How to Solve Set Constraints

Jens Palsberg

October 1, 2008

We will explain how to solve a particular class of set constraints in cubic time.

1 Set Constraints

Let C be a finite set of constants, and let V be a set of variables. A set constraint is a conjunction of constraints of the forms:

$$c \in v$$
$$c \in v \Rightarrow v' \subseteq v''$$

where $c \in C$ and $v, v', v'' \in V$. For a particular set constraint, we use C to denote the finite set of constants that occur in that set constraint, and we use V to denote the finite set of variables that occur in that set constraint.

We use 2^C to denote the powerset of C. A set constraint has solution $\varphi: V \to 2^C$ if

- for each conjunct of the form $c \in v$, we have $c \in \varphi(v)$ and
- for each conjunct of the form $c \in v \Rightarrow v' \subseteq v''$, we have $c \in \varphi(v) \Rightarrow \varphi(v') \subseteq \varphi(v'')$.

We say that a set constraint is satisfiable if it has a solution.

Theorem 1 Every set constraint is satisfiable.

Proof. The mapping $\lambda v : V.C$ is a solution of the set constraint.

For two mappings $\varphi, \psi: V \to 2^C$, we define the binary intersection $\varphi \sqcap \psi$ as:

$$\varphi \sqcap \psi = \lambda v : V.(\varphi(v) \cap \psi(v))$$

Theorem 2 For a given set constraint, the binary intersection of two solutions is itself a solution.

Proof. Suppose $\varphi, \psi: V \to 2^C$ are both solutions. Let us examine each of the conjuncts of the set constraint.

- For a conjunct of the form $c \in v$, we have from φ, ψ being solutions that $c \in \varphi(v)$ and $c \in \psi(v)$. From $c \in \varphi(v)$ and $c \in \psi(v)$ we have $c \in (\varphi(v) \cap \psi(v))$ which is equivalent to $c \in (\varphi \sqcap \psi)(v)$.
- for each conjunct of the form $c \in v \Rightarrow v' \subseteq v''$, we have from φ, ψ being solutions that $c \in \varphi(v) \Rightarrow \varphi(v') \subseteq \varphi(v'')$ and $c \in \psi(v) \Rightarrow \psi(v') \subseteq \psi(v'')$. We want to show $c \in (\varphi \sqcap \psi)(v) \Rightarrow (\varphi \sqcap \psi)(v') \subseteq (\varphi \sqcap \psi)(v'')$. Suppose $c \in (\varphi \sqcap \psi)(v)$. From $c \in (\varphi \sqcap \psi)(v)$ and the definition of \sqcap , we have $c \in (\varphi(v) \cap \psi(v))$, so we have $c \in \varphi(v)$ and $c \in \psi(v)$. From $c \in \varphi(v)$ and $c \in \varphi(v) \Rightarrow \varphi(v') \subseteq \varphi(v'')$, we have $\varphi(v') \subseteq \varphi(v'')$. From $c \in \psi(v)$ and $c \in \psi(v) \Rightarrow \psi(v') \subseteq \psi(v'')$, we have $\psi(v') \subseteq \psi(v'')$. From $\varphi(v') \subseteq \varphi(v'')$ and $\psi(v') \subseteq \psi(v'')$, we have $(\varphi(v') \cap \psi(v')) \subseteq (\varphi(v'') \cap \psi(v''))$, which is equivalent to $(\varphi \sqcap \psi)(v') \subseteq (\varphi \sqcap \psi)(v'')$, and that is the desired result.

For the space $V \to 2^C$, we define an ordering \subseteq as follows. We say that $\varphi \subseteq \psi$ if and only if for all $v \in V : \varphi(v) \subseteq \psi(v)$. For a set $S \subseteq (V \to 2^C)$, we say that an element $\varphi \in S$ is the \subseteq -least element of S if for all $\psi \in S : \varphi \subseteq \psi$.

Theorem 3 Every set constraint has a \subseteq -least solution.

Proof. Let a particular set constraint be given. The space of possible solutions of the set constraint is $V \to 2^C$, which is a finite set. Let $S \subseteq (V \to 2^C)$ be the set of solutions of the set constraint. From $V \to 2^C$ being finite, we have that S is finite. From Lemma 1 we have that S is nonempty. So, S is a nonempty, finite set. Let $S = \{\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_n\}$. Let $\varphi = (\ldots ((\varphi_1 \sqcap \varphi_2) \sqcap \varphi_3) \ldots \sqcap \varphi_n)$. From Lemma 2 we have that φ is a solution of the set constraint, so $\varphi \in S$. Additionally, we have that for all $i \in 1..n : \varphi \subseteq \varphi_i$. So, φ is the \subseteq -least solution of the set constraint.

2 Solving Set Constraint Efficiently

We will translate the problem of solving a set constraint into a graph problem. For a given set constraint, the initial graph has one node for each element of V, and an empty set of edges. Additionally, each node v is associated with a bit vector B_v of length |C| in which every bit is initially 0. For a bit vector B_v and a constant $c \in C$, we denote the entry for c in B_v by $B_v[c]$. Finally, every bit in every bit vector is associated with a list $L_v[c]$ of constraints of the form $v' \subseteq v''$; all those lists are initially empty.

The initial graph represents the mapping $\lambda v : V.\emptyset$. The idea is that for a given $v \in V$, the bit vector B_v represents a subset of C. When all the bits are 0, the bit vector B_v represents the empty set. An edge in the graph implies a subset relationship. If the graph has an edge from v to v', it implies that the set represented by the bit vector associated with v is a subset of the set represented by the bit vector associated with v'.

We now process each conjunct of the set constraint in turn. The processing will add edges, change bits from 0 to 1, and add constraints to the constraint lists associated with bits in the bit vectors. We will use the following two procedures propagate and insert-edge as key subroutines.

```
procedure propagate(v:Node, c:Constant) {

if (B_v[c] == 0) {

B_v[c] = 1

for (each element (v' \subseteq v'') of L_v[c]) { insert-edge(v', v'') }

for (each edge (v, v')) { propagate(v', c) }

}

procedure insert-edge(v, v':Node) {

insert an edge (v, v')

for (c \text{ such that } B_v[c] == 1) { propagate(v', c) }

}
```

For a constraint of the form $c \in v$, we execute $\operatorname{propagate}(v,c)$. For a constraint of the form $c \in v \Rightarrow v' \subseteq v''$, we execute:

if $(B_v[c] == 0)$ { add the constraint $(v' \subseteq v'')$ to the list $L_v[c]$ } else { insert-edge(v', v'') }

When we have processed all the constraints, the resulting graph represents the \subseteq -least solution of the set constraint.

We will now analyze the time complexity of the constraint processing. We will do the analysis for a set constraint with |C| = O(n), |V| = O(n), O(n) conjuncts of the form $c \in v$, and $O(n^2)$ conjuncts of the form $c \in v \Rightarrow v' \subseteq v''$. For each conjunct, the algorithm performs a constant amount of work except for the calls to the propagate subroutine. So, the total time is $O(n^2)$ plus the time spent by propagate. The propagate routine itself performs a constant amount of work except for recursive calls! So the key to analyzing the time spent by propagate is to determine the *number of calls* to the propagate routine. The processing of the set constraint generates $O(n^3)$ immediate calls to propagate. The recursion involved in each call to propagate stops when finding a bit that is 1. So, for each $c \in C$, the total work of all calls of the form propagate(v,c) is given by the number of edges in the graph, which is $O(n^2)$. To sum up, the total time is $O(n^2) + (O(n) \times O(n^2)) = O(n^3)$.



Available online at www.sciencedirect.com



Information Processing Letters 98 (2006) 150-155

Information Processing Letters

www.elsevier.com/locate/ipl

Optimal register allocation for SSA-form programs in polynomial time

Sebastian Hack*, Gerhard Goos

Institut für Programmstrukturen und Datenorganisation, Adenauerring 20a, 76131 Karlsruhe, Germany

Received 11 February 2005; received in revised form 23 November 2005

Available online 17 February 2006

Communicated by F. Meyer auf der Heide

Abstract

This paper gives a constructive proof that the register allocation problem for a uniform register set is solvable in polynomial time for SSA-form programs.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Combinatorial problems; Graph algorithms

1. Introduction

Register allocation is the task in a compiler, which maps (temporary) variables to processor registers. The most prominent approach is to map this task to a graph coloring problem. The nodes of the so-called *interference graph* are formed by the temporaries of the program. Whenever the compiler finds out, that two temporaries cannot be held in the same register (due to *interference*), an edge is drawn between the corresponding nodes in the interference graph. Colors correspond to processor registers. Thus, having *k* registers, a *k*-coloring of the interference graph forms a correct register assignment.

Chaitin et al. [4] show that for each undirected graph a program can be given which has this graph as its interference graph. So, since graph coloring is NP-complete,

* Corresponding author.

register allocation must also be. However, if one only considers programs in SSA-form (e.g., refer to [5]) the situation changes. It turns out, that interference graphs of SSA-form programs belong to the class of *chordal* graphs, which is in turn a subset of the class of *perfect* graphs. It is well known, that chordal graphs can be colored in quadratic time. This also answers the question posed by Andersson [1] whether interference graphs are perfect.

This paper is structured as follows: First, we describe the model of a program used in this paper. Then we give a new definition of liveness for SSA-form programs. After quoting basic facts from graph theory, we will prove that interference graphs of SSA-form programs always have perfect elimination orders and show how they are determined.

2. SSA-form programs

Here, we consider a program (in SSA-form) given by its control flow graph (CFG) whose nodes are made up of labeled single instructions. We will therefore identify

E-mail addresses: hack@ipd.info.uni-karlsruhe.de (S. Hack), ggoos@ipd.info.uni-karlsruhe.de (G. Goos).

^{0020-0190/\$ –} see front matter @ 2006 Elsevier B.V. All rights reserved. doi:10.1016/j.ipl.2006.01.008

if then	if then
$\ell_2: x \leftarrow \cdots$	$\ell_2: x_1 \leftarrow \cdots$
else	else
$\ell_3: x \leftarrow \cdots$	$\ell_3: x_2 \leftarrow \cdots$
end	end
$\ell_4: y \leftarrow x+1$	$\ell_4' \colon x_3 \leftarrow \phi(x_1, x_2)$
	$\ell_4: v_1 \leftarrow x_3 + 1$

Fig. 1. Program fragment and its equivalent in SSA-form.

the node and its label in the following. Let us call the set of labels \mathcal{L} . The CFG has one distinct start node which has no predecessor nodes and is denoted by **start**. The instruction corresponding to a node is of the following form:

$$\ell: (d_1,\ldots,d_m) \leftarrow \tau(u_1,\ldots,u_n).$$

We denote the operation τ at a label ℓ by Op_{ℓ} . We call $\operatorname{Res}_{\ell} = \{d_1, \ldots, d_m\}$ the ordered set of result values and $\operatorname{Arg}_{\ell} = \{u_1, \ldots, u_n\}$ the ordered set of argument values at the label ℓ . All d_1, \ldots, d_m and u_1, \ldots, u_n are elements of the set of (abstract) values \mathcal{V} of the considered program. Given a label ℓ , let us denote $\operatorname{Arg}_{\ell}(i)$ the *i*th argument to the operation at ℓ . Each label has an ordered set of *k* predecessor labels which we will denote by $P_{\ell}^1, \ldots, P_{\ell}^k$.

We will also write $\ell' \to {}^i \ell$ if $\ell' = P_{\ell}^i$. If we do not care about the position, we simply write $\ell' \to \ell$ to denote that ℓ' is a predecessor to ℓ . A path p is an ordered set $\{\ell_1, \ldots, \ell_n\}$ of at least two nodes, for which $\ell_1 \to \ell_2, \ell_2 \to \ell_3, \ldots, \ell_{n-1} \to \ell_n$ holds. To indicate that a set $p = \{\ell_1, \ldots, \ell_n\}$ is a path, we write $p: \ell_1 \to \cdots \to \ell_n$.

Finally, we say a label ℓ dominates another label ℓ' if each path from **start** to ℓ' contains ℓ , writing $\ell \leq \ell'$. Note, since \leq is reflexive, $\ell \leq \ell$.

We require the program to be given in SSA-form, which means that each variable is statically only assigned once. Usually, when a program is transferred into SSA-form, for each definition of a non-SSA variable x, a SSA variable x_i is created. The usages of the variables then have to be adjusted to refer to the corresponding SSA variable. This is not always possible. It might be dependent on the control flow, which definition of the variable is applicable at the usage. Consider the left program in Fig. 1. At label ℓ_4 , it is dependent on the control flow whether the definition at ℓ_2 or the one at ℓ_3 is relevant for ℓ_4 . In SSA-form programs, a special instruction, called ϕ -function is used to disambiguate multiple definitions by control flow. A ϕ function copies its *i*th parameter to its result, if it was reached via P_{ℓ}^k . Note, that a basic block can have multiple ϕ -functions. So, since the program is in SSA-form,

$$\ell_1: a \leftarrow \cdots$$

if ... then
$$\ell_2: \cdots$$

else
$$\ell_3: b \leftarrow \cdots$$

end
$$\ell_4: y \leftarrow \phi'(a, b)$$

Fig. 2. Liveness at
$$\phi'$$
-functions.

for each SSA-variable¹ v there is exactly one label D_v for which $v \in \text{Res}_{D_v}$.

Since we allow only one instruction per label, we replace the set of all ϕ -operations in a basic block

$$y_1 = \phi(x_{11}, \dots, x_{1n}),$$
$$\dots$$
$$y_m = \phi(x_{m1}, \dots, x_{mn})$$

by the more concise ϕ' -operation:

$$\ell: (y_1, \ldots, y_m) = \phi'(x_{11}, \ldots, x_{1n}, \ldots, x_{m1}, \ldots, x_{mn})$$

which sets $y_i = x_{ij}$ if ℓ was reached via P_{ℓ}^j . It is convenient to define $\operatorname{Arg}_{\ell}^{\prime}(j) = \{x_{ij} \mid 1 \leq i \leq m\}$ subsuming all operands of a ϕ' -operation which refer to P_{ℓ}^j . Note, that ϕ' is totally equivalent to the traditional ϕ , since SSA semantics states that all ϕ -operations in a basic block are evaluated *simultaneously*. In the following, everything stated for ϕ' -operations implicitly holds for ϕ -operations. Thus we will only use ϕ' .

2.1. Liveness in SSA-form programs

To perform register allocation on SSA-form programs, a precise notion of liveness is needed. The standard definition of liveness

A variable v is live at a label ℓ , if there is a path from ℓ to a usage of v not containing a definition of v.

cannot be straightforwardly transferred to SSA-form programs. The problem arises with ϕ - and accordingly ϕ' -operations. Consider the program in Fig. 2. Surely, *a* is not live at label ℓ_3 although there is a path from ℓ_3 to a usage of *a*, namely ℓ_4 . The cause for this oddity is that the usual notion of *usage* does not hold for ϕ' -operations. In addition to their arguments, a ϕ' -operation also uses control flow information to produce its result. To make the traditional definition of liveness work, we have to incorporate the predecessor by which a label was reached into the notion of usage:

¹ SSA variables are often called *values*.

Definition 1 (Usage).

$$\text{usage} : \mathbb{N} \times \mathcal{L} \times \mathcal{V} \to \mathbb{B}, \\ (i, \ell, v) \mapsto \begin{cases} v \in \operatorname{Arg}_{\ell} & \text{if } \operatorname{Op}_{\ell} \neq \phi', \\ v \in \operatorname{Arg}_{\ell}'(i) & \text{if } \operatorname{Op}_{\ell} = \phi'. \end{cases}$$

Now, a usage is not only dependent on a label and a value but also on a number which represents the predecessor by which the label was reached. In our example in Fig. 2, usage $(1, \ell_4, a)$ is true, since *a* is indeed used if ℓ_4 is entered via ℓ_2 . usage $(2, \ell_4, a)$ is false, since *a* is *not* used if ℓ_4 is reached through ℓ_3 . If the operation at a label is not ϕ' , this definition resembles the common concept of usage by simply ignoring the predecessor index.

The traditional definition of liveness quoted above, uses paths which end in usages of some variable to define liveness. In this traditional setting, usages and paths are unrelated. With Definition 1, paths and usages are no longer unrelated. So it is straightforward to merge them in one term.

Definition 2 (*Usepath*). A path $p: \ell_1 \to \cdots \to \ell_n$ is a *usepath* from ℓ_1 to ℓ_n concerning a value v, if v is used at ℓ_n regarding this path. More formally:

usepath:
$$\mathcal{L}^{n} \times \mathcal{V} \to \mathbb{B}$$
,
 $(p: \ell_{1} \to \dots \to \ell_{n}, v)$
 $\mapsto \begin{cases} \text{usage}(i, \ell_{n}, v) & \text{if } p = \ell_{1} \to^{i} \ell_{n}, \\ \text{usepath}(\ell_{2} \to \dots \to \ell_{n}, v) & \text{otherwise.} \end{cases}$

Referring to the example in Fig. 2, ℓ_1 , ℓ_2 , ℓ_4 is a usepath of *a* and ℓ_1 , ℓ_3 , ℓ_4 is a usepath of *b*.

Using this definition of usage together with the traditional definition of liveness stated above, one obtains a realistic model of liveness in SSA programs:

Definition 3 (*Liveness*). A value v is live at a label ℓ_1 iff there exists a label ℓ_n with usepath($\ell_1 \rightarrow \ell_2 \rightarrow \cdots \rightarrow \ell_n, v$) and $D_v \notin \{\ell_2, \ldots, \ell_n\}$.

We use the definition of usepaths to re-formulate the notion of a strict program coined by Budimlić et al. [3].

Definition 4 (*Strict program*). A program is called *strict*, iff for each value v each path from **start** to some label ℓ with usepath(**start** $\rightarrow \cdots \rightarrow \ell, v$) contains the definition of v.

From now on, we will only consider strict programs.² The next lemma is essential for the rest of this paper and has also been given by Budimlić relying on a slightly different liveness definition.

Lemma 5. Each label ℓ at which a value v is live is dominated by D_v .

Proof. Suppose, ℓ is not dominated by D_v . Then, there is path from **start** to ℓ not containing D_v . From the fact, that v is live at ℓ follows, that there is a usepath of v from ℓ to some ℓ' . So there is a usepath from **start** to ℓ' not containing D_v which contradicts the definition of a strict program. \Box

2.2. Common facts about SSA-form programs

Since the definition of liveness given above seems rather unusual, we shortly derive some well-known facts about SSA-form programs from our definition. These facts are not vital for the rest of the paper and are only given to clarify certain properties of SSA-form programs.

Corollary 6. Each value v, used in a non- ϕ' -operation at a label ℓ is dominated by its definition.

Proof. Then, for each predecessor of P_{ℓ}^{j} of ℓ , usage (j, ℓ, v) holds. With Definition 3, v is live at each P_{ℓ}^{j} . With Lemma 5, D_{v} dominates each predecessor of ℓ . Thus D_{v} also dominates ℓ .

Corollary 7. If a value $v \in \operatorname{Arg}_{\ell}^{\prime}(i)$ for a ϕ^{\prime} -operation at a label ℓ and some *i*, then the definition of *v* dominates P_{ℓ}^{i} .

Proof. Surely, usage (i, ℓ, v) holds. So $p: P_{\ell}^{i} \to \ell$ is a usepath concerning v. So, after Definition 3, v is live at P_{ℓ}^{i} . Thus, with Lemma 5, $D_{v} \leq P_{\ell}^{i}$.

Corollary 8. Let ℓ be a label with $Op_{\ell} \neq \phi'$. Each pair of values $v, w \in Arg_{\ell}$ interfere.

Proof. Due to Definition 3, v and w are live at each predecessor of that label. So v and w interfere.

Often, one can read statements like:

 $^{^2}$ Surely, each non-strict program can be turned into a strict one by inserting instructions which initially define the variables by an arbitrary value.

- ϕ' -operations do not cause interferences.
- Concerning liveness, φ'-operations can be treated as if they had no arguments.
- φ'-operations extend the lifetimes of their arguments to the end of the respective predecessor label.

All these statements try to describe Corollary 8 the other way around. With the definition of usepaths, they are covered implicitly. In our model the basic assumption is, that the property of usage is always tied to a value *and* a path, which makes Corollary 8 the "special" case.

3. Graph theory

Here we quote definitions from basic graph theory and the theory of perfect graphs important to this paper. Let G = (V, E) be an undirected graph. If there is an edge from $v \in V_G$ to $w \in V$, we write $vw \in E_G$. We leave out the *G* in E_G , V_G if it is clear from the context, which graph is considered. We call a graph *G* complete, iff for each $v, w \in V$, there is an edge $vw \in E$. We call *G'* an induced subgraph of *G*, if $V_{G'} \subseteq V_G$ and for all nodes $v, w \in V_{G'}, vw \in E_G \rightarrow vw \in E_{G'}$ holds.

Definition 9 (*Simplicial vertex*). A vertex $v \in V_G$ is called *simplicial*, if v and its neighbors induce a complete subgraph in G.

Definition 10 (*Perfect Elimination Order, PEO*). We call a linearly ordered sequence of vertices v_1, \ldots, v_n a *perfect elimination order*, if each v_i is simplicial in $G - \{v_1, \ldots, v_{n-1}\}$ where $G - \{a_1, \ldots, a_m\}$ is the graph obtained by deleting all vertices $\{a_1, \ldots, a_m\}$ and their incident edges from the graph.

The class of graphs for which perfect elimination orders exist are also called *chordal* or *triangulated* graphs. Gavril [6] gives an algorithm for coloring chordal graphs in $O(|V|^2)$. The algorithm constructs a PEO for a given chordal graph by searching and removing a simplicial node from the graph each step. Afterwards, the nodes are inserted into the graph in reverse order. Each node is assigned a color which is not occupied by a neighbor of the node to insert. It is further proven, that this algorithm leads to a minimal coloring of the graph.

4. Interference graphs of SSA-programs

We say two values v and v' interfere, iff there is a label ℓ where v and v' are live (regarding Definition 3). Now, we can define the interference graph IG = (V, E) of an SSA-form program. The set of vertices is made up by the values occurring in the program, $V_{IG} = \mathcal{V}$. Since nodes in the interference graph and values are identical, we identify both terms in the following. We draw an edge between to values v and v' iff they interfere and write $vv' \in E_{IG}$. The following lemmas lead to a theorem that connects the dominance relation of a program to perfect elimination orders in the interference graph of that program. Lemmas 11 and 12 have also been shown by Budimlić et al. [3] and are given for the sake of completeness, here.

Lemma 11 shows that each edge in the interference graph is directed according to the dominance relationship of the values their nodes represent.

Lemma 11. If two values v and w are live at some label ℓ , either D_v dominates D_w or vice versa.

Proof. By Lemma 5, D_v and D_w dominate ℓ . Thus, either D_v dominates D_w or D_w dominates D_v . \Box

The next lemma shows what is trivial in basic blocks also holds for complete programs in SSA-form: if one value starts living before another (it dominates the other) and both interfere, the value is still alive at the definition of the other.

Lemma 12. If v and w interfere and $D_v \leq D_w$, then v is live at D_w .

Proof. Assume, v is not live at D_w . Then there is no usepath from D_w to some ℓ' concerning v. Since all labels where w is live are dominated by D_w , there is no label where v and w are simultaneously live. So v and w do not interfere which contradicts the proposition. \Box

Lemma 13 shows how the dominance order relation is reflected by the interference graph. It says that all values dominating a value v and interfering with v form a clique in the interference graph. This is used later on to connect the perfect elimination order to the dominance relation.

Lemma 13. $ab, bc \in E$ and $ac \notin E$. If $D_a \preceq D_b$, then $D_b \preceq D_c$.

Proof. Due to Lemma 11, either $D_b \leq D_c$ or $D_c \leq D_b$. Assume $D_c \leq D_b$. Then (with Lemma 12), *c* is live at D_b . Since *a* and *b* also interfere and $D_a \leq D_b$, *a* is also live at D_b . So, *a* and *c* are live at D_b which cannot be by precondition. \Box **Lemma 14.** A value v can extend a perfect elimination order, if each value whose definition is dominated by D_v is already contained in the PEO.

Proof. To extend a PEO, v must be simplicial. Assume v is not simplicial. Then there exist two neighbors a, b for which $va, vb \in E$ but $ab \notin E$ (by Definition 9). Due to the proposition, all values whose definitions are dominated by D_v have already been removed from *IG*. Thus, D_a dominates D_v . By Lemma 13, D_v dominates D_b which contradicts the proposition. Thus, v is simplicial. \Box

Theorem 15. *The interference graph of a SSA-form program P is chordal.*

Proof. Consider the tree T of immediate dominators (cf. [9]) concerning the control flow graph of P. We start with an empty PEO and apply Lemma 14 recursively on T starting at the leaves. This constructs a PEO for the interference graph of P. Since each graph which has a PEO is chordal, cf. [7], the interference graph of P is chordal.

As we can see by Theorem 15, a post-order visitation of the dominator tree of a program yields a PEO. Since the vertices are colored reversely along a PEO, a preorder visitation of the dominator tree defines a sequence in which the values can be colored optimally using the algorithm described in [6]. Since the liveness analysis annotates the set of live values to each label, we always have the set of neighbors present upon coloring a value. Thus, we do not have to construct the interference graph itself.

5. Leaving the SSA-form

As no real-world processor has a ϕ -instruction the compiler has to destroy the SSA-form of the program at some point in time. Conventionally, ϕ -functions are replaced by copies in its predecessor blocks to implement the control flow dependent copy as described in Section 2. In doing so, one modifies the interference graph of the program since new interferences are introduced, as shown in Fig. 3. x_3 is now interfering with y_1 and y_2 which has not been the case in the interference graph of the SSA-form program. These interferences are introduced due to the fact, that the atomic, simultaneous evaluation by ϕ' -functions (as mentioned in Section 2) is broken down to a sequential set of operations. In the worst case, these new interferences render the interference graph un-chordal which also might invalidate our

if then	if then
$\ell_2: x_1 \leftarrow \cdots$	$\ell_2: x_1 \leftarrow \cdots$
$\ell_3: y_1 \leftarrow \cdots$	$\ell_3: y_1 \leftarrow \cdots$
else	$\ell_4: x_3 \leftarrow x_1$
$\ell_4: x_2 \leftarrow \cdots$	$\ell_5: y_3 \leftarrow y_1$
$\ell_5: y_2 \leftarrow \cdots$	else
end	$\ell_4: x_2 \leftarrow \cdots$
$\ell_6: (x_3, y_3) \leftarrow \phi'(x_1, y_1, x_2, y_2)$	$\ell_5: y_2 \leftarrow \cdots$
	$\ell_4: x_3 \leftarrow x_2$
	$\ell_5: y_3 \leftarrow y_2$
	end

Fig. 3. A program fragment in SSA-form and after destroying SSA using copy instructions.

register allocation. We can however destroy the SSAform of the program and convert a register allocation with k registers of a SSA-form program into a register allocation with exactly the same number of registers for the resulting non-SSA-form program.

Consider a ϕ' -function

$$\ell:(y_1,\ldots,y_m) \leftarrow \phi'(x_{11},\ldots,x_{1n},\ldots,x_{m1},\ldots,x_{mn})$$

at some label ℓ . Arriving at ℓ and coming from P_{ℓ}^{J} , the x_{ij} are copied *at once* into the y_i according to SSA semantics. Consider a valid register allocation The simultaneous assignment given by the ϕ' -function corresponds to a *l*-to-*m* mapping of registers, where $l \leq m \leq k$, and *k* represents the number of register available.³ Furthermore, all y_i are assigned to different registers, since all y_i interfere. So the question of removing a ϕ' -function reduces to implementing *l*-to-*m* mappings between registers on the control flow edges to the ϕ 's label using ordinary processor instructions *and m* registers.

Theorem 16. Any simultaneous assignment from l registers to m registers, where $l \leq m$, can be implemented with m registers using only copy and swap instructions.

Proof. Consider following simultaneous assignment:

$$(y_1,\ldots,y_m) \leftarrow (\underbrace{x_1,\ldots,x_1,\ldots,x_l,\ldots,x_l}_{m}).$$

In general, there may be multiple y_i to which the same x_j is assigned. For each x_j we arbitrarily pick one of the y_i to which it is assigned and denote it by $[x_j]$. Note, that this induces an equivalence relation \sim on the y_1, \ldots, y_n : $y_i \sim y_j$ if there is some x_k which is assigned to y_i and y_j . Thus, y_i and y_j are members of the equiv-

³ Note, that the same value *x* can be assigned to different y_i by a ϕ' -function. E.g., $(y_1, y_2) = \phi'(a_1, b_1, a_1, b_2)$.

155

alence class $[x_k]$. We denote the set of the x_j by X and the set of the equivalence classes $[x_j]$ by [X].

Consider a register allocation $\rho: \mathcal{V} \to \mathcal{R}$ of the SSAform program obtained by the algorithm described in the last section. Let π by a function mapping $\rho(x_j)$ to $\rho([x_j])$. Note, that since all y_i interfere, all $[x_j]$ also interfere and by the fact that all x_j interfere, π is injective. π may also be partial, since l might be smaller than $k = |\mathcal{R}|$.

Each register in $\rho([X])$ which is not in $\rho(X)$ can be assigned immediately since its value is not needed anymore. So we apply the following recursive scheme: Let $y = \pi(x)$. If $y \in [X]$ and $y \notin X$ we issue a copy from $\rho(x)$ to $\rho(y)$ and recursively consider the mapping $\pi|_{X \setminus \{x\}}$.

At the end of this recursive procedure, either all elements of [X] are processed and thus all of X since π is injective or the remaining subset of [X] equals the one of X. Thus this rest represents a permutation of registers which can be, as known from basic linear algebra, implemented by a sequence of swap instructions. If the processor does not possess swap instructions, one can use three xors, adds or subs (cf. [10]).

Finally, each $y_i \in [x_j]$ can be processed by copying $\rho([x_i])$ to $\rho(y_i)$. \Box

6. Conclusions

We have shown that the interference graphs of strict SSA-form programs are chordal which leads to a coloring algorithm running in quadratic time. Furthermore, the coloring algorithm does not need to have the interference graph materialized but uses a coloring sequence induced by the dominance relation of the program. We also showed, how a register allocation of a SSA-form program using *m* registers can be turned into a register allocation of a corresponding non-SSA program using also no more than *m* registers, by implementing the ϕ' -functions properly.

7. Related work

At the time this paper was submitted, chordal graphs played no role in register allocation. Meanwhile, they have drawn the attention of other researchers in the area. The paper which initiated our research on the topic is by Andersson [1] who investigated interference graphs in real-world compilers and found that all of them were 1-perfect (1-perfectness means that the chromatic number of the graph equals the size of its largest clique). The result of the quest for a proof of this observation is this paper. Independently of us, Brisk proved the perfectness of strict SSA-form programs [2]. In his proof he also shows their chordality without referring to it. Pereira and Palsberg extended Andersson's studies and found that, with SSA-optimizations enabled, 95% of the (non-SSA!) interference graphs of the Java standard library were chordal. They use this fact to derive new spilling and coalescing heuristics for graph coloring register allocators. Finally, the authors of this paper published a more technical proof (without using perfect elimination orders) of this paper's result in a technical report [8].

Acknowledgements

We thank our colleagues Michael Beck, Marco Gaertler, Götz Lindenmaier and especially Rubino Geiß for many fruitful discussions. We also thank the anonymous referees for their suggestions helping to improve this paper.

References

- C. Andersson, Register allocation by optimal graph coloring, in: G. Hedin (Ed.), CC 2003, Lecture Notes in Comput. Sci., vol. 2622, Springer-Verlag, Heidelberg, 2003, pp. 33–45.
- [2] P. Brisk, F. Dabiri, J. Macbeth, M. Sarrafzadeh, Polynomial time graph coloring register allocation, in: 14th Internat. Workshop on Logic and Synthesis, ACM Press, New York, 2005.
- [3] Z. Budimlić, K.D. Cooper, T.J. Harvey, K. Kennedy, T.S. Oberg, S.W. Reeves, Fast copy coalescing and live-range identification, in: Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM Press, New York, 2002, pp. 25–32.
- [4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, Register allocation via coloring, J. Comput. Languages 6 (1981) 45–57.
- [5] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, An efficient method of computing static single assignment form, in: Symp. on Principles of Programming Languages, ACM Press, New York, 1989, pp. 25–35.
- [6] F. Gavril, Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and independent set of a chordal graph, SIAM J. Comput. 1 (2) (1972) 180–187.
- [7] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.
- [8] S. Hack, Interference graphs of programs in SSA-form, Tech. Rep. 2005-25, Universität Karlsruhe, June 2005.
- [9] T. Lengauer, R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, Trans. Programm. Languages Systems 1 (1) (1979) 121–141.
- [10] H.S. Warren, Hacker's Delight, Addison-Wesley, Reading, MA, 2003.

Register Allocation via Coloring of Chordal Graphs

Fernando Magno Quintão Pereira and Jens Palsberg

UCLA Computer Science Department University of California, Los Angeles

Abstract. We present a simple algorithm for register allocation which is competitive with the iterated register coalescing algorithm of George and Appel. We base our algorithm on the observation that 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A greedy algorithm can optimally color a chordal graph in time linear in the number of edges, and we can easily add powerful heuristics for spilling and coalescing. Our experiments show that the new algorithm produces better results than iterated register coalescing for settings with few registers and comparable results for settings with many registers.

1 Introduction

Register allocation is one of the oldest and most studied research topics of computer science. The goal of register allocation is to allocate a finite number of machine registers to an unbounded number of temporary variables such that temporary variables with interfering live ranges are assigned different registers. Most approaches to register allocation have been based on graph coloring. The graph coloring problem can be stated as follows: given a graph G and a positive integer K, assign a color to each vertex of G, using at most K colors, such that no two adjacent vertices receive the same color. We can map a program to a graph in which each node represents a temporary variable and edges connect temporaries whose live ranges interfere. We can then use a coloring algorithm to perform register allocation by representing colors with machine registers.

In 1982 Chaitin [8] reduced graph coloring, a well-known NP-complete problem [18], to register allocation, thereby proving that also register allocation is NP-complete. The core of Chaitin's proof shows that the interference relations between temporary variables can form any possible graph. Some algorithms for register allocation use integer linear programming and may run in worst-case exponential time, such as the algorithm of Appel and George [2]. Other algorithms use polynomial-time heuristics, such as the algorithm of Briggs, Cooper, and Torczon [5], the Iterated Register Coalescing algorithm of George and Appel [12], and the Linear Scan algorithm of Poletto and Sarkar [16]. Among the polynomial-time algorithms, the best in terms of resulting code quality appears to be iterated register coalescing. The high quality comes at the price of handling spilling and coalescing of temporary variables in a complex way. Figure 1



Fig. 1. The iterated register coalescing algorithm.



Fig. 2. (a) A chordal graph. (b-c) Two non-chordal graphs.

illustrates the complexity of iterated register coalescing by depicting the main phases and complicated pattern of iterations of the algorithm. In this paper we show how to design algorithms for register allocation that are simple, efficient, and competitive with iterated register coalescing.

We have observed that the interference graphs of real-life programs tend to be *chordal* graphs. For example, 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler. A graph is chordal if every cycle with four or more edges has a *chord*, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. (Chordal graphs are also known as 'triangulated', 'rigid-circuit', 'monotone transitive', and 'perfect elimination' graphs.) The graph in Figure 2(a) is chordal because the edge *ac* is a chord in the cycle *abcda*. The graph in Figure 2(b) is nonchordal because the cycle *abcda* is chordless. Finally, the graph in Figure 2(c) is non-chordal because the cycle *abcda* is chordless, just like in Figure 2(b).

Chordal graphs have several useful properties. Problems such as minimum coloring, maximum clique, maximum independent set and minimum covering by cliques, which are NP-complete in general, can be solved in polynomial time for chordal graphs [11]. In particular, optimal coloring of a chordal graph G = (V, E) can be done in O(|E| + |V|) time.

In this paper we present an algorithm for register allocation, which is based on a coloring algorithm for chordal graphs, and which contains powerful heuristics for spilling and coalescing. Our algorithm is simple, efficient, and modular, and it performs as well, or better, than iterated register coalescing on both chordal graphs and non-chordal graphs.

The remainder of the paper is organized as follows: Section 2 discusses related work, Section 3 summarizes some known properties and algorithms for chordal graphs, Section 4 describes our new algorithm, Section 5 presents our experimental results, and Section 6 concludes the paper.

2 Related Work

We will discuss two recent efforts to design algorithms for register allocation that take advantage of properties of the underlying interference graphs. Those efforts center around the notions of perfect and 1-perfect graphs. In a 1-perfect graph, the chromatic number, that is, the minimum number of colors necessary to color the graph, equals the size of the largest clique. A perfect graph is a 1-perfect graph with the additional property that every induced subgraph is 1-perfect. Every chordal graph is perfect, and every perfect graph is 1-perfect.

Andersson [1] observed that all the 27,921 interference graphs made publicly available by George and Appel [3] are 1-perfect, and we have further observed that 95.6% of those graphs are chordal when the interferences between precolored registers and temporaries are not considered. Andersson also showed that an optimal, worst-case exponential time algorithm for coloring 1-perfect graphs is faster than iterated register coalescing when run on those graphs.

Recently, Brisk et al. [6] proved that *strict* programs in SSA-form have *per-fect* interference graphs; independently, Hack [14] proved the stronger result that strict programs in SSA-form have *chordal* interference graphs. A strict program [7] is one in which every path from the initial block until the use of a variable v passes through a definition of v. Although perfect and chordal graphs can be colored in polynomial time, the practical consequences of Brisk and Hack's proofs must be further studied. SSA form uses a notational abstraction called *phi-function*, which is not implemented directly but rather replaced by copy instructions during an SSA-elimination phase of the compiler. Register allocation after SSA elimination is NP-complete [15].

For example, Figure 3(a) shows a program with a non-chordal interference graph, Figure 3(b) shows the program in SSA form, and Figure 3(c) shows the program after SSA elimination. The example program in Figure 3(a) has a cycle of five nodes without chords: a-d-e-c-b-a. In the example in Figure 3(b), $e = phi(e_1, e_2)$ will return e_2 if control reaches block 8 through block 7, and will return e_1 if control reaches block 8 through block 4. The SSA semantics states that all phi-functions at the beginning of a block must be evaluated simultaneously as the first operation upon entering that block; thus, live ranges that reach block 8 do not interfere with live ranges that leave block 8. Hack [14] used this observation to show that phi-functions break chordless cycles so strict programs in SSA-form have chordal interference graphs. The example program after SSA elimination, in Figure 3(c), has an interference graph which is non-chordal, non-perfect, and even non-1-perfect: the largest clique has two nodes but three colors are needed to color the graph. Note that the interference graph has a cycle of seven nodes without chords: a-d-e1-c1-e-c2-b-a.

For 1-perfect graphs, recognition and coloring are NP-complete. Perfect graphs can be recognized and colored in polynomial time, but the algorithms are highly complex. The recognition of perfect graphs is in $O(|V|^9)$ time [9]; the complexity of the published coloring algorithm [13] has not been estimated accurately yet. In contrast, chordal graphs can be recognized and colored in O(|E| + |V|) time, and the algorithms are remarkably simple, as we discuss next.



Fig. 3. (a) A program with a non-chordal interference graph, (b) the program in SSA form, (c) the program after SSA elimination.

3 Chordal Graphs

We now summarize some known properties and algorithms for chordal graphs. For a graph G, we will use $\Delta(G)$ to denote the maximum outdegree of any vertex in G, and we will use N(v) to denote the set of neighbors of v, that is, the set of vertices adjacent to v in G. A clique in an undirected graph G = (V, E) is a subgraph in which every two vertices are adjacent. A vertex $v \in V$ is called simplicial if its neighborhood in G is a clique. A Simplicial Elimination Ordering of G is a bijection $\sigma : V(G) \to \{1 \dots |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \dots, v_i\}$. For example, the vertices b, d of the graph shown in Figure 2(a) are simplicial. However, the vertices a and c are not, because b and d are not connected. In this graph, $\langle b, a, c, d \rangle$ is a simplicial elimination ordering. There is no simplicial elimination ordering ending in the nodes a or c. The graphs depicted in Figures 2(b) and 2(c) have no simplicial elimination orderings.

Theorem 1. (Dirac [10]) An undirected graph without self-loops is chordal if and only if it has a simplicial elimination ordering.

The algorithm greedy coloring, outlined in Figure 4, is a O(E) heuristic for graph coloring. Given a graph G and a sequence of vertices ν , greedy coloring assigns to each vertex of ν the next available color. Each color is a number c where $0 \leq c \leq \Delta(G) + 1$. If we give greedy coloring a simplicial elimination ordering of the vertices, then the greedy algorithm yields an optimal coloring [11]. In other words, greedy coloring is optimal for chordal graphs.

The algorithm known as Maximum Cardinality Search (MCS)[17] recognizes and determines a simplicial elimination ordering σ of a chordal graph in O(|E| + |V|) time. MCS associates with each vertex v of G a weight $\lambda(v)$, which initially

proc	edure greedy coloring
1	<i>input:</i> $G = (V, E)$, a sequence of vertices ν
2	output: a mapping $m, m(v) = c, 0 \le c \le \Delta(G) + 1, v \in V$
3	For all $v \in \nu$ do $m(v) \leftarrow \perp$
4	For $i \leftarrow 1$ to $ \nu $ do
5	let c be the lowest color not used in $N(\nu(i))$ in
6	$m(u(i)) \leftarrow c$

Fig. 4. The greedy coloring algorithm.

proc	cedure MCS
1	input: $G = (V, E)$
2	<i>output:</i> a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
3	For all $v \in V$ do $\lambda(v) \leftarrow 0$
4	For $i \leftarrow 1$ to $ V $ do
5	let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \ge \lambda(u)$ in
6	$\sigma(i) \leftarrow v$
7	For all $u \in V \cap N(v)$ do $\lambda(u) \leftarrow \lambda(u) + 1$
8	$V \leftarrow V - \{v\}$

Fig. 5. The maximum cardinality search algorithm.

is 0. At each stage MCS adds to σ the vertex v of greatest weight not yet visited. Subsequently MCS increases by one the weight of the neighbors of v, and starts a new phase. Figure 5 shows a version of MCS due to Berry et al. [4].

The procedure MCS can be implemented to run in O(|V| + |E|) time. To see that, notice that the first loop executes |V| iterations. In the second loop, for each vertex of G, all its neighbors are visited. After a vertex is evaluated, it is removed from the remaining graph. Therefore, the weight λ is increased exactly |E| times. By keeping vertices in an array of buckets indexed by λ , the vertex of highest weight can be found in O(1) time.

4 Our Algorithm

Our algorithm has several independent phases, as illustrated in Figure 6, namely coloring, spilling, and coalescing, plus an optional phase called *pre-spilling*. Coalescing must be the last stage in order to preserve the optimality of the coloring algorithm, because, after merging nodes, the resulting interference graph can be non-chordal. Our algorithm uses the *MCS* procedure (Figure 5) to produce an ordering of the nodes, for use by the pre-spilling and coloring phases. Our approach yields optimal colorings for chordal graphs, and, as we show in Section 5, it produces competitive results even for non-chordal graphs. We have implemented heuristics, rather than optimal algorithms, for spilling and coalescing. Our experimental results show that our heuristics perform better than those used in the iterated register coalescing algorithm.



Fig. 6. The main phases of our algorithm.



Fig. 7. (a) Euclid's algorithm. (b) Interference graph generated for gcd().

In order to illustrate the basic principles underlying our algorithm, we will as a running example show how our algorithm allocates registers for the program in Figure 7 (a). This program calculates the greatest common divisor between two integer numbers using Euclid's algorithm. In the intermediate representation adopted, instructions have the form op, t, p_1, p_2 . Such an instruction defines the variable t, and adds the temporaries p_1 and p_2 to the chain of used values. The interference graph yielded by the example program is shown in Figure 7 (b). Solid lines connecting two temporaries indicate that they are simultaneously alive at some point in the program, and must be allocated to different registers. Dashed lines connect move related registers.

Greedy Coloring In order to assign machine registers to variables, the greedy coloring procedure of Figure 4 is fed with an ordering of the vertices of the interference graph, as produced by the MCS procedure. From the graph shown in Figure 7 (b), MCS produces the ordering: \langle T7, R1, R2, T1, R5, R4, T8, R6, T9 \rangle , and greedy coloring then produces the mapping between temporaries and colors that is outlined in Figure 8 (a). If the interference graph is chordal, then the combination of MCS and Greedy Coloring produces a minimal coloring. The coloring phase uses an unbounded number of colors so that the interference graph can always be colored. The excess of colors will be removed in the postspilling stage.

Post-spilling Given an instance of a register allocation problem, it may be possible that the number of available registers is not sufficient to accommodate all the



Fig. 8. (a) Colored interference graph. (b) Interference graph after spilling the highest colors.

temporary variables. In this case, temporaries must be removed until the remaining variables can be assigned to registers. The process of removing temporaries is called spilling. A natural question concerning spilling when the interference graph is chordal is if there is a polynomial algorithm to determine the minimum number of spills. The problem of determining the maximum K-colorable subgraph of a chordal graph is NP-complete [20], but has polynomial solution when the number of colors (K) is fixed. We do not adopt the polynomial algorithm because its complexity seems prohibitive, namely $O(|V|^K)$ time.

Iterated register coalescing performs spilling as an iterative process. After an unsuccessful attempt to color the interference graph, some vertices are removed, and a new coloring phase is executed. We propose to spill nodes in a single iteration, by removing in each step all nodes of a chosen color from the colored interference graph. The idea is that given a K-colored graph, if all the vertices sharing a certain color are removed, the resulting subgraph can be colored with K-1 colors. We propose two different heuristics for choosing the next color to be removed: (i) remove the least-used color, and (ii) remove the highest color assigned by the greedy algorithm.

The spilling of the highest color has a simpler and more efficient implementation. The heuristic is based on the observation that the greedy coloring tends to use the lower colors first. For a chordal graph, the number of times the highest color is used is bounded by the number of maximal cliques in the interference graph. A maximal clique is a clique that cannot be augmented. In other words, given a graph G = (V, E), a clique Q is maximal if there is no vertex $v, v \in V-Q$, such that v is adjacent to all the vertices of Q. For our running example, Figure 8 (b) shows the colored interference graph after the highest colors have been removed, assuming that only two registers are available in the target machine. Coincidentally, the highest colors are also the least-used ones.

Coalescing The last phase of the algorithm is the coalescing of move related instructions. Coalescing helps a compiler to avoid generating redundant copy instructions. Our coalescing phase is executed in a greedy fashion. For each

proc	edure coalescing
1	<i>input:</i> list l of copy instructions, $G = (V, E)$, K
2	output: G' , the coalesced graph G
3	let $G' = G$ in
4	for all $x := y \in l$ do
5	let S_x be the set of colors in $N(x)$
6	let S_y be the set of colors in $N(y)$
7	if there exists $c, c < K, c \notin S_x \cup S_y$ then
8	let $xy, xy \notin V$ be a new node
9	add xy to G' with color c
10	make xy adjacent to every $v, v \in N(x) \cup N(y)$
11	replace occurrences of x or y in l by xy
12	remove x from G'
13	remove y from G'

Fig. 9. The greedy coalescing algorithm.

instruction a := b, the algorithm looks for a color c not used in $N(a) \cup N(b)$, where N(v) is the set of neighbors of v. If such a color exists, then the temporaries a and b are coalesced into a single register with the color c. This algorithm is described in Figure 9. Our current coalescing algorithm does not use properties of chordal graphs; however, as future work, we plan to study how coalescing can take benefit from chordality.

Pre-spilling To color a graph, we need a number of colors which is at least the size of the largest clique. We now present an approach to removing nodes that will bring the size of the largest clique down to the number of available colors and guarantee that the resulting graph will be colorable with the number of available colors (Theorem 2). Gavril [11] has presented an algorithm maximalCl, shown in Figure 10, which lists all the maximal cliques of a chordal graph in O(|E|) time. Our pre-spilling phase first runs maximalCl and then the procedure pre-spilling shown in Figure 11. Pre-spilling uses a map ω which maps each vertex to an approximation of the number of maximal cliques that contain that vertex. The objective of pre-spilling is to minimize the number of spills. When an interference graph is non-chordal, the maximalCl algorithm may return graphs that are not all cliques and so pre-spilling may produce unnecessary spills. Nevertheless, our experimental results in Section 5 show that the number of spills is competitive even for non-chordal graphs.

The main loop of pre-spilling performs two actions: (i) compute the vertex v that appears in most of the cliques of ξ and (ii) remove v from the cliques in which it appears. In order to build an efficient implementation of the pre-spilling algorithm, it is helpful to define a bidirectional mapping between vertices and the cliques in which they appear. Because the number of maximal cliques is bounded by |V| for a chordal graph, it is possible to use a bucket list to compute $\omega(v), v \in V$ in O(1) time. After a temporary is deleted, a number of cliques may become K-colorable, and must be removed from ξ . Again, due to the bidirectional

pro	cedure maximalCl
1	input: $G = (V, E)$
2	<i>output:</i> a list of cliques $\xi = \langle Q_1, Q_2, \dots, Q_n \rangle$
3	$\sigma \gets \operatorname{MCS}(\operatorname{G})$
4	For $i \leftarrow 1$ to n do
5	Let $v \leftarrow \sigma[i]$ in
6	$Q_i \leftarrow \{v\} \cup \{u \mid (u, v) \in E, u \in \{\sigma[1], \dots, \sigma[i-1]\}\}$

Fig. 10. Listing maximal cliques in chordal graphs.

pro	cedure pre-spilling
1	<i>input:</i> $G = (V, E)$, a list of subgraphs of $G: \xi = \langle Q_1, Q_2, \dots, Q_n \rangle$,
	a number of available colors K , a mapping ω
2	output: a K-colorable subgraph of G
3	$R_1 = Q_1; R_2 = Q_2; \dots R_n = Q_n$
4	while there is R_i with more than K nodes do
5	let $v \in R_i$ be a vertex such that $\forall u \in R_i, \omega(v) \ge \omega(u)$ in
6	remove v from all the graphs R_1, R_2, \ldots, R_n
7	$\mathbf{return} R_1 \cup R_2 \cup \ldots \cup R_n$

Fig. 11. Spilling intersections between maximal cliques.

mapping between cliques and temporaries, this operation can be performed in O(|N(v)|), where N(v) is the set of vertices adjacent to v. Overall, the spilling algorithm can be implemented in O(|E|).

Theorem 2. The graph pre-spilling(G,maximalCl(G),K, ω) is K-colorable.

Proof. Let $\langle Q_1, Q_2, \ldots, Q_n \rangle$ be the output of maximalCl(G). Let $R_1 \cup R_2 \cup \ldots \cup R_n$ be the output of $pre-spilling(G, maximalCl(G), K, \omega)$. Let $R_i^{\bullet} = R_1 \cup R_2 \cup \ldots \cup R_i$ for $i \in 1..n$.

We will show that for all $i \in 1..n$, R_i^{\bullet} is K-colorable. We proceed by induction on i.

In the base case of i = 1, we have $R_1^{\bullet} = R_1 \subseteq Q_1$ and Q_1 has exactly one node. We conclude that R_1^{\bullet} is K-colorable.

In the induction step we have from the induction hypothesis that R_i^{\bullet} is Kcolorable so let c be a K-coloring of R_i^{\bullet} . Let v be the node $\sigma[i+1]$ chosen in line
5 of maximalCl. Notice that v is the only vertex of Q_{i+1} that does not appear in Q_1, Q_2, \ldots, Q_i so c does not assign a color to v. Now there are two cases. First,
if v has been removed by pre-spilling, then $R_{i+1}^{\bullet} = R_i^{\bullet}$ so c is a K-coloring of R_{i+1}^{\bullet} . Second, if v has not been removed by pre-spilling, then we use that R_{i+1} has at most K nodes to conclude that the degree of v in R_{i+1} is at most K-1.
We have that c assigns a color to all neighbors for v in R_{i+1} so we have a color
left to assign to v and can extend c to a K-coloring of R_{i+1}^{\bullet} .

Figure 12 (a) shows the mapping between temporaries and maximal cliques that is obtained from the gcd(x, y) method, described in Figure 7 (a). Assuming that the target architecture has two registers, the cliques must be pruned



Fig. 12. (a) Mapping between nodes and maximal cliques. (b) Mapping after pruning node R1. (c) Interference graph after spilling R1 and R2.



Fig. 13. (a) Coloring produced by the greedy algorithm. (b) Coalescing R6 and T9. (c) Coalescing R4 and T7. (d) Coalescing R5 and T8.

until only cliques of size less than two remain. The registers R1 and R2 are the most common in the maximal cliques, and, therefore, should be deleted. The configuration after removing register R1 is outlined in Figure 12 (b). After the pruning step, all the cliques are removed from ξ . Figure 12 (c) shows the interference graph after the spilling phase.

Figure 13 outlines the three possible coalescings in this example. Coincidentally, two of the move related registers were assigned the same color in the greedy coloring phase. Because of this, their colors do not had to be changed during the coalescing stage. The only exception is the pair (R4, T7). In the coalescing phase, the original color of R4 is changed to the same color of T7. Afterwards, the registers are merged.

Complexity Analysis The coloring phase, as a direct application of maximum cardinality search and greedy coloring, can be implemented to run in O(|V|+|E|) time.

Our heuristics for spilling can all can be implemented to run in O(|E|) time. In order to implement spilling of the least-used color, it is possible to order the colors with bucket sort, because the maximum color is bounded by the highest degree of the interference graph plus one. The same technique can be used to order the weight function for the pre-spilling algorithm because the size of the list ξ , produced by the procedure *maximalCl*, is bounded by |V|.

Coalescing is the phase with the highest complexity, namely $O(t^3)$, where t is the number of temporaries in the source code. Our coalescing algorithm inspects, for each pair of move related instructions, all their neighbors. It is theoretically possible to have up to t^2 pairs of move related instructions in the target code. However, the number of these instructions is normally small, and our experimental results show that the coalescing step accounts for less than 10% of the total running time (see Figure 14 (a)).

5 Experimental Results

We have built an evaluation framework in Java, using the JoeQ compiler [19], in order to compare our algorithm against the iterated register coalescing. When pre-spilling is used, post-spilling is not necessary (Theorem 2). Our benchmark suite is the entire run-time library of the standard Java 1.5 distribution, i.e. the set of classes in rt.jar. In total, we analyzed 23,681 methods. We analyzed two different versions of the target code. One of them is constituted by the intermediate representation generated by JoeQ without any optimization. In the other version, the programs are first converted to single static assignment form (SSA), and them converted back to the JoeQ intermediate representation, by substituting the *phi* functions by copy instructions. In the former case, approximately 91% of the interference graphs produced are chordal. In the latter, the percentage of chordal graphs is 95.5%.

Table 1 shows results obtained by the iterative algorithm (IRC), and our non-iterative register allocator (NIA). The implementation of both algorithms attempts to spill the minimum number of registers. As it can be seen in the table, our technique gives better results than the traditional register allocator. It tends to use less registers per method, because it can find an optimum assignment whenever the interference graph is chordal. Also, it tends to spill less temporaries, because, by removing intersections among cliques, it decreases the chromatic number of several clusters of interfering variables at the same time. Notably, for the method coerceData, of the class java.awt.image.ComponentColorModel, with 6 registers available for allocation, the pre-spilling caused the eviction of 41 temporaries, whereas Iterated Register Coalescing spilled 86. Also, because our algorithm tends to spill fewer temporaries and to use fewer registers in the allocation, it is able to find more opportunities for coalescing. The Iterated register coalescing and our algorithm have similar running times. The complexity of a single iteration of the IRC is O(|E|), and the maximum number of iterations observed in the tests was 4; thus, its running time can be characterized as linear. Furthermore, both algorithms can execute a cubic number of coalescings, but, in the average, the quantity of copy instructions per program is small when compared to the total number of instructions.

Table 2 compares the two algorithms when the interference graphs are chordal and non-chordal. This data refers only to target programs after SSA elimination.

Algorithm	SSA	number of	register/	spill/	Total	maximum	coalescing/	running
		registers	method	method	spills	# spills	moves	time (s)
NIA	no	18	4.20	0.0044	102	15	0.38	2645.1
Post-spilling	yes	18	4.13	0.0034	81	14	0.72	2769.9
least-used	no	6	3.79	0.43	$10,\!218$	30	0.37	2645.0
color	yes	6	3.75	0.51	$12,\!108$	91	0.73	2781.7
NIA	no	18	4.20	0.0048	115	15	0.34	2641.5
Post-spilling	yes	18	4.13	0.010	246	63	0.72	2767.0
highest	no	6	3.80	0.50	$11,\!923$	33	0.35	2674.3
used color	yes	6	3.75	0.80	$19,\!018$	143	0.69	2764.2
NIA	no	18	4.20	0.0044	105	15	0.34	2640.5
Pre-spilling	yes	18	4.13	0.0039	94	17	0.72	2763.2
	no	6	3.78	0.45	10,749	34	0.35	2645.8
	yes	6	3.75	0.49	$11,\!838$	43	0.70	2765.1
	no	18	4.25	0.0050	115	16	0.31	2644.1
IRC	yes	18	4.17	0.0048	118	27	0.70	2823.2
1110	no	6	3.81	0.50	$11,\!869$	32	0.31	2641.5
	yes	6	3.77	0.57	$13,\!651$	86	0.66	2883.7

Table 1. Comparison between our algorithm (NIA) and Iterated Register Coalescing (IRC), including results for the three different spilling heuristics in Section 4.

In general, non-chordal interference graphs are produced by complex methods. For instance, methods whose interference graphs are non-chordal use, on average, 80.45 temporaries, whereas the average for chordal interference graphs is 13.94 temporaries.

The analysis of methods whose interference graphs are chordal gives some insight about the structure of Java programs. When an interference graph is chordal, the mapping between temporaries and registers is optimal, i.e. it uses the smallest possible number of registers. Figure 14 (b) shows the relation between number of methods of the Java Library and the minimum number of registers necessary to handle them. Only methods that could be colored with less than 18 colors (99.6%) are shown. Allocation results for methods whose interference graph are non-chordal are also presented, even though these may not be optimal.

Figure 14 (a) compares the amount of time spent on each phase of the algorithm when different spilling heuristics are adopted. The time used in the allocation process is a small percentage of the total running time presented in Table 1 because the latter includes the loading of class files, the parsing of bytecodes, the liveness analysis and the construction of the interference graph. When pre-spilling is used, it accounts for more than half the allocation time.

We have also tested our register allocation algorithm on the 27,921 interference graphs published by George and Appel. Those graphs were generated by the standard ML compiler of New Jersey compiling itself [3]. Our tests have shown that 95.7% of the interference graphs are chordal when the interferences between pre-colored registers and temporaries are not taken into consideration. The compilation results are outlined in Table 3. The graphs contain 21 pairwise

Algorithm	chordal	number of	register/	spill/	Total	maximum	coalescing/
	graph	registers	method	method	spills	# spills	moves
NIA	no	18	8.17	0.054	61	17	0.75
Pre-spilling	no	6	5.77	4.55	5173	43	0.79
	yes	18	3.92	0.0015	33	6	0.69
	yes	6	3.65	0.29	6665	31	0.68
	no	18	8.39	0.062	71	27	0.74
IRC	no	6	5.79	4.89	5562	86	0.66
	yes	18	3.97	0.0015	34	6	0.67
	ves	6	3.68	0.39	8089	45	0.67

 Table 2. Comparative performance of our spilling heuristics for chordal and nonchordal interference graphs.



Fig. 14. (a) Time spent on coloring, spilling and coalescing in the different heuristics. (b) Number of registers assigned to methods of the Java 1.5 Standard Library.

interfering pre-colored registers, which represent the machine registers available for the allocation. Because of these cliques, all the graphs, after spilling, demanded exactly 21 colors. When the graphs are chordal, pre-spilling gives the best results; however, this heuristic suffers a penalty when dealing with the non-chordal graphs, because they present a 21-clique, and must be colored with 21 registers. In such circumstances, the procedure *maximalCl* from Figure 10 have listed some false maximal cliques, and unnecessary spills have been caused. Overall, the spilling of the least-used colors gives the best results. The execution times for analyzing the ML-compiler-based benchmarks are faster than those for analyzing the Java Library because the latter set of timings includes the times to construct the interference graphs.

	chordal	Total of	maximum	coalescing/	allocation
Algorithm	graph	spills	number of	moves	time
			spills		(s)
Post-spilling least	yes	1,217	84	0.97	<u> </u>
used color	no	63	14	0.94	223.8
Post-spilling highest	yes	1,778	208	0.97	222.0
used color	no	80	20	0.94	222.9
Pre-spilling	yes	1,127	86	0.97	100.2
	no	1.491	23	0.93	402.0

 Table 3. Results obtained from the allocation of registers to 27,921 interference graphs generated from ML code.

6 Conclusion

This paper has presented a non-iterative algorithm for register allocation based on the coloring of chordal graphs. Chordal graphs present an elegant structure and can be optimally colored in O(|V| + |E|) time. For the register allocation problem, we can find an optimal allocation in time linear in the number of interferences between live ranges, whenever the interference graph is chordal. Additionally, our algorithm is competitive even when performing register allocation on non-chordal inputs.

In order to validate the algorithm, we compared it to iterated register coalescing. Our algorithm allocates fewer registers per method and spills fewer temporaries. In addition, our algorithm can coalesce about the same proportion of copy instructions as iterated register coalescing.

In addition to being efficient, our algorithm is modular and flexible. Because it is non-iterative, it presents a simpler design than traditional algorithms based on graph coloring. The spill of temporaries can happen before or after the coloring phase. By performing spilling before coloring, it is possible to assign different weights to temporaries in order to generate better code. Our implementation and a set of interference graphs generated from the Java methods tested can be found at http://compilers.cs.ucla.edu/fernando/projects/.

Acknowledgments. We thank Ben Titzer and the reviewers for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. We were supported by the National Science Foundation award number 0401691.

References

- 1. Christian Andersson. Register allocation by optimal graph coloring. In 12th Conference on Compiler Construction, pages 34–45. Springer, 2003.
- Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In International Conference on Programming Languages Design and Implementation, pages 243–253. ACM Press, 2001.
- Andrew W Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09-http://www.cs.princeton.edu/~appel/graphdata/, 2005.
- Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287– 298, 2004.
- Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems* (TOPLAS), 16(3):428–455, 1994.
- Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomialtime graph coloring register allocation. In 14th International Workshop on Logic and Synthesis. ACM Press, 2005.
- Zoran Budimlic, Keith D Cooper, Timothy J Harvey, Ken Kennedy, Timothy S Oberg, and Steven W Reeves. Fast copy coalescing and live-range identification. In International Conference on Programming Languages Design and Implementation, pages 25–32. ACM Press, 2002.
- G J Chaitin. Register allocation and spilling via graph coloring. Symposium on Compiler Construction, 17(6):98–105, 1982.
- Maria Chudnovsky, Gerard Cornuejols, Xinming Liu, Paul Seymour, and Kristina Vuskovic. Recognizing berge graphs. *Combinatorica*, 25:143–186, 2005.
- G A Dirac. On rigid circuit graphs. In Abhandlungen aus dem Mathematischen Seminar der Universiat Hamburg, volume 25, pages 71–75. University of Hamburg, 1961.
- 11. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180–187, 1972.
- 12. Lal George and Andrew W Appel. Iterated register coalescing. Transactions on Programming Languages and Systems (TOPLAS), 18(3):300–324, 1996.
- M Grotschel, L Lovasz, and A Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
- 14. Sebastian Hack. Interference graphs of programs in SSA-form. Technical report, Universitat Karlsruhe, 2005.
- 15. Fernando M Q Pereira and Jens Palsberg. Register allocation after SSA elimination is NP-complete. Manuscript, 2005.
- Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. ACM Transactions on Programming Languages and Systems, 21(5):895–913, 1999.
- 17. Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
- 18. Douglas B West. Introduction to Graph Theory. Prentice Hall, 2nd edition, 2001.
- 19. John Whaley. Joeq:a virtual machine and compiler infrastructure. In Workshop on Interpreters, virtual machines and emulators, pages 58–66. ACM Press, 2003.
- 20. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.

Register Allocation after Classical SSA Elimination is NP-complete

Fernando Magno Quintão Pereira Jens Palsberg

UCLA University of California, Los Angeles

Abstract. Chaitin proved that register allocation is equivalent to graph coloring and hence NP-complete. Recently, Bouchez, Brisk, and Hack have proved independently that the interference graph of a program in static single assignment (SSA) form is chordal and therefore colorable in linear time. Can we use the result of Bouchez et al. to do register allocation in polynomial time by first transforming the program to SSA form, then performing register allocation, and finally doing the classical SSA elimination that replaces ϕ -functions with copy instructions? In this paper we show that the answer is no, unless P = NP: register allocation after classical SSA elimination is NP-complete. Chaitin's proof technique does not work for programs after classical SSA elimination; instead we use a reduction from the graph coloring problem for circular arc graphs.

1 Introduction

In Section 1.1 we define three central notions that we will use in the paper: the core register allocation problem, static single assignment (SSA) form, and post-SSA programs. In Section 1.2 we explain why recent results on programs in SSA form might lead one to speculate that we can solve the core register allocation problem in polynomial time. Finally, in Section 1.3 we outline our result that register allocation is NP-complete for post-SSA programs produced by the classical approach that replaces ϕ -functions with copy instructions.

1.1 Background

Register Allocation. In a compiler, register allocation is the problem of mapping temporaries to machine registers. In this paper we will focus on:

Core register allocation problem:

Instance: a program P and a number K of available registers. Problem: can each of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers? Notice that K is part of the input to the problem. Fixing K would correspond to the register allocation problem solved by a compiler for a fixed architecture. Our core register allocation problem is related to the kind of register allocation problem solved by gcc; the problem does not make assumptions about the number of registers in the target architecture.

Chaitin et al. [8] showed that the core register allocation problem is NPcomplete by a reduction from the graph coloring problem. The essence of Chaitin et al.'s proof is that every graph is the interference graph of some program.

SSA form. Static single assignment (SSA) form [21] is an intermediate representation used in many compilers, including gcc version 4. If a program is in SSA form, then every variable is assigned exactly once, and each use refers to exactly one definition. A compiler such as gcc version 4 translates a given program to SSA form and later to an executable form.

SSA form uses ϕ -functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of ϕ -functions using the matrix notation introduced by Hack et al. [17]. Figure 1 (a) outlines the general representation of a ϕ -matrix. And Figure 1 (c) gives the intuitive semantics of the matrix shown in Figure 1 (b).

An equation such as $V = \phi M$, where V is a n-dimensional vector, and M is a $n \times m$ matrix, contains $n \phi$ -functions such as $a_i \leftarrow \phi(a_{i1}, a_{i2}, \ldots, a_{im})$. Each possible execution path has a corresponding column in the ϕ -matrix, and adds one parameter to each ϕ -function. The ϕ symbol works as a multiplexer. It will assign to each element a_i of V an element a_{ij} of M, where j is determined by the actual path taken during the program's execution.

All the ϕ -functions are evaluated simultaneously at the beginning of the basic block where they are located. As noted by Hack et al. [17], the live ranges of temporaries in the same column of a ϕ -matrix overlap, while the live ranges of temporaries in the same row do not overlap. Therefore, we can allocate the same register to two temporaries in the same row. For example, Figure 2 shows a program, its SSA version, and the program after classical SSA elimination. If the control flow reaches block 2 from block 1, $\phi(v_{11}, v_{12})$ will return v_{11} ; v_{12} being returned otherwise. Variables i_2 and v_{11} do not interfere. In contrast, the variables v_{11} and i_1 interfere because both are alive at the end of block 1.

Post-SSA programs. SSA form simplifies many analyses that are performed on the control flow graph of programs. However, traditional instruction sets do not implement ϕ -functions [10]. Thus, in order to generate executable code, compilers have a phase called *SSA-elimination* in which ϕ -functions are destroyed. Henceforth, we will refer to programs after SSA elimination as *post-SSA* programs.

The classical approach to SSA-elimination replaces the ϕ -functions with copy instructions [1, 5, 7, 10, 18, 20]. For example, consider $v_1 = \phi(v_{11}, ..., v_{1m})$ in block b. The algorithm explained by Appel [1] adds at the end of each block *i* that precedes *b*, one copy instruction such as $v_1 = v_{1i}$.



Fig. 1. (a) ϕ -functions represented as a matrix equation. (b) Matrix equation representing two ϕ -functions and three possible execution paths. (c) Control flow graph illustrating the semantics of ϕ -functions.

In this paper we concentrate on SSA programs whose control flow graphs have the structure outlined in Figure 3 (a). The equivalent post-SSA programs, generated by the classical approach to SSA-elimination, are given by the grammar in Figure 3 (b). We will say that a program generated by the grammar in Figure 3 (b) is a *simple* post-SSA program. For example, the program in Figure 2(c) is a simple post-SSA program. A simple post-SSA program contains a single loop. Just before the loop, and at the end of it (see Figure 3 (b)), the program contains copy instructions that correspond to the elimination of a ϕ -matrix such as:

$\binom{v_1}{}$		v_{11}	v_{21}
$\begin{pmatrix} \cdots \\ v_K \\ i \end{pmatrix}$	$=\phi$	v_{1K} i_1	$\begin{bmatrix} v_{2K} \\ i_2 \end{bmatrix}$

1.2 Programs in SSA-form have chordal interference graphs

The core register allocation problem is NP-complete and a compiler can transform a given program into SSA form in cubic time [9]. Thus we might expect that the core register allocation problem for programs in SSA form is NP-complete. However, that intuition would be wrong, unless P = NP, as demonstrated by the following result.

In 2005, Brisk et al. [6] proved that *strict* programs in SSA form have *perfect* interference graphs; independently, Bouchez [4] and Hack [16] proved the stronger result that strict programs in SSA form have *chordal* interference graphs. In a



Fig. 2. (a) A program. (b) The same program in SSA form. (c) The program after classical SSA elimination.

strict program, every path from the initial block to the use of a variable v passes through a definition of v [7]. The proofs presented in [4, 16] rely on two wellknown facts: (i) the chordal graphs are the intersection graphs of subtrees in trees [14], and (ii) live ranges in an SSA program are subtrees of the dominance tree [16].

We can color a chordal graph in linear time [15] so we can solve the core register allocation problem for programs in SSA form in linear time. Thus, the transformation to SSA form seemingly maps an NP-complete problem to a polynomial-time problem in polynomial time! The key to understanding how such a transformation is possible lies in the following observation. Given a program P, its SSA-form version P', and a number of registers K, the core register allocation problem (P, K) is not equivalent to (P', K). While we can map a (P, K)-solution to a (P', K)-solution, we can not necessarily map a (P', K)solution to a (P, K)-solution. The SSA transformation splits the live ranges of temporaries in P in such a way that P' may need fewer registers than P.

Given that the core register allocation problem for programs in SSA form can be solved in polynomial time and given that a compiler can do classical SSA elimination in linear time, we might expect that the core register allocation problem after classical SSA elimination is in polynomial time. In this paper we show that also that intuition would be wrong!

1.3 Our Result

We prove that the core register allocation problem for simple post-SSA programs is NP-complete. Our result has until now been a commonly-believed folk theorem without a published proof. The recent results on register allocation for programs



Fig. 3. (a) Control flow representation of simple SSA programs. (b) The grammar for simple post-SSA programs.

in SSA form have increased the interest in a proof. Our result implies that the core register allocation problem for post-SSA programs is NP-complete for any language with loops or with jumps that can implement loops.

The proof technique used by Chaitin et al. [8] does not work for post-SSA programs. Chaitin et al.'s proof constructs programs from graphs, and if we transform those programs to SSA form and then post-SSA form, we can color the interference graph of each of the post-SSA programs with just three colors. For example, in order to represent C_4 , their technique would generate the graph in the upper part of Figure 4 (b). The minimal coloring of such graph can be trivially mapped to a minimal coloring of C_4 , by simply deleting node x. Figure 4 (a) shows the control flow graph of the program generated by Chaitin et al.'s proof technique, and Figure 4 (c) shows the program in post-SSA form. The interference graph of the transformed program is shown in the lower part of Figure 4 (b); that graph is chordal, as expected.

We prove our result using a reduction from the graph coloring problem for circular arc graphs, henceforth called simply *circular graphs*. A circular graph can be depicted as a set of intervals around a circle (e.g. Figure 5 (a)). The idea of our proof is that the live ranges of the variables in the loop in a simple post-SSA program form a circular graph. From a circular graph and an integer K we build a simple post-SSA program such that the graph is K-colorable if and only if we can solve the core register allocation problem for the program and K + 1 registers. Our reduction proceeds in two steps. In Section 2 we define the notion of SSA-circular graphs and we show that the coloring problem for SSA-



Fig. 4. (a) Chaitin et al.'s program to represent C_4 . (b) The interference graph of the original program (top) and of the program in SSA form (bottom). (c) The program of Chaitin et al. in SSA form.

circular graphs is NP-complete. In Section 3 we present a reduction from coloring of SSA-circular graphs to register allocation for simple post-SSA programs. An SSA-circular graph is a special case of a circular graph in which some of the intervals come in pairs that correspond to the copy instructions at the end of the loop in a simple post-SSA program. From a circular graph we build an SSAcircular graph by splitting some arcs. By adding new intervals at the end of the loop, we artificially increase the color pressure at that point, and ensure that two intervals that share an extreme point receive the same color. In Section 4 we give a brief survey of related work on complexity results for a variety of register allocation problems, and in Section 5 we conclude.

Recently, Hack et al. [17] presented an SSA-elimination algorithm that does not use move instructions to replace ϕ -functions. Instead, Hack et al.'s algorithm uses xor instructions to permute the values of the parameters of the ϕ -functions in a way that preserves both the semantics of the original program and the chordal structure of the interference graph, without demanding extra registers. As a result, register allocation after the Hack et al.'s approach to SSA elimination is in polynomial time. In contrast, register allocation after the classical approach to SSA elimination is NP-complete.

2 From circular graphs to SSA-circular graphs

Let N denote the set of positive, natural numbers $\{1, 2, 3, \ldots\}$. A *circular graph* is an undirected graph given by a finite set of vertices $V \subseteq N \times N$, such that $\forall d \in N : (d, d) \notin V$ and $\forall (d, u), (d', u') \in V : d = d' \Leftrightarrow u = u'$. We sometimes refer to a vertex (d, u) as an *interval*, and we call d, u extreme points. The set of vertices of a circular graph defines the edges implicitly, as follows. Define

 $\begin{array}{l} b \ : \ N \times N \to \ \text{ finite unions of intervals of the real numbers} \\ b(d,u) = \begin{cases} \]d,u[& \text{if } d < u \\ \]0,u[\cup]d,\infty[& \text{if } d > u. \end{cases} \end{array}$



Fig. 5. (a) C_5 represented as a set of intervals. (b) The set of intervals that represent $W = \mathcal{F}(C_5, 3)$. (c) W represented as a graph.

Two vertices (d, u), (d', u') are connected by an edge if and only if $b(d, u) \cap b(d', u') \neq \emptyset$. We use \mathcal{V} to denote the set of such representations of circular graphs. We use $\max(V)$ to denote the largest number used in V, and we use $\min(V)$ to denote the smallest number used in V. We distinguish three subsets of vertices of a circular graph, V_l , V_i and V_z :

$$V_{i} = \{ (d, u) \in V \mid d < u \}$$
$$V_{l} = \{ (d, u) \in V \mid d > u \}$$
$$V_{z} = \{ (d, y) \in V_{i} \mid \exists (y, u) \in V_{l} \}$$

Notice that $V = V_i \cup V_l$, $V_i \cap V_l = \emptyset$, and $V_z \subseteq V_i$.

Figure 5 (a) shows a representation of $C_5 = (\{a, b, c, d, e\}, \{ab, bc, cd, de, ea\})$ as a collection of intervals, where $a = (14, 7), b = (6, 9), c = (8, 11), d = (10, 13), e = (12, 5), V_i = \{b, c, d\}, V_l = \{a, e\}, and V_z = \emptyset$. Intuitively, when the intervals of the circular graph are arranged around a circle, overlapping intervals determine edges between the corresponding vertices.

An SSA-circular graph W is a circular graph with two additional properties:

$$\forall (y,u) \in W_l : \exists d \in N : (d,y) \in W_z \tag{1}$$

$$\forall (d, u) \in W_i \setminus W_z : \forall (d', u') \in W_l : u < d'$$

$$\tag{2}$$

We use \mathcal{W} to denote the set of SSA-circular graphs.

Let W be an SSA-circular graph. Property (1) says that for each interval in W_l there is an interval in W_z so that these intervals share an extreme point y. In Section 3 it will be shown that the y points represent copy instructions used to propagate the parameters of ϕ -functions. Henceforth, the y points will be called *copy points*. Figure 5 (b) shows $W \in W$ as an example of SSA-circular graph. $W_l = \{(18, 1), (21, 5), (22, 7)\}, W_i = \{(6, 9), (8, 11), (10, 13)\} \cup W_z$, and $W_z = \{(15, 18), (12, 21), (14, 22)\}$. Notice that for every interval $(y, u) \in W_l$,



Fig. 6. The critical points created by $\mathcal{F}(V, K)$.

there is an interval $(d, y) \in W_z$. Figure 5 (c) exhibits W using the traditional representation of graphs.

Let $n = |V_l|$. We will now define a mapping \mathcal{F} on pairs (V, K):

$$\mathcal{F} : \mathcal{V} \times N \to \mathcal{W}$$

$$\mathcal{F}(V, K) = V_i \cup \mathcal{G}(V_l, K, \max(V))$$

$$\mathcal{G} : \mathcal{V} \times N \times N \to \mathcal{V}$$

$$\mathcal{G}(\{ (d_i, u_i) \mid i \in 1...n \}, K, m) = \{ (m+i, m+K+i) \mid i \in 1...K-n \} \quad (3)$$

$$\cup \{ (m+K+i,i) \mid i \in 1...K-n \} \quad (4)$$

$$\cup \{ (d_i, m+2K+i) \mid i \in 1...n \} \quad (5)$$

$$\mapsto \{ (m+2K+i, m) \mid i \in 1...n \} \quad (6)$$

 $\cup \{ (m+2K+i, u_i) \mid i \in 1..n \}$ (6)

Given V, the function \mathcal{F} splits each interval of V_l into two nonadjacent intervals that share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the main vertices. Given V, the function \mathcal{F} also creates 2(K - n) new intervals, namely K - n pairs of intervals such that the two intervals of each pair are nonadjacent and share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the *auxiliary* vertices. Figures 5 (b) and 5 (c) represent $\mathcal{F}(C_5, 3)$, and Figure 6 outlines the critical points between $m = \max(V)$ and K - n.

Lemma 1. If V is a circular graph and $\min(V) > K$, then $\mathcal{F}(V, K)$ is an SSA-circular graph.

Proof. Let $W = \mathcal{F}(V, K)$. Notice first that W is a circular graph because the condition $\min(V) > K$ ensures that rules (4) and (6) define vertices that don't share any extreme points. To see that W is an SSA-circular graph let us consider in turn the two conditions (1) and (2). Regarding condition (1), W_l consists of the sets defined by (4), (6), while W_z consists of the sets defined by (3), (5), and for each $(y, u) \in W_l$, we can find $(d, y) \in W_z$. Regarding condition (2), we have that if $(d, u) \in W_i \setminus W_z$ and $(d', u') \in W_l$, then $u \leq \max(V) < \max(V) + K + 1 \leq d'$.

Lemma 2. If $W = \mathcal{F}(V, K)$ is K-colorable, then two intervals in $W_z \cup W_l$ that share an extreme point must be assigned the same color by any K-coloring of W.

Proof. Let c be a K-coloring of W. Let $v_1 = (d, y)$ and $v_2 = (y, u)$ be a pair of intervals in $W_z \cup W_l$. From the definition of $\mathcal{F}(V, K)$ we have that those two intervals are not connected by an edge. The common copy point y is crossed by exactly K - 1 intervals (see Figure 6). Each of those K - 1 intervals must be assigned a different color by c so there remains just one color that c can assign to v_1 and v_2 . Therefore, c has assigned the same color to v_1 and v_2 .

Lemma 3. Suppose V is a circular graph and $\min(V) > K$. We have V is K-colorable if and only if $\mathcal{F}(V, K)$ is K-colorable.

Proof. Let $V_l = \{ (d_i, u_i) \mid i \in 1...n \}$ and let $m = \max(V)$.

First, suppose c is a K-coloring of V. The vertices of V_l form a clique so c must use $|V_l|$ colors to color V_l . Let $n = |V_l|$. Let $\{x_1, \ldots, x_{K-n}\}$ be the set of colors *not* used by c to color V_l . We now define a K-coloring c' of $\mathcal{F}(V, K)$:

$$c'(v) = \begin{cases} c(v) & \text{if } v \in V_i \\ x_i & \text{if } v = (m+i, m+K+i), i \in 1..K-n \\ x_i & \text{if } v = (m+K+i, i), i \in 1..K-n \\ c(d_i, u_i) & \text{if } v = (d_i, m+2K+i), i \in 1..n \\ c(d_i, u_i) & \text{if } v = (m+2K+i, u_i), i \in 1..n \end{cases}$$

To see that c' is indeed a K-coloring of $\mathcal{F}(V, K)$, first notice that the colors of the main vertices don't overlap with the colors of the auxiliary vertices. Second, notice that since $\min(V) > K$, no auxiliary edge is connected to a vertex in V_i . Third, notice that since c is a K-coloring of V, the main vertices have colors that don't conflict with their neighbors in V_i .

Conversely, suppose c' is a K-coloring of $\mathcal{F}(V, K)$. We now define a K-coloring c of V:

$$c(v) = \begin{cases} c'(v) & \text{if } v \in V_i \\ c'(d_i, m + 2K + i) & \text{if } v = (d_i, u_i), i \in 1..n \end{cases}$$

To see that c is indeed a K-coloring of V, notice that from Lemma 2 we have that c' assigns the same color to the intervals $(d_i, m + 2K + i), (m + 2K + i, u_i)$ for each $i \in 1..n$. So, since c' is a K-coloring of $\mathcal{F}(V, K)$, the vertices in V_l have colors that don't conflict with their neighbors in V_i .

Lemma 4. Graph coloring for SSA-circular graphs is NP-complete.

Proof. First notice that graph coloring for SSA-circular graphs is in NP because we can verify any color assignment in polynomial time. To prove NP-hardness, we will do a reduction from graph coloring for circular graphs, which is known to be NP-complete [12, 19]. Given a graph coloring problem instance (V, K) where V is a circular graph, we first transform V into an isomorphic graph V' by adding K to all the integers used in V. Notice that $\min(V') > K$. Next we produce the graph coloring problem instance $(\mathcal{F}(V', K), K)$, and, by Lemma 3, V' is K-colorable if and only if $\mathcal{F}(V', K)$ is K-colorable.

3 From SSA-circular graphs to post-SSA programs

We now present a reduction from coloring of SSA-circular graphs to the core register allocation problem for simple post-SSA programs. In this section we use a representation of circular graphs which is different from the one used in Section 2. We represent a circular graph by a finite list of elements of the set $\mathcal{I} = \{ \operatorname{def}(j), \operatorname{use}(j), \operatorname{copy}(j,j'), | j,j' \in N \}$. Each j represents a temporary name in a program. We use ℓ to range over finite lists over \mathcal{I} . If ℓ is a finite list over \mathcal{I} and the d-th element of ℓ is either $\operatorname{def}(j)$ or $\operatorname{copy}(j,j')$, then we say that j is defined at index d of ℓ . Similarly, if the u'th element of ℓ is either use(j') or $\operatorname{copy}(j,j')$, then we say that j' is used at index u of ℓ . We define \mathcal{X} as follows:

 $\mathcal{X} = \{ \ell \mid \text{for every } j \text{ mentioned in } \ell, j \text{ is defined exactly once and used} \\ \text{exactly once } \land \text{ for every } \operatorname{copy}(j,j') \text{ in } \ell, \text{ we have } j \neq j' \}$

We will use X to range over \mathcal{X} . The sets \mathcal{X} and \mathcal{V} are isomorphic; the function α is an isomorphism which maps \mathcal{X} to \mathcal{V} :

$$\alpha : \mathcal{X} \to \mathcal{V}$$

$$\alpha(X) = \{ (d, u) \mid \exists j \in N : j \text{ is defined at index } d \text{ of } X \text{ and } j \text{ is used}$$

at index u of X }

We define $\mathcal{Y} = \alpha^{-1}(\mathcal{W})$, and we use Y to range over \mathcal{Y} . The graph W shown in Figure 5 (b) is shown again in Figure 7 (a) in the new representation:

$$Y = \langle \operatorname{use}(t), \operatorname{use}(e), \operatorname{def}(b), \operatorname{use}(a), \operatorname{def}(c), \operatorname{use}(b), \operatorname{def}(d), \operatorname{use}(c), \operatorname{def}(e), \\ \operatorname{use}(d), \operatorname{def}(a), \operatorname{def}(t_2), \operatorname{copy}(t, t_2), \operatorname{copy}(e, e_2), \operatorname{copy}(a, a_2) \rangle.$$

Figure 8 presents a mapping \mathcal{H} from \mathcal{Y} -representations of SSA-circular graphs to simple post-SSA programs. Given an interval (d, u) represented by def(j) and use(j), we map the initial point d to a variable definition $\mathbf{v_j} = \mathbf{i_2} + \mathcal{C}$, where $\mathbf{i_2}$ is the variable that controls the loop. We assume that all the constants are chosen to be different. The final point u is mapped to a variable use, which we implement by means of the conditional statement **if** $(\mathbf{v_j} > \mathcal{C})$ **break**. We opted for mapping uses to conditional commands because they do not change the live ranges inside the loop, and their compilation do not add extra registers to the final code. An element of the form copy(j, j'), which is mapped to the assignment j = j', is used to simulate the copy of one of the parameters of a ϕ -function, after classical SSA elimination. Figure 7 (b) shows the program $P = \mathcal{H}(Y, 3)$, where $Y = \mathcal{F}(C_5, 3)$.

Lemma 5. We can color an SSA-circular graph Y with K colors if and only if we can solve the core register allocation problem for $\mathcal{H}(Y, K)$ and K+1 registers.

Proof. First, assume Y has a K-coloring. The intervals in Y match the live ranges in the loop of $\mathcal{H}(Y, K)$, except for the control variables *i*, and *i*₂, which have nonoverlapping live ranges. Therefore, the interference graph for the loop



Fig. 7. (a) $Y = \mathcal{F}(C_5, 3)$ represented as a sequence of instructions and as a graph. (b) $P = \mathcal{H}(Y, 3)$.

can be colored with K+1 colors. The live ranges of the variables declared outside the loop form an interval graph of width K+1. We can extend the K+1-coloring of that interval graph to a K+1-coloring of the entire graph in linear time.

Now, assume that there is a solution of the core register allocation problem for $\mathcal{H}(Y, K)$ that uses K+1 registers. The intervals in Y represent the live ranges of the variables in the loop. The control variables i and i_2 demand one register, which cannot be used in the allocation of the other live ranges inside the loop. Therefore, the coloring of $\mathcal{H}(Y, K)$ can be mapped trivially to the nodes of Y.

Theorem 1. The core register allocation problem for simple post-SSA programs is NP-complete.

Proof. Combine Lemmas 4 and 5.

 \square

As an illustrative example, to color C_5 with three colors is equivalent to determining a 3-coloring to the graph Y in Figure 7 (a). Such colorings can be found if and only if the core register allocation problem for $P = \mathcal{H}(Y,3)$ can be solved with 4 registers. In this example, a solution exists. One assignment of registers would be $\{a, a_1, a_2, c, p_1\} \rightarrow R1$, $\{b, d, t_1, t_2, t, p_3\} \rightarrow R2$, $\{e, e_1, e_2, p_2\} \rightarrow R3$, and $\{i, i_1, i_2, p_4\} \rightarrow R4$. This corresponds to coloring the arcs a and c with the first color, arcs b and d with the second, and e with the third.

```
\operatorname{gen}(\operatorname{def}(j)) = \operatorname{int} \mathbf{v}_j = \mathbf{i}_2 + \mathcal{C};
      gen(use(j)) = if(v_j > C) break;
gen(copy(j,j')) = v_j = v_{j'};
                     \mathcal{H} : \mathcal{Y} \times N \rightarrow \text{simple post-SSA program}
           \mathcal{H}(Y,K) = \mathbf{int} \ \mathrm{m}(\mathbf{int} \ \mathrm{p}_1, \ldots, \mathbf{int} \ \mathrm{p}_{K+1}) \ \{
                                    int v_{11} = p_1; \ldots; int v_{1K} = p_K;
                                     int i_1 = p_{K+1};
                                     int v_1 = v_{11}; \ldots; int v_K = v_{1K};
                                     int i = i1;
                                     while (i < C) {
                                           int i_2 = i+1;
                                           map(Y, gen)
                                           i = i_2;
                                     }
                                     return v_1;
                              }
```

Fig. 8. The mapping of circular graphs to simple post-SSA programs

4 Related Work

The first NP-completeness proof of a register allocation related problem was published by Sethi [22]. Sethi showed that, given a program represented as a set of instructions in a directed acyclic graph and an integer K, it is an NP-complete problem to determine if there is a computation of the DAG that uses at most K registers. Essentially, Sethi proved that the placement of loads and stores during the generation of code for a straight line program is an NP-complete problem if the order in which instructions appear in the target code is not fixed.

Much of the literature concerning complexity results for register allocation deals with two basic questions. The first is the core register allocation problem, which we defined in Section 1. The second is the core spilling problem which generalizes the core register allocation problem:

Core spilling problem:

Instance: a program P, number K of available registers, and a number M of temporaries.

Problem: can at least M of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers?

Farach and Liberatore [11] proved that the core spilling problem is NP-complete even for straight line code and even if rescheduling of instructions is not allowed. Their proof uses a reduction from set covering.

For a straight line program, the core register allocation problem is linear in the size of the interference graph. However, if the straight line program contains pre-colored registers that can appear more than once, then the core register
allocation problem is NP-complete. In this case, register allocation is equivalent to pre-coloring extensions of interval graphs, which is NP-complete [2].

In the core register allocation problem, the number of registers K is not fixed. Indeed, the problem used in our reduction, namely the coloring of circular graphs, has a polynomial-time solution if the number of colors is fixed. Given n circular arcs determining a graph G, and a fixed number K of colors, Garey et al. [12] have given an $O(n \cdot K! \cdot K \cdot \log K)$ time algorithm for coloring G if such a coloring exists. Regarding general graphs, the coloring problem is NP-complete for every fixed value of K > 2 [13].

Bodlaender et al. [3] presented a linear-time algorithm for the core register allocation problem with a fixed number of registers for structured programs. Their result holds even if rescheduling of instructions is allowed. If registers of different types are allowed, such as integer registers and floating point registers, for example, then the problem is no longer linear, although it is still polynomial.

Researchers have proposed different algorithms for inserting copy instructions, particularly for reducing the number of copy instructions [7, 5, 10, 18]. Rastello et al. [10] have proved that the optimum replacement of ϕ -functions by copy instructions is NP-complete. Their proof uses a reduction from the maximum independent set problem.

5 Conclusion

We have proved that the core register allocation problem is NP-complete for post-SSA programs generated by the classical approach to SSA-elimination that replaces ϕ -functions with copy instructions. In contrast, Hack et al.'s recent approach to SSA-elimination [17] generates programs for which the core register allocation problem is in polynomial time. We conclude that the choice of SSAelimination algorithm matters.

We claim that compiler optimizations such as copy propagation and constant propagation cannot improve the complexity of the core register allocation problem for simple post-SSA programs. Inspecting the code in Figure 8 we perceive that the number of loop iterations cannot be easily predicted by a local analysis because all the control variables are given as function parameters. In the statement **int** $v_j = i+C$; the variable i limits the effect of constant propagation and the use of different constants C limits the effect of copy propagation. Because all the K + 1 variables alive at the end of the loop have different values, live ranges cannot be merged at that point. In contrast, rescheduling of instructions might improve the complexity of the core register allocation problem for simple post-SSA programs. However, rescheduling combined with register allocation is an NP-complete problem even for straight line programs [22].

Theorem 1 continues to hold independent on the ordering in which copy instructions are inserted, because the function \mathcal{G} , defined in Section 2, can be modified to accommodate any ordering of the copy points. In more detail, let $W = \mathcal{F}(V, K)$ be a SSA-circular graph, let $n \in [0 \cdots \max(W)]$, and let ovl(n) be



Fig. 9. (a) SSA graph W that represents $\mathcal{F}(C_3, K)$. (b) A program P that represents W with a single **if**-statement. (c) Schematic view of the live ranges of P.

the number of intervals that overlap at point n.

$$\forall n \in [max(V) \cdots max(W)] : \operatorname{ovl}(n) = K \tag{7}$$

Any ordering that ensures property 7 suffices for the proof of Lemma 2. Figure 6 shows the region around the point 0 of a SSA-circular graph. Given $W = \mathcal{F}(V, K)$, exactly K copy points are inserted in the interval between $\max(V)$ and $\max(W)$.

Our proof is based on a reduction from the coloring of circular graphs. We proved our result for programs with a loop because the core of the interference graph of such programs is a circular graph. The existence of a back edge in the control flow graph is not a requirement for Theorem 1 to be true. For example, SSA-circular graphs can be obtained from a language with a single **if**-statement. Figure 9 (a) shows a SSA-circular graph that represents C_3 , when K = 2, and Figure 9 (b) shows a program whose live ranges represent such graph. The live ranges are outlined in Figure 9 (c).

Acknowledgments. We thank Fabrice Rastello, Christian Grothoff and the anonymous referees for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. Jens Palsberg is supported by the National Science Foundation award number 0401691.

References

- 1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
- M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. I: interval graphs. In Discrete Mathematics, pages 267–279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".
- Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In SIAM Symposium on Discrete Algorithms, pages 574–583, 1998.

- Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.
- Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. Software Practice and Experience, 28(8):859–881, 1998.
- Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomialtime graph coloring register allocation. In 14th International Workshop on Logic and Synthesis. ACM Press, 2005.
- Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In International Conference on Programming Languages Design and Implementation, pages 25–32. ACM Press, 2002.
- Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451–490, 1991.
- François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. ST Journal of Research Processor Architecture and Compilation for Embedded Systems, 1(2):81–96, 2004.
- Martin Farach and Vincenzo Liberatore. On local register allocation. In 9th ACM-SIAM symposium on Discrete Algorithms, pages 564 – 573. ACM Press, 1998.
- M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Algebraic Discrete Methods*, 1(2):216– 227, 1980.
- M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. *Theoretical Computer Science*, 1(3):193–267, 1976.
- Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SICOMP, 1(2):180–187, 1972.
- 15. Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46–56, 1974.
- Sebastian Hack. Interference graphs of programs in SSA-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005.
- Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In 15th International Conference on Compiler Construction. Springer-Verlag, 2006.
- Allen Leung and Lal George. Static single assignment form for machine code. In Conference on Programming Language Design and Implementation, pages 204–214. ACM Press, 1999.
- 19. Daniel Marx. A short proof of the NP-completeness of circular arc coloring, 2003.
- Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems, pages 315–329, 2005.
- B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In ACM SIGPLAN-SIGACT symposium on Principles of Programming languages, pages 12–27. ACM Press, 1988.
- 22. Ravi Sethi. Complete register allocation problems. In 5th annual ACM symposium on Theory of computing, pages 182–195. ACM Press, 1973.

A Verifiable SSA Program Representation for Aggressive Compiler Optimization

Vijay S. Menon¹

¹Intel Labs

Santa Clara, CA 95054

Neal Glew¹ Brian R. Murphy² Ali-Reza Adl-Tabatabai¹

Andrew McCreight³*
 Leaf Petersen¹

Tatiana Shpeisman¹

²Intel China Research Center Beijing, China ³Dept. of Computer Science, Yale University New Haven, CT 06520

{vijay.s.menon, brian.r.murphy, tatiana.shpeisman, ali-reza.adl-tabatabai, leaf.petersen}@intel.com aglew@acm.org andrew.mccreight@yale.edu

Abstract

We present a verifiable low-level program representation to embed, propagate, and preserve safety information in high performance compilers for safe languages such as Java and C#. Our representation precisely encodes safety information via static singleassignment (SSA) [11, 3] proof variables that are first-class constructs in the program.

We argue that our representation allows a compiler to both (1) express aggressively optimized machine-independent code and (2) leverage existing compiler infrastructure to preserve safety information during optimization. We demonstrate that this approach supports standard compiler optimizations, requires minimal changes to the implementation of those optimizations, and does not artificially impede those optimizations to preserve safety.

We also describe a simple type system that formalizes type safety in an SSA-style control-flow graph program representation. Through the types of proof variables, our system enables compositional verification of memory safety in optimized code.

Finally, we discuss experiences integrating this representation into the machine-independent global optimizer of STARJIT, a high-performance just-in-time compiler that performs aggressive control-flow, data-flow, and algebraic optimizations and is competitive with top production systems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Compilers; D.3.4 [*Programming Languages*]: Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Performance, Design, Languages, Reliability, Theory, Verification

Keywords Typed Intermediate Languages, Proof Variables, Safety Dependences, Check Elimination, SSA Formalization, Type Systems, Typeability Preservation, Intermediate Representations

POPL'06 January 11-13, 2006, Charleston, South Carolina, USA.

Copyright © 2006 ACM 1-59593-027-2/06/0001...\$5.00.

1. Introduction

In the past decade, safe languages have become prevalent in the general software community and have gained wide acceptance among software developers. Safe languages such as Java and C# are particularly prominent. These languages provide a C++-like syntax and feature set in conjunction with verifiable safety properties. Foremost among these properties is memory safety, the guarantee that a program will only read or write valid memory locations. Memory safety is crucial to both robustness and security. It prevents common programmer memory errors and security exploits such as buffer overruns through a combination of compile-time and run-time checks.

Both Java and C# were designed to allow programs to be compiled and distributed via bytecode formats. These formats retain the crucial safety properties of the source language and are themselves statically verifiable. Managed runtime environments (MRTEs), such as the Java Virtual Machine (JVM) or the Common Language Infrastructure (CLI), use static verification to ensure that no memory errors have been introduced inadvertently or maliciously before executing bytecode programs.

Bytecodes, however, are still rather high-level compared to native machine code. Runtime checks (e.g., array bounds checks) are built into otherwise potentially unsafe operations (e.g., memory loads) to ease the verification process. To obtain acceptable performance, MRTEs compile programs using a just-in-time (JIT) compiler. A JIT compiler performs several control- and data-flow compiler transformations and produces optimized native machine code. In the process, runtime checks are often eliminated or separated from the potentially unsafe operations that they protect. As far as we are aware, all production Java and CLI JIT compilers remove safety information during the optimization process: optimized low level code or generated machine code is not easily verifiable. From a security perspective, this precludes the use of optimized low level code as a persistent and distributable format. Moreover, from a reliability perspective it requires that the user trust that complex compiler transformations do not introduce memory errors.

In recent years, researchers have developed proof languages (e.g., PCC [20] and TAL [19]) that allow a compiler to embed safety proofs into low-level code, along with verification techniques to validate those proofs. They have demonstrated certifying compilers that can compile Java and safe C-like languages [21, 8, 18, 13] while both performing optimizations and generating safety proofs. Nevertheless, although the proof language and verification process is well-developed, implementing or modifying existing optimizations to correctly generate and/or preserve safety information is still an arduous and poorly understood process.

^{*} Supported in part by NSF grants CCR-0208618 and CCR-0524545.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

In this paper, we introduce a new program representation framework for safe, imperative, object-oriented languages to aid in the generation, propagation, and verification of safety information through aggressive compiler optimization. In this representation we encode *safety dependences*, the dependences between potentially unsafe operations and the control points that guarantee their safety, as abstract proof variables. These proof variables are purely static: they have no runtime semantics. Nevertheless, they are first class constructs produced by control points and consumed by potentially unsafe instructions. From the perspective of most compiler transformations, they are the same as any other variable.

We argue that this representation is particularly well-suited to use as an intermediate representation for an aggressively optimizing compiler. We demonstrate that it supports common advanced compiler optimizations without artificially constraining or extensively modifying them. In particular, we demonstrate that by carrying proof values in normal variables a compiler can leverage existing transformations such as SSA construction, copy propagation, and dead code elimination to place, update and eliminate proof variables.

We illustrate our ideas in the context of the machine-independent global optimizer of STARJIT [1], a dynamic optimizing compiler for Java and C#. STARJIT was designed as a high-performance optimizing compiler and is competitive in performance with the best production MRTE systems. We describe a prototype integration of our ideas into STARJIT's internal representation, and we discuss how it is able to preserve safety information through a varied set of aggressive optimizations. The original motivation for the safety dependence representation described in this paper was for optimization rather than safety. However, a prototype implementation of a verifier has also been developed, and this paper is intended to provide both a description of the safety dependence mechanism and a theoretical development of a type system based upon it.

In particular, our paper makes the following contributions:

- We introduce a safe low-level imperative program representation that combines static single-assignment (SSA) form with explicit safety dependences, and we illustrate how it can be used to represent highly optimized code.
- 2. We present a simple type system to verify memory safety of programs in this representation. To the best of our knowledge, this type system is the first to formalize type checking in an SSA representation. While SSA is in some sense equivalent to CPS, the details are sufficiently different that our type system is quite unlike the usual lambda-calculus style type systems and required new proof techniques.
- 3. We demonstrate the utility of this program representation in a high-performance compiler, and we describe how a compiler can leverage its existing framework to preserve safety information. In particular, we demonstrate that only optimizations that directly affect memory safety, such as bounds check elimination and strength reduction of address calculations, require significant modification.

The remainder of the paper is organized as follows. In Section 2, we motivate the explicit representation of safety dependence in an optimizing compiler and describe how to do this via proof variables in a low-level imperative program representation. In Section 3, we describe a formal core language specifically dealing with arraybounds checks and present a type system with which we can verify programs in SSA form. In Section 4, we demonstrate how a compiler would lower a Java program to the core language and illustrate how aggressive compiler optimizations produce efficient and verifiable code. In Section 5, we informally describe extensions to our core language to capture complete Java functionality. In Section 6,

if (a!=null)
<pre>while (!done) {</pre>
b = (B)a;
$\cdots = \cdots b \cdot x \cdots$
}

Figure 1. Field load in loop

we discuss the status of our current implementation, and, finally, in Sections 7 and 8 we discuss related work and conclude.

2. Motivation

We define a *potentially unsafe instruction* as any instruction that, taken out of context, might fault or otherwise cause an illegal memory access at runtime. Some instructions, taken independently, are inherently unsafe. A load instruction may immediately fault if it accesses protected memory or may trigger an eventual crash by reading an incorrectly typed value. A store may corrupt memory with an illegal value (e.g., if an arbitrary integer replaces an object's virtual table).

Consider, for example, the field access in Figure 1. Assuming C++-like semantics, the operation $b \cdot x$ dereferences memory with no guarantee of safety. In general, C++ does not guarantee that b refers to a real object of type B: b may hold an an integer that faults when used as a pointer.

Assuming Java semantics, however, the field access itself checks at runtime that b does not point to a null location. If the check succeeds, the field access executes the load; otherwise, it throws an exception, bypassing the load. By itself, this built-in check does not ensure safety: the load also depends on the preceding cast, which dynamically checks that the runtime type of b is in fact compatible with the type B. If the check succeeds, the cast executes the load; otherwise, it throws an exception, bypassing the load.

Typically, the safety of a potentially unsafe instruction depends on a set of control flow points. We refer to this form of dependence as *safety dependence*. In this example, the safety of the load depends on the cast that establishes its type. We call an instruction *contextually safe* when its corresponding safety dependences guarantee its safety. To verify the output of a compiler optimization, we must prove that each instruction is contextually safe.

2.1 Safety In Java

In Java and the verifiable subset of CLI, a combination of static verification and runtime checks guarantee the contextual safety of individual bytecode instructions. Static type checking establishes that variables have the appropriate primitive or object type. Runtime checks such as type tests (for narrowing operations), null pointer tests, and array bounds tests detect conditions that would cause a fault or illegal access and throw a language-level runtime exception instead.

Figure 2 shows Java-like bytecode instructions (using pseudoregisters in place of stack locations for clarity) for the code of Figure 1. The Java type system guarantees that variable b has type B at compile time, while the getfield instruction guarantees nonnull access by testing for null at runtime. The check and the static verifier together guarantee that the load operation will not trigger an illegal memory access.

2.2 Safety in a Low-Level Representation

The Java bytecode format was not intended to be an intermediate program representation for an optimizing compiler. There are a number of reasons why such a format is not suitable, but here we

```
L: \qquad \begin{array}{c} \text{ifnull } a \text{ goto } EXIT \\ L: \\ \text{ifeq } done \text{ goto } EXIT \\ b:= \text{checkcast}(a, B) \\ t_1:= \texttt{getfield}(b, B::x) \\ \cdots \\ \text{goto } L \\ EXIT: \end{array}
```

```
if a = null goto EXIT
L:
if done = 0 goto EXIT
checkcast(a, B)
checknull(a)
t_2 := getfieldaddr(a, B::x)
t_1 := ld(t_2)
\dots
goto L
EXIT
```



will focus only on those related to safety. First, bytecodes hide redundant check elimination opportunities. For example, in Figure 2, optimizations can eliminate the null check built into the getfield instruction because of the ifnull instruction. Even though several operations have built-in exception checks, programmers usually write their code to ensure that these checks never fail, so such optimization opportunities are common in Java programs.

Second, extraneous aliasing introduced to encode safety properties hides optimization opportunities. In Figures 1 and 2, variable *b* represents a copy of *a* that has the type *B*. Any use of *a* that requires this type information must use *b* instead. While this helps static verification, it hinders optimization. The field access must establish that *b* is not null, even though the ifnull statement establishes that property on *a*. To eliminate the extra check, a redundancy elimination optimization must reason about aliasing due to cast operations; this is beyond the capabilities of standard algorithms [16, 5].

In the absence of a mechanism for tracking safety dependences, STARJIT would lower a code fragment like this to one like that in Figure 3. Note that the 1d operation is potentially unsafe and is safety dependent on the null check. In this case, however, the safety dependence between the null check and the load is not explicit. Although the instructions are still (nearly) adjacent in this code, there is no guarantee that future optimizations will leave them so. Figure 4 roughly illustrates the code that STARJIT would produce for our example. Redundant checks are removed by a combination of partial loop peeling (to expose redundant control flow) and common subexpression elimination. The invariant address field calculation is hoisted via code motion. In this case, the dependence of the load on the operations that guarantee its safety (specifically, the if and checkcast statements) has become obscured. We refer to this as an erasure-style low-level representation, as safety information is effectively erased from the program.

An alternative representation embeds safety information directly into the values and their corresponding types. The Java language already does this for type refinement via cast operations. This approach also applies to null checks, as shown in Figure 5. The SafeTSA representation takes this approach, extending it to array bounds checks [25, 2] as well. We refer to this as a *refinement-style* representation. In this representation, value dependences preserve the safety dependence between a check and a load. To preserve

```
\begin{array}{l} t_2 := \texttt{getfieldaddr}(a,B::x)\\ \texttt{if} a = \texttt{null goto} EXIT\\ \texttt{if} done = 0 \texttt{goto} EXIT\\ \texttt{checkcast}(a,B)\\ L:\\ t_1 := \texttt{ld}(t_2)\\ \cdots\\ \texttt{if} done \neq 0 \texttt{goto} L\\ EXIT: \end{array}
```



 $\label{eq:linear_state} \begin{array}{l} \text{if } a = \texttt{null goto } EXIT \\ L: \\ & \\ \text{if } done = 0 \texttt{ goto } EXIT \\ & \\ b:= \texttt{checkcast}(a,B) \\ & \\ t_3:= \texttt{checknull}(b) \\ & \\ t_2:= \texttt{getfieldaddr}(t_3,B::x) \\ & \\ t_1:= \texttt{ld}(t_2) \\ & \\ & \\ \cdots \\ & \\ \texttt{goto } L \\ EXIT: \end{array}$

Figure 5. Field load lowered in refinement-style representation

safety, optimizations must preserve the value flow between the check and the load. Check elimination operations (such as the checknull in Figure 5) may be eliminated by optimization, but the values they produce (e.g., t_2) must be redefined in the process.

From an optimization standpoint, a refinement-style representation is not ideal. The safety dependence between the check and the load is not direct. Instead, it is threaded through the address field calculation, which is really just an addition operation. While the load itself cannot be performed until the null test, the address calculation is always safe. A code motion or instruction scheduling compiler optimization should be free to move it above the check if it is deemed beneficial. In Figure 3, it is clearly legal. In Figure 5, it is no longer possible. The refinement-style representation adds artificial constraints to the program to allow safety to be checked. In this case, the address calculation is artificially dependent on the check operation.

A refinement-style representation also obscures optimization opportunities by introducing multiple names for the same value. Optimizations that depend on syntactic equivalence of expressions (such as the typical implementation of redundancy elimination) become less effective. In Figure 3, *a* is syntactically compared to null twice. In Figure 5, this is no longer true. In general, syntactically equivalent operations in an erasure-style representation may no longer be syntactically equivalent in a refinement-style representation.

2.3 A Proof Passing Representation

Neither the erasure-style nor refinement-style representations precisely represent safety dependences. The erasure-style representation omits them altogether, while the refinement-style representation encodes them indirectly. As a result, the erasure-style representation is easy to optimize but difficult to verify, while the refinement-style is difficult to optimize but easy to verify.

To bridge this gap, we propose the use of a *proof passing* representation that encodes safety dependence directly into the program representation through proof variables. Proof variables act as capabilities for unsafe operations (similar to the capabilities of Walker et al. [26]). The availability of a proof variable represents the availability of a proof that a safety property holds. A potentially unsafe instruction must use an available proof variable to ensure

$$\begin{bmatrix} s_1, s_2 \end{bmatrix} \text{ if } a = \text{null goto } EXIT \\ L: \\ \text{ if } done = 0 \text{ goto } EXIT \\ s_3 := \text{checkcast}(a, B) \\ s_4 := \text{checknull}(a) \\ t_2 := \text{getfieldaddr}(a, B :: x) \\ s_5 := \text{pfand}(s_3, s_4) \\ t_1 := \text{ld}(t_2) [s_5] \\ \dots \\ \text{goto } L \\ EXIT \\ \end{bmatrix}$$

Figure 6. Field load lowered in a proof passing representation

```
t_2 := \texttt{getfieldaddr}(a, B :: x)
           [s_1, s_2] if a = null goto EXIT
L:
           if done = 0 goto EXIT
           s_3 := \texttt{checkcast}(a, B)
           s_4 := s_1
           s_5 := pfand(s_3, s_4)
           t_1 := \operatorname{ld}(t_2) [s_5]
           \verb"goto" L
EXIT :
```



contextual safety. This methodology relates closely to mechanisms proposed for certified code by Crary and Vanderwaart [10] and Shao et al. [23] in the context of the lambda calculus. We discuss the relationship of our approach to this work in Section 7.

Proof variables do not consume any physical resources at runtime: they represent abstract values and only encode safety dependences. Nevertheless, they are first-class constructs in our representation. They are generated by interesting control points and other relevant program points, and consumed by potentially unsafe instructions as operands guaranteeing safety. Most optimizations treat proof variables like other program variables.

Figure 6 demonstrates how we represent a load operation in a proof passing representation. As in Figure 5, we represent safety through value dependences, but instead of interfering with existing values, we insert new proof variables that directly model the safety dependence between the load and both check operations.

Figures 7 to 10 represent the relevant transformations performed by STARJIT to optimize this code. In Figure 7, we illustrate two optimizations. First, STARJIT's common subexpression elimination pass eliminates the redundant checknull operation. When STAR-JIT detects a redundant expression in the right hand side of an instruction, it replaces that expression with the previously defined variable. The if statement defines the proof variable s_1 if the test fails. This variable proves the proposition $a \neq \text{null}$. At the definition of s_4 , the compiler detects that $a \neq \text{null}$ is available, and redefines s_4 to be a copy of s_1 . STARJIT updates a redundant proof variable the same way as any other redundant variable.

Second, STARJIT hoists the definition of t_2 , a loop invariant address calculation, above the loop. Even though the computed address may be invalid at this point, the address calculation is always safe; we require a proof of safety only on a memory operation that dereferences the address.

Figure 8 shows a step of copy propagation, which propagates s_1 into the load instruction and eliminates the use of s_4 , allowing dead code elimination to remove the definition of s_4 .

Figure 9 illustrates the use of partial loop peeling to expose redundant control flow operations within the loop. This transforma-

```
t_2 := \texttt{getfieldaddr}(a, B::x)
           [s_1, s_2] if a = null goto EXIT
L:
           if done = 0 goto EXIT
           s_3 := \texttt{checkcast}(a, B)
           s_5 := pfand(s_3, s_1)
           t_1 := \operatorname{Id}(t_2)[s_5]
           goto L
EXIT :
```



```
t_2 := \texttt{getfieldaddr}(a, B::x)
            [s_1, s_2] if a = null goto EXIT
            \texttt{if } done = 0 \texttt{ goto } \textit{EXIT}
            s_3^1 := \texttt{checkcast}(a, B)
            s_3^2 := \phi(s_3^1, s_3^3)
            s_5 := \operatorname{pfand}(s_3^2, s_1)
            t_1 := 1d(t_2) [s_5]
            if done = 0 goto EXIT
            s_3^3 := \text{checkcast}(a, B)
            goto L
EXIT :
```

L:



 $t_2 := \texttt{getfieldaddr}(a, B::x)$ $[s_1, s_2]$ if a =null goto $E\!X\!I\!T$ if done = 0 goto EXIT $s_3 := \texttt{checkcast}(a, B)$ $s_5 := pfand(s_3, s_1)$ L: $t_1 := \operatorname{ld}(t_2) [s_5]$ if *done* $\neq 0$ goto *L* EXIT :

Figure 10. Field load with 2nd CSE and Branch Reversal

tion duplicates the test on done and the checkcast operation, and makes the load instruction the new loop header. The proof variable s_3 is now defined twice, where each definition establishes that a has type B on its corresponding path. The compiler leverages SSA form to establish that the proof variable is available within the loop.

Finally, in Figure 10, another pass of common subexpression elimination eliminates the redundant checkcast. Copy propagation propagates the correct proof variable, this time through a redundant phi instruction. Note, that this final code is equivalent to the erasure-style representation in Figure 4 except that proof variables provide a direct representation of safety. In Figure 10, it is readily apparent that the if and checkcast statements establish the safety of the load instruction.

In the next section we formalize our approach as a small core language, and the following sections show its use and preservation across compiler optimizations and extension to full Java.

3. **Core Language**

In this section we describe a small language that captures the main ideas of explicit safety dependences through proof variables. As usual with core languages, we wish to capture just the essence of the problem and no more. The issue at hand is safety dependences, and to keep things simple we will consider just one such depen-

P(b.i)	L_2	n_2	pc	Side conditions
\overline{p}	$L_1\{\overline{x}_1 := L_1(\overline{x}_2)\}$	n_1	b.(i+1)	$\overline{p}[n_1] = \overline{x}_1 := \overline{x}_2$
x: au: au:=i	$L_1\{x := i\}$	n_1	b.(i+1)	
$x_1:\tau:=x_2$	$L_1\{x_1 := L_1(x_2)\}$	n_1	b.(i+1)	
$x_1: au:=\mathtt{newarray}(x_2,x_3)$	$L_1\{x_1 := v_1\}$	n_1	b.(i+1)	$L_1(x_2) = n, L_1(x_3) = v_3, v_1 = \langle v_3, \dots, v_3 \rangle$
$x_1: \tau := \texttt{newarray}(x_2, x_3)$	$L_1\{x_1 := v_1\}$	n_1	b.(i+1)	$L_1(x_2) = i, i < 0, v_1 = \langle \rangle$
$x_1:\tau:=\texttt{len}(x_2)$	$L_1\{x_1 := n\}$	n_1	b.(i+1)	$L_1(x_2) = \langle v_0, \dots, v_{n-1} \rangle$
$x_1: au := \mathtt{base}(x_2)$	$L_1\{x_2 := v@0\}$	n_1	b.(i+1)	$L_1(x_2) = v, v = \langle \overline{v'} \rangle$
$x_1:\tau:=x_2 \ bop \ x_3$	$L_1\{x_1 := i_4\}$	n_1	b.(i+1)	$L_1(x_2) = i_2, \ L_1(x_3) = i_3, \ i_4 = i_2 \ bop \ i_3$
$x_1:\tau:=x_2 \ bop \ x_3$	$L_1\{x_1 := v@i_4\}$	n_1	b.(i+1)	$L_1(x_2) = v@i_2, L_1(x_3) = i_3, i_4 = i_2 \text{ bop } i_3$
$x_1:\tau:=\mathtt{ld}(x_2)\ [x_3]$	$L_1\{x_1 := v_i\}$	n_1	b.(i+1)	$L_1(x_2) = \langle v_0, \dots, v_n \rangle @i, 0 \le i \le n$
$x_1: au:=\texttt{pffact}(x_2)$	$L_1\{x_1 := \texttt{true}\}$	n_1	b.(i+1)	
$x: au:=\mathtt{pfand}(\overline{y})$	$L_1\{x := \texttt{true}\}$	n_1	b.(i+1)	
$[x_1: au_1,x_2: au_2]$ if $x_3 \ rop \ x_4$ goto b'	$L_1\{x_1 := \texttt{true}\}$	$edge_P(b, b+1)$	(b+1).0	$L_1(x_3) = i_3, L_1(x_4) = i_4, \neg(i_3 \text{ rop } i_4)$
$[x_1: au_1,x_2: au_2]$ if $x_3 \ rop \ x_4$ goto b'	$L_1\{x_2 := \texttt{true}\}$	$edge_P(b, b')$	b'.0	$L_1(x_3) = i_3, L_1(x_4) = i_4, i_3 \text{ rop } i_4$
goto b'	L_1	$edge_P(b,b')$	b'.0	

 $(P, L_1, n_1, b.i) \mapsto (P, L_2, n_2, pc)$ where:

Figure 11. Operational semantics

dence, namely, bounds checking for arrays. In particular, we consider a compiler with separate address arithmetic, load, and store operations, where the type system must ensure that a load or store operation is applied only to valid pointers. Moreover, since the basic safety criteron for a store is the same as for a load, namely, that the pointer is valid, we consider only loads; adding stores to our core language adds no interesting complications. Not considering stores further allows us to avoid modelling the heap explicitly, but to instead use a substitution semantics which greatly simplifies the presentation.

The syntax of our core language is given as follows:

Prog. States	S	::=	(P, L, n, pc)
Programs	P	::=	\overline{B}
Blocks	B	::=	$\overline{p}; \overline{\iota}; c$
Phi Instructions	p	::=	$x: \tau := \phi(\overline{x})$
Instructions	ι	::=	$x: \tau := r$
Right-hand sides	r	::=	$i \mid x \mid \texttt{newarray}(x_1, x_2) \mid$
			$len(x) \mid base(x) \mid$
			$x_1 \ bop \ x_2 \ \ ld(x_1) \ [x_2] \ $
			$pffact(x) \mid pfand(\overline{x})$
Binary Ops	bop	::=	+ -
Transfers	c	::=	goto $n \mid \texttt{halt} \mid$
			$[x_1: au_1, x_2: au_2]$ if $x_3 \ rop \ x_4$
			goto n
Relations	rop	::=	< ≤ = ≠
Environments	L	::=	$\overline{x} := \overline{v}$
Values	v	::=	$i \mid \langle \overline{v} angle \mid \langle \overline{v} angle @i \mid extsf{true}$
Prog. Counters	pc	::=	$n_1.n_2$

Here *i* ranges over integer constants, *x* ranges over variables, *n* ranges over natural numbers, and ϕ is the phi-operation of SSA. We use the bar notation introduced in Featherweight Java [15]: \overline{B} abbreviates $B_0, \ldots, B_n, \overline{x} := \overline{v}$ abbreviates $x_0 := v_0, \ldots, x_n := v_n$, *et cetera*. We also use the bar notation in type rules to abbreviate a sequence of typing judgements in the obvious way. In addition to the grammar above, programs are subject to a number of context-sensitive restrictions. In particular, the *n* in $[x_1:\tau_1, x_2:\tau_2]$ if x_3 rop x_4 goto *n* and goto *n* must be a block number in the program (i.e., if the program is B_0, \ldots, B_m then $0 \le n \le m$); the transfer in the last block must be a goto or halt; the number of variables in a phi instruction must equal the number of incoming edges (as defined below) to the block in which it appears; the variables assigned in the phi instructions of a block must be distinct.

Informally, the key features of our language are the following. The operation base(x) takes an array and creates a pointer to the element at index zero. The arithmetic operations can be applied to such pointers and an integer to compute a pointer to a different index. The $ld(x_1)$ $[x_2]$ operation loads the value pointed to by the pointer in x_1 . The variable x_2 is a proof variable and conceptually contains a proof that x_1 is a valid pointer: that is, that it points to an in-bounds index. The typing rules ensure that x_1 is valid by requiring x_2 to contain an appropriate proof. The operations pffact(x)and $pfand(\overline{x})$ construct proofs. For pffact(x) a proof of a formula based on the definition of x is constructed. For example, if x's definition is x : int := len(y) then pffact(x) constructs a proof of x = len(y). A complete definition of the defining facts of instructions appears in Figure 14. For $pfand(x_1, \ldots, x_n), x_1$ through x_n are also proof variables, and a proof of the conjunction is returned. Values of the form $\langle v_0, \ldots, v_n \rangle @i$ represent pointers to array elements: in this case a pointer to the element at index iof an array of type $\langle v_0, \ldots, v_n \rangle$. Such a pointer is valid if *i* is in bounds (that is, if $0 \le i \le n$) and invalid otherwise. The typing rules must ensure that only valid pointers are loaded from, with proof variables used to provide evidence of validity. The final unusual aspect of the language is that branches assign proofs to proof variables that reflect the condition being branched on. For example, in the branch $[x_1:\tau_1, x_2:\tau_2]$ if $x_3=x_4$ goto n, a proof of $x_3 \neq x_4$ is assigned to x_1 along the fall-through edge, and a proof of $x_3 = x_4$ is assigned to x_2 along the taken edge. These proofs can then be used to discharge validity requirements for pointers.

To state the operational semantics and type system we need a few definitions. The program counters of a program pcs(P) are $\{b.i \mid P = B_0, \ldots, B_m \land b \leq m \land B_b = \overline{p}; \iota_1; \cdots; \iota_n; c \land i \leq n+1\}$. We write P(b) for B_b when $P = B_0, \ldots, B_n$ and $b \leq n$; if $P(b) = \overline{p}; \iota_1; \ldots; \iota_n; c$ then P(b.n) is \overline{p} when n = 0, and ι_n when $1 \leq n \leq m$ and c when n = m + 1. The edges of a program P, edges(P), are as follows. The entry edge is (-1, 0). If P(n) ends in $[x_1:\tau_1, x_2:\tau_2]$ if x_3 rop x_4 goto n' then there are edges (n, n+1), called the fall-through edge, and (n, n'), called the taken edge. If P(n) ends in goto n' then there is an edge (n, n'). For a given P and n_2 the edges $(n_1, n_2) \in edges(P)$ are numbered from zero in the order given by n_1 ; $edge_P(n_1, n_2)$ is this number, also called the incoming edge number of (n_1, n_2) into n_2 .

Operational Semantics A program P is started in the state $(P, \emptyset, 0, 0.0)$. The reduction relation that maps one state to the next is given in Figure 11. Note that the third component of a pro-

gram state tracks which incoming edge led to the current program counter—initially this is the entry edge (-1,0), and is updated by transfers. It is used by phi instructions to select the correct variable. The notation $\overline{p}[i]$ denotes $x_1 := x_{1i}, \ldots, x_n := x_{ni}$ when $\overline{p} = x_1 : \tau_1 := \phi(x_{11}, \ldots, x_{1m}), \ldots, x_n : \tau_n := \phi(x_{n1}, \ldots, x_{nm})$. A program terminates when in a state of the form (P, L, n, pc)where P(pc) = halt. A program state is stuck if it is irreducible and not a terminal state. Stuck states all represent type errors that the type system should prevent. Note that the array creation operation must handle negative sizes. Our implementation would throw an exception, but since the core language does not have exceptions, it simply creates a zero length array if a negative size is requested.

In the operational semantics, the proof type has the single inhabitant true, upon which no interesting operations are defined. Proofs in this sense are equivalent to unit values for which nonescaping occurrences can be trivially erased when moving to an untyped setting. This "proof erasure" property is precisely analogous to the "coercion erasure" property of the coercion language of Vanderwaart et al. [24]. In practice, uses of proof variables in the STARJIT compiler are restricted such that all proof terms can be elided during code generation and consequently impose no overhead at run time. While we believe that it would be straightforward to formalize the syntactic restrictions that make this possible, we choose for the sake of simplicity to leave this informal here.

Type System The type system has two components: the SSA property and a set of typing judgements. The SSA property ensures both that every variable is assigned to at most once in the program text (the single assignment property) and that all uses of variables are dominated by definitions of those variables. In a conventional type system, these properties are enforced by the typing rules. In particular, the variables that are listed in the context of the typing judgement are the ones that are in scope. For SSA IRs, it is more convenient to check these properties separately.

The type checker must ensure that during execution each use of a variable is preceded by an assignment to that variable. Since the *i*th variable of a phi instruction is used only if the *i*-th incoming edge was used to get to the block, and the proof variables in an if transfer are assigned only on particular out-going edges, we give a rather technical definition of points at which variables are assigned or used. These points are such that a definition point dominating a use point implies that assignment will always precede use. These points are based on an unconventional notion of control-flow graph, to avoid critical edges which might complicate our presentation. For a program P with blocks 0 to m, the control-flow graph consists of the nodes $\{0, \ldots, m\} \cup edges(P)$ and edges from each original node n to each original edge (n, n') and similarly from (n, n')to n'. The definition/use points, du(P), are $pcs(P) \cup \{b.0.i\}$ $P(b.0) = p_0, \dots, p_n \land 0 \le i \le n \} \cup \{e.i \mid e \in \mathsf{edges}(P) \land i \in$ $\{0,1\}\}.$

Figure 13 gives the formal definition of dominance, definition/use points, and the SSA property.

The syntax of types is:

 $\begin{array}{lll} \text{Types} & \tau ::= \texttt{int} \mid \texttt{array}(\tau) \mid \texttt{ptr}_? \langle \tau \rangle \mid \texttt{S}(x) \mid \texttt{pf}_{(F)} \\ \text{Facts} & F ::= e_1 \ \textit{rop} \ e_2 \mid F_1 \wedge F_2 \\ \text{Fact Exps.} & e ::= i \mid x \mid \texttt{len}(x) \mid e_1 \ \textit{bop} \ e_2 \mid x@e \\ \text{Environments} \ \Gamma ::= \overline{x} : \overline{\tau} \end{array}$

The type $ptr_{?}\langle \tau \rangle$ is given to pointers that, if valid, point to values with type τ (the ? indicates that they might not be valid). The singleton type S(x) is given to things that are equal to x. The type $pf_{(F)}$ is given to proof variables that contain a proof of the fact F. Facts include arithmetic comparisons and conjunction. Fact expressions include integers, variables, array lengths, arithmetic operations, and a subscript expression—the fact expression x@estands for a pointer that points to the element at index e of array x.

Judgement	Meaning
$\Gamma \vdash \tau_1 \le \tau_2$	τ_1 is a subtype of τ_2 in Γ
$\vdash F_1 \implies F_2$	F_1 implies F_2
$\Gamma \vdash \overline{p}$	\overline{p} is safe in environment Γ
$\Gamma \vdash_P \iota$	ι is safe in environment Γ
$\Gamma \vdash c$	c is safe in environment Γ
$\vdash_P \tau$ at du	au well-formed type at du in P
$\vdash_P \Gamma$	environment Γ well-formed in P
$\vdash P$	P is safe

Figure 12. Typing judgements

The judgements of the type system are given in figure 12. Most of the typing rules are given in Figure 14. Typing environments Γ state the types that variables are supposed to have. The rules check that when assignments are made to a variable, the type of the assigned value is compatible with the variable's type. For example, the judgement $\Gamma \vdash \text{int} \leq \Gamma(x)$ in the rule for $x : \tau := i$ checks that integers are compatible with the type of x. The rules also check that uses of a variable have a type compatible with the operation. For example, the rule for load expects a proof that the pointer, x_2 , is valid, so the rule checks that x_3 's type $\Gamma(x_3)$ is a subtype of $pf_{(x \otimes 0 \leq x_2 \wedge x_2 < x \otimes 1en(x))}$ for some x. It is this check along with the rules for proof value generation and the SSA property that ensure that x_2 is valid.

Given these remarks, the only other complicated rule is for phi instructions. In a loop a phi instruction might be used to combine two indices, and the compiler might use another phi instruction to combine the proofs that these indices are in bounds. For example, consider this sequence:

$$x_1 : int := \phi(x_2, x_3) y_1 : pf_{(0 \le x_1)} := \phi(y_2, y_3)$$

where y_2 : $pf_{(0 \le x_2)}$ and y_3 : $pf_{(0 \le x_3)}$. Here the types for y_1 , y_2 , and y_3 are different and in some sense incompatible, but are intuitively the correct types. The rule for phi instructions allows this typing. In checking that y_2 has a compatible type, the rule substitutes x_2 for x_1 in y_1 's type to get $pf_{(0 \le x_2)}$, which is the type that y_2 has; similarly for y_3 .

For a program P that satisfies the SSA property, every variable mentioned in the program has a unique definition point, and that definition point is decorated with a type. Let vt(P) denote the environment formed from extracting these variable/type pairs. A program P is well formed ($\vdash P$) if:

- 1. P satisfies the SSA property,
- 2. $\vdash_P \mathsf{vt}(P)$,
- 3. $vt(P) \vdash \overline{p}$ for every \overline{p} in P,
- 4. $vt(P) \vdash_P \iota$ for every instruction ι in P, and
- 5. $vt(P) \vdash c$ for every transfer c in P.

The type system is safe:

THEOREM 1 (Type Safety). If $\vdash P$ and $(P, \emptyset, 0, 0.0) \mapsto^* S$ then S is not stuck.

A proof of this theorem appears in the companion technical report [17]. The proof takes the standard high-level form of showing preservation and progress lemmas, as well as some lemmas particular to an SSA language. It is important to note that safety of the type system is contingent on the soundness of the decision procedure for $\vdash F_1 \implies F_2$. In the proof, a judgement corresponding to truth of facts in an environment is given. In this setting, the assumption of logical soundness corresponds to the restriction that in any environment in which F_1 is true, F_2 is also true.

Defs and Uses:

If $P(b.i) = x : \tau := r$ then program counter b.i defines x, furthermore, b.i is a use of the ys where r has the following forms:

 $y \mid \texttt{newarray}(y_1, y_2) \mid \texttt{len}(y) \mid \texttt{base}(y) \mid y_1 \; bop \; y_2 \mid \texttt{ld}(y_1) \; [y_2] \mid \texttt{pffact}(y) \mid \texttt{pfand}(\overline{y})$

If $P(b,i) = (p_0, \ldots, p_n)$ and $p_j = x_j : \tau_j := \phi(y_{j1}, \ldots, y_{jm})$ then b.i.j defines each x_j and $e_k.1$ uses each y_{jk} where e_k is the k-th incoming edge of b. If $P(b,i) = [x_1 : \tau_1, x_2 : \tau_2]$ if y_1 nop y_2 goto n then $e_1.0$ defines x_1 and $e_2.0$ defines x_2 where e_1 and e_2 are the fall-through and taken edges respectively, and b.i uses y_1 and y_2 . If x has a unique definition/use point in P that defines it, then def_P(x) is this point.

Dominance:

- In program P, node n dominates node m, written $dom_P(n, m)$, if every path in the control-flow graph of P from (-1, 0) to m includes n.
- In program P, definition/use point $n_1.i_1$ strictly dominates definition/use point $n_2.i_2$, written $sdom_P(n_1.i_1, n_2.i_2)$ if $n_1 = n_2$ and $i_1 < i_2$ (here i_1 or i_2 might be a dotted pair 0.j, so we take this inequality to be lexicographical ordering) or $n_1 \neq n_2$ and $dom_P(n_1, n_2)$.

Single Assignment:

A program satisfies the single-assignment property if every variable is defined by at most one definition/use point in that program.

In Scope:

A program P satisfies the *in-scope property* if for every definition/use point du_1 that uses a variable there is a definition/use point du_2 that defines that variable and sdom_P(du_2, du_1).

SSA:

A program satisfies the *Single Static Assignment (SSA) property* if it satisfies the single-assignment and in-scope properties. Note that a program that satisfies SSA has a unique definition for each variable mentioned in the program.



Figure 14. Typing rules

The typing rules presented are for the most part syntax-directed, and can be made algorithmic. A consideration is that the rule for load must determine the actual array variable, which is not apparent from the conclusion. In general, the decision prodecure only needs to verify that the rule holds for one of the arrays available at that program point. In practice, the correct array can be inferred by examining the type of the proof variable. We believe that judgements on facts may be efficiently decided by an integer linear programming tool such as the Omega Calculator [22] with two caveats. First, such tools reason over \mathbb{Z} rather than 32- or 64-bit integers. Second, they restrict our fact language for integer relations (and, thus, compiler reasoning) to affine expressions. This is, however, sufficient to capture current STARJIT optimizations.

4. Compiler optimizations

In this section we examine compiler optimizations in the context of the core language. We demonstrate how an optimizing compiler can preserve both proof variables and their type information. We argue that our ideas greatly simplify this process. In previous work, an implementer would need to modify each optimization to update safety information. In our representation, we leverage existing compiler infrastructure to do the bulk of the work. In particular, most controlflow or data-flow optimizations require virtually no changes at all. Others that incorporate algebraic properties only need to be modified to record the compiler's reasoning. In the next section we will discuss how these ideas can be extended from the core language to full Java.

In general, there are two ways in which an optimization can maintain the correctness of the proofs embedded in the program. First, it can apply the transformation to both computation and proof simultaneously. This is sufficient for the majority of optimizations. Second, it can create new proofs for the facts provided by the original computation. As we show below, this is necessary for the few optimizations that infer new properties that affect safety. In the rest of this section we show how these general principles apply to individual compiler optimizations on a simple example. For this example, we show how to generate a low-level intermediate representation that contains safety information and how to preserve this information through several compiler optimizations, such as loop invariant code motion, common subexpression elimination, array bounds check elimination, strength reduction of array element pointer, and linear function test replacement.

The example we will consider, in pseudo code, is:

Where we assume that a is a non-null integer array, that a is not modified in the loop, and that the pseudo code array subscripting has an implicit bounds check. Although this example does not reflect the full complexity of Java, it is sufficient to illustrate the main ideas of propagating safety information through the compiler optimizations. Section 5 discusses additional issues in addressing full Java.

The first compilation step for our example lowers the program into a low-level representation suitable for optimization, as shown in Figure 15. In our system, lowering generates instructions that express the computation and any required proofs of the computation's safety. For example, a typical compiler would expand an array element access a[i] into the following sequence: array bounds checks, computation of the array element address, and a potentially unsafe load from that address. In our system, the compiler also generates proof variables that show that the array index *i* is within the array bounds (q_4 for the lower bound and q_6 for the upper bound) and that the load accesses an element *i* of the array *a* (proof vari-

$uB: intLOOP:i_2: int[q_1:pf_{(i_2 \leq uR)}, q_2:] :=$	$\begin{split} &:= \texttt{len}(a) \\ &:= \phi(i_1, i_3) \\ & \texttt{if } uB \leq i_2 \texttt{ goto } EXIT \\ &:= \texttt{len}(a) \\ &:= \texttt{pffact}(aLen) \\ &:= \texttt{checkLowerBound}(i_2, 0) \end{split}$
$LOOP:$ $i_2: int$ $[q_1: pf_{(i_2 < uR)}, q_2: \ldots] :=$	$\begin{split} &:= \phi(i_1, i_3) \\ &\text{if } uB \leq i_2 \text{ goto } EXIT \\ &:= \texttt{len}(a) \\ &:= \texttt{pffact}(aLen) \\ &:= \texttt{checkLowerBound}(i_2, 0) \end{split}$
$i_2: int [q_1: pf_{(i_2 < uB)}, q_2: \ldots] :=$	$\begin{split} &:= \phi(i_1, i_3) \\ &\text{if } uB {\leq} i_2 \text{ goto } EXIT \\ &:= \texttt{len}(a) \\ &:= \texttt{pffact}(aLen) \\ &:= \texttt{checkLowerBound}(i_2, 0) \end{split}$
$[q_1: pf_{(i_0 < \mu B)}, q_2: \ldots] :=$	$\begin{array}{l} \text{if } uB \leq i_2 \text{ goto } EXIT \\ := \texttt{len}(a) \\ := \texttt{pffact}(aLen) \\ := \texttt{checkLowerBound}(i_2, 0) \end{array}$
··· · · · · · · · · · · · · · · · · ·	:=len(a) :=pffact(aLen) :=checkLowerBound(i ₂ ,0)
aLen: int	:= pffact(aLen) := checkLowerBound(i ₂ , 0)
$q_3: pf_{(aLen=len(a))}$	$:=$ checkLowerBound $(i_2, 0)$
$q_4: pf_{(0 \le i_2)}$	(
$q_5: pf_{(i_2 < aLen)}$	$:=$ checkUpperBound $(i_2, aLen)$
$q_6: pf_{(i_2 < len(a))}$	$:=\!\texttt{pfand}(q_3,q_5)$
$aBase: ptr_{2}(int)$:=base (a)
$q_7: pf_{(aBase=a@0)}$:=pffact(aBase)
$addr: ptr_{?}(int)$	$:= aBase + i_2$
$q_8: pf_{(addr=aBase+i_2)}$:= pffact(addr)
$q_9: pf_{(addr=a@i_2)}$	$:= \texttt{pfand}(q_7, q_8)$
$q_{10}: pf_{(a@0 \le addr \le a@len(a))}$	$:=$ pfand (q_4, q_6, q_9)
val:int	$:= \operatorname{ld}(addr) [q_{10}]$
:	:= val
$i_3: \texttt{int}$	$:=i_2+1$
	goto LOOP
EXIT:	

Figure 15. Low-level representation for array load in loop

$i_1: \texttt{int}$:= 0
uB:int	:= len(a)
$q_3: pf_{(uB=len(a))}$:= pffact(uB)
$aBase: ptr_2(int)$:=base (a)
$q_7: pf_{(aBase=a@0)}$:= pffact(aBase)
LOOP:	-
$i_2: \texttt{int}$	$:= \phi(i_1, i_3)$
$[q_1 : pf_{(i_2 \le uB)}, q_2 : \ldots] :=$	if $uB \leq i_2$ goto $EXIT$
$q_4: pf_{(0 \le i_2)}$	$:=$ checkLowerBound $(i_2,0)$
$q_5: pf_{(i_2 \leq uB)}^{(i_2 \leq uB)}$	$:= \texttt{checkUpperBound}(i_2, uB)$
$q_6: pf_{(i_2 < len(a))}$	$:=\!\texttt{pfand}(q_3,q_5)$
$addr: ptr_{?}(int)$	$:= aBase + i_2$
$q_8: pf_{(addr=aBase+i_2)}$:= pffact(addr)
$q_9: pf_{(addr=a@i_2)}$	$:= \texttt{pfand}(q_7, q_8)$
$q_{10}: pf_{(a@0 \le addr \le a@len(a))}$	$:=$ pfand (q_4, q_6, q_9)
val : int	$:= \operatorname{ld}(addr) [q_{10}]$
:	:=val
$i_3: \texttt{int}$	$:=i_2+1$
	goto LOOP
EXIT:	

Figure 16. IR after CSE and loop invariant code motion

able q_9). The conjunction of these proofs is sufficient to type check the load instruction according to the typing rules in Figure 14. The proof variables are generated by the explicit array bounds checks (which we use as syntactic sugar for the branches that transfer control to a halt instruction if the bounds check fails) and by pffact and pfand statements that encode arithmetic properties of the address computation as the types of proof variables.

Next, we take the example in Figure 15 through several common compiler optimizations that are employed by STARJIT to generate efficient code for loops iterating over arrays (Figures 16 - 19). The result is highly-optimized code with an embedded proof of program safety.

We start, in Figure 16, by applying several basic data-flow optimizations such as CSE, dead code elimination, and loop invariant code motion. An interesting property of these optimizations in our system is that they require no modification to preserve the

$i_1: \texttt{int}$:= 0
$q_{11}: pf_{(i_1=0)}$	$:= \texttt{pffact}(i_1)$
uB: int	:= len(a)
$q_3: pf_{(uB=len(a))}$:= pffact(uB)
$aBase: ptr_{?}(int)$:= base(a)
$q_7: pf_{(aBase=a@0)}$:= pffact(aBase)
LOOP:	
$i_2: \texttt{int}$	$:= \phi(i_1, i_3)$
$q_4: pf_{(0 \le i_2)}$	$:= \phi(q_{11}, q_{13})$
$[q_1: pf_{(i_2 < uB)}, q_2: \ldots] :=$	if $\mathit{uB}{\leq}i_2$ goto $EXIT$
$q_6: pf_{(i_2 < len(a))}$	$:=\!\texttt{pfand}(q_3,q_1)$
$addr: \tilde{\mathtt{ptr}}_{?}\langle \mathtt{int} \rangle$	$:= aBase + i_2$
$q_8: pf_{(addr=aBase+i_2)}$:= pffact(addr)
$q_9: pf_{(addr=a@i_2)}$	$:= \texttt{pfand}(q_7, q_8)$
$q_{10}: pf_{(a@0 \leq addr \leq a@len(a))}$	$:=$ pf and (q_4, q_6, q_9)
val:int	$:= \operatorname{ld}(addr) [q_{10}]$
:	:= val
i_3 : int	$:=i_2+1$
$q_{12}: \mathtt{pf}_{(i_3=i_2+1)}$	$:= \texttt{pffact}(i_3)$
$q_{13}: \mathtt{pf}_{(0 \le i_3)}$	$:= \texttt{pfand}(q_4, q_{12})$
(_))	goto $LOOP$
EXIT:	

Figure 17. IR after bound check elimination

safety proofs. They treat proof variables identically to other terms, and, thus, are automatically applied to both the computation and the proofs. For example, common subexpression elimination and copy propagation replace *all* occurrences of *aLen* with *uB*, including those that occur in proof types. The type of the proof variable q_3 is updated to match its new definition pffact(*uB*).

In Figure 17, we illustrate array bounds check elimination. In the literature [4], this optimization is typically formulated to remove redundant bounds checks without leaving any trace of its reasoning in the program. In such an approach, a verifier must effectively repeat the optimization reasoning to prove program safety. In our system, an optimization cannot eliminate an instruction that defines a proof variable without constructing a new definition for that variable or removing all uses of that variable. Intuitively, the compiler must record in a new definition its reasoning about why the eliminated instruction was redundant. Consider the bounds checks in Figure 16. The lower bound check that verifies that $0 \le i_2$ is redundant because i_2 is a monotonically increasing variable with the initial value 0. Formally, the facts that $i_1=0$, $i_2=\phi(i_1,i_3)$ and $i_3=i_2+1$ imply that $0 \le i_2$. This reasoning is recorded in the transformed program through a new definition of the proof variable q_A and the additional proof variables q_{11} and q_{13} . We use SSA to connect these proofs at the program level. The upper bound check that verifies that $i_2 < len(a)$ (proof variable q_5) is redundant because the if statement guarantees the same condition (proof variable q_1). Because the new proof for the fact q_5 is already present in the program, the compiler simply replaces all uses of of q_5 with q_1 .

In Figure 18, we perform operator strength reduction (OSR) [9] to find a pointer that is an affine expression of a monotonically increasing or decreasing loop index variable and to convert it into an independent induction variable. In our example, OSR eliminates *i* from the computation of *addr* by incrementing it directly. Because variable *addr* is used in the $q_8 := pffact(addr)$ statement, the compiler cannot modify the definition of *addr* without also modifying the definition of q_8 (otherwise, the transformed program would not type check). Informally, the compiler must reestablish the proof that the fact trivially provided by the original definition still holds. In our system, OSR is modified to construct a new proof for the fact trivially implied by the original pointer definition by

:= 0 i_1 : int $q_{11}:{\tt pf}_{(i_1=0)}$ $:= pffact(i_1)$ uB : int := len(a):= pffact(uB) $q_3: pf_{(uB=len(a))}$:=base(a) $aBase: ptr_? (int)$ $q_7: pf_{(aBase=a@0)}$:=pffact(aBase) $addr_1: ptr_? (int)$ $:= aBase + i_1$ $q_{14}: \mathtt{pf}_{(addr_1=aBase+i_1)}$ $:= pffact(addr_1)$ LOOP: i_2 : int $:= \phi(i_1, i_3)$ $q_4:\mathtt{pf}_{(0\leq i_2)}$ $:=\phi(q_{11},q_{13})$ $addr_2: \mathtt{ptr}_2(\mathtt{int})$ $:= \phi(addr_1, addr_3)$ $q_8: pf_{(addr_2=aBase+i_2)}$ $:= \phi(q_{14}, q_{16})$ if $uB \leq i_2$ goto EXIT $[q_1: \mathtt{pf}_{(i_2 < uB)}, q_2: \ldots] :=$ $q_6: pf_{(i_2 < len(a))}$ $:= \texttt{pfand}(q_3, q_1)$ $:=\!\texttt{pfand}(q_7,q_8)$ $q_9: pf_{(addr_2=a@i_2)}$ $q_{10}: \texttt{pf}_{(a@0 \leq addr < a@\texttt{len}(a))}$ $:=\!\texttt{pfand}(q_4,q_6,q_9)$ val: int $:= 1d(addr_2) [q_{10}]$. . . : . . . := val $:=i_2+1$ i_3 : int $q_{12}:\mathtt{pf}_{(i_3=i_2+1)}$ $:= pffact(i_3)$ $addr_3: ptr_?\langle int \rangle$ $:= addr_2 + 1$ $:= pffact(addr_3)$ $q_{15}: pf_{(addr_3=addr_2+1)}$ $:=\!\texttt{pfand}(q_4,q_{12})$ $q_{13}: pf_{(0 \le i_3)}$ $q_{16}: \mathtt{pf}_{(\mathit{addr}_3 = \mathit{aBase} + i_3)}$ $:=\!\mathtt{pfand}(q_8,q_{12},q_{15})$ goto LOOP EXIT:

Figure 18. IR after strength reduction of element address

$uB: int$ $q_3: pf_{(uB=1en(a))}$ $aBase: ptr_? \langle int \rangle$ $q_7: pf_{(aBase=a@0)}$ $addr_1: ptr_? \langle int \rangle$ $q_{14}: pf_{(addr_1=aBase)}$ $addrUB: ptr_? \langle int \rangle$ $q_{17}: pf_{(addrUB=aBase+uB)}$ $LOOPP:$:= len(a) := pffact(uB) := pffact(aBase) := aBase $:= pffact(addr_1)$:= aBase+uB := pffact(addrUB)
$ \begin{array}{l} addr_2: \mathtt{ptr}_? \langle \mathtt{int} \rangle \\ q_4: \mathtt{pf}_{(aBase \leq addr_2)} \\ [q_1: \mathtt{pf}_{(addr_2 < addrUB)}, q_2: \ldots] := \end{array} $	$ \begin{array}{l} := \phi(addr_1, addr_3) \\ := \phi(q_{14}, q_{13}) \\ \texttt{if} \ addrUB \leq addr_2 \\ \texttt{goto} \ EXIT \end{array} $
$\begin{array}{l} q_6: \texttt{pf}_{(addr_2 < aBase + \texttt{len}(a))} \\ q_{10}: \texttt{pf}_{(a@0 \leq addr_2 < a@\texttt{len}(a))} \\ val: \texttt{int} \\ \dots: \dots \\ addr_3: \texttt{ptr}_7 \langle \texttt{int} \rangle \end{array}$	$\begin{split} &:= \texttt{pfand}(q_3, q_1, q_{17}) \\ &:= \texttt{pfand}(q_4, q_6, q_7) \\ &:= \texttt{ld}(addr_2) \; [q_{10}] \\ &:= val \\ &:= addr_2 + 1 \end{split}$
$\begin{array}{l} q_{15}: \texttt{pf}_{(addr_3 = addr_2 + 1)} \\ q_{13}: \texttt{pf}_{(aBase \leq addr_3)} \\ EXIT: \end{array}$	$\begin{array}{l} := \texttt{pffact}(addr_3) \\ := \texttt{pfand}(q_4, q_{15}) \\ \texttt{goto} \ LOOP \end{array}$
	•••

Figure 19. IR after linear function test replacement

induction on that fact. Again, we leverage SSA to establish the new proof. In this case, $q_8 : pf_{(addr_2=aBase+i_2)}$ is defined by the phi instruction that merges proof variables $q_{14} : pf_{(addr_1=aBase+i_1)}$ and $q_{16} : pf_{(addr_3=aBase+i_3)}$.

Finally, we illustrate linear function test replacement (LFTR) [9] in Figure 19.¹ Classical LFTR replaces the test $uB \le i_2$ in the branch by a new test $addrUB \leq addr_2$. If our program contained no proof variables, this would allow the otherwise unused base variable *i* to be removed from the loop. We augment the usual LFTR procedure, which rewrites occurrences of the base induction variable i_2 in loop exit tests (and exits) in terms of the derived induction variable $addr_2$, to also rewrite occurrences of i_2 in the types of proof variables. Finally, to eliminate the original induction variable altogether, the compiler must replace the inductive proofs on the original variable (expressed through ϕ instructions) with proofs in terms of the derived induction variable. In this case, the compiler must replace the proof that $0 \le i_2$ (established by q_{11} and q_{12}) with one that proves $aBase \leq addr_2$ (established by q_{14} and q_{15}). After the replacement, the loop induction variable i and any proof variables that depend upon it are no longer live in the loop, so all definitions of the variable can be removed. The compiler must remove the proof variables whose types reduce to tautologies and apply further CSE to yield Figure 19.

5. Extensions

Our core language can easily be extended to handle other interesting aspects of Java and CLI. In this section we describe several of these extensions.

Firstly, we can handle object-model lowering through the use of our singleton types. Consider an invoke virtual operation. It is typically lowered into three operations: load the virtual dispatch table (vtable), load the method pointer from the vtable, call the method pointer passing the object as an additional argument. In our system, these operations would look like this:

$$\begin{array}{l} x: SomeClass := \cdots \\ t_1: \texttt{vtable}(x) := \texttt{vtable}(x) \\ t_2: (\texttt{S}(x), \texttt{int}) \rightarrow \texttt{int} := \texttt{method}(foo: (\texttt{int}) \rightarrow \texttt{int}, t_1) \\ t_3: \texttt{int} := \texttt{call}(t_2)(x, 10) \end{array}$$

Here the method foo (taking an integer and returning an integer) is being invoked on variable x. In the lowered code, variable t_1 gets the dependent type vtable(x) meaning that it contains the vtable from the object currently in x. Variable t_2 gets the loaded method pointer. From the type vtable(x), the typing rules can determine a precise function type for this method pointer, namely $(S(x), int) \rightarrow int$, where the first argument must be x. The actual call is the last operation, and here we pass x as an explicit argument. Since x has type S(x), this operation type checks.

By using singleton types based on term variables, we achieve a relatively simple type system and still avoid the well known typing problems with the explicit "this" argument (see [12] and references). The existing solutions to this typing problem have much more complicated type systems, with one exception. Chen and Tarditi [7] have a similarly simple type system for a lowered IR for class-based object-oriented languages. Like our system, theirs also has class names as types, and keeps around information about the class hierarchy, fields, and methods. They also have existentials with subclass bounds (type variables can be bounded above by a class, and range over any subclass of that class). They use these existentials to express the unknown runtime type of any given object, and thus the type of the explicit "this" argument. They also have a class representation function that maps class names to a record type for objects in the class, and they have coercions to convert between the two. These ideas could be adapted to our system instead of our vtable types, and our vtable types could be adapted to their type system. In summary, both systems are simpler than existing, more foundational, object encodings. Theirs has type variables and bounded existentials, ours has singleton types based on term variables.

Java and CLI also allow null as a value in any class type, and at runtime this null value must be checked and an exception thrown before any invocation or field access on an object. We can use our proof variable technique to track and ensure that these null checks are done. We simply add a null constant to the fact expression language. We can add an operation like $p : pf_{(x \neq null)} := chknull(x)$ to check that x is not null. If x is null then it throws an exception, if not then it assigns a proof of $x \neq null$ to p. Similarly to arraybounds check elimination, we can eliminate redundant null checks.

To handle exceptions we simply add explicit control flow for them. Each potentially exception throwing operation will end a basic block and there will be edges coming out of the block corresponding to exceptions that go to blocks corresponding to the exception handlers. An important point is that exceptions typically occur before the assignment of the potentially exception throwing operation, so like the conditional branches of our core language, we must treat the definition point as occuring on the fallthrough edge rather than at the end of the basic block. So in both $x : \tau := \text{chknull}(y)$ and $x : \tau := \text{call}(y)(\overline{y})$, the variable x is assigned on the fall-through edge.

We can easily deal with stores to pointers by adding a store operation of the form $\mathtt{st}(x, y)$ [p] where x holds the pointer, y the value to store, and p a proof that x is valid. The type rule for this operation is:

$$\begin{split} & \Gamma \vdash \Gamma(x) \leq \mathtt{ptr}_?\langle \tau \rangle \quad \Gamma \vdash \Gamma(y) \leq \tau \\ & \Gamma \vdash \Gamma(p) \leq \mathtt{pf}_{(z@0 \leq x \land x < z@\mathtt{len}(z))} \\ & \Gamma \vdash_P \mathtt{st}(x,y) \; [p] \end{split}$$

Modifying our formalisation and type soundness proof to accomodate stores would be straightforward.

Java and CLI have mutable covariant arrays, and thus require array-store checks at runtime. In particular, when storing into an array, the runtime must check that the object being stored is compatible with the runtime element type of the array (which could be a subtype of the static element type). In our implementation we use types of the form elem(x) to stand for the runtime element type of array x. The load base operation on x actually returns something of type $ptr_{\gamma}(elem(x))$. The array-store check produces a proof value that can be used to prove that some other variable has type elem(x)and we have a coercion to use the proof value to change the variable's type. The end of a lowered array store would look something like this:

$$\begin{array}{l} x: \mathtt{array}(C) := \cdots \\ y: C := \cdots \\ \cdots \\ p_1: \mathtt{pf}_{(x \neq \mathtt{null} \land x @ 0 \leq t \land t < x @ \mathtt{len}(x))} := \cdots \\ p_2: \mathtt{pf}_{(y: \mathtt{elem}(x))} := \mathtt{chkst}(x, y) \\ \mathtt{st}(t, \mathtt{retype}(y, p_2)) \ [p_1] \end{array}$$

One technicality is worth noting. In order to avoid circularities between the type system and the fact language, and to avoid making the fact language's decision procedure mutually dependent upon the subtype checker, we restrict the types that can appear in a fact of the form $x : \tau$ to those that do not mention proof types.

Downcasts are similar to store checks, and we can treat them in a similar way. A chkcast(x : C) operation checks that x is in type C and returns a proof of this fact, otherwise it throws an exception. The actual subtype checks performed at runtime in our implementa-

¹Note that the code resulting from LFTR is not typable in our core language, since we do not allow conditional branches on pointers. Extending the language to handle this is straightforward, but requires a total ordering on pointer values which essentially requires moving to a heap-based semantics. Note though that the fact language does permit *reasoning* about pointer comparison, as used in the previous examples.

tion are generally done by the virtual machine itself, and the virtual machine is not type checked by the type system of our JIT. However, we do partially inline this operation to include some common fast cases, and to expose some parts to redundant elimination and CSE. For example, if a object is null then it is in any reference type and can be stored into any reference array or downcast to any reference type. Another example is comparing the vtable of an object against the vtable of a specific class, if these are equal then that object is in that class. Such comparisons produce facts in our system of the form x=null or vtable(x)=vtable(C). We can simply add axioms to our fact language like $\vdash x=$ null $\implies x: C$ or \vdash vtable(x)=vtable(C) $\implies x: C$.

6. Implementation Status

The current implementation of the STARJIT compiler generates and maintains proof variables throughout its compilation process to enable safe implementation of certain optimizations in the presence of check elimination (to be described in a forthcoming paper). For their initially designed role in optimizations, proof variables did not require proof types: optimizations do not need to know the reason an optimization was safe, but only its safety dependences. As such, the current STARJIT representation is similar to that described in Section 2 with some of the extensions in Section 5.

STARJIT implements all of the optimizations discussed in this paper as well as more described in [1]. We modified each optimization, if necessary, to correctly handle proof variables. Array bounds check elimination and operator strength reduction required the most significant modification, as described in Section 4. For partial inlining of virtual machine type checking functions, as described in Section 5, we updated the definition of proof variables to established that a variable has the checked type. We also modified method inlining to properly establish the type of inlined methods. For each parameter of a method, we added a proof variable that established that it had the correct type. When a method is compiled independently, that proof variable is trivially defined at the method entry (as parameter types to a method are guaranteed by the runtime environment). When the method is inlined, the corresponding proof variables must be defined by the calling method instead. As method call operations require proof variables for each parameter in our system, this information is readily available. Most optimizations, however, did not require significant changes for the reasons outlined in this paper.

An early version of a type verifier which inferred proof types itself was implemented. This implementation was particularly helpful in finding bugs within STARJIT, but was insufficient for complete verification of optimized code. In particular, the inference algorithm was insufficient for some more complicated optimization situations, such as the LFTR example (without proof type information) in Section 4. We are confident that extending the compiler to use precise proof types for proof variables will be straightforward, using the framework developed in this paper.

7. Related Work

As far as we are aware, SafeTSA [25, 2] is the only other example of a type-safe SSA representation in the literature. The motivation of their work is rather different than ours. SafeTSA was designed as an alternative to Java bytecode, whereas our representation is designed to be a low-level intermediate language for a bytecode compiler. SafeTSA can represent certain optimizations, such as CSE and limited check elimination, that Java bytecode does not. However, in our classification in Section 2, SafeTSA is a refinementstyle representation and, thus, cannot represent the effect of many of the low-level optimizations we discuss here. For example, it cannot represent the safety of check elimination based upon a previous branch or the construction of an unsafe memory address as illustrated in Figure 7. On the other hand, we do not support their notion of referential security: the property that a program must be safe by construction.

While most of the work on certified code focuses on the final machine code representation, there has been previous work on intermediate representations that allow verification of the memory safety of highly optimized machine level code. One of the major differences between the various approaches lies in the degree to which safety information is made explicit.

On the side of less explicit information are the SpecialJ compiler [8] and DTAL [27]. Both approaches record loop invariants, but not explicit safety dependences. This makes verification harder (all available invariants must be considered by the decision procedure), interferes with more optimizations (such as loop peeling) than our approach, and makes removing dead invariants much more difficult (because invariants never have explicit uses).

At the other end of the spectrum, there are other systems that not only represent dependences explicitly as we do, but also record exactly why the dependences imply safety for each instruction, using proofs, instead of relying on a decision procedure during checking, as in our system. The LTT system of Crary and Vanderwaart [10] and the TSCB system of Shao et al. [23], developed independently, both take this approach, albeit in the setting of a functional or mostly-functional language. Both systems are designed around the idea of incorporating a logic into a type theory, in order to combine the benefits of proof-carrying code [20] with the convenience of a type system. LTT and TSCB adopt the linear logical framework LLF and the Calculus of Inductive Constructions, respectively, as their proof languages. Incorporating a proof system also gives them more flexibility, as they can express a variety of properties within a single framework.

The lack of explicit proofs in the representation forces us to use a decision procedure during typechecking. This limits us to decidable properties, and may be less suited for certified code applications where the added complexity of a decision procedure in the verifier may be undesirable.

On the other hand, a system such as ours is much more suited to use in the internals of an optimizing compiler. For the limited use that we need proofs for—to verify the correctness of checks which are eliminated by a real optimizing compiler—we can get away with a vastly simpler system, one that imposes much less of a burden on the compiler than more syntactically heavy systems. Moreover, for applications of certified code, we believe that it should be possible to take optimized intermediate code in the style presented here and translate it, as part of code generation, to a more explicit form in the style of LTT or TSCB, thereby reaping the benefits of both approaches, perhaps by following the Special J model of using a proof generating theorem prover. However, this remains future work.

Finally, our proof variables are also similar to the Jalapeño Java system's condition registers as described in [6, 14]. Both are mechanisms to represent control-flow information as abstract value dependences. Their usage, however, is more limited. Condition registers are not used to express general safety information or to support verification of code. Instead, they are used by the compiler to model control flow between a check operation and all (rather than just potentially unsafe) instructions that follow it. Jalapeño uses condition registers to collapse control flow due to exceptions into a single extended block and, in that block, to prevent instruction reordering that would violate control flow dependences.

8. Conclusions

This paper has shown a typed low-level program representation that preserves memory safety dependences in highly-optimizing type-preserving compilers. Our representation encodes safety dependences as first-class term-level proof variables that capture the essential memory-safety dependences in the program without artificially constraining optimizations-previous approaches that piggyback safety dependence on top of value dependence inhibit optimization opportunities. Our representation encodes proofs of memory safety as dependent types associated with proof variables. Experience implementing this representation in the STARJIT compiler has demonstrated that a highly-optimizing Java JIT compiler can easily generate and maintain this representation in the presence of aggressive SSA-based optimizations such as bounds check elimination, value numbering, strength reduction, linear function test replacement, and others. Using explicit proof values and proof types, modern optimizing compilers for type-safe languages can now generate provably safe yet low-level intermediate representations without constraining optimizations.

References

- [1] ADL-TABATABAI, A.-R., BHARADWAJ, J., CHEN, D.-Y., GHU-LOUM, A., MENON, V. S., MURPHY, B. R., SERRANO, M., AND SHPEISMAN, T. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal 7*, 1 (February 2003).
- [2] AMME, W., DALTON, N., VON RONNE, J., AND FRANZ, M. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of* the ACM SIGPLAN 2001 conference on Programming language design and implementation (Snowbird, UT, USA, 2001), pp. 137–147.
- [3] BILARDI, G., AND PINGALI, K. Algorithms for computing the static single assignment form. J. ACM 50, 3 (2003), 375–425.
- [4] BODÍK, R., GUPTA, R., AND SARKAR, V. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (Vancouver, British Columbia, Canada, 2000), pp. 321–333.
- [5] BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. Value numbering. Software—Practice and Experience 27, 6 (June 1996), 701–724.
- [6] CHAMBERS, C., PECHTCHANSKI, I., SARKAR, V., SERRANO, M. J., AND SRINIVASAN, H. Dependence analysis for Java. In Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (1999), vol. 1863 of Lecture Notes in Computer Science, pp. 35–52.
- [7] CHEN, J., AND TARDITI, D. A simple typed intermediate language for object-oriented languages. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Programming Languages* (Long Beach, CA, USA, Jan. 2005), ACM Press, pp. 38–49.
- [8] COLBY, C., LEE, P., NECULA, G. C., BLAU, F., PLESKO, M., AND CLINE, K. A certifying compiler for Java. In *PLDI '00: Proceedings* of the ACM SIGPLAN 2000 conference on Programming language design and implementation (New York, NY, USA, 2000), ACM Press, pp. 95–107.
- [9] COOPER, K. D., SIMPSON, L. T., AND VICK, C. A. Operator strength reduction. ACM Transactions on Programming Languages and Systems (TOPLAS) 23, 5 (September 2001), 603–625.
- [10] CRARY, K., AND VANDERWAART, J. An expressive, scalable type theory for certified code. In ACM SIGPLAN International Conference on Functional Programming (Pittsburgh, PA, 2002), pp. 191–205.
- [11] CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, K. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages* (Austin, TX, Jan. 1989).
- [12] GLEW, N. An efficient class and object encoding. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages (Minneapolis, MN, USA, Oct. 2000), ACM Press, pp. 311–324.

- [13] GROSSMAN, D., AND MORRISETT, J. G. Scalable certification for typed assembly language. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation* (London, UK, 2001), Springer-Verlag, pp. 117–146.
- [14] GUPTA, M., CHOI, J.-D., AND HIND, M. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming - ECOOP '00 (Lecture Notes in Computer Science, Vol. 1850)* (June 2000), Springer-Verlag, pp. 422–446.
- [15] IGARASHI, A., PIERCE, B., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS) 23, 3 (May 2001), 396–560. First appeared in OOPSLA, 1999.
- [16] KNOOP, J., RÜTHING, O., AND STEFFEN, B. Lazy code motion. In Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation (San Francisco, CA, June 1992).
- [17] MENON, V., GLEW, N., MURPHY, B., MCCREIGHT, A., SHPEIS-MAN, T., ADL-TABATABAI, A.-R., AND PETERSEN, L. A verifiable SSA program representation for aggressive compiler optimization. Tech. Rep. YALEU/DCS/TR-1338, Department of Computer Science, Yale University, 2005.
- [18] MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. TALx86: A realistic typed assembly language. In Second ACM SIGPLAN Workshop on Compiler Support for System Software (Atlanta, Georgia, 1999), pp. 25–35. Published as INRIA Technical Report 0288, March, 1999.
- [19] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. ACM Transactions on Programming Languages and Systems (TOPLAS) 21, 3 (May 1999), 528—569.
- [20] NECULA, G. Proof-carrying code. In *POPL1997* (New York, New York, January 1997), ACM Press, pp. 106–119.
- [21] NECULA, G. C., AND LEE, P. The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (New York, NY, USA, 1998), ACM Press, pp. 333– 344.
- [22] PUGH, W. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomput*ing '91 (Albuquerque, NM, Nov. 1991).
- [23] SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. A type system for certified binaries. In *Proceedings of the 29th Annual* ACM Symposium on Principles of Programming Languages (January 2002), ACM Press, pp. 216–232.
- [24] VANDERWAART, J. C., DREYER, D. R., PETERSEN, L., CRARY, K., AND HARPER, R. Typed compilation of recursive datatypes. In Proceedings of the TLDI 2003: ACM SIGPLAN International Workshop on Types in Language Design and Implementation (New Orleans, LA, January 2003), pp. 98–108.
- [25] VON RONNE, J., FRANZ, M., DALTON, N., AND AMME, W. Compile time elimination of null- and bounds-checks. In 3rd Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3) (December 2000).
- [26] WALKER, D., CRARY, K., AND MORISETT, G. Typed memory management via static capabilities. ACM Transactions on Programming Languages and Systems (TOPLAS) 22, 4 (July 2000), 701–771.
- [27] XI, H., AND HARPER, R. Dependently typed assembly language. In *International Conference on Functional Programming* (September 2001), pp. 169–180.

Object-Oriented Type Inference

Jens Palsberg and Michael I. Schwartzbach

palsberg@daimi.aau.dk and mis@daimi.aau.dk

Computer Science Department, Aarhus University Ny Munkegade, DK-8000 Århus C, Denmark

Abstract

We present a new approach to inferring types in untyped object-oriented programs with inheritance, assignments, and late binding. It guarantees that all messages are understood, annotates the program with type information, allows polymorphic methods, and can be used as the basis of an optimizing compiler. Types are finite sets of classes and subtyping is set inclusion. Using a *trace graph*, our algorithm constructs a set of conditional type constraints and computes the least solution by least fixed-point derivation.

1 Introduction

Untyped object-oriented languages with assignments and late binding allow rapid prototyping because classes inherit implementation and not specification. Late binding, however, can cause programs to be unreliable, unreadable, and inefficient [27]. Type inference may help solve these problems, but so far no proposed inference algorithm has been capable of checking most common, completely untyped programs [9].

We present a new type inference algorithm for a basic object-oriented language with inheritance, assignments, and late binding.

In Proc. ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) October 6-11, 1991, pages 146–161. ©1991 ACM. Copied by permission. The algorithm guarantees that all messages are understood, annotates the program with type information, allows polymorphic methods, and can be used as the basis of an optimizing compiler. Types are finite sets of classes and subtyping is set inclusion. Given a concrete program, the algorithm constructs a finite graph of type constraints. The program is *typable* if these constraints are solvable. The algorithm then computes the least solution in worst-case exponential time. The graph contains all type information that can be derived from the program without keeping track of nil values or flow analyzing the contents of instance variables. This makes the algorithm capable of checking most common programs; in particular, it allows for polymorphic methods. The algorithm is similar to previous work on type inference [18, 14, 27, 1, 2, 19, 12, 10, 9] in using type constraints, but it differs in handling late binding by *conditional* constraints and in resolving the constraints by least fixed-point derivation rather than unification.

The example language resembles SMALLTALK [8] but avoids metaclasses, blocks, and primitive methods. Instead, it provides explicit new and if-thenelse expressions; classes like Natural can be programmed in the language itself.

In the following section we discuss the impacts of late binding on type inference and examine previous work. In later sections we briefly outline the example language, present the type inference algorithm, and show some examples of its capabilities.

2 Late Binding

Late binding means that a message send is dynamically bound to an implementation depending on the class of the receiver. This allows a form of polymorphism which is fundamental in object-oriented programming. It also, however, involves the danger that the class of the receiver does *not* implement a method for the message—the receiver may even be nil. Furthermore, late binding can make the control flow of a program hard to follow and may cause a time-consuming run-time search for an implementation.

It would significantly help an optimizing compiler if, for each message send in a program text, it could infer the following information.

- Can the receiver be nil?
- Can the receiver be an instance of a class which does not implement a method for the message?
- What are the classes of all possible non-nil receivers in any execution of the program?

Note that the available set of classes is induced by the particular program. These observations lead us to the following terminology.

Terminology:

Type: A type is a finite set of classes.

- **Induced Type:** The induced type of an expression in a concrete program is the set of classes of all possible non-nil values to which it may evaluate in any execution of that particular program.
- **Sound approximation:** A sound approximation of the induced type of an expression in a concrete program is a *superset* of the induced type.

Note that a sound approximation tells "the whole truth", but not always "nothing but the truth" about an induced type. Since induced types are generally uncomputable, a compiler must make do with sound approximations. An induced type is a *subtype* of any sound approximation; subtyping is set inclusion. Note also that our notion of type, which we also investigated in [22], differs from those usually used in theoretical studies of types in object-oriented programming [3, 7]; these theories have difficulties with late binding and assignments.

The goals of type inference can now be phrased as follows.

Goals of type inference:

- **Safety guarantee:** A guarantee that any message is sent to either nil or an instance of a class which implements a method for the message; and, given that, also
- **Type information:** A sound approximation of the induced type of any receiver.

Note that we ignore checking whether the receiver is nil; this is a standard data flow analysis problem which can be treated separately.

If a type inference is successful, then the program is *typable*; the error messageNotUnderstood will not occur. A compiler can use this to avoid inserting some checks in the code. Furthermore, if the type information of a receiver is a *singleton* set, then the compiler can do early binding of the message to the only possible method; it can even do in-line substitution. Similarly, if the type information is an *empty* set, then the receiver is known to always be nil. Finally, type information obtained about variables and arguments may be used to annotate the program for the benefit of the programmer.

SMALLTALK and other untyped object-oriented languages are traditionally implemented by interpreters. This is ideal for prototyping and exploratory development but often too inefficient and space demanding for real-time applications and embedded systems. What is needed is an optimizing compiler that can be used near the end of the programming phase, to get the required efficiency and a safety guarantee. A compiler which produces good code can be tolerated even it is slow because it will be used much less often than the usual programming environment. Our type inference algorithm can be used as the basis of such an optimizing compiler. Note, though, that both the safety guarantee and the induced types are sensitive to small changes in the program. Hence, separate compilation of classes seems impossible. Typed objectoriented languages such as SIMULA [6]/BETA [15], C++ [26], and EIFFEL [17] allow separate compilation but sacrifice flexibility. The relations between types and implementation are summarized in figure 1.

When programs are:	Their implementation is:
Untyped	Interpretation
Typable	Compilation
Typed	Separate Compilation

Figure 1: Types and implementation.

Graver and Johnson [10, 9], in their type system for SMALLTALK, take an intermediate approach between "untyped" and "typed" in requiring the programmer to specify types for instance variables whereas types of arguments are inferred. Suzuki [27], in his pioneering work on inferring types in SMALLTALK, handles late binding by assuming that each message send may invoke all methods for that message. It turned out, however, that this yields an algorithm which is not capable of checking most common programs.

Both these approaches include a notion of *method* type. Our new type inference algorithm abandons this idea and uses instead the concept of conditional constraints, derived from a finite graph. Recently, Hense [11] addressed type inference for a language O'SMALL which is almost identical to our example language. He uses a radically different technique, with type schemes and unification based on work of Rémy [24] and Wand [29]. His paper lists four programs of which his algorithm can type-check only the first three. Our algorithm can type-check all

four, in particular the fourth which is shown in figure 11 in appendix B. Hense uses record types which can be extendible and recursive. This seems to produce less precise typings than our approach, and it is not clear whether the typings would be useful in an optimizing compiler. One problem is that type schemes always correspond to either singletons or infinite sets of monotypes; our finite sets can be more precise. Hense's and ours approaches are similar in neither keeping track of nil values nor flow analyzing the contents of variables. We are currently investigating other possible relations.

Before going into the details of our type inference algorithm we first outline an example language on which to apply it.

3 The Language

Our example language resembles SMALLTALK, see figure 2.

A program is a set of classes followed by an expression whose value is the result of executing the program. A class can be defined using inheritance and contains instance variables and methods; a method is a message selector $(m_{1-} \ldots m_{n-})$ with formal parameters and an expression. The language avoids metaclasses, blocks, and primitive methods. Instead, it provides explicit new and if-then-else expressions (the latter tests if the condition is non-nil). The result of a sequence is the result of the last expression in that sequence. The expression "self class new" yields an instance of the class of self. The expression "E instanceOf ClassId" yields a run-time check for class membership. If the check fails, then the expression evaluates to nil.

The SMALLTALK system is based on some primitive methods, written in assembly language. This dependency on primitives is not necessary, at least not in this theoretical study, because classes such as True, False, Natural, and List can be programmed in the language itself, as shown in appendix A.

(Program)	P ::=	$C_1 \ldots C_n E$
(Class)	C ::=	class ClassId [inherits ClassId] var $Id_1 \dots Id_k M_1 \dots M_n$ end ClassId
(Method)	M ::=	method $m_1 \operatorname{Id}_1 \dots m_n \operatorname{Id}_n E$
(Expression)	E ::=	$\begin{split} \mathrm{Id} &:= \mathrm{E} \mid \mathrm{E} \ m_1 \ \mathrm{E}_1 \ \ldots \ m_n \ \mathrm{E}_n \mid \mathrm{E} \ ; \ \mathrm{E} \mid if \ \mathrm{E} \ then \ \mathrm{E} \ else \ \mathrm{E} \mid \\ \mathrm{ClassId} \ new \mid self \ class \ new \mid \mathrm{E} \ instanceOf \ \mathrm{ClassId} \mid \\ self \mid super \mid \mathrm{Id} \mid nil \end{split}$

Figure 2: Syntax of the example language.

4 Type Inference

Our type inference algorithm is based on three fundamental observations.

Observations:

- **Inheritance:** Classes inherit implementation and not specification.
- **Classes:** There are finitely many classes in a program.

Message sends: There are finitely many syntactic message sends in a program.

The first observation leads to separate type inference for a class and its subclasses. Notionally, this is achieved by expanding all classes before doing type inference. This expansion means removing all inheritance by

- Copying the text of a class to its subclasses
- Replacing each message send to super by a message send to a renamed version of the inherited method
- Replacing each "self class new" expression by a "ClassId new" expression where ClassId is the enclosing class in the expanded program.

This idea of expansion is inspired by Graver and Johnson [10, 9]; note that the size of the expanded

program is at most quadratic in the size of the original.

The second and third observation lead to a finite representation of type information about all executions of the expanded program; this representation is called the *trace graph*. From this graph a finite set of type constraints will be generated. Typability of the program is then solvability of these constraints. Appendix B contains seven example programs which illustrate different aspects of the type inference algorithm, see the overview in figure 3. The program texts are listed together with the corresponding constraints and their least solution, if it exists. Hense's program in figure 11 is the one he gives as a typical example of what he cannot type-check [11]. We invite the reader to consult the appendix while reading this section.

A trace graph contains three kinds of type information.

Three kinds of type information:

- **Local constraints:** Generated from method bodies; contained in nodes.
- **Connecting constraints:** Reflect message sends; attached to edges.
- **Conditions:** Discriminate receivers; attached to edges.

Example program in:	Illustrates:	Can we type it?
Figure 10	Basic type inference	Yes
Figure 11	Hense's program	Yes
Figure 12	A polymorphic method	Yes
Figure 13	A recursive method	Yes
Figure 14	Lack of flow analysis	No
Figure 15	Lack of nil detection	No
Figure 16	A realistic program	Yes

Figure 3: An overview of the example programs.

4.1 Trace Graph Nodes

The nodes of the trace graph are obtained from the various methods implemented in the program. Each method yields a number of different nodes: one for each syntactic message send with the corresponding selector. The situation is illustrated in figure 4, where we see the nodes for a method m that is implemented in each of the classes C_1, C_2, \ldots, C_n . Thus, the number of nodes in the trace graph will at most be quadratic in the size of the program. There is also a single node for the main expression of the program, which we may think of as a special method without parameters.

Methods do not have types, but they can be provided with type annotations, based on the types of their formal parameters and result. A particular method implementation may be represented by several nodes in the trace graph. This enables it to be assigned several different type annotations—one for each syntactic call. This allows us effectively to obtain method polymorphism through a finite set of method "monotypes".

4.2 Local Constraints

Each node contains a collection of *local constraints* that the types of expressions must satisfy. For each syntactic occurrence of an expression E in the implementation of the method, we regard its type as

an unknown variable [E]]. Exact type information is, of course, uncomputable. In our approach, we will ignore the following two aspects of program executions.

Approximations:

Nil values: It does not keep track of nil values.

Instance variables: It does not flow analyze the contents of instance variables.

The first approximation stems from our discussion of the goals of type inference; the second corresponds to viewing an instance variable as having a single possibly large type, thus leading us to identify the type variables of different occurrences of the same instance variable. In figures 14 and 15 we present two program fragments that are typical for what we cannot type because of these approximations. In both cases the constraints demand the false inclusion {True} \subseteq {Natural}. Suzuki [27] and Hense [11] make the same approximations.

For an expression E, the local constraints are generated from all the phrases in its derivation, according to the rules in figure 5. The idea of generating constraints on type variables from the program syntax is also exploited in [28, 25].

The constraints guarantee safety; only in the cases 4) and 8) do the approximations manifest themselves. Notice that the constraints can all be ex-



Figure 4: Trace graph nodes.

pressed as inequalities of one of the three forms: "constant \subseteq variable", "variable \subseteq constant", or "variable \subseteq variable"; this will be exploited later.

Each different node employs unique type variables, except that the types of instance variables are common to all nodes corresponding to methods implemented in the same class. A similar idea is used by Graver and Johnson [10, 9].

4.3 Trace Graph Edges

The *edges* of the trace graph will reflect the possible connections between a message send and a method that may implement it. The situation is illustrated in figure 6.

If a node corresponds to a method which contains a message send of the form X m: A, then we have an edge from that sender node to any other receiver node which corresponds to an implementation of a method m. We label this edge with the condition that the message send may be executed, namely $C \in [X]$ where C is the class in which the particular

method m is implemented. With the edge we associate the *connecting constraints*, which reflect the relationship between formal and actual parameters and results. This situation generalizes trivially to methods with several parameters. Note that the number of edges is again quadratic in the size of the program.

4.4 Global Constraints

To obtain the *global constraints* for the entire program we combine local and connecting constraints in the manner illustrated in figure 7. This produces *conditional constraints*, where the inequalities need only hold if all the conditions hold. The global constraints are simply the union of the conditional constraints generated by all paths in the graph, originating in the node corresponding to the main expression of the program. This is a finite set, because the graph is finite; as shown later in this section, the size of the constraint set may in (extreme) worst-cases become exponential.

If the set of global constraints has a solution, then

	Expression:	Constraint:
1)	Id := E	$\llbracket \mathrm{Id} \rrbracket \supseteq \llbracket \mathrm{E} \rrbracket \land \llbracket \mathrm{Id} := \mathrm{E} \rrbracket = \llbracket \mathrm{E} \rrbracket$
2)	$\mathbf{E} \mathbf{m}_1 \mathbf{E}_1 \dots \mathbf{m}_n \mathbf{E}_n$	$\llbracket \mathbb{E} \rrbracket \subseteq \{ C \mid C \text{ implements } m_1 \dots m_n \}$
3)	E_1 ; E_2	$\llbracket \mathrm{E}_1 \ ; \ \mathrm{E}_2 rbracket = \llbracket \mathrm{E}_2 rbracket$
4)	if E_1 then E_2 else E_3	$\llbracket if \ \mathrm{E}_1 \ then \ \mathrm{E}_2 \ else \ \mathrm{E}_3 rbracket \supseteq \llbracket \mathrm{E}_2 rbracket \cup \llbracket \mathrm{E}_3 rbracket$
5)	C new	$\llbracket C \text{ new} \rrbracket = \{C\}$
6)	${ m E}$ instanceOf C	$\llbracket E \text{ instanceOf } C \rrbracket = \{C\}$
7)	self	$[self] = \{the enclosing class\}$
8)	Id	$\llbracket \mathrm{Id} \rrbracket = \llbracket \mathrm{Id} \rrbracket$
9)	nil	$\llbracket nil \rrbracket = \{ \}$

Figure 5: The local constraints.



Figure 6: Trace graph edges.

this provides approximate information about the dynamic behavior of the program.

Consider any execution of the program. While observing this, we can trace the pattern of method executions in the trace graph. Let E be some expression that is evaluated at some point, let VAL(E) be its value, and let CLASS(b) be the class of an object b. If L is some solution to the global constraints, then the following result holds.

Soundness Theorem:

If $VAL(E) \neq nil$ then $CLASS(VAL(E)) \in L(\llbracket E \rrbracket)$

It is quite easy to see that this must be true. We sketch a proof by induction in the number of message sends performed during the trace. If this is zero, then we rely on the local constraints alone; given a dynamic semantics [4, 5, 23, 13] one can easily verify that their satisfaction implies the above property. If we extend a trace with a message send X m: A implemented by a method in a class C, then we can inductively assume that $C \in L([X]]$). But this implies that the local constraints in the node corresponding to the invoked method must hold, since all their conditions now hold and L is a solution. Since the relationship between actual and formal parameters and results is soundly represented by the connecting constraints, which also must hold, the result follows.

Note that an expression E occurring in a method that appears k times in the trace graph has ktype variables $[\![\mathbf{E}]\!]_1, [\![\mathbf{E}]\!]_2, \ldots, [\![\mathbf{E}]\!]_k$ in the global constraints. A sound approximation to the induced



Figure 7: Conditional constraints from a path.

type of E is obtained as

$$\bigcup_{i} L(\llbracket \mathbf{E} \rrbracket_{i})$$

Appendix C gives an efficient algorithm to compute the smallest solution of the extracted constraints, or to decide that none exists. The algorithm is at worst quadratic in the size of the constraint set.

The complete type inference algorithm is summarized in figure 8.

4.5 Type Annotations

Finally, we will consider how a solution L of the type constraints can produce a *type annotation* of the program. Such annotations could be provided for the benefit of the programmer.

An instance variable x has only a single associated type variable. The type annotation is simply $L(\llbracket x \rrbracket)$. The programmer then knows an upper bound of the set of classes whose instances may reside in x.

A method has finitely many type annotations, each of which is obtained from a corresponding node in the trace graph. If the method, implemented in the class C, is

Input: A program in the example language.

- **Output:** Either: a safety guarantee and type information about all expressions; or: "unable to type the program".
- 1) Expand all classes.
- 2) Construct the trace graph of the expanded program.
- 3) Extract a set of type constraints from the trace graph.
- 4) Compute the least solution of the set of type constraints. If such a solution exists, then output it as the wanted type information, together with a safety guarantee; otherwise, output "unable to type the program".

Figure 8: Summary of the type inference algorithm.

then each type annotation is of the form

$$\{\mathsf{C}\} \times L(\llbracket\mathsf{F}_1\rrbracket) \times \cdots \times L(\llbracket\mathsf{F}_n\rrbracket) \to L(\llbracket\mathsf{E}\rrbracket)$$

The programmer then knows the various manners in which this method may be used.

A constraint solution contains more type informa-

tion about methods than the method types used by Suzuki. Consider for example the polymorphic identity function in figure 12. Our technique yields both of the method type annotations

$$\begin{array}{l} \mathsf{id} : \ \{\mathsf{C}\} \times \{\mathsf{True}\} \rightarrow \{\mathsf{True}\}\\ \mathsf{id} : \ \{\mathsf{C}\} \times \{\mathsf{Natural}\} \rightarrow \{\mathsf{Natural}\} \end{array}$$

whereas the method type using Suzuki's framework is

$$\mathsf{id}: \{\mathsf{C}\} \times \{\mathsf{True}, \mathsf{Natural}\} \to \{\mathsf{True}, \mathsf{Natural}\}$$

which would allow neither the succ nor the isTrue message send, and, hence, would lead to rejection of the program.

4.6 An Exponential Worst-Case

The examples in appendix B show several cases where the constraint set is quite small, in fact linear in the size of the program. While this will often be the situation, the theoretical worst-case allows the constraint set to become exponential in the size of the program. The running time of the inference algorithm depends primarily on the topology of the trace graph.

In figure 9 is shown a program and a sketch of its trace graph. The induced constraint set will be exponential since the graph has exponentially many different paths. Among the constraints will be a family whose conditions are similar to the words of the regular language

$$(CCC + DCC)^{\frac{n}{3}}$$

the size of which is clearly exponential in n.

Note that this situation is similar to that of type inference in ML, which is also worst-case exponential but very useful in practice. The above scenario is in fact not unlike the one presented in [16] to illustrate exponential running times in ML. Another similarity is that both algorithms generate a potentially exponential constraint set that is always solved in polynomial time.



Figure 9: A worst-case program.

5 Conclusion

Our type inference algorithm is sound and can handle most common programs. It is also conceptually simple: a set of uniform type constraints is constructed and solved by fixed-point derivation. It can be further improved by an orthogonal effort in data flow analysis.

The underlying type system is simple: types are finite sets of classes and subtyping is set inclusion.

An implementation of the type inference algorithm is currently being undertaken. Future work includes extending this into an optimizing compiler. The inference algorithm should be easy to modify to work for full SMALLTALK, because metaclasses are simply classes, blocks can be treated as objects with a single method, and primitive methods can be handled by stating the constraints that the machine code must satisfy. Another challenge is to extend the algorithm to produce type annotations together with type substitution, see [20, 21, 22].

Appendix A: Basic classes

```
class Object
end Object
class True
   method isTrue
      Object new
end True
class False
   method isTrue
      nil
end False
Henceforth, we abbreviate "True new" as "true",
and "False new" as "false".
class Natural
   var rep
   method isZero
      if rep then false else true
   method succ
      (Natural new) update: self
   method update: x
      rep := x; self
   method pred
      if (self isZero) isTrue then self else rep
   method less: i
      if (i isZero) isTrue
      then false
      else if (self isZero) isTrue then true
      else (self pred) less: (i pred)
end Natural
Henceforth, we abbreviate "Natural new" as "0",
and, recursively, "n succ" as "n + 1".
```

class List var head, tail method setHead: h setTail: t head := h: tail := t method cons: x (self class new) setHead: x setTail: self method isEmpty if head then false else true method car head method cdr tail method append: aList if (self isEmpty) isTrue then aList else (tail append: aList) cons: head method insert: x if (self isEmpty) isTrue then self cons: xelse if (head less: x) isTrue then self cons: xelse (tail insert: x) cons: head method sort if (self isEmpty) isTrue then self else (tail sort) insert: head method merge: aList if (self isEmpty) isTrue then aList else if (head less: (aList car)) isTrue then (tail merge: aList) cons: head else (self merge: (aList cdr)) cons: (aList car) end List

class Comparable var key method getKey key method setKey: k key := k method less: c key less: (c getKey) end Comparable

Appendix B: Example Programs

class A method f 7 end A class B method f true $\mathbf{end}\ \mathsf{B}$ x := A new; (x f) succ **Constraints:** $\llbracket A \text{ new} \rrbracket = \{A\}$ $[x] \supseteq [A \text{ new}]$ $[\![\mathsf{x} := \mathsf{A} \ \mathbf{new}]\!] = [\![\mathsf{A} \ \mathbf{new}]\!]$ $\llbracket x \rrbracket \subseteq \{A, B\}$ $\mathsf{A} \in [\![\mathsf{x}]\!] \Rightarrow [\![\mathsf{x}\ \mathsf{f}]\!] = [\![\mathsf{7}]\!]$ $\mathsf{A} \in \llbracket \mathsf{x} \rrbracket \Rightarrow \llbracket \mathsf{7} \rrbracket = \{\mathsf{Natural}\}$ $\mathsf{B} \in \llbracket \mathsf{x} \rrbracket \Rightarrow \llbracket \mathsf{x} \ \mathsf{f} \rrbracket = \llbracket \mathsf{true} \rrbracket$ $\mathsf{B} \in \llbracket \mathsf{x} \rrbracket \Rightarrow \llbracket \mathsf{true} \rrbracket = \{\mathsf{True}\}$ $\mathbf{x} f \subseteq \{ \text{Natural} \}$ $\mathsf{Natural} \in \llbracket \mathsf{x} \ \mathsf{f} \rrbracket \Rightarrow \llbracket (\mathsf{x} \ \mathsf{f}) \ \mathsf{succ} \rrbracket = \{\mathsf{Natural}\}$ [x := A new; (x f) succ] = [(x f) succ]**Smallest Solution:** $[x] = [A \text{ new}] = [x := A \text{ new}] = \{A\}$ [x f] = [(x f) succ] = $[x := A \text{ new}; (x f) \text{ succ}] = [7] = \{\text{Natural}\}$ $[true] = {True}$ Trace graph sketch: f_A f_B

Figure 10: Conditions at work.

class A $\mathbf{method}\ \mathbf{m}$ 0 end A class B inherits A method n 0 end B a := A new; b := B new; a := b;a m **Constraints:** $\llbracket A \text{ new} \rrbracket = \{A\}$ $[a] \supseteq [A new]$ $\llbracket \mathsf{B} \ \mathbf{new} \rrbracket = \{\mathsf{B}\}\$ $\llbracket b \rrbracket \supseteq \llbracket B \text{ new} \rrbracket$ $[\![a]\!]\supseteq[\![b]\!]$ $\llbracket a \rrbracket \subseteq \{A, B\}$ $\mathsf{A} \in \llbracket \mathsf{a} \rrbracket \Rightarrow \llbracket \mathsf{a} \ \mathsf{m} \rrbracket = \llbracket \mathsf{0} \rrbracket$ $\mathsf{B} \in [\![\mathsf{a}]\!] \Rightarrow [\![\mathsf{a}\ \mathsf{m}]\!] = [\![\mathsf{0}]\!]$ $\llbracket 0 \rrbracket = \{ Natural \}$ **Smallest Solution:** $[a] = \{A, B\}$ $[\![b]\!] = \{B\}$ $[a m] = {Natural}$ $[A \text{ new}] = \{A\}$ $[[B new]] = \{B\}$ Trace graph sketch: mA mB nВ

Figure 11: Hense's program.

```
class C
       method id: x
               Х
end C
((C new) id: 7) succ;
((C new) id: true) isTrue
Constraints:
  [C new]_1 = \{C\}
  \llbracket \mathsf{C} \operatorname{\mathbf{new}} \rrbracket_1 \subseteq \{\mathsf{C}\}
 \mathsf{C} \in \llbracket \mathsf{C} \ \mathbf{new} \rrbracket_1 \Rightarrow \llbracket \mathsf{7} \rrbracket = \llbracket \mathsf{x} \rrbracket_1
  \mathsf{C} \in \llbracket \mathsf{C} \operatorname{\mathbf{new}} \rrbracket_1 \Rightarrow \llbracket \mathsf{x} \rrbracket_1 = \llbracket (\mathsf{C} \operatorname{\mathbf{new}}) \text{ id: } 7 \rrbracket
  \llbracket 7 \rrbracket = \{ \mathsf{Natural} \}
  \llbracket (C \text{ new}) \text{ id}: 7 \rrbracket \subseteq \{\text{Natural}\}
  Natural \in [(C \text{ new}) \text{ id}: 7] \Rightarrow \{\text{Natural}\} = [((C \text{ new}) \text{ id}: 7) \text{ succ}]
  [[C new]]_2 = \{C\}
  \llbracket \mathsf{C} \operatorname{\mathbf{new}} \rrbracket_2 \subseteq \{\mathsf{C}\}
  \mathsf{C} \in \llbracket \mathsf{C} \ \mathbf{new} \rrbracket_2 \Rightarrow \llbracket \mathsf{true} \rrbracket = \llbracket \mathsf{x} \rrbracket_2
  \mathsf{C} \in \llbracket \mathsf{C} \ \mathbf{new} \rrbracket_2 \Rightarrow \llbracket \mathsf{x} \rrbracket_2 = \llbracket (\mathsf{C} \ \mathbf{new}) \text{ id: true} \rrbracket
  [true] = {True}
  [(C new) id: true] \subseteq {True, False}
  True \in [(C new) id: true] \Rightarrow {Object} = [((C new) id: true) isTrue]
  \mathsf{False} \in \llbracket (\mathsf{C} \ \mathbf{new}) \ \mathsf{id} \colon \mathsf{true} \rrbracket \Rightarrow \{\} = \llbracket ((\mathsf{C} \ \mathbf{new}) \ \mathsf{id} \colon \mathsf{true}) \ \mathsf{isTrue} \rrbracket
Smallest Solution:
  [[C new]]_1 = [[C new]]_2 = \{C\}
```

Trace graph sketch:



Figure 12: A polymorphic method.

```
class D
         method f: x
                  if \times then self f: \times else nil
end D
(D new) f: nil
Constraints:
   [D \text{ new}] = \{D\}
   \llbracket \mathsf{D} \operatorname{\mathbf{new}} \rrbracket \subseteq \{\mathsf{D}\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket \Rightarrow \llbracket \mathsf{nil} \rrbracket = \llbracket \mathsf{x} \rrbracket_1
  D \in [D \text{ new}] \Rightarrow [if \times then \text{ self f: } \times else \text{ nil}]_1 = [(D \text{ new}) \text{ f: nil}]
  D \in [D \text{ new}] \Rightarrow [if x \text{ then self f: } x \text{ else nil}]_1 \supseteq [self f: x]_1 \cup [nil]_1
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket \Rightarrow \llbracket \mathsf{nil} \rrbracket_1 = \{\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket \Rightarrow \llbracket \mathsf{self} \rrbracket_1 = \{\mathsf{D}\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket \Rightarrow \llbracket \mathsf{self} \rrbracket_1 \subseteq \{\mathsf{D}\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1 \Rightarrow \llbracket \mathsf{x} \rrbracket_1 = \llbracket \mathsf{x} \rrbracket_2
   \mathsf{D} \in \llbracket \mathsf{D} \text{ new} \rrbracket, \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1 \Rightarrow \llbracket \mathsf{if} \times \mathsf{then} \mathsf{ self f: } \times \mathsf{else} \mathsf{ nil} \rrbracket_2 = \llbracket \mathsf{self f: } \times \rrbracket_1
  D \in [D \text{ new}], D \in [\text{self}]_1 \Rightarrow [\text{if } x \text{ then self } f: x \text{ else } nil]_2 \supseteq [\text{self } f: x]_2 \cup [\text{nil}]_2
   \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1 \Rightarrow \llbracket \mathsf{nil} \rrbracket_2 = \{\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1 \Rightarrow \llbracket \mathsf{self} \rrbracket_2 = \{\mathsf{D}\}
   \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1 \Rightarrow \llbracket \mathsf{self} \rrbracket_2 \subseteq \{\mathsf{D}\}
  \mathsf{D} \in \llbracket \mathsf{D} \ \mathbf{new} \rrbracket, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_1, \ \mathsf{D} \in \llbracket \mathsf{self} \rrbracket_2 \Rightarrow \llbracket \mathsf{x} \rrbracket_2 = \llbracket \mathsf{x} \rrbracket_2
  D \in [D \text{ new}], D \in [\text{self}]_1, D \in [\text{self}]_2 \Rightarrow [\text{if } x \text{ then self } f: x \text{ else } nil]_2 = [\text{self } f: x]_2
  [nil] = \{\}
```

Smallest Solution:

 $\begin{bmatrix} D & \mathbf{new} \end{bmatrix} = \llbracket \mathsf{self} \rrbracket_1 = \llbracket \mathsf{self} \rrbracket_2 = \{D\} \\ \llbracket \mathsf{nil} \rrbracket = \llbracket x \rrbracket_1 = \llbracket \mathsf{nil} \rrbracket_1 = \llbracket \mathbf{if} \times \mathbf{then} \ \mathsf{self} \ \mathsf{f:} \times \mathbf{else} \ \mathsf{nil} \rrbracket_1 = \llbracket \mathsf{self} \ \mathsf{f:} \times \rrbracket_1 = \\ \llbracket (D & \mathbf{new}) \ \mathsf{f:} \ \mathsf{nil} \rrbracket_1 = \llbracket x \rrbracket_2 = \llbracket \mathsf{nil} \rrbracket_2 = \llbracket \mathbf{if} \times \mathbf{then} \ \mathsf{self} \ \mathsf{f:} \times \mathbf{else} \ \mathsf{nil} \rrbracket_2 = \\ \llbracket \mathsf{self} \ \mathsf{f:} \ x \rrbracket_2 = \llbracket (D & \mathbf{new}) \ \mathsf{f:} \ \mathsf{nil} \rrbracket_2 = \{\}$

Trace graph sketch:



Figure 13: A recursive method.

x := 7;
x succ;
x := true;
x isTrue
Constraints:
$\llbracket x \rrbracket \supseteq \llbracket 7 \rrbracket$
$\llbracket 7 \rrbracket = \{ Natural \}$
$\llbracket x \rrbracket \subseteq \{ Natural \}$
$\llbracket x \rrbracket \supseteq \llbracket true \rrbracket$
$\llbracket true \rrbracket = \{True\}$
$\llbracket x \rrbracket \subseteq \{True,False\}$
:

Figure 14: A safe program rejected.

```
(if nil then true else 7) succ

Constraints:

[[if nil then true else 7]] \subseteq {Natural}

[[if nil then true else 7]] \supseteq [[true]] \cup [[7]]

[[true]] = {True}

[[7]] = {Natural}

:
```

Figure 15: Another safe program rejected.

```
class Student inherits Comparable
...
end Student
class ComparableList inherits List
method studentCount
if (self isEmpty) isTrue
then 0
else
if (self car) instanceOf Student
then ((self cdr) studentCount) succ
else (self cdr) studentCount
end ComparableList
```

Figure 16: An example program.

Appendix C: Solving Systems of Conditional Inequalities

This appendix shows how to solve a finite system of conditional inequalities in quadratic time.

Definition C.1: A CI-system consists of

- a finite set \mathcal{A} of *atoms*.
- a finite set $\{\alpha_i\}$ of variables.
- a finite set of *conditional inequalities* of the form

$$C_1, C_2, \ldots, C_k \Rightarrow Q$$

Each C_i is a *condition* of the form $a \in \alpha_j$, where $a \in \mathcal{A}$ is an atom, and Q is an *inequality* of one of the following forms

$$\begin{array}{rccc} A & \subseteq & \alpha_i \\ \alpha_i & \subseteq & A \\ \alpha_i & \subseteq & \alpha_j \end{array}$$

where $A \subseteq \mathcal{A}$ is a set of atoms.

A solution L of the system assigns to each variable α_i a set $L(\alpha_i) \subseteq \mathcal{A}$ such that all the conditional inequalities are satisfied. \Box

In our application, \mathcal{A} models the set of classes occurring in a concrete program.

Lemma C.2: Solutions are closed under intersection. Hence, if a CI-system has solutions, then it has a unique minimal one.

Proof: Consider any conditional inequality of the form $C_1, C_2, \ldots, C_k \Rightarrow Q$, and let $\{L_i\}$ be all solutions. We shall show that $\cap_i L_i$ is a solution. If a condition $a \in \cap_i L_i(\alpha_j)$ is true, then so is all of $a \in L_i(\alpha_j)$. Hence, if all the conditions of Q are true in $\cap_i L_i$, then they are true in each L_i ; furthermore, since they are solutions, Q is also true in each L_i . Since, in general, $A_k \subseteq B_k$ implies $\cap_k A_k \subseteq \cap_k B_k$, it follows that $\cap_i L_i$ is a solution. Hence, if there are any solutions, then $\cap_i L_i$ is the unique smallest one. \Box

Definition C.3: Let C be a CI-system with atoms



Figure 17: The lattice of assignments.

 \mathcal{A} and *n* distinct variables. An *assignment* is an element of $(2^{\mathcal{A}})^n \cup \{error\}$ ordered as a lattice, see figure 17. If different from *error*, then it assigns a set of atoms to each variable. If *V* is an assignment, then $\tilde{\mathcal{C}}(V)$ is a new assignment, defined as follows. If V = error, then $\tilde{\mathcal{C}}(V) = error$. An inequality is *enabled* if all of its conditions are true under *V*. If for any enabled inequality of the form $\alpha_i \subseteq A$ we do *not* have $V(\alpha_i) \subseteq A$, then $\tilde{\mathcal{C}}(V) = error$; otherwise, $\tilde{\mathcal{C}}(V)$ is the smallest pointwise extension of *V* such that

- for every enabled inequality of the form $A \subseteq \alpha_j$ we have $A \subseteq \tilde{\mathcal{C}}(V)(\alpha_j)$.
- for every enabled inequality of the form $\alpha_i \subseteq \alpha_j$ we have $V(\alpha_i) \subseteq \tilde{\mathcal{C}}(V)(\alpha_j)$.

Clearly, $\tilde{\mathcal{C}}$ is monotonic in the above lattice. \Box

Lemma C.4: An assignment $L \neq error$ is a solution of a CI-system C iff $L = \tilde{C}(L)$. If C has no solutions, then *error* is the smallest fixed-point of \tilde{C} .

Proof: If L is a solution of C, then clearly \tilde{C} will not equal *error* and cannot extend L; hence, L is a fixed-point. Conversely, if L is a fixed-point of \tilde{C} , then all the enabled inequalities must hold. If

there are no solutions, then there can be no fixed-point below *error*. Since *error* is by definition a fixed-point, the result follows. \Box

This means that to find the smallest solution, or to decide that none exists, we need only compute the least fixed-point of \tilde{C} .

Lemma C.5: For any CI-system \mathcal{C} , the least fixedpoint of $\tilde{\mathcal{C}}$ is equal to

$$\lim_{k\to\infty}\tilde{\mathcal{C}}^k(\emptyset,\emptyset,\ldots,\emptyset)$$

<u>Proof</u>: This is a standard result about monotonic functions on complete lattices. \Box

Lemma C.6: Let n be the number of *different* conditions in a CI-system C. Then

$$\lim_{k\to\infty}\tilde{\mathcal{C}}^k(\emptyset,\emptyset,\ldots,\emptyset)=\tilde{\mathcal{C}}^{n+1}(\emptyset,\emptyset,\ldots,\emptyset)$$

Proof: When no more conditions are enabled, then the fixed-point is obtained by a single application. Once a condition is enabled in an assignment, it will remain enabled in all larger assignments. It follows that after n iterations no new conditions can be enabled; hence, the fixed-point is obtained in at most n + 1 iterations. \Box

Lemma C.7: The smallest solution to any CIsystem, or the decision that none exists, can be obtained in quadratic time.

Proof: This follows from the previous lemmas. \Box

References

- Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In Ninth Symposium on Principles of Programming Languages, pages 133–141, 1982.
- [2] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–68. Springer-Verlag (*LNCS* 173), 1984.
- [3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–522, December 1985.

- [4] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information* and Computation, 114(2):329–350, 1994. Also in Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 433–443, New Orleans, Louisiana, October 1989.
- [5] William R. Cook. A Denotational Semantics of Inheritance. PhD thesis, Brown University, 1989.
- [6] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.
- [7] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. ACM Computing Surveys, 20(1):29–72, March 1988.
- [8] Adele Goldberg and David Robson. Smalltalk-80—The Language and its Implementation. Addison-Wesley, 1983.
- [9] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In Seventeenth Symposium on Principles of Programming Languages, pages 136–150, 1990.
- [10] Justin Owen Graver. Type-Checking and Type-Inference for Object-Oriented Programming Languages. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1989. UIUCD-R-89-1539.
- [11] Andreas V. Hense. Polymorphic type inference for a simple object oriented programming language with state. Technical Report No. A 20/90, Fachbericht 14, Universität des Saarlandes, December 1990.
- [12] Ralph E. Johnson. Type-checking Smalltalk. In Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications, pages 315–321. Sigplan Notices, 21(11), November 1986.
- [13] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles* of Programming Languages, pages 80–87, 1988.
- [14] Marc A. Kaplan and Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In *Fifth Symposium on Principles of Programming Lan*guages, pages 60–75, 1978.
- [15] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.
- [16] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In Seventeenth Symposium on Principles of Programming Languages, pages 382–401, 1990.
- [17] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs, NJ, 1988.

- [18] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17:348–375, 1978.
- [19] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Twelfth Symposium on Principles of Programming Languages*, pages 7–21, 1985.
- [20] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming, pages 151–160, Ottawa, Canada, October 1990.
- [21] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming, pages 325–341. Springer-Verlag (LNCS 512), Geneva, Switzerland, July 1991.
- [22] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. *Science of Computer Programming*, 23(1):19–53, 1994.
- [23] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In Proc. ACM Conference on Lisp and Functional Programming, pages 289–297, 1988.
- [24] Didier Rémy. Typechecking records and variants in a natural extension of ML. In Sixteenth Symposium on Principles of Programming Languages, pages 77–88, 1989.
- [25] Michael I. Schwartzbach. Type inference with inequalities. In Proc. TAPSOFT'91, pages 441–455. Springer-Verlag (LNCS 493), 1991.
- [26] Bjarne Stroustrup. The C⁺⁺ Programming Language. Addison-Wesley, 1986.
- [27] Norihisa Suzuki. Inferring types in Smalltalk. In Eighth Symposium on Principles of Programming Languages, pages 187–199, 1981.
- [28] Mitchell Wand. A simple algorithm and proof for type inference. Fundamentae Informaticae, X:115–122, 1987.
- [29] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, 1989.

Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis

Jeffrey Dean, David Grove, and Craig Chambers

Department of Computer Science and Engineering University of Washington Seattle, WA 98195-2350 USA {jdean,grove,chambers}@cs.washington.edu

Abstract. Optimizing compilers for object-oriented languages apply static class analysis and other techniques to try to deduce precise information about the possible classes of the receivers of messages; if successful, dynamicallydispatched messages can be replaced with direct procedure calls and potentially further optimized through inline-expansion. By examining the complete inheritance graph of a program, which we call *class hierarchy* analysis, the compiler can improve the quality of static class information and thereby improve run-time performance. In this paper we present class hierarchy analysis and describe techniques for implementing this analysis effectively in both statically- and dynamically-typed languages and also in the presence of multi-methods. We also discuss how class hierarchy analysis can be supported in an interactive programming environment and, to some extent, in the presence of separate compilation. Finally, we assess the bottom-line performance improvement due to class hierarchy analysis alone and in combination with two other "competing" optimizations, profileguided receiver class prediction and method specialization.

1 Introduction

Object-oriented languages foster the development of reusable, extensible class libraries and frameworks [Johnson 92]. For example, the InterViews graphics framework [Linton *et al.* 89] defines a collection of interacting base classes. The base classes define a set of messages that are to be defined or overridden in subclasses. Clients of the framework specialize it to their use by providing application-specific subclasses of the framework's base classes with the appropriate operations defined. Other frameworks have a similar structure, exploiting inheritance and dynamic binding of messages to make library code customizable and malleable.

Heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable, but it also imposes a significant performance overhead, compared to an equivalent but non-extensible program written in a non-object-oriented manner. In some domains, such as structured graphics packages, the performance cost of the extra flexibility provided by using a heavily object-oriented style is acceptable. However, in other domains, such as basic data structure libraries, numerical computing packages, rendering libraries, and trace-driven simulation frameworks, the cost of message passing can be too great, forcing the programmer to avoid object-oriented programming in the "hot spots" of their application. For example, hybrid languages like C++ [Stroustrup 91], Modula-3 [Nelson 91, Harbison 92], and CLOS [Bobrow *et al.* 88, Gabriel *et al.* 91] provide non-object-oriented built-in array data structures that are more

Appeared in ECOOP'95, August 1995.

efficient than would be a typical class-based extensible implementation using dynamically-dispatched fetch and store operations, Sather [Omohundro 94, Szypersky *et al.* 93] allows the programmer to explicitly select where subtype polymorphism is allowed, trading away reusability for performance, and it is common practice in C++ programming to avoid virtual function calls along common execution paths, sometimes leading to contorted, hard-to-understand and hard-to-extend code.

Compilers can reduce the cost of dynamically-dispatched messages in a number of ways. For example, *static class analysis* identifies a superset of the set of possible classes of objects that can be stored in variables and returned from expressions. Sometimes class analysis determines that the receiver of a message can be an instance of only one class, allowing the dynamically-dispatched message to be replaced with a direct procedure call (i.e., *statically-bound*) at compile-time and further optimized using inline expansion if the target procedure is small. If static class analysis determines that the receiver of a message can be one of a small set of classes, the dynamically-dispatched message can be replaced with a "type-case" expression, implemented with a series of run-time class tests, each branching to direct procedure calls implementing that case; executing one or two run-time class tests followed by an inlined version of the called procedure can be faster than performing a general run-time method lookup, particularly if additional optimizations of the called and calling methods can take place after inlining. Several other compiler techniques have been investigated for reducing the cost of message passing:

- *Profile-guided receiver class prediction* can support a type-casing-style optimization where static analysis is unable to determine precise information about the receiver of a message. The profile information representing the expected receiver class distribution of particular messages or call sites can be hard-wired into the compiler [Deutsch & Schiffman 84, Chambers *et al.* 89], gathered and exploited on-line [Hölzle & Ungar 94], and/or gathered off-line and exploited via recompilation [Garrett *et al.* 94, Calder & Grunwald 94].
- *Method specialization* can produce faster specialized versions of a method for particular inheriting subclasses; each specialized version can be optimized for the particular class or classes of the receiver for which the method is being specialized. Specializations for a given source method can be produced obliviously for each inheriting subclass [Kilian 88, Chambers & Ungar 89, Lea 90, Lim & Stolcke 91] or they can be produced selectively for groups of inheriting subclasses guided by execution frequency profiles [Dean *et al.* 95].

Class hierarchy analysis is another idea for speeding messages. When the compiler compiles a method, it knows statically that the receiver of the method is some subclass S of the class C containing the method. Unfortunately, in the absence of additional information, the compiler cannot optimize messages sent to the method's receiver, because the subclass S may override any of C's dynamically-dispatched methods. Class hierarchy analysis resolves this dilemma by supplying the compiler with complete knowledge of the program's class inheritance graph and the set of methods defined on each class. In the presence of this global information about the program being compiled, the compiler can infer statically a specific set of possible classes given that the receiver is a subclass of the class C, and messages sent to the method's receiver can be optimized. In particular, if there are no overriding methods in subclasses, a message sent to the method's receiver can be replaced with a direct procedure call and perhaps inlined. This sort of optimization would be especially important in the case of highly-extensible frameworks, where a great deal of flexibility is incorporated in the form of dynamically-dispatched messages within the framework base classes, but where only a limited portion of the potential flexibility is exploited by any particular application. For example, InterViews

supports the display and manipulation of arbitrary graphical shapes, but if a particular application only implements a rectangle concrete subclass, then all the dynamicallydispatched calls within the framework for manipulating arbitrary shapes can be replaced with direct calls to the appropriate rectangle methods.

Class hierarchy analysis has long been known informally as a possible optimization among implementors of optimizing compilers for object-oriented languages, but we are unaware of any previous studies of the effectiveness and costs of this technique. Moreover, class hierarchy analysis is just one of a number of candidate optimizations that could be incorporated into an optimizing compiler, and the question remains as to which is the most cost-effective combination to include. In this paper we perform such a study:

- We describe several implementation techniques for efficiently incorporating class hierarchy analysis into a compiler, in particular into an existing static class analysis framework. Our techniques scale to support multi-method-based languages; efficient compile-time method lookup in the presence of multi-methods is substantially harder than for mono-methods.
- We address programming environment concerns of achieving fast turnaround for programming changes and supporting independent development of libraries, which could be adversely affected by a whole-program analysis such as class hierarchy analysis.
- We measure the run-time performance benefit and compile-time cost of class hierarchy analysis on several large programs written in Cecil [Chambers 92, Chambers 93], a pure object-oriented language with multi-methods. Moreover, we also measure the run-time performance benefits and compile-time costs of profile-guided receiver class prediction and method specialization separately and in combination with class hierarchy analysis.

The next section of this paper describes our integration of class hierarchy analysis into static class analysis and addresses programming environment concerns. Section 3 reports on our experimental evaluation of class hierarchy analysis, profile-guided receiver class prediction, and method specialization. Section 4 describes some related work and Section 5 offers some conclusions.

2 Class Hierarchy Analysis

By exploiting information about the structure of the class inheritance graph, including where methods are defined (but not depending on the implementation of any method nor on the instance variables of the classes), the compiler can gain valuable static information about the possible classes of the receiver of each method being compiled. To illustrate, consider the following class hierarchy:



Consider the situation where the method p in the class F contains a send of the m message to self. m is declared to be a virtual function (there are several implementations of m for subclasses of A, and the right implementation should be selected dynamically). As a result, with only static intraprocedural class analysis the m message in F:p must be implemented as a general message send. However, by examining the subclasses of F and determining that there are no overriding implementations of m, the m message can be replaced with a direct procedure call to C::m and then further optimized with inlining, interprocedural analysis, or the like. This reasoning depends not on knowing the exact class of the receiver, as with most previous techniques, but rather on knowing that no subclasses of F override the version of m inherited by F. Class hierarchy analysis is a direct method for determining this without programmer intervention.

2.1 Alternatives to Class Hierarchy Analysis

Other languages have alternative approaches for achieving a similar effect. In C++ a programmer can declare whether or not a method is virtual (methods default to being non-virtual). When a method is not declared to be virtual, the compiler can infer that no subclass will override the method,¹ thus enabling it to implement invocations of the method as direct procedure calls. However, this approach suffers from three weaknesses relative to class hierarchy analysis:

• The C++ programmer must make explicit decisions of which methods need to be virtual, making the programming process more difficult. When developing a reusable framework, the framework designer must make decisions about which

^{1.} Actually, C++ non-virtual functions can be overridden, but dynamic binding will not be performed: the static type of the receiver determines which version of the non-virtual method to invoke, not the dynamic class.

operations will be overridable by clients of the framework, and which will not. The decisions made by the framework designer may not match the needs of the client program; in particular, a well-written highly-extensible framework will often provide flexibility that goes unused for any particular application, incurring an unnecessary run-time performance cost. In contrast, class hierarchy analysis is automatic and adapts to the particular framework/client combination being optimized.

- The virtual/non-virtual annotations are embedded in the source program. If
 extensions to the class hierarchy are made that require a non-virtual function to
 become overloaded and dynamically dispatched, the source program must be
 modified. This can be particularly difficult in the presence of separately-developed
 frameworks which clients may not be able to change. Class hierarchy analysis, as
 an automatic mechanism, requires no source-level modifications.
- A function may need to be virtual, because it has multiple implementations that need to be selected dynamically, but within some particularly subtree of the inheritance graph, there will be only one implementation that applies. In the example above, the *m* method must be declared virtual, since there are several implementations, but there is only one version of *m* that is called from *F* or any of its subclasses. In C++, *m* must be virtual and consequently implemented with a dynamically-bound message, but class hierarchy analysis can identify when a virtual function "reverts" to a non-virtual one with a single implementation for a particular class subtree, enabling better optimization. In particular, it is always the case that a message sent to the receiver of a method defined in a leaf class will have only one target implementation and hence can be implemented as a direct procedure call, regardless of whether or not the target method is declared virtual. For the benchmark programs discussed in Section 3, slightly more than half of the message sends that were statically bound through class hierarchy analysis could not have been made non-virtual in C++ (i.e., had more than a single definition of the routine).

In a similar vein, Trellis [Schaffert *et al*. 85, Schaffert *et al*. 86] allows a class to be declared with the no_subtypes annotation and Dylan [Dyl92] allows a class to be sealed, both of which inform the compiler that no subclasses exist. These annotations allow the compiler to treat the class as a leaf class and compile all messages sent to objects statically known to be of the class as direct procedure calls. Sealing has similar weaknesses relative to class hierarchy analysis as do non-virtual functions in C++: programmers have to predict in advance, in the source code, which classes are to be sealed, and opportunities for static binding will be missed, relative to class hierarchy analysis, when a class has unknown subclasses but none of the subclasses override certain methods.

2.2 Implementation

To make class hierarchy analysis effective, it must be integrated with intraprocedural static class analysis. Static class analysis is a kind of data flow analysis that computes a set of classes for each variable and expression in a method; the compiler uses this information to optimize dynamically-bound messages, type-case statements as in Modula-3 and Trellis, and other run-time type checks. Previous frameworks for static
class analysis in dynamically-typed object-oriented languages have defined several representations for sets of classes [Chambers & Ungar 90]:

Representation	Description	Source	Use
Unknown	the set of all classes	method arguments; results of non- inlined message sends; contents of instance variables	
Class(C)	the singleton set { <i>C</i> }	true branch of run- time class tests; lit- erals	supports static bind- ing of sends; elimi- nating run-time type checks
$Union(S_1,, S_n)$	union of class sets	control flow merges	supports "type-cas- ing" if small union of classes
Difference(S_1, S_2)	difference of two class sets	false branch of run- time class tests	avoids repeated tests

Earlier frameworks focused on the singleton class set as the primary source of optimization: if the receiver of a message is a singleton class set, then the message lookup can be resolved at compile-time and replaced with a direct procedure call to the target method. Unions of class sets were optimized only through a type-casing optimization, if the union combined a small number of classes.

2.2.1 Cone Class Sets

Class hierarchy analysis changes the flavor of static class analysis. The initial class set associated with the receiver of the method being analyzed is the set of classes inheriting from the class containing the method; in the earlier example, the receiver of F s p method is associated with the set {F, G, H}. It would be possible to use the Union set representation to represent the class set of the method receiver, but this could be space-inefficient for the large receiver class sets of methods declared high up in the inheritance hierarchy. Consequently we introduce a new representation for the kind of regular class sets inferred by class hierarchy analysis, the Cone:

Representation	Description	Source	Use
Cone(<i>C</i>)	the set of all sub- classes of the class <i>C</i> , including <i>C</i>	class hierarchy anal- ysis of method receiver; static type declarations	supports static bind- ing of sends

Class hierarchy analysis annotates the method's receiver with a cone set representation for the class containing the method. A simple optimization of this representation is to use the

Class(*C*) representation rather than Cone(*C*) if *C* is a leaf class. (This framework for representing static class analysis information is similar to Palsberg and Schwartzbach's static type system [Palsberg & Schwartzbach 94].) Cones tend to be concise summaries of sets of classes: in our implementation, when compiling a 52,000-line benchmark program with 957 classes, the average cone used for optimization purposes contained 12 concrete classes, and some cones included as many as 93 concrete classes.

In a statically-typed language, cones can be used to integrate static type declarations into the static class analysis framework: for a variable declared to be of static type C, any static class information inferred for the variable is intersected with Cone(C). This integration is crucial to adapting techniques developed for dynamically-typed objectoriented languages to work effectively for statically-typed object-oriented languages. For hybrid languages, built-in non-object-oriented data types like integers and arrays can be considered their own separate classes, as far as static class analysis is concerned; CLOS takes a similar view on integrating the standard Lisp data types with user-defined classes.

2.2.2 Method Applies-To Sets

If only singleton class sets support static binding of messages, then only leaf classes would benefit from class hierarchy analysis. However, this is unnecessarily conservative: even if the receiver of a message has multiple potential classes, if all the classes inherit the same method, then the message send can be statically bound and replaced with a direct procedure call. For instance, in the earlier example, the class set computed for the *m* message sent to the receiver of the *F*::*p* method is {*F*, *G*, *H*}, but all three classes inherit the same implementation of *m*, *C*::*m*. Our measurements indicate that nearly 50% of the messages statically bound using class hierarchy analysis have receiver class sets containing more than a single class. To receive the most benefit from class hierarchy analysis, static binding of messages whose receivers are sets of classes should be supported. One approach would be to iterate through all elements of Union and Cone sets, performing method lookup for each class, and checking that each class inherits the same method; however, this could be slow for large sets (e.g., cones of classes with many subclasses).

We have pursued an alternative approach that compares whole sets of classes at once. We first precompute for each method the set of classes for which that method is the appropriate target; we call this set the *applies-to* set. (In our compiler, we compute the applies-to sets of methods on demand, the first time a message with a particular name and argument count is analyzed, to spread out the cost of this computation.) Then at a message send, we take the class set inferred for the receiver and test whether this set overlaps each potentially-invoked method's applies-to set. If only one method's applies-to set overlaps the receiver's class set, then that is the only method that can be invoked and the message send can be replaced with a direct procedure call to that method. (To avoid repeatedly checking a large number of methods for applicability at every call site in the program, our compiler incorporates a compile-time method lookup cache that memoizes the function mapping receiver class set to set of target methods. In practice, the size of this cache is reasonable: for a 52,000-line program, this cache contained 7,686 entries, and a total of 54,211 queries of the cache were made during compilation.)

The efficiency of this approach to compile-time method lookup depends on the ability to precompute the applies-to sets of each method and the implementation of the set overlaps test for the different representations of sets. To precompute the applies-to sets, we first construct a partial order over the set of methods, where one method M_1 is less than

another M_2 in the partial ordering iff M_1 overrides M_2 . For the running example, we construct the following partial order:



Then for each method defined on class C, we initialize its applies-to set to Cone(C). Finally, we traverse the partial order top-down. For each method M, we visit each of the immediately overriding methods and subtract off their (initial) applies-to sets from M's applies-to set. In general, the resulting applies-to set for a method C::M is represented as Difference(Cone(C), Union(Cone(D_1), ..., Cone(D_n))), where D_1 , ..., D_n are the classes containing the directly-overriding methods. If a method has many directly-overriding methods, the representation of the method's applies-to set can become quite large. To avoid this problem, the subtracting can be ignored at any point, it is safe though conservative for applies-to sets to be larger than necessary.

The efficiency of overlaps testing depends on the representation of the two sets being compared. Overlaps testing for two arbitrary Union sets of size N is $O(N^2)$,¹ but overlaps testing among Cone and Class representations takes only constant time (assuming that testing whether one class can inherit from another takes only constant time [AK et al. 89, Agrawal *et al*. 91, Caseau 93]): for example, Cone(C1) overlaps Class(C2) iff C1 = C2or C2 inherits from C1. Overlaps testing of arbitrary Difference sets is complex and can be expensive. Since applies-to sets in general are Differences, overlaps testing of a receiver class set against a collection of applies-to Difference sets could be expensive. To represent irregular applies-to sets more efficiently, we convert Difference sets into a flattened BitSet representation. Overlaps testing of two BitSet class sets requires O(N)time, where N is the number of classes in the program. In practice, this check is fast: even for a large program with 1,000 classes, if bit sets use 32 bit positions per machine word, only 31 machine word comparisons are required to check whether two bit sets overlap. In our implementation, we precompute the BitSet representation of Cone(C) for each class C, and we use these bit sets when computing differences of Cones, overlaps of Cones, and membership of a class in a Cone.

When compiling a method and performing intraprocedural static class analysis, the static class information for the method's receiver is initialized to Cone(C), where C is the class containing the method. It might appear that the applies-to set computed for the method would be more precise initial information. Normally, this would be the case. However, if an overriding method contains a SUPEr send (or the equivalent) to invoke the overridden method, the overridden method can be called with objects other than those in the method's applies-to set; the applies-to set only applies for normal dynamically-dispatched message sends. If it is known that none of the overriding methods contain super sends that would invoke the method, then applies-to would be a more precise and legal initial class set.

^{1.} Since the set of classes is fixed, Union sets whose elements are singleton classes could be stored in a sorted order, reducing the overlaps computation to O(N).

2.2.3 Support for Dynamically-Typed Languages

In a dynamically-typed language, there is the possibility that for some receiver classes a message send will result in a run-time message-not-understood error. When attempting to optimize a dynamic dispatch, we need to ensure that we will not replace such a message send with a statically bound call even if there is only one applicable source method. To handle this, we introduce a special "error" method defined on the root class, if there is no default method already defined. Once error methods are introduced, no special efforts need be made to handle the possibility of run-time method lookup errors. For example, if only one (source) method is applicable, but a method lookup error is possible, our framework will consider this case as if two methods (one real and one error) were applicable and hence block static binding to the one real method. Similarly, if a message is ambiguously defined for some class, more than one method will include the class in its applies-to set, again preventing static binding to either method.

Knowledge of the class hierarchy and the location of defined methods can improve the results of receiver class prediction, a common technique used when the available static class information is not precise enough to lead to static binding of a message send. If the compiler can predict the expected class(es) of the message's receiver, either based on the name of the message and a hard-wired table in the compiler (as in Smalltalk-80 and the Self-91 system) or on dynamic profile data (as in the Self-93 system and Cecil), then it can insert run-time class tests for the expected classes. The compiler generates a full message send to handle any unexpected classes that occur at run-time:

After Class Prediction:

Before Class Prediction:

a := s.area(); if (s.class == Rectangle) { // statically bind to rectangle's area; inline if small a := s.Rectangle::area(); } else if (s.class == Circle) { // statically bind to circle's area; inline if small a := s.Circle::area(); } else { // a full message send to handle unexpected cases a := s.area(); }

In dynamically-typed languages, if the compiler can prove statically that the classes being tested exhaust the set of classes for which the message is correctly defined, then the final "unexpected" case can be replaced with a run-time message lookup error trap. (In statically-typed languages, using class hierarchy information to convert static type declarations into Cone class set representations accomplishes a similar purpose.) Such a lookup error trap might take up less compiled code space than a full message send, but more importantly in some languages it is known not to return to the caller. Thus, the error branch never merges back into the main stream of the program, and the compiler learns that only the predicted class(es) are possible after the message. In the above example, if analysis of the class hierarchy reveals that Rectangle and Circle are the only classes implementing the area message, then the third case can be replaced with an error trap. After the area message, the compiler will know that s is either a Rectangle or a Circle, enabling it to better implement later messages sent to S. (In a statically-typed language, class hierarchy analysis coupled with static type declarations would have shown s to refer to either a Rectangle or a Circle all along.) In the absence of class hierarchy information, the compiler must assume that some other class could implement the area message (or, in a statically-typed language, that some other class could be a subtype of the Shape static type), and consequently include support for the third "unexpected" case. When compiling our 52,000-line benchmark program, elimination of unexpected cases using class hierarchy analysis occurred 3,232 times; 3,004 of these occurrences optimized basic messages such as if and not, which might not be necessary in a less pure language lacking user-defined control structures.

2.2.4 Support for Multi-Methods

The above strategy for static class analysis in the presence of class hierarchy analysis and/ or static type declarations works for singly-dispatched languages with one message receiver, but it does not support languages with multi-methods, such as CLOS, Dylan, and Cecil. To support multi-methods, we associate methods not with sets of classes but sets of *k*-tuples of classes, where *k* is the number of dispatched arguments of the method.¹ To represent many common sets of tuples of classes concisely, we use *k*-tuples of class sets: a *k*-tuple $<S_1, ..., S_k>$, where the S_i are class sets, represents the set of tuples of classes that is the cartesian product of the S_i class sets. To represent other irregular sets of tuples of classes, we support a union of class set tuples as a basic representation.

Static class analysis is modified to support multi-methods as follows. For each method, we precompute the method's applies-to *tuple* of class sets; this tuple describes the combinations of classes for which the method should be invoked. For a multi-method specialized on the classes C_1 , ..., C_k , the method's applies-to tuple is initialized to $<Cone(C_1)$, ..., $Cone(C_k)>$. When visiting the directly-overriding methods, the overriding method's applies-to tuple is subtracted from the overriden method's tuple. When determining which methods apply to a given message, the *k*-tuple is formed from the class sets inferred for the *k* dispatched message arguments, and then the applies-to tuples of the candidate methods are checked to see if they overlap the tuple representing the actual arguments to the message.

Efficient multi-method static class analysis relies on efficient overlaps testing and difference operations on tuples. Testing whether one tuple overlaps another is straightforward: each element class set of one tuple must overlap the corresponding class set of the other tuple. Computing the difference of two tuples of class sets efficiently is trickier. The pointwise difference of the element class sets, though concise, would not be a correct implementation. One straightforward and correct representation would be a union of k k-tuples, where each tuple has one element class set difference taken:

$$- \equiv \bigcup_{i=1..k} < S_1, ..., S_{i-1}, S_i - T_i, S_{i+1}, ..., S_k >$$

If the $S_i - T_i$ element set is empty, then the *i*-th *k*-tuple is dropped from the union: its cartesian-product expansion is the empty set. Also, if two tuples in the union are identical except for one position, they can be merged into a single tuple by taking the union of the element class sets. Both optimizations are important in practice.

^{1.} We assume that the compiler can determine statically which subset of a message's arguments can be examined as part of method lookup. In CLOS, for instance, all methods in a generic function have the same set of dispatched arguments. In Cecil, the compiler examines all methods with the same name and number of arguments and finds all argument positions that any of the methods is specialized upon. It would be possible to consider all arguments as potentially dispatched, but this would be substantially less efficient, both at compile-time and at run-time, particularly if the majority of methods are specialized on a single argument.

For example, consider the following class hierarchy and multi-methods (x @ X is the syntax we use for indicating that the x formal argument of a multi-method is specialized for the class X):



Under both CLOS's and Cecil's method overriding rules, the partial order constructed for these methods is the following:

The applies-to tuples constructed for these methods, using the formula above, are:

(The third tuple of the first method's applies-to union drops out, since one of the tuple's elements is the empty class set.)

Unfortunately, for a series of difference operations, as occurs when computing the applies-to tuple of a method by subtracting off each of the applies-to tuples of the overriding methods, this representation tends to grow in size exponentially with the number of differences taken. For example, if a third method is added to the existing class hierarchy, which overrides the first method:

then the applies-to tuple of the first method becomes the following:

$$\begin{array}{ll} \mathfrak{m}(@A, @A, @A): &<\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{A,C\}, \{A,C\}, \{A,B,C\}> \cup \\ &<\{A,C\}, \{A,B,C\}, \{A,B,C\}> \cup <\{A,B\}, \{A,B\}, \{A,B,C\}> \cup \\ &<\{A,B,C\}, \{A\}, \{A,B,C\}> \cup <\{A,B,C\}, \{A,B\}, \{A,B\}> \end{array}$$

To curb this exponential growth problem, we have developed (with help from William Pugh) a more efficient way to represent the difference of two overlapping tuples of class sets:

$$- \equiv \bigcup_{i=1..k} < S_1 \cap T_1, ..., S_{i-1} \cap T_{i-1}, S_i - T_i, S_{i+1}, ..., S_k >$$

By taking the intersection of the first *i*-1 elements of the *i*th tuple in the union, we avoid duplication among the element tuples of the union. As a result, the element sets of the tuples are smaller and tend to drop out more often for a series of tuple difference operations. For the three multi-method example, the applies-to tuple of the first method is simplified to the following:

$$\begin{array}{ll} \mathfrak{m}(@A, @A, @A): & <\{A\}, \{A,B,C\}, \{A,B,C\}> \cup <\{C\}, \{A,C\}, \{A,B,C\}> \cup \\ & <\{C\}, \{B\}, \{A,B\}> \cup <\{B\}, \{A,B\}, \{A,B,C\}> \end{array}$$

As a final guard against exponential growth, we impose a limit on the number of class set terms in the resulting tuple representation, beyond which we stop narrowing (through subtraction) a method's applies-to set. We rarely resort to this final ad hoc measure: when compiling a 52,000-line Cecil program, only one applies-to tuple, for a message with 5 dispatched argument positions, crossed our implementation's threshold of 64 terms. The intersection-based representation is crucial for conserving space: without it, using the simpler representation described first, many applies-to sets would have exceeded the 64-term threshold.

2.3 Incremental Programming Changes

Class hierarchy analysis might seem to be in conflict with incremental compilation: the compiler generates code containing embedded assumptions about the structure of the program's class inheritance hierarchy and method definitions, and these assumptions might change whenever the class hierarchy is altered or a method is added or removed. A simple approach to overcoming this obstacle is to perform class hierarchy analysis and its dependent optimizations only after program development ceases. A final batch optimizing compilation could be applied to frequently-executed software just prior to shipping it to users, as a final performance boost.

Class hierarchy analysis can be applied even during active program development, however, if the compiler maintains enough intermodule dependency information to be able to selectively recompile those parts of a program invalidated after some change to the class hierarchy or the set of methods. In previous work, we have developed a framework for maintaining intermodule dependency information [Chambers *et al.* 95]. This framework is effective at representing the compilation dependencies introduced by class hierarchy analysis.

In the dependency framework, intermodule dependencies are represented by a directed, acyclic graph structure. Nodes in this graph represent information, including pieces of the program's source and information resulting from various interprocedural analyses such as class hierarchy analysis, and an edge from one node to another indicates that the information represented by the target node is derived from or depends on the information represented by the source node. Depending on the number of incoming and outgoing edges, we classify nodes into three categories:



- *Source nodes* have only outgoing dependency edges. They represent information present in the source modules comprising the program, such as the source code of procedures and the class inheritance hierarchy.
- *Target nodes* have only incoming dependency edges. They represent information that is an end product of compilation, such as compiled . o files.
- *Internal nodes* have both incoming and outgoing edges. They represent information that is derived from some earlier information and used in the derivation of some later information.

The dependency graph is constructed incrementally during compilation. Whenever a portion of the compilation process uses a piece of information that could change, the compiler adds an edge to the dependency graph from the node representing the

information used to the node representing the client of the information. When changes are made to the source program, the compiler computes what source dependency nodes have been affected and propagates invalidations downstream from these nodes. This invalidates all information (including compiled code modules) that depended on the changed source information.

In our compiler, static class analysis queries a compile-time method lookup cache to attempt to determine the outcome of message lookups statically; this cache is indexed with a message name and a tuple of argument class sets and returns the set of methods that might be called by such a message. To compute an entry in the method lookup cache, the compiler tests the applies-to tuples of methods with a matching name, in turn examining the BitSet representation of the set of classes represented by a Cone class set, which was computed from the underlying class inheritance graph. To support selective recompilation of optimized code, dependency graph nodes are introduced to model information derived from the source code:



- one kind of dependency node represents the BitSet representation of the set of subclasses of a class (one product of class hierarchy analysis),
- another kind of dependency node represents the set of methods with a particular name (another product of class hierarchy analysis),
- a third kind of dependency node represents the applies-to tuples of the methods, which is derived from the previous two pieces of information, and
- a fourth kind of dependency node guards each entry in the compile-time method lookup cache.

If the set of subclasses of a given class is changed or if the set of methods with a particular name and argument count is changed, the corresponding source dependency nodes are invalidated. This causes all downstream dependency nodes to be invalidated recursively, eventually leading to the appropriate compiled code being invalidated and subsequently recompiled.

To support greater selectivity and avoid unnecessarily invalidating any compiled code, some of the internal nodes in the dependency framework are *filtering nodes*. When invalidated, a filtering node will first check whether the information it represents really has changed; only if the information it represents has changed will a filtering node invalidate its successor dependency nodes. The compile-time method lookup cache entries are guarded by such filtering nodes. If part of the inheritance graph is changed or a new method is added, then downstream method lookup results *may* have changed, but often the source changes do not affect all potentially dependent method lookup cache entries. By rechecking the method lookup when invalidated, and squashing the invalidation if the method lookup outcome was unaffected by a particular source change, many unnecessary recompilations are avoided.

Empirical evaluation using a trace of a month's worth of actual program development indicates that the dependency-graph-based approach reduces the amount of recompilation required during incremental compiles by a factor of seven over a coarser-grained C++-style header file scheme, in the presence of class hierarchy analysis, and by a factor of two

over the Self compiler's previous state-of-the-art fine-grained dependency mechanism [Chambers & Ungar 91]. Of course, more recompilation occurs in the presence of class hierarchy analysis than would occur without it, but for these traces the number of files recompiled after a programming change is often no more than the number of files directly modified by the changes. A more important concern with our current implementation is that many filtering nodes may need to be checked after some programming changes, and even if few compiled files are invalidated, a fair amount of compilation time is expended in checking caches. The size of the dependency graph is about half as large as the executable for the program being compiled, which is acceptable in our program development environment; coarser-grained dependency graphs could be devised that save space at the cost of reduced selectivity. Further details are available elsewhere [Chambers *et al.* 95].

2.4 Optimization of Incomplete Programs

Class hierarchy analysis is most effective in situations where the compiler has access to the source code of the entire program, since the whole inheritance hierarchy can be examined and the locations of all method definitions can be determined; having access to all source code also provides the compiler with the option of inlining any routine once a message send to the routine has been statically-bound. Although today's integrated programming environments make it increasingly likely that the whole program is available for analysis, there are still situations where having source code for the entire program is unrealizable. In particular, a library may be developed separately from client applications, and the library developer may not wish to share source code for the library with clients. For example, many commercial C++ class libraries provide only header files and compiled . \circ files and do not provide complete source code for the library.

Fortunately, having full source code access is not a requirement for class hierarchy analysis: as long as the compiler has knowledge of the class hierarchy and where methods are defined in the hierarchy (but not their implementations), class hierarchy analysis can still be applied, and this information usually is available in the header files provided for the library. When compiling the client application, the compiler can perform class hierarchy analysis of the whole application, including the library, and statically bind calls within the client application. If source code for some methods in the library is unavailable, then statically-bound calls to those methods simply won't be able to be inlined. Static binding alone still provides significant performance improvements, particularly on modern RISC processors, where dynamically-dispatched message send implementations stall the hardware pipeline. Furthermore, some optimizing linkers are able to optimize static calls by inlining the target routine's machine code at link time [Fernandez 95], although the resulting code is not as optimized as what could be done in the compiler.

Using class hierarchy analysis when compiling a library in isolation is more difficult, since the client program might create subclasses of library classes that override methods defined in the library. The sealing approach of Dylan and Trellis can provide the compiler with information about what library classes won't be subclassed by client applications, supporting class hierarchy-based optimizations for those classes at the cost of reduced extensibility. Alternatively, the compiler could choose to compile specialized versions of methods applicable only to classes present in the library. For example, in a data structure library, the compiler could compile specialized versions of methods would also be compiled to support any subclasses of these library classes defined by client applications. In previous work, we have developed a profile-guided algorithm that examines the potential targets of sends in a routine and derives a set of profitable specializations based

on where in the class hierarchy these target routines are defined [Dean *et al.* 95]. This specialization algorithm detects call sites where class hierarchy analysis is insufficient to statically-bind message sends, and produces versions of methods specialized to truncated cones of the class hierarchy. Empirical measurements indicate that this algorithm applied to a complete 52,000-line Cecil program improves performance by 50% or more with only a 5% to 10% compiled code space increase, although we would expect a larger relative space overhead if the algorithm were applied to a library in isolation.

In summary, although class hierarchy analysis is most effective when the whole program is available, it can still be applied in situations where only portions of the program are available. Using the techniques described above, it can be applied to incomplete programs and to libraries independent of client applications, at some cost in missed optimization opportunities and/or increased compiled code space.

3 Empirical Assessment

Class hierarchy analysis, method specialization, and profile-guided receiver class prediction are all techniques for increasing the amount of class information available to the optimizer at compile time. All three represent different, and partially overlapping, approaches to solving the same fundamental problem: enabling the static binding of dynamic dispatches. In this section, we examine the effectiveness of these three approaches in isolation and in combination, focusing on the following questions:

- What is the impact of class hierarchy analysis on program performance?
- How effective is class hierarchy analysis in comparison to specialization? Can additional benefit be gained from combining class hierarchy analysis and specialization?
- How much benefit does class hierarchy analysis confer to a system that already performs profile-guided receiver class prediction?

We examine these issues in the context of the Vortex optimizing compiler for Cecil, a pure object-oriented language with multi-methods. Table 1 describes the five medium-to-large Cecil programs that we used as benchmarks. Appendix A includes the raw performance data.

Program Lines ^a		Description	
Richards (Rich)	400	Operating systems simulation	
Deltablue (Delta)	lue (Delta) 650 Incremental cons		
InstrSched (Instr)	2,400	MIPS global instruction scheduler	
Typechecker (Type)	17,000 ^b	Cecil static typechecker	
Compiler (Comp)	43,800 ^b	Vortex optimizing compiler	

Table 1: Cecil Benchmarks

a. Not including 8,500-line standard library.

b. The typechecker and compiler share approximately 12,000 lines of code.



Figure 1: Number of dynamic dispatches and execution speed

3.1 Effectiveness of Class Hierarchy Analysis

Since class hierarchy analysis provides the compiler with additional information about the classes of program variables (in particular the receiver(s) of the message being compiled), we would expect that the compiler would be able to statically bind more dynamic dispatches. Additionally, as discussed in Section 2.2.3, in some situations the compiler can determine when unexpected cases of a message send are guaranteed to fail, thus improving the quality of static analysis downstream of the send.

To measure the impact of class hierarchy analysis, we compiled the benchmark programs using the following set of compiler optimizations:

- unopt: No optimizations.
- std: Standard static intraprocedural analyses, including iterative intraprocedural class analysis, inlining, hard-wired class prediction for a small set of common messages, closure optimizations, extended splitting [Chambers & Ungar 90] and other standard intraprocedural optimizations such as CSE, constant folding and propagation, and dead code elimination.
- *cha-ct-only*: Standard (*std*) augmented by a limited usage of class hierarchy analysis. The results of class hierarchy analysis are used only to optimize the uncommon cases after run-time class tests, as described in Section 2.2.3.
- *cha*: Standard augmented by class hierarchy analysis. Class hierarchy analysis is used to provide class information about the receiver(s) of a method and to determine when messages sends are guaranteed to fail.

Figure 1 shows the dynamic number of dynamic dispatches and the execution speeds of the benchmark programs, normalized to those of *std*. Augmenting standard intraprocedural techniques with class hierarchy analysis resulted in a 23% to 89% improvement in execution speed for these applications. Since the performance difference between *std* and *cha-ct-only* was negligible, we can conclude that almost all of the runtime benefits seen in *cha* are due to additional receiver class information and not from optimizing the unexpected branches of class tests inserted by hard-wired class prediction, therefore we expect the improvements due to class hierarchy analysis to be significant even in object-oriented languages lacking used-defined control structures.



Figure 2: Compiled Code Space

In addition to improving execution speed, class hierarchy analysis reduces compiled code size, as shown in figure 2. Executables compiled with class hierarchy analysis were 12% to 21% smaller than *std* executables. By comparing the relative heights of the bars we can see that most of this reduction in code space was due to the replacement of call sites which are guaranteed to result in a message-not-understood error by a simple call to an error routine. Such call sites mainly occurred in the off branches of class tests inserted by hard-wired class prediction and thus, one would expect that class hierarchy analysis would have a smaller impact on compiled code space in languages without user-defined control structures.

3.2 Class Hierarchy Analysis and Specialization

Method specialization creates multiple copies of a single source method, each one of which is compiled with more precise static class information about the method receiver(s) thus enabling static binding and inlining of messages sent to Self. Class hierarchy analysis makes a similar contribution. In some sense, specialization and class hierarchy analysis are competing approaches to gaining the same sort of information. An important question, then, is "what are the relative benefits and costs of the two techniques?" Since a specialized method has exact class information about the receiver(s) of the method, we would expect that specialization would yield better results than class hierarchy analysis, but specialization accrues its benefits at the cost of increased compiled code space. In this section, we examine the impact of class hierarchy analysis and method specialization, both in isolation and in combination, using the following set of compiler optimizations:

- std: Standard static intraprocedural analyses, as described in Section 3.1.
- cha: Standard augmented by class hierarchy analysis.
- *cust-k*: Standard augmented by the *customization* form of method specialization. Customization is the specialization strategy used in Self, Trellis, and Sather implementations where a specialized version of a method is generated for each inheriting subclass. We extended customization to handle multi-methods by specializing on all combinations of subclasses of the dispatched arguments.¹

^{1.} We used profile data to determine which of the specializations produced by *cust-k* actually needed to be generated, since it was infeasible to actually compile them all. In effect, this simulates Self-style dynamic compilation.



Figure 3: Number of dynamic dispatches and execution speed

- *cust-1*: Standard augmented by customization on only the first receiver. Customization on only the first receiver is more feasible in practice than customizing on all combinations of receivers in the case of multi-methods.
- *cha-1*: Standard augmented by class hierarchy analysis for only the first receiver, to compare against *cust-1*.
- *cust+cha*: Standard augmented with both *cust-k* and class hierarchy analysis.
- *selective*: Standard augmented with class hierarchy analysis and a selective specialization algorithm that combines a profile-derived weighted call graph and class hierarchy analysis to select candidates for specialization [Dean *et al.* 95].

Figure 3 shows the normalized number of dynamic dispatches and execution speeds for the benchmark programs compiled with these seven configurations. As expected, in most cases customization had a larger impact than class hierarchy analysis, speeding up programs by 48% to 87%. Combining class hierarchy analysis and naive customization (*cust+cha*) yielded only small additional benefits. However, the runtime benefits of customization came at the cost of a large increase in compiled code space and compile time. *Cust-1* executables were 2.5 to 3.5 times larger than *std* executables and *cust-k* executables are too large to actually be built using standard static compilation techniques; even in a system with dynamic compilation, *cust-k* would require compiling roughly 1.5 to 2 times as many methods as *std*. In contrast, *cha* executables were 12% to 21% smaller than *std*. By far, the best results were achieved by *selective*, which increased execution speed by 52% to 210% while increasing compiled code space by only 4% to 10%.

3.3 Class Hierarchy Analysis and Profile-Guided Receiver Class Prediction

Profile-guided receiver class prediction has been shown to substantially improve the performance of applications written in pure object-oriented languages. It is unclear whether or not adding class hierarchy analysis to a system that already performs profile-guided receiver class prediction would result in any significant improvements [Hölzle 94]. To examine this question, we utilized the following compiler configurations:

- std: Standard intraprocedural optimizations as described in Section 3.1.
- cha: Standard augmented by class hierarchy analysis



Figure 4: Number of dynamic dispatches and execution speed

- profile: Standard augmented with profile-guided receiver class prediction.
- profile+cha: Standard augmented with profile-guided receiver class prediction and class hierarchy analysis.

Figure 4 shows the normalized number of dynamic dispatches and execution speeds of the applications. These numbers show that profile-guided receiver class prediction is the most effective of the three techniques in isolation, improving application execution speeds by 90% to 410% over std. However, augmenting profile-guided receiver class prediction with class hierarchy analysis (profile+cha) yielded surprisingly large additional improvements of 45% to 125% over profile alone. Part of this improvement can be explained by a reduction in the number of call sites at which the compiler, due to a lack of sufficient static class information, is forced to fall back on profile information and insert explicit class tests. However, the improvements can not solely be explained by this effect, since combining the two optimizations yields larger benefits than can be explained even by assuming that their benefits in isolation are completely additive. For example, in the Compiler benchmark, cha is 41% faster than std and profile is 142% faster than std. Multiplying these two speedups results in a projected speedup of 241% over *std*, which is smaller than the observed speedup of 272%. By examining the number of dynamic dispatches, we can see that it is not the case that *profile+cha* is able to eliminate more dispatches than the sum of *profile* and *cha* (in the Compiler benchmark, *cha* eliminated 45% of the dynamic dispatches present in std and profile eliminated 58%, but profile+cha only eliminated 88%). We believe that the large speedups observed in profile+cha are due to the increased effectiveness of other compiler optimizations, such as CSE and register allocation, that was enabled by the simplified control flow graphs produced by this configuration.

In our implementation of profile-guided receiver class prediction, we only insert class tests if the target method is a desirable candidate for inlining; if the target method is not going to be inlined, then we leave the call site unchanged. In the Richards benchmark, several of the frequently-called methods are too large to be inlined, but can be statically-bound by class hierarchy analysis. This explains why *profile* eliminated fewer dynamic dispatches than *cha* for the Richards benchmark.

4 Other Related Work

An alternative to performing whole-program optimizations such as class hierarchy analysis at compile-time is to perform optimizations at link-time. Recent work by Fernandez has investigated using link-time optimization of Modula-3 programs to convert dynamic dispatches to statically bound calls when no overriding methods were defined [Fernandez 95]. This optimization is similar to class hierarchy analysis. An advantage of performing optimizations at link-time is that, because the optimizations operate on machine code, they can be applied to the whole program, including libraries for which source code is unavailable. However, there are two drawbacks of link-time optimizations. First, because the conversion of message sends to statically-bound calls happens in the linker, rather than the compiler, the compiler's optimization tools cannot be brought to bear on the now statically-bound call site; the indirect benefits of post-inlining optimizations in the compiler can be more important than the direct benefit of eliminating procedure call/return sequences. Second, care must be taken to prevent linking from becoming a bottleneck in the edit-compile-debug cycle. For example, Fernandez's linktime optimizer for Modula-3 performs code generation from a machine-independent intermediate representation for the entire program at every link; Fernandez acknowledges that this design penalizes turnaround time for small programming changes. Additional link-time optimizations would only increase this penalty. In contrast, class hierarchy analysis coupled with a selective invalidation mechanism supports incremental recompilation, fast linking, and compile-time optimization of call sites where source code of target methods is available.

Srivastava has developed an algorithm to prune unreachable procedures from C++ programs at link-time [Srivastava 92]. Although the described algorithm is only used to prune code and not to optimize dynamic dispatches, it would be relatively simple to convert some virtual function calls into direct procedure calls using the basic infrastructure used to perform the procedure pruning.

Interprocedural class analysis algorithms are an important area of current research [Palsberg & Schwartzbach 91, Oxhøj *et al.* 92, Agesen *et al.* 93, Plevyak & Chien 94]. By examining the whole program and solving an interprocedural data flow problem to determine the set of classes that might be stored in each program variable, these algorithms can provide more accurate class sets than intraprocedural static class analysis or class hierarchy analysis. However, current interprocedural class analysis algorithms are relatively expensive to run, assume access to the source code of the entire program (including method bodies), and are not incremental in the face of programming changes. Nevertheless, as these algorithms mature, it will be interesting to compare the run-time benefits and compile-time costs of interprocedural class analysis against class hierarchy analysis and the other techniques examined in Section 3. Agesen and Hölzle have compared the effectiveness of profile-guided receiver class prediction alone to interprocedural class analysis alone for a suite of small-to-medium sized Self programs, but they did not report on the effectiveness of combining the two techniques [Agesen & Hölzle 95].

5 Conclusions

Class hierarchy analysis is a promising technique for eliminating dynamically-dispatched message sends automatically. Unlike language-level mechanisms such as non-virtual functions in C_{++} and sealed classes in Dylan, class hierarchy analysis improves performance while preserving the source-level semantics of message passing and the ability for clients to subclass any class. To integrate class hierarchy analysis effectively

into existing static class analysis frameworks, we introduced the cone representation for a class and its subclasses and constructed applies-to sets for each method to support compile-time method lookup in the presence of cones and other unions of classes. Cones also provide a means for static type declarations to be integrated into static class analysis. Class hierarchy analysis imposes some requirements on the underlying programming environment, particularly to support incremental compilation, but these costs appear to be manageable in practice. These techniques have been implemented in the Vortex compiler for Cecil since the Spring of 1994. In this compiler class hierarchy analysis is always performed as a matter of course, and intermodule dependency links support selective recompilation. The Vortex compiler is itself a 52,000-line Cecil program, undergoing rapid continuous development and extension, providing some evidence that class hierarchy analysis is compatible with a program development environment.

Class hierarchy analysis is only one of a number of optimizations proposed for objectoriented languages; others include method specialization and profile-guided receiver class prediction. We implemented and measured these techniques separately and in combination, on a collection of medium-to-large Cecil programs, to try to determine which techniques are most effective and where the techniques could profitably be combined. Of the techniques that we examined, profile-guided class prediction was the most effective in isolation at improving program performance. However, performing class hierarchy analysis in addition to profile-guided class prediction provided substantial performance improvements over profile-guided class prediction alone. Class hierarchy analysis consumed far less compiled code space than customization, but with smaller performance gains; the best results are achieved by a profile-guided selective specialization algorithm integrated with class hierarchy analysis.

There are two interesting future directions for this work. First, it would be very interesting to extend this performance study to include interprocedural static class analysis algorithms as they mature. Second, the effectiveness of these techniques in pure object-oriented languages like Cecil has been demonstrated, but their effectiveness and relative value when applied to hybrid, statically-typed object-oriented languages such as C++ and Modula-3 remains an open question. We are in the process of porting compiler front-ends for C++ and Modula-3 to the Vortex optimizing compiler back-end in order to be able to perform such experiments.

Acknowledgments

We thank William Pugh for his help in devising a more efficient tuple difference implementation. Charles Garrett performed many of the initial studies of profile-guided receiver class prediction. Vitaly Shmatikov, Ben Teitelbaum, Tina Wong, and Matthew Parker have contributed to the Cecil implementation.

This research is supported in part by an NSF Research Initiation Award (contract number CCR-9210990), an NSF Young Investigator Award (contract number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, and Pure Software. Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories provided us with dot, a program for automatic graph layout.

Other papers on the Cecil programming language and the Vortex optimizing compiler are available via anonymous ftp from cs.washington.edu:/pub/chambers and via the World Wide Web URL http://www.cs.washington.edu/research/projects/cecil.

References

- [Agesen & Hölzle 95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. Technical Report TRCS 95-04, Department of Computer Science, University of California, Santa Barbara, March 1995.
- [Agesen *et al.* 93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzback. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings ECOOP '93*, July 1993.
- [Agrawal *et al*. 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings OOPSLA '91*, pages 113–128, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
- [AK et al. 89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. ACM Transactions on Programming Languages and Systems, 11(1):115–146, January 1989.
- [Bobrow et al. 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. SIGPLAN Notices, 28(Special Issue), September 1988.
- [Calder & Grunwald 94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 397–408, Portland, Oregon, January 1994.
- [Caseau 93] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In Proceedings OOPSLA'93, pages 271–287, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.
- [Chambers & Ungar 89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language. SIGPLAN Notices, 24(7):146–160, July 1989. In Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.
- [Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. SIGPLAN Notices, 25(6):150–164, June 1990. In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.
- [Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings OOPSLA '91*, pages 1–15, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
- [Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP* '92, LNCS 615, pages 33– 56, Utrecht, The Netherlands, July 1992. Springer-Verlag.
- [Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

- [Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In Proceedings OOPSLA '89, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.
- [Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In 17th International Conference on Software Engineering, Seattle, WA, April 1995.
- [Dean *et al.* 95] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. *SIGPLAN Notices*, June 1995. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language* Design and Implementation.
- [Deutsch & Schiffman 84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pages 297–302, Salt Lake City, Utah, January 1984.
- [Dyl92] Dylan, an Object-Oriented Dynamic Language, April 1992. Apple Computer.
- [Fernandez 95] Mary Fernandez. Simple and Effective Link-time Optimization of Modula-3 Programs. SIGPLAN Notices, June 1995. In Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.
- [Gabriel *et al.* 91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):28–38, September 1991.
- [Garrett et al. 94] Charlie Garrett, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report UW-CS 94-03-05, University of Washington, March 1994.
- [Harbison 92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Notices*, 29(6):326– 336, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
- [Hölzle 94] Urs Hölzle. Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming. PhD thesis, Stanford University, August 1994.
- [Johnson 92] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings OOPSLA '92*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.
- [Kilian 88] Michael F. Kilian. Why Trellis/Owl Runs Fast. Unpublished manuscript, March 1988.
- [Lea 90] Doug Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, CA, April 1990.
- [Lim & Stolcke 91] Chu-Cheow Lim and Andreas Stolcke. Sather Language Design and Performance Evaluation. Technical Report TR 91-034, International Computer Science Institute, May 1991.
- [Linton et al. 89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 2(2):8–22, February 1989.

- [Nelson 91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [Omohundro 94] Stephen Omohundro. The Sather 1.0 Specification. Unpublished manuscript from International Computer Science Institute, Berkeley, CA, 1994.
- [Oxhøj et al. 92] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In O. Lehrmann Madsen, editor, *Proceedings* ECOOP '92, LNCS 615, pages 329–349, Utrecht, The Netherlands, July 1992. Springer-Verlag.
- [Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings OOPSLA '91*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
- [Palsberg & Schwartzbach 94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
- [Plevyak & Chien 94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings OOPSLA '94*, pages 324–340, Portland, Oregon, October 1994.
- [Schaffert *et al.* 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985.
- [Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings OOPSLA '86*, pages 9–16, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.
- [Srivastava 92] Amitabh Srivastava. Unreachable Procedures in Object-Oriented Programming. *ACM Letters on Programming Languages and Systems*, 1(4):355–364, December 1992.
- [Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.
- [Szypersky et al. 93] Clemens Szypersky, Stephen Omohundro, and Stephan Murerzw. Engineering a Programming Language: The Type and Class System of Sather. Technical Report 93-064, International Computer Science Institute, Berkeley, CA, 1993.

Appendix A Raw Data

Configuration	Richards	Deltablue	InstSched	Typechecker	Compiler
unopt	13.410	31.830	25.080	294	2314
std	1.780	11.880	11.870	113	1176
cha-ct-only	1.830	11.490	11.960	110	1152
cha	0.940	8.940	9.670	81	834
cust-1	0.950	7.620	8.710	73	735
cha-1	1.060	9.160	10.030	83	852
cust-k	0.950	8.050	8.870	73	712
cust-k+cha	0.910	6.220	8.160	71	700
selective	0.560	3.810	6.850	68	650
profile	0.930	2.310	5.830	48	484
profile+cha	0.380	1.200	4.000	40	316

Table 2: Execution Times (seconds)

Table 3: Dynamically Dispatched Message Sends (x1000)

Configuration	Richards	Deltablue	InstSched	Typechecker	Compiler
unopt	10,006	13,461	7,505	75,053	470,488
std	3,697	6,980	4,676	36,131	231,659
cha-ct-only	3,697	6,980	4,664	36,028	231,529
cha	1,583	4,668	3,004	18,566	128,400
cust-1	1,414	3,874	2,655	19,301	129,729
cha-1	1,583	4,668	3,192	20,632	143,578
cust-k	1,414	3,874	2,637	19,341	129,701
cust- <i>k</i> +cha	1,414	3,624	2,516	19,156	125,095
selective	1,278	3,021	1,969	15,478	101,929
profile	619	159	1,890	15,426	98,435
profile+cha	545	157	474	3,902	28,640

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch Xerox PARC, Software Concepts Group

Allan M. Schiffman Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern programming systems are among the most difficult to implement efficiently, even individually. A new implementation of the Smalltalk-80 system, hosted on a small microprocessor-based computer, achieves high performance while retaining complete (object code) compatibility with existing implementations. This paper discusses the most significant optimization techniques developed over the course of the project, many of which are applicable to other languages. The key idea is to represent certain runtime state (both code and data) in more than one form, and to convert between forms when needed.

*Smalltalk-80 is a trademark of the Xerox Corporation.

BACKGROUND

The Smalltalk-80 system is an object-oriented programming language and interactive programming environment. The Smalltalk-80 language includes many of the most difficult-toimplement features of modern programming languages: dynamic storage allocation, full upward funargs, and call-time binding of procedure names to actual procedures based on dynamic type information, sometimes called *message-passing*. The interactive environment includes a full complement of programming tools: compiler, debugger, editor, window system, and so on, all written in the Smalltalk-80 language itself. A detailed overview of the system appears in [SCG 81]. [Goldberg 83] is a technical reference for both the non-interactive programmer and the system implementor; [Goldberg 84] is a reference manual for the interactive system.

SPECIAL DIFFICULTIES

The standard Smalltalk-80 system implementation is based on an ideal *virtual machine* or v-machine. The compiler generates code for this machine, and the implementor's documentation describes the system as an interpreter for the v-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 vmachine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure activations just like any other data objects.

The Smalltalk-80 language approaches programming with generic data types through message-passing and dynamic typing. To invoke a procedure (method in Smalltalk-80 terminology), a message is sent to a data object (the receiver), which selects the method to be executed. This means that a method address must be found at runtime. At a given lexical point in the code, only the message name (selector) is known. To perform a messagesend, the data type (class) of the receiver is extracted, and the selector is used as a hash index into a table of the message dictionary of the class, which maps selectors to methods. The task of method-lookup is complicated by the inheritance property of classes -- a class may be defined as a subclass to another, inheriting all of the methods of the superclass. If the initial method-lookup fails, the lookup algorithm tries again using the message dictionary of the superclass of the receiver's class, continuing in this way up the class hierarchy until a method corresponding to the selector is found or the top of the inheritance hierarchy is reached.

The Smalltalk-80 language uses the organization of objects into classes to provide strong information hiding. Only the methods associated with a given class (and its subclasses) can access directly the state of an instance of that class. All access from "outside" must be through messages. Because of this, a Smalltalk-80 program must often make procedure calls to access state where languages such as Pascal could compile a direct access to a field of a record. This makes the performance of the method-lookup algorithm even more critical.

IMPLEMENTATION OUTLINE

The purpose of the research described here was to build a Smalltalk-80 system with acceptable performance on a relatively inexpensive, microprocessor-based computer; specifically, to discover how to implement the basic data and code objects of the Smalltalk-80 system in a way that still conformed to the v-machine specification, but were more suitable for conventional hardware. (As of early 1982, the only implementations that ran at acceptable speed were on non-commercial, user-microprogrammable machines, as described in [Krasner 83] [Lampson 81].) The system specification in [Goldberg 83] includes the definition of internal data structures and object code representation for the virtual machine. Indeed, much of the system code depends on these definitions. We chose to take

This was motivated partly by a desire to retain object-code portability, and partly by a desire not to complicate the description of the Smalltalk-80 machine model.

The single principle that underlies all the results reported here is dynamic change of representation. By this we mean that the same information is represented in more than one (structurally different) way during its lifetime, being converted transparently between representations as needed for efficient use at any moment. An important special case of this idea is *caching*: one can think of information in a cache as a different representation of the same information (considering contents and accessing information together) in the backup memory. In the implementation described in this paper, we applied this principle to several different kinds of runtime information in the Smalltalk-80 system.

* We dynamically translate v-code (i.e., code in the instruction set of the v-machine) into code that executes directly on the hardware without interpretation, the native code or *n-code*. Translated code is cached: it is regenerated rather than paged.

* We represent procedure activation records (*contexts* in Smalltalk-80 parlance) in either a machine-oriented form, when they are being used to hold execution state, or in the form of Smalltalk-80 data objects, when they are being treated as such.

* We use several different caches to speed up the polymorphic search required at each procedure invocation. In the best case, which applies over 90% of the time, a Smalltalk-80 procedure invocation requires only one comparison operation in addition to a conventional procedure linkage.

* Using the techniques in [Deutsch&Bobrow 76], we represent reference count information for automatic storage management in a way that climinates approximately 85% of the reference counting operations required by a standard implementation.

CODE TRANSLATION

Targeting code to a portable v-machine has been used in other language implementations. Usually v-code targeting is used only to avoid having multiple (one per target machine) code-generation phases of the compiler; a secondary benefit is that v-code is usually much more compact than code for any real machine. Since the Smalltalk-80 compiler is just one tool available in the same interactive environment used for execution, and other tools besides the compiler must be able to examine the machine state, the v-machine approach is even more attractive in reducing the cost of rehosting.

PERFORMANCE ISSUES

To rehost the system, an implementor must emulate the vmachine on the target hardware, either in microcode or in software. This normally incurs a severe performance penalty arising from several factors.

> Processors have specialized hardware for fetching, decoding, and dispatching their own native instruction set. This hardware is typically not available to the programmer (although it may be available at the microprogram level), and therefore not useful to the vmachine interpreter in its time-consuming operation of instruction fetching, decoding, and dispatching.

> • The v-machine architecture may be substantially different from that of the underlying hardware. For example, many v-machines, including both the P-system

and Smalltalk-80 v-machines, use a stack-oriented architecture for convenience in code generation, but most available hardware machines execute registeroriented code much more efficiently than stack-oriented code.

* The basic operations of the v-machine may be relatively expensive to implement, even though the overall algorithm represented by a v-code program may not be much more expensive than if it were implemented in the hardware instruction set. For example, even though a naive interpreter for the Smalltalk-80 v-code must perform reference counting operations every time it pushes a variable value onto the stack, a sequence of several instructions often has no net effect on reference counts.

If the v-code were translated to n-code after normal compilation of a source program to v-code, the interpreter's overhead could be eliminated and some optimizations become possible. One technique for eliminating part of the overhead of interpretation is *threaded code* [Bell 73] [Moore 74]. In this approach, v-code consists of an actual sequence of subroutine calls on runtime routines. This technique does reduce the overhead for fetching and dispatching v-code instructions, although it does not help with operand decoding, or enable optimizations that span more than one v-instruction. We prefer to translate v-code to in-line n-code in a more sophisticated way.

Naive translation from v-code to n-code is a process something like macro-expansion. In fact, [Mitchell 71] observed that a translator can be derived very simply from an interpreter by having the interpreter save its action-routine code in a buffer rather than executing it. If the computation performed by individual action routines is small relative to the computation needed for the interpreter loop, the benefit of even this simple kind of translation will be great.

Translation-time can also be considered an opportunity for peephole optimization or even mapping stack references to registers [Pittman 80]. Translation back-ends for portable compilers have been implemented [Zellweger 79].

DYNAMIC TRANSLATION

Because the Smalltalk-80 v-code is a compact representation that captures the basic semantics of the language, n-code will typically take up much more space than v-code. (In the implementation discussed in this paper, n-code takes about 5 times as much space as v-code.) This would place severe stress on a virtual memory system if the n-code were being paged. However, since n-code is derived algorithmically from v-code, there is no need to keep it permanently: it can be recomputed when needed, if this is more efficient than swapping it in from secondary storage. This leads us to the idea of translating at runtime. (The idea of dynamic translation appears in [Rau 78], where it is applied to translation from v-code to microcode.) When a procedure is about to be executed, it must exist in ncode form. If it does not, the call faults and the translator takes control. The translator finds the corresponding v-code routine, translates it, and completes the call. Since, as mentioned earlier, the translation process is more akin to macro-expansion than compilation, translation time for a v-code byte is comparable to the time taken to interpret it.

We consider the translation approach, and dynamic translation in particular, to be the most interesting part of our research, since it motivated the work on multiple state representations described below. A later section of this paper presents the experimental results that support our contention that dynamic translation is an effective technique in a substantial region of current technological parameters.

MAPPING STATE AT RUNTIME

Since the definition of the Smalltalk-80 v-machine makes runtime state such as procedure activations visible to the programmer as data objects, an implementation based on n-code must find a way to make the state appear to the programmer as though it were the state of a v-machine, regardless of the actual representation. The system must maintain a mapping of nmachine state to v-machine state; in particular, it must keep the v-code available for inspection.

How can we guarantee that all attempts to access a quantity requiring representation mapping are detected? The structure of the Smalltalk-80 language guarantees that the only code that can access an object of a given class directly is the code that implements messages sent to that class. Thus, the only code that can directly access the parts of an object requiring mapping is code associated with that object's class. Recall that all the code in the Smalltalk-80 system is written in the Smalltalk-80 language, hence compiled into v-code. When we translate a procedure from v-code to n-code that is associated with a class whose representation may require mapping, we generate special n-code that calls a subroutine to ensure that the object is represented in a form where accesses to its named parts are meaningful.

The most obvious quantity requiring mapping is the return address (PC) in an activation record, which refers to a location in the n-code procedure rather than in the v-code. Although there is no simple algorithmic correspondence between the v-PC and the n-PC values, the v-PC need only be available when a program attempts to inspect an activation as a data object. At that moment, the system can consult (or compute) a table associated with the procedure that gives the correspondence between n- and v-PC values.

We can greatly reduce the size of the mapping tables for PC values by observing that the PC can only be accessed when an activation is suspended, i.e., at a procedure call or interrupt/process-switch. If we are willing to accept somewhat greater latency in a Smalltalk-80 program's response to interrupts, we can choose a restricted but sufficient set of allowable interrupt points, and only store the mapping tables for those points. This is what our implementation does: interrupts are only allowed at, and PC map entries are only stored for, all procedure calls and backward branches (the latter since interrupts must be allowed inside loops).

MULTIPLE REPRESENTATIONS OF CONTEXTS

As mentioned earlier, the format of procedure activation records are part of the Smalltalk-80 v-machine specification. Contexts are full-fledged data objects; they have identifiable fields which can be accessed and they respond to messages. A context is created for every message-send. There is also syntax in the language for creating contexts whose activation is deferred, called *block contexts* in Smalltalk-80 terminology, which correspond to the *functionals, closures*, or *funargs* of other languages. Most control structures in the Smalltalk-80 system are implemented with block contexts.

The fact that contexts are standard data objects implies that they must be created like data objects, i.e., allocated on a heap and reclaimed by garbage collection or reference counting. Unfortunately, conventional machines are adapted for calling sequences that create a new activation record as a stack frame, storing suspended state in predefined slots in the frame. Actually implementing contexts as heap objects results in a serious performance penalty.

Measurements show that even in Smalltalk-80 programs, more than 85% of all contexts behave like procedure activations in conventional languages: they are created by a call, never referenced as a data object, and can be freed as soon as control returns from them. (Note that any context in which a block context is created does not satisfy this criterion.) Such contexts are candidates for stack-frame representation. (An unpublished experimental implementation of an earlier Smalltalk system used linear stacks, but did not deal properly with contexts that outlived their callers.)

Stack allocation of contexts solves one of the two major efficiency problems associated with treating contexts like other objects, namely the overhead of allocating the contexts themselves. [Deutsch&Bobrow 76] shows how to solve the other problem, of reference counting operations apparently being required on every store into a local variable. With these two problems solved, we can use the hardware subroutine call, return, and store instructions directly.

Our system has several types of context representations. A message-send creates a new context in a representation optimized for execution: a frame is allocated on the machine's stack (with some spare slots) by the usual machine instructions. In the simple case, where no reference is ever made to the context as a data object, the machine's return instruction simply pops the frame off the stack when control returns from the context. This kind of context, which lives its life as a stack frame, we call *volatile*.

At the other extreme, we store contexts in a format compliant with the virtual machine specification, which can be manipulated as data items. We call this representation *stable*.

The third representation of a context, called *hybrid*, is a stack frame that incorporates header information to make it look partly like an ordinary data object. A volatile context is converted to hybrid when a pointer is generated to it. Since this makes it possible for programs to refer to the context as an object, we fill in slots in the frame corresponding to the header fields in an ordinary object. This pseudo-object is tagged as being of a class we name "DummyContext." A block of memory is allocated, and its address is stored in the context in case the context must be stabilized in the future. Since there may be pointers to this context, it cannot be returned from in a normal way, so the return address is copied to another slot in the frame and replaced with the address of a clean-up routine that stabilizes the context on return.

When a message is sent to a hybrid context, the send fails (there are no procedures defined for the DummyContext class), and a routine is called to convert the hybrid context to the stabilized form. At this point PC mapping comes into play; the n-PC in the activation is converted to a v-PC for the stabilized representation. Pointers to the hybrid context are switched to refer to the stable context (this is simple in our system, which uses an indirection table for all objects). After the context has been stabilized, the failed message is re-sent to the stable form.

A stable context is not suitable for execution. Before a stabilized context can be resumed, it is reconstituted on the stack as hybrid. Again, this means that the n-PC must be reconstructed from the v-PC. Usually the v-PC does not change during the stable period, so our system includes a one-element cache in each n-code procedure for the most recent v-PC/n-PC pair, to avoid having to run the mapping algorithm.

Block contexts are "born" in stable form, since the whole purpose of closures is to provide a representation for an execution context which can be invoked later.

IN-LINE CACHING OF METHOD ADDRESSES

Message-passing is applied down to the simplest operations in Smalltalk. The system provides a variety of predefined classes: the most basic operations on elementary data types (such as addition of integers) are performed by *primitives* implemented by the kernel of the system, rather than by Smalltalk routines, but there is no distinction drawn at the language level. Since message-sends are so ubiquitous, they must be fast; the operation of method-lookup is both expensive and critical.

• All existing Smalltalk-80 implementations accelerate methodlookup by using a *method cache*, a hash table of popular method addresses indexed by the pair (receiver class, message selector). This simple technique typically improves system performance by 20-30%. More extensive measurements of this improvement appear in [Krasner 83].

Further performance improvements are suggested by the observation of dynamic locality of type usage. That is, at a given point in code, the receiver is often the same class as the receiver at the same point when the code was last executed. If we cache the looked-up method address at the point of send, subsequent execution of the send code has the method address at hand, and method-lookup can be avoided if the class of the receiver is the same as it was at the previous execution of this particular send. Of course, the class of the receiver may have changed, and must be checked against the class corresponding to the cached method address.

In the implementation described here, the translator generates n-code for sends *unlinked* -- as a call to the methodlookup routine, with the selector as an in-line argument. The method-lookup routine *links* the call by finding the receiver class, storing it in-line at the call point, and doing the methodlookup (like other implementations, it uses a selector/classmethod cache). When the n-code method address is found, it is placed in-line with a call instruction, overwriting the former call to the lookup routine. The call is then re-executed. (Of course, there may be no corresponding n-code method, in which case the translator is called first.) Note that this is a kind of dynamic code modification, which is generally condemned in modern practice. The n-method address can just as well be placed outof-line and accessed indirectly; code modification is more efficient, and we are using it in a well-confined way.

The entry code of an n-code method checks the stored receiver class from the point of call against the actual receiver class. If they do not match, relinking must occur, just as if the call had not yet been linked.

Since linked sends have n-code method addresses bound inline, this address must be invalidated if the called n-code method is being discarded from memory. The idea of scanning all ncode routines to invalidated linked addresses was initially so daunting that we almost rejected the scheme. However, since ncode only exists in main memory, invalidation cannot produce time-consuming page faults. Furthermore, since the PC mapping tables described earlier contain precisely the addresses of calls in the n-code, no searching of the n-code is required: it is only necessary to go through the mapping tables and overwrite the call instructions to which the entries point. (A scheme similar to this may be found in [Moon 73].)

For a few special selectors like +, the translator generates in-line code for the common case along with the standard send code. For example, + generates a class check to verify that both arguments are small integers, native code for integer addition, and an overflow check on the result. If any of the checks fail, the send code is executed. This is a space-time tradeoff justified by measurements that indicate that the overwhelming majority of arithmetic operations involve only small integers, even though they are (in principle) polymorphic like all other operations in the language.

EXPERIMENTAL RESULTS

Three aspects of our results deserve experimental validation: the use of stable and volatile context representations, the use of

the one-element in-line cache and linked sends for accelerating method-lookup, and the technique of v-code to n-code translation (specifically, dynamic translation).

CONTEXT REPRESENTATIONS

The dramatic drop in reference counting overhead obtained by treating contexts specially has been documented elsewhere (e.g., [Krasner 83], section 19). We also obtain a striking efficiency improvement by allocating contexts on a stack, and by keeping their contents in execution-oriented form. Offsetting these advantages, in our implementation there is an added overhead of converting contexts between volatile/hybrid and stable forms, and of ensuring that a context accessed as a data object (either by sending it a message or directly while running a method implemented in a context class) is in stable form.

To evaluate the performance advantage of linear context allocation and volatile representation, we compared our code for allocating and deallocating contexts against code based on a hypothetical design that used the standard object representation for contexts, but did not reference-count their contents. This code appears to take about 8 times as long to execute, which would make it consume 12% of total execution time compared to 1.5% for our present code.

Less than 10% of all contexts ever exist in other than volatile form. Block contexts, which are created in stable form, and their enclosing context, which must be made hybrid so the block context can refer to it, account for two-thirds of these; nearly all of the remainder arise from an implementation detail regarding linking together fixed-size stack segments. In all of our measured examples, the time required for the conversion between the stable and volatile form was under 3% of total execution time.

If the receiver of a message is not a hybrid context, there is no overhead for making the check because it happens as part of the normal method-lookup (recall that hybrid contexts appear to be objects of a special class DummyContext with no associated methods). Only when method-lookup fails is a check made whether the receiver was actually a DummyContext. In the normal operation of the system, messages are only sent to contexts by the debugger and for cleanup during destruction of a process, so the overall impact is negligible.

As discussed above, methods associated with context classes must be translated specially, so that each reference to an instance variable checks to make sure the receiver is in stable form. The time required for this check is negligible.

IN-LINE CACHE AND LINKED SENDS

Independent measurements by us and by a group at U.C. Berkeley confirm that the one-element in-line cache is effective about 95% of the time. Measurements reported in [Krasner 83] indicate that a more conventional global cache of a reasonable size is effective about 85-90% of the time. It may be that an inline cache tends to lower the effectiveness of the global cache, since most of the lookups that would succeed in the global cache are now handled by the in-line cache, but we have no direct evidence on this point.

Adding an in-line cache to the simple translator described below improved overall performance by only 9%. On a benchmark consisting almost entirely of message sends where the in-line cache is guaranteed valid, the in-line cache only improved performance by 11%. The improvement obtained by adding an in-line cache to the optimizing translator was also about 10%. Our original hand-analysis indicated that the overall improvement should be closer to 20%, and we cannot yet account for the discrepancy. The code produced by the optimizing translator for the activate-and-return benchmark is a remarkable 47% faster than the code from the simple translator with the inline cache, suggesting that operations other than the overhead eliminated by the in-line cache still dominates overall execution time.

DYNAMIC CODE TRANSLATION

Our implementation of the Smalltalk-80 v-machine is designed to be easily switchable between different execution strategies. We have implemented a straightforward interpreter, a simple translator with almost no optimization, and a more sophisticated translator. Both translators exist in two variants, with and without the in-line cache described above. Switching between strategies simply requires relinking the implementation with a different set of modules; the price in execution speed paid for this flexibility is negligible.

Our first experiment in code translation was a simple translator that does little peephole optimization and always generates exactly 4 n-bytes per v-byte. (The latter restriction eliminated the need for the PC mapping tables described earlier.)

Our second experiment was a translator that does significant peephole optimization. The code it generates keeps the top element of the v-machine stack in a machine register whenever possible, and implements all v-instructions in-line except sends and a few rare instructions like load current context. Even arithmetic and relational operations are implemented in-line, with a call on an out-of-line routine if the operands are not small integers. The resulting code is bulky but fast.

To estimate the space required by translated methods, we have observed that the average v-method consists of 55% pointers (literal constants, message selectors, and references to global variables) and 45% v-instructions. Since our simple translator expands each v-code byte to 4 n-code bytes, the expansion factor for the method as a whole is .55+(.45*4)=2.35. The version of the simple translator that uses an in-line cache simply triples the size of the pointer area, leaving room for a cached class and n-method pointer regardless of whether the pointer is a selector or something else. This expands the total size of methods by a factor of (3*.55)+(4*.45)=3.45. The observed expansion factors for the optimizing translators appear in the table below.

We ran the standard set of Smalltalk-80 benchmarks described in [Krasner 83], section 9, using each of our five execution strategies. The normalized results are summarized in the following table:

Strategy	Space	Time
Interpreter	1.00	1.000
Simple translator, no in-líne cache	2.35	0.686
Simple translator with in-line cache	3.45	0.625
Optimizing translator, no in-line cache	5.0	0.564
Optimizing translator with in-line cache	5.03	0.515

The space figure for the optimizing translator without the inline cache could be reduced at the expense of further slowing the code down.

With respect to paging behavior in a virtual memory environment, we would like to compare the following three execution strategies: * Pure interpretation: only v-code exists; it is brought into main memory as needed.

* Static translation: n-code is generated simultaneously with v-code. Only n-code is needed at execution time. N-code is brought into memory as needed.

* Dynamic translation: n-code is kept in a cache in main memory; v-code is brought into memory for translation as needed.

Note that space taken by n-code in main memory trades off against space for data. When main memory space is needed (either for n-code or for data), we have the option of replacing data pages or discarding n-code. Unfortunately, since the work described here has been carried out in a non-virtual memory environment, we have no experimental results on this topic.

CONCLUSIONS AND RELATED WORK

Perhaps the most important observation from our research is that we have demonstrated that it is possible to implement an interactive system based on a demanding high-level language, with only a modest increase in memory requirements and without the use of any of the special hardware (special-purpose microcode, tagged memory architecture, garbage collection coprocessor) often advocated for such systems, and with resulting performance that users judge excellent. We have achieved this by careful optimization of the observed common cases and by the plentiful use of caches and other changes of representation.

A related research project [Patterson 83] is investigating a Smalltalk-80 implementation that uses only n-code, on a specially designed VLSI processor called SOAR. As discussed above, this implementation requires rewriting the compiler, debugger, and other tools that manipulate compiled code and contexts. We expect some interesting comparisons between the two approaches sometime in 1984, when the SOAR implementation becomes operational.

We believe the techniques described in this paper are applicable in varying degrees to other late-bound languages such as Lisp, and to portable V-code-based language implementations such as the Pascal P-system, but we have no current plans to investigate these other languages.

ACKNOWLEDGMENTS

Thanks are due to Mike Braca, who programmed the I/O kernel of our implementation; Bob Hagmann, who programmed the optimizing code translator and made many contributions to the design of the system; and Mark Roberts, who implemented the disk file system and virtual memory capabilities. Bob Hagmann, Dan Ingalis, and Paul McCullough contributed helpful comments on this paper. The Smalltalk-80 system itself is owed to PARC SCG. Butler Lampson gave helpful suggestions during the early project design phase.

REFERENCES

[Ammann 75] Ammann, U., Nori, Jensen, K., Nageli, H., "The Pascal (P) Compiler Implementation Notes." Institut Fur Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

[Ammann 77] Ammann, U., "On code generation in a Pascal compiler." Software Practice and Experience v7 #3, June/July 1977, pp. 391-423.

[Bell 73] Bell, J. R., "Threaded Code." Communications of the ACM, v16 (1973) pp. 370-372.

[Deutsch & Bobrow 76] Deutsch, L. P., Bobrow, D. G., "An efficient, incremental, real-time garbage collector." Communications of the ACM, October 1976. [Goldberg 83] Goldberg, A., Robson, D., "Smalltalk-80: The Language and its Implementation." Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Goldberg. A., "Smalltalk-80: The Interactive Programming Environment." Addison-Wesley, Reading, MA, 1984.

[Krasner 83] Krasner, Glenn, Ed., "Smalltalk-80: Bits of History, Words of Advice." Addison-Wesley, Reading, MA, 1983.

[Lampson 81] Lampson, B. W., Ed., "The Dorado: A High-Performance Personal Computer." Xerox PARC Report CSL-81-1, Palo Alto, CA, January 1981.

[Mitchell 71] Mitchell, J. G., "The Design and Construction of Flexible and Efficient Interactive Programming Systems," Ph.D. dissertation, 1971, NTIS AD 712-721, in Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York (1978).

[Moon 73] Moon D., Ed., Maclisp Manual pp. 3-75 to 3-77, MIT AI Laboratory Technical Report (1973).

[Moore 74] Moore, C. H., "FORTH: a New Way to Program a Computer." Astronomy and Astrophysics Supplement, #15 (1974) pp 497-511.

[Patterson 83] Patterson, D., Ed., "Smalltalk on a RISC: Architectural Investigations (Proceedings of CS 292R)." University of California, Berkeley, April 1983.

[Perkins 79] Perkins, D. R., Sites, R. I., "Machine independent Pascal code optimization." ACM SIGPLAN Notices v14 #8 (August 1979) pp. 201-207.

[Pittman 80] Pittman, T.J., "A Practical Optimizer: Zero-Address to Multi-Address Code." M.S. thesis, University of California, Santa Cruz, June 1980.

[Rau 78] Rau, B. R., "Levels of Representation of Programs and the Architecture of Universal Host Machines." Proceedings of Micro-11, Asilomar, CA, November 1978.

[Richards 75] Richards, M., "The portability of the BCPL compiler." Software, Practice and Experience v1 (1971) pp. 135-146.

[SCG 81] Software Concepts Group, special issue on Smalltalk. BYTE Magazine, volume 6, number 8, August 1981.

[XSIS 83] Masinter, L. M., Ed., "Interlisp Reference Manual." Xerox Special Information Systems, Pasadena, CA, 1983.

[Zellweger 79] Zellweger, P. T., "Machine-Independent Optimization in SOPAIPII.LA." The S-1 Project 1979 Annual Report (Chapter 8), Lawrence Livermore Laboratory (1979).

Fast Static Analysis of C++ Virtual Function Calls

David F. Bacon and Peter F. Sweeney

IBM Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598 Email: {dfb,pfs}@watson.ibm.com

Abstract

Virtual functions make code easier for programmers to reuse but also make it harder for compilers to analyze. We investigate the ability of three static analysis algorithms to improve C++ programs by resolving virtual function calls, thereby reducing compiled code size and reducing program complexity so as to improve both human and automated program understanding and analysis. In measurements of seven programs of significant size (5000 to 20000 lines of code each) we found that on average the most precise of the three algorithms resolved 71% of the virtual function calls and reduced compiled code size by 25%. This algorithm is very fast: it analyzes 3300 source lines per second on an 80 MHz PowerPC 601. Because of its accuracy and speed, this algorithm is an excellent candidate for inclusion in production C++compilers.

1 Introduction

A major advantage of object-oriented languages is abstraction. The most important language feature that supports abstraction is the dynamic dispatch of methods based on the run-time type of an object. In dynamically typed languages like Smalltalk and SELF, all dispatches are considered dynamic, and eliminating these dynamic dispatches has been essential to obtaining high performance [9, 14, 24].

C++ is a more conservatively designed language. Programmers must explicitly request dynamic dispatch by declaring a method to be virtual. C++ programs therefore suffer less of an initial performance penalty, at the cost of reduced flexibility and increased program-

Appears in the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), October 1996, San Jose, California. SIGPLAN Notices volume 31 number 10. Copyright © 1996 Association for Computing Machinery.

mer effort. However, virtual function calls still present a significant source of opportunities for program optimization.

The most obvious opportunity, and the one on which the most attention has been focused, is execution time overhead. Even with programmers specifying virtual functions explicitly, the execution time overhead of virtual function calls in C++ has been measured to be as high as 40% [16]. In addition, as programmers become familiar with the advantages of truly object-oriented design, use of virtual functions increases. The costs associated with developing software are so high that the performance penalty of virtual functions is often not sufficient to deter their use. Therefore, unless compilers are improved, the overhead due to virtual function calls is likely to increase as programmers make more extensive use of this feature.

Other researchers have shown that virtual function call resolution can result in significant performance improvements in execution time performance for C++ programs [6, 3, 16]; in this paper we concentrate on comparing algorithms for resolving virtual function calls, and investigating the reasons for their success or failure.

Another opportunity associated with virtual functions is code size reduction. For a program without virtual function calls (or function pointers), a complete call graph can be constructed and only the functions that are used need to be linked into the final program. With virtual functions, each virtual call site has multiple potential targets. Without further knowledge, all of those targets and any functions they call transitively must be included in the call graph.

As a result, object-code sizes for C++ programs have become a major problem in some environments, particularly when a small program is statically linked to a large object library. For instance, when a graphical "hello world" program is statically linked to a GUI object library, even though only a very small number of classes are actually instantiated by the program, the entire library can be dragged in.

Finally, virtual function calls present an analogous

problem for browsers and other program-understanding tools: if every potential target of a virtual function call is included in the call graph, the user is presented with a vastly larger space of object types and functions that must be comprehended to understand the meaning of the program as a whole.

In this paper, we compare three fast static analysis algorithms for resolving virtual function calls and evaluate their ability to solve the problems caused by virtual function calls in C++. We also use dynamic measurements to place an upper bound on the potential of static analysis methods, and compare the analysis algorithms against more sophisticated analyses like alias analysis. Finally, we present measurements of the speed of the analysis algorithms, which demonstrate that they are fast enough to be included in commercial-quality compilers.

1.1 Outline

Section 2 briefly describes and compares the mechanics of the three static analysis algorithms that are evaluated in this paper. Section 3 describes our benchmarks, presents the results of our measurements, and explains the reason behind the success or failure of the analysis algorithms. Section 4 describes related work, and Section 5 presents our conclusions.

2 Static Analysis

In this paper we will be comparing three static analysis algorithms, called *Unique Name* [6], *Class Hierarchy Analysis* [11, 13], and *Rapid Type Analysis* [4]. We will sometimes abbreviate them as UN, CHA, and RTA, respectively.

In this section we give a brief overview of the three algorithms, and use a small example program to illustrate the differences between them. We then briefly compare them in power to other static analyses, and discuss the interaction of type safety and analysis.

2.1 Unique Name

The first published study of virtual function call resolution for C++ was by Calder and Grunwald [6]. They were attempting to optimize C++ programs at link time, and therefore had to confine themselves to information available in the object files. They observed that in some cases there is only one implementation of a particular virtual function anywhere in the program. This

```
class A {
 public:
    virtual int foo() { return 1; };
};
class B: public A {
 public:
    virtual int foo() { return 2; };
    virtual int foo(int i) { return i+1; };
};
void main() {
    B* p = new B;
    int result1 = p->foo(1);
    int result2 = p->foo();
    A * q = p;
    int result3 = q->foo();
}
```

Figure 1: Program illustrating the difference between the static analysis methods.

can be detected by comparing the mangled names 1 of the C++ functions in the object files.

When a function has a unique name (really a unique signature), the virtual call is replaced with a direct call. While it can be used within a compiler in the same manner as the other algorithms evaluated in this paper, Unique Name has the advantage that it does not require access to source code and can optimize virtual calls in library code. However, when used at link-time, Unique Name operates on object code, which inhibits optimizations such as inlining.

Figure 1 shows a small program which illustrates the power of the various static analyses. There are three virtual calls in main(). Unique Name is able to resolve the first call (that produces result1) because there is only one virtual function called foo that takes an integer parameter - B::foo(int). There are many foo functions that take no parameters, so it can not resolve the other calls.

2.2 Class Hierarchy Analysis

Class Hierarchy Analysis [11, 13] uses the combination of the statically declared type of an object with the class hierarchy of the program to determine the set of possible targets of a virtual function call. In Figure 1, p is a

¹The *mangled name* of a function is the name used by the linker. It includes an encoding of the class and argument types to distinguish it from other identically named functions.

pointer whose static type is B*. This means that p can point to objects whose type is B or any of B's derived classes.

By combining this static information with the class hierarchy, we can determine that there are no derived classes of B, so that the only possible target of the second call (that produces result2) is int B::foo().

Class Hierarchy Analysis is more powerful than Unique Name for two reasons: it uses static information (as in Figure 1), and it can ignore identically-named functions in unrelated classes.

Class Hierarchy Analysis must have the complete program available for analysis, because if another module defines a class C derived from B that overrides foo(), then the call can not be resolved.

In the process of performing Class Hierarchy Analysis, we build a call graph for the program. The call graph includes functions reachable from main() as well as those reachable from the constructors of global-scope objects. Note that some other researchers use the term "Class Hierarchy Analysis" to denote only the resolution of virtual calls, not the building of the call graph.

2.3 Rapid Type Analysis

Rapid Type Analysis [4] starts with a call graph generated by performing Class Hierarchy Analysis. It uses information about instantiated classes to further reduce the set of executable virtual functions, thereby reducing the size of the call graph.

For instance, in Figure 1, the virtual call $q \rightarrow foo()$ (which produces result3) is not resolved by Class Hierarchy Analysis because the static type of q is A*, so the dynamic type of the object could be either A or B. However, an examination of the entire program shows that no objects of type A are created, so A::foo() can be eliminated as a possible target of the call. This leaves only B::foo().

Note that RTA must not consider instantiation of subobjects as true object instantiations: when an object of type B is created, A's constructor is called to initialize the A sub-object of B. However, the virtual function table of the contained object still points to B's foo() method.

Rapid Type Analysis builds the set of possible instantiated types *optimistically*: it initially assumes that no functions except main are called and that no objects are instantiated, and therefore no virtual call sites call any of their target functions. It traverses the call graph created by Class Hierarchy Analysis starting at main. Virtual call sites are initially ignored. When a constructor for an object is found to be callable, any of the virtual methods of the corresponding class that were left out are then traversed as well. The live portion of the call graph and the set of instantiated classes grow iteratively in an interdependent manner as the algorithm proceeds.

Rapid Type Analysis inherits the limitations and benefits of Class Hierarchy Analysis: it must analyze the complete program. Like CHA, RTA is flow-insensitive and does not keep per-statement information, making it very fast.

Rapid Type Analysis is designed to be most effective when used in conjunction with class libraries. For instance, a drawing library defines numerous objects derived from class shape, each with their own draw() method. A program that uses the library and only ever creates (and draws) squares will never invoke any of the methods of objects like circle and polygon. This will allow calls to draw() to be resolved to calls to square::draw(), and none of the other methods need to be linked into the final program. This leads to both reduced execution time and reduced code size.

Another approach to customizing code that uses class libraries is to use class slicing [23].

2.4 Other Analyses

There are several other levels of static analysis that can be performed. First, a simple local flow-sensitive analysis would be able to resolve this call:

because it will know that q points to an object of type A. Rapid Type Analysis would not resolve the call because both A and B objects are created in this program.

An even more powerful static analysis method is alias analysis, which can resolve calls even when there is intervening code which could potentially change an object's type. Alias analysis is discussed more fully in Section 4.2, with related work.

2.5 Type Safety Issues

An important limitation of CHA and RTA is that they rely on the type-safety of the programs. Continuing to use the class hierarchy from Figure 1, consider the following code fragment:

```
void* x = (void*) new B;
B* q = (B*) x;
int case1 = q->foo();
```

Despite the fact that the pointer is cast to void* and then back to B*, the program is still type-safe because we can see by inspection that the down-cast is actually to the correct type. However, if the original type is A, as in

then the program is not type-safe, and the compiler would be justified in generating code that raises an exception at the point of the virtual function call to foo(). However, because foo() is in fact defined for A, most existing compilers will simply generate code that calls A::foo(); this may or may not be what the programmer intended. If the call had instead been

int case3 =
$$q$$
->foo(666);

then the program will result in a undefined run-time behavior (most likely a segmentation fault) because A's virtual function table (VFT) does not contain an entry for foo(int).

The computation of case1 is clearly legal, and the computation of case3 is clearly illegal. In general it is not possible to distinguish the three cases statically. Unfortunately, in case2, Class Hierarchy Analysis would determine that the call was resolvable to B::foo(), which is incorrect. Rapid Type Analysis would determine that there are no possible call targets, which is correct according to the C++ language definition but different from what is done by most compilers.

Therefore, Class Hierarchy Analysis and Rapid Type Analysis either need to be disabled whenever a downcast is encountered anywhere in the program, or they can be allowed to proceed despite the downcast, with a warning printed to alert the programmer that optimization could change the results of the program if the downcasts are truly unsafe (as in case2 or case3).

We favor the latter alternative because downcasting is very common in C++ programs. This can be supplemented by pragmas or compiler switches which allow virtual function call resolution to be selectively disabled at a call site or for an entire module. We will discuss this issue further when we present the results for one of our benchmarks, lcom, which contained some unsafe code.

3 Experimental Results

In this section we evaluate the ability of the three fast static analysis methods to solve the problems that were outlined in the introduction: execution time performance, code size, and perceived program complexity. Where possible, we will use dynamic measurement information to place an upper limit on what could be achieved by perfect static analysis.

3.1 Methodology

Our measurements were gathered by reading the C++ source code of our benchmarks into a prototype C++ compiler being developed at IBM. After type analysis is complete, we build a call graph and analyze the code. Since the prototype compiler is not yet generating code reliably enough to run large benchmarks, we compile the programs with the existing IBM C++ compiler on the RS/6000, xlC. The benchmarks are traced, and their executions are simulated from the instruction trace to gather relevant execution-time statistics. We then use line number and type information to match up the call sites in the source and object code.

We used both optimized and unoptimized compiled versions of the benchmarks. The unoptimized versions were necessary to match the call sites in the source code and the object code, because optimization includes inlining, which distorts the call graph. However, existing compilers can not resolve virtual function calls, so optimization does not change the number of virtual calls, although it may change their location, especially when inlining is performed. Therefore, turning optimization (and inlining) off does not affect our results for virtual function resolution. Unoptimized code was only used for matching virtual call sites. All measurements are for optimized code unless otherwise noted.

Because our tool analyzes source code, virtual calls in library code were not available for analysis. Only one benchmark, simulate, contained virtual calls in the library code. They are not counted when we evaluate the efficacy of static analysis, since had they been available for analysis they might or might not have been resolved.

The information required by static analysis is not large, and could be included in compiled object files and libraries. This would allow virtual function calls in library code to be resolved, although it would not confer the additional benefits of inlining at the virtual call site.

3.2 Benchmarks

Table 1 describes the benchmarks we used in this study. Of the nine programs, we consider seven to be "real" programs (sched, ixx, lcom, hotwire, simulate, idl and taldict) which can be used to draw meaningful conclusions about how the analysis algorithms will perform. idl and taldict are both programs made up of production code with demo drivers;

Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	$11,\!157$	IDL specification to C++ stub-code translator
lcom	$17,\!278$	Compiler for the "L" hardware description language
hotwire	$5,\!335$	Scriptable graphical presentation builder
simulate	$6,\!672$	Simula-like simulation class library and example
idl	$30,\!288$	SunSoft IDL compiler with demo back end
taldict	$11,\!854$	Taligent dictionary benchmark
deltablue	$1,\!250$	Incremental dataflow constraint solver
$\operatorname{richards}$	606	Simple operating system simulator

Table 1: Benchmark Programs. Size is given in non-blank lines of code

the rest are all programs used to solve real problems. The remaining two benchmarks, richards and deltablue, are included because they have been used in other papers and serve as a basis for comparison and validation.

Table 2 provides an overview of the static characteristics of the programs in absolute terms. Library code is not included. The number of functions, call sites, and virtual call arcs gives a composite picture of the static complexity of the program. Live call sites are those which were executed in our traces. Non-dead virtual call sites are those call sites, both resolved and unresolved, that remained in the program after our most aggressive analysis (RTA) removed some of the dead functions and the virtual call sites they contained.

Table 3 provides an overview of the dynamic (execution time) program characteristics for optimized code. Once again, all numbers are for user code only. The number of instructions between virtual function calls is an excellent (though crude) indication of how much potential there is for speedup from virtual function resolution. Under IBM's AIX operating system and C++run-time environment a virtual function call takes 12 instructions, meaning that the user code of taldict could be sped up by a factor of two if all virtual calls are resolved (as they in fact are).

The graphs in the paper all use percentages because the absolute numbers vary so much. Tables 2 and 3 include the totals for all subsequent graphs, with the relevant figure indicated in square brackets at the top of the column.

Figure 2 is a bar graph showing the distribution of types of live call sites contained in the user code of the programs; Figure 3 shows the analogous figures for the number of dynamic calls in user code. Direct (non-virtual) method calls account for an average of 51% of the static call sites in the seven large applications, but

only 39% of the dynamic calls. Virtual method calls account for only 21% of the static call sites, but a much more significant 36% of the total dynamic calls.

Indirect function calls are used sparely except by deltablue, and pointer-to-member calls are only used by ixx, and then so infrequently that they do not appear on the bar chart.

Since non-virtual and virtual method calls are about evenly mixed, and direct (non-method) calls are less frequent, we conclude that the programs are written in a relatively object-oriented style. However, only some of the classes are implemented in a highly reusable fashion, because half of the method calls are non-virtual. The exception is taldict, with 89% of the dynamic function calls virtual: taldict uses the Taligent frameworks, which are designed to be highly re-usable. As use of C++ becomes more widespread and code reuse becomes more common, we expect that programs will become more like taldict, although probably not to such an extreme.

Note that we assume that trivially resolvable virtual function calls are implemented as direct calls, and count them accordingly throughout our measurements. That is, the call to foo() in

A a; a.foo();

is considered a direct call even if foo() is a virtual function. This is consistent with the capabilities of current production C++ compilers, but different from some related work.

Our results differ, in some cases significantly, from those reported in two previous studies of C++ virtual function call resolution [6, 3]. This would seem to indicate that there is considerable variation among applications.

Program	Code Size	Functions	Call	Live Call	Virtual	Non-Dead	Virtual
	(bytes) [6]	[7]	Sites	Sites [2]	Call Sites	V-Call Sites [4]	Call Arcs [8]
sched	99,888	237	530	184	34	33	58
ixx	$178,\!636$	1,108	$3,\!601$	767	467	399	1,752
lcom	$164,\!032$	779	2,794	$1,\!653$	458	446	$3,\!661$
hotwire	45,416	230	$1,\!204$	550	48	6	83
simulate	28,900	242	580	141	36	23	41
idl	243,748	856	$3,\!671$	882	$1,\!248$	$1,\!198$	$3,\!486$
taldict	20,516	429	783	47	79	14	116
deltablue	N.A.	103	372	201	3	3	11
richards	9,744	78	174	68	1	1	5

Table 2: Totals for static (compile-time) quantities measured in this paper. All quantities are measured for user code only (libraries linked to the program are not included). Numbers in brackets are the numbers of subsequent figures for which the column gives the total.

Program	Instrs.	Function	Virtual	Instrs. per
	Executed	Calls [3]	Calls $[5]$	Virtual Call
sched	$106,\!901,\!207$	$2,\!302,\!003$	967,789	110
ixx	$7,\!919,\!945$	$248,\!391$	$47,\!138$	168
lcom	$107,\!826,\!169$	4,210,059	1,099,317	98
hotwire	$4,\!842,\!856$	189,160	$33,\!504$	145
simulate	$1,\!230,\!305$	$57,\!537$	$10,\!848$	113
idl	776,792	$33,\!826$	$14,\!211$	55
taldict	$837,\!496,\!535$	$39,\!401,\!445$	35,060,980	23
deltablue	$10,\!492,\!752$	558,028	$205,\!100$	51
richards	$86,\!916,\!173$	2,407,782	657,900	132

Table 3: Totals for dynamic (run-time) quantities measured in this paper. All quantities are for user code only (libraries linked to the program are not included). Numbers in brackets are the numbers of subsequent figures for which the column gives the total.

Considerable additional work remains to be done for benchmarking of C++ programs. While the SPEC benchmark suite has boiled down "representative" C code to a small number of programs, it may well be that such an approach will not work with C++ because it is a more diverse language with more diverse usage patterns.

3.3 Resolution of Virtual Function Calls

When a virtual call site always calls the same function during one or more runs of the program, we say that it is *monomorphic*. If it calls multiple functions, it is *polymorphic*. If the optimizer can prove that a monomorphic call will *always* call the same function, then it can be resolved statically. Polymorphic call sites can not be resolved unless the enclosing code is cloned or type tests are inserted.

The performance of the analyses for resolving virtual function calls is shown in Figures 4 (which presents the static information for the call sites) and 5 (which presents the dynamic information for the calls in our program traces). Together with the remaining graphs they compare the performance of the three static analysis algorithms, and they all use a consistent labeling to aid in interpretation. Black is always used to label the things that could not possibly be handled by static analysis; in the case of virtual function resolution, black represents the call sites or calls that were polymorphic. White represents the region of possible opportunity for finer analysis; for virtual function resolution, this is the call sites or calls that were dynamically monomorphic but were not resolved by any of the static analysis methods we implemented. For graphs of static quantities, the diagonally striped section labels an additional region of opportunity in unexecuted code; for virtual function resolution, this is the call sites that were not resolved and were not executed at run-time. They may be dead, monomorphic, or polymorphic.

Since Class Hierarchy Analysis (CHA) resolves a superset of the virtual calls resolved by Unique Name (UN), and Rapid Type Analysis (RTA) resolves a superset of the virtual calls resolved by CHA, we show their cumulative effect on a single bar in the chart. Therefore, to see the effect of RTA, the most powerful analysis, include all the regions labeled as "resolved" (they are outlined with a thick line).

If the region of opportunity is very small, then the dynamic trace has given us a tight upper bound: we *know* that no static analysis could do much better. On the other hand, if the white region (and for static graphs, the striped region) is large, then the dynamic trace has only given us a loose upper bound: more powerful static analysis might be able to do better, or it might not.

Call sites identified as dead by Rapid Type Analysis were not counted, regardless of whether they were resolved. This was done so that the static and dynamic measurements could be more meaningfully compared, and because it seemed pointless to count as resolved a call site in a function that can never be executed. However, this has relatively little effect on the overall percentages.

Figure 5 shows that for for five out of seven of the large benchmarks, the most powerful static analysis, RTA, resolves all or almost all of the virtual function calls. In other words, in five out of seven cases, RTA does an essentially perfect job. On average, RTA resolves 71% of the dynamic virtual calls in the seven large benchmarks. CHA is also quite effective, resolving an average of 51%, while UN performs relatively poorly, resolving an average of 15% of the dynamic virtual calls.

We were surprised by the poor performance of Unique Name, since Calder and Grunwald found that Unique Name resolved an average of 32% of the virtual calls in their benchmarks. We are not sure why this should be so; possibly our benchmarks, being on average of a later vintage, contain more complex class hierarchies. UN relies on there only being a single function in the entire application with a particular signature.

Our benchmarks are surprisingly monomorphic; only two of the large applications (ixx and lcom) exhibit a significant degree of polymorphism. We do not expect this to be typical of C++ applications, but perhaps monomorphic code is more common than is generally believed.

A problem arose with one program, lcom, which is not type-safe: applying CHA or RTA generates some specious call site resolutions. We examined the program and found that many virtual calls were *potentially unsafe*, because the code used down-casts. However, most of these potentially unsafe calls *are in fact safe*, because the program uses a collection class defined to hold pointers of type void*. Usually, inspection of the code shows that the down-casts are simply being used to restore a void* pointer to the original type of the object inserted into the collection.

We therefore selectively turned off virtual function call resolution at the call sites that could not be determined to be safe; only 7% of the virtual calls that would have been resolved by static analysis were left unresolved because of this (they are counted as unresolved monomorphic calls). We feel that this is a reasonable course because a programmer trying to optimize their own program might very well choose to follow this course rather than give up on optimization altogether; readers will have to use their own judgment as to whether this would be an acceptable programming practice in their environment.

The only benchmark to use library code containing virtual calls was simulate, which uses the task library supplied with AIX. Slightly less than half of the virtual calls were made from the library code, and about half of those calls were monomorphic (and therefore potentially resolvable). We have not included virtual calls in library code in the graphs because the corresponding code was not available to static analysis.

3.3.1 Why Rapid Type Analysis Wins

Since Class Hierarchy Analysis is a known and accepted method for fast virtual function resolution, it is important to understand why RTA is able to do better.

RTA does better on four of seven programs, although for idl the improvement is minor. For ixx, RTA resolves a small number of additional static call sites (barely visible in Figure 4), which account for almost 20% of the total dynamic virtual function calls. The reason is that those calls are all to frequently executed string operations. There is a base class String with a number of virtual methods, and a derived class UniqueString, which overrides those methods. RTA determines that no UniqueString objects are created in ixx, and so it is able to resolve the virtual call sites to String methods. These call sites are in inner loops, and therefore account for a disproportionate number of the dynamic virtual calls.

RTA also makes a significant difference for taldict, resolving the remaining 19% of unresolved virtual calls. RTA is able to resolve two additional call sites because they are calls where a hash table class is calling the method of an object used to compare key values. The comparison object base class provides a default comparison method, but the derived class used in taldict overrides it. RTA finds that no instances of the base class are created, so it is able to resolve the calls.

The hotwire benchmark is a perfect example of the class library scenario: a situation in which an application is built using only a small portion of the functionality of a class library. The application itself is a simple dynamic overhead transparency generator; it uses a library of window management and graphics routines. However, it only creates windows of the root type, which can display text in arbitrary fonts at arbitrary locations. All of the dynamic dispatch occurs on redisplay of subwindows, of which there are none in this application. Therefore, all of the live virtual call sites are resolved.

3.3.2 Why Fast Static Analysis Fails

One benchmark, sched, stands out for the poor performance of all three static analysis algorithms evaluated in this paper. Only 10% of the dynamic calls are resolved, even though 30% of the static call sites are resolved, and 100% of the dynamic calls are monomorphic. Of course, a function may be monomorphic with one input but not with another. However, sched appears to actually be completely monomorphic.

The unresolved monomorphic virtual call sites are all due to one particular programming idiom: sched defines a class Base and two derived classes Derived1 and Derived2 (not their real names). Base has no data members, and defines a number of virtual functions whose implementation is always assert(false) – in other words, they will raise an exception when executed. In essence, Base is a strange sort of abstract base class.

Derived1 and Derived2 each implement a mutually exclusive subset of the methods defined by Base, and since Base has no data members, this means that these two object types are totally disjoint in functionality. It is not clear why the common base class is being used at all.

RTA determines that no objects of type Base are ever created. However, the calls to the methods of Derived1 and Derived2 are always through pointers of type Base*. Therefore, there are always two possible implementations of each virtual function: the one defined by one of the derived classes, and the one inherited from Base by the other derived class.

Depending on your point of view, this is either an example of the inability of static analysis to handle particular coding styles, or another excellent reason not to write strange code.

The other benchmark for which none of the static analyses do a very good job is 1com: 45% of the virtual calls are monomorphic but unresolved. 40% of the virtual calls are from a single unresolved call site. These calls are all through an object passed in from a single procedure, further up in the call graph. That procedure creates the object with **new**, and it is always of the same type. While it would probably not be resolved by simple flow analysis, it could be resolved by alias analysis.

What kinds of programming idioms are not amenable to fast static analysis? CHA will resolve monomorphic virtual calls for which there is only a single possible target. RTA will also eliminate monomorphic calls when only one of the possible target object types is used in the program. The kind of monomorphic calls that can't be resolved by RTA occur when multiple related object types are used independently, for instance if Square and Circle objects were each kept on their own linked list, instead of being mixed together. We call this *disjointed* polymorphism.

Disjointed polymorphism is what occurs in lcom and, in a degenerate fashion, in sched. While there are certainly situations in which it does make sense to use disjointed polymorphism, we believe it to be relatively uncommon, and this is borne out by our benchmarks. Disjointed polymorphism presents the major opportunity for alias analysis to improve upon the fast static techniques presented in this paper, since it can sometimes determine that a pointer can only point to one type of object even when multiple possible object types have been created.

3.4 Code Size

Because they build a call graph, Class Hierarchy Analysis and Rapid Type Analysis identify some functions as dead: those that are not reachable in the call graph. RTA is more precise because it removes virtual call arcs to methods of uninstantiated objects from the call graph.

Figure 6 shows the effect of static analysis on user code size. As before, white represents the region of opportunity for finer analysis – those functions that were not live during the trace and were not eliminated by static analysis.

Our measurements include only first-order effects of code size reduction due to the elimination of entire functions. There is a secondary code-size reduction caused by resolving virtual call sites, since calling sequences for direct calls are shorter than for virtual calls. We also did not measure potential code expansion (or contraction) caused by inlining of resolved call sites. Finally, due to technical problems our code size measurements are for unoptimized code, and we were not able to obtain measurements for deltablue.

On average, 42% of the code in the seven large benchmarks is not executed during our traces. Class Hierarchy Analysis eliminates an average of 24% of the code from these benchmarks, and Rapid Type Analysis gets about one percent more.

CHA and RTA do very well at reducing code size: in five of the seven large benchmarks, less than 20% of the code is neither executed nor eliminated by static analysis. Only ixx and idl contain significant portions of code that was neither executed nor eliminated (about 40%).

We were surprised to find that despite the fact that RTA does substantially better than CHA at virtual function resolution, it does not make much difference in reducing code size. Unique Name does not remove any functions because it only resolves virtual calls; it does not build a call graph.

3.5 Static Complexity

Another important advantage of static analysis is its use in programming environments and compilers. For instance, in presenting a user with a program browser, the task of understanding the program is significantly easier if large numbers of dead functions are not included, and if virtual functions that can not be reached are not included at virtual call sites.

In addition, the cost and precision of other forms of static analysis and optimization are improved when the call graph is smaller and less complex.

Figure 7 shows the effect of static analysis on eliminating functions from the call graph. This is similar to Figure 6, except that each function is weighted equally, instead of being weighted by the size of the compiled code. As we stated above, since Unique Name does not build a call graph, it does not eliminate any functions.

Once again, Class Hierarchy Analysis eliminates a large number of functions, and Rapid Type Analysis eliminates a few more.

Figure 8 shows the effect of static analysis on the number of virtual call arcs in the call graph. At a virtual call site in the call graph for a C++ program, there is an arc from the call site to each of the possible virtual functions that could be called.

Class Hierarchy Analysis removes call arcs because it eliminates functions, and so any call arcs that they contain are also removed. Rapid Type Analysis can both remove dead functions and remove virtual call arcs in live functions. For example, refer back to Figure 1 at the beginning of this paper: even though main() is a live function, RTA removes the call arc to A::foo() at the call that produces result3 because it discovers that no objects of type A are ever created.

Surprisingly, despite the large number of virtual call sites that are resolved in most programs, relatively few virtual call arcs are removed in three of the seven large benchmarks. In those programs, the virtual function resolution is due mostly to Class Hierarchy Analysis. CHA, by definition, resolves a function call when there is statically only a single possible target function at the call site. Therefore, the call site is resolved, but the call arc is not removed. On the other hand, because RTA actually removes call arcs in live functions, it may eliminate substantial numbers of call arcs, as is seen in the case of hotwire.

	Size	Analysis Time		Compile	RTA
Benchmark	(lines)	СНА	RTA	Time	Overhead
sched	5,712	1.90	1.94	921	< 0.1%
ixx	$11,\!157$	5.12	5.22	367	1.4%
lcom	$17,\!278$	6.27	6.50	218	3.0%
hotwire	5,335	2.05	2.06	160	1.3%
simulate	$6,\!672$	2.67	2.75	49	5.6%
idl	30,288	5.71	6.42	450	1.4%
taldict	$11,\!854$	1.66	1.78	45	4.0%
deltablue	$1,\!250$	0.42	0.44	18	2.4%
richards	606	0.30	0.32	9	3.6%

Table 4: Compile-Time Cost of Static Analysis (timings are in seconds on an 80 MHz PowerPC 601). Compile time is for optimized code, and includes linking. Rightmost column shows the overhead of adding RTA to the compilation process.

3.6 Speed of Analysis

We have claimed that a major advantage of the algorithms described in this paper is their speed. Table 4 shows the cost of performing the Class Hierarchy Analysis and Rapid Type Analysis algorithms on an 80 MHz PowerPC 601, a modest CPU by today's standards. The total time to compile and link the program is also included for comparison. We do not include timings for Unique Name because we implemented it on top of CHA, which would not be done in a real compiler. Since Unique Name performed poorly compared to CHA and RTA, we did not feel it was worth the extra effort of a "native" implementation.

RTA is not significantly more expensive than CHA. This is because the major cost for both algorithms is that of traversing the program and identifying all the call sites. Once this has been done, the actual analysis proceeds very quickly.

RTA analyzes an average of 3310 non-blank source lines per second, and CHA is only marginally faster. The entire 17,278-line lcom benchmark was analyzed in 6.5 seconds, which is only 3% of the time required to compile and link the code. On average, RTA took 2.4% of the total time to compile and link the program.

We expect that these timings could be improved upon significantly; our implementation is a prototype, designed primarily for correctness rather than speed. No optimization or tuning has been performed yet.

Even without improvement, 3300 lines per second is fast enough to include in a production compiler without significantly increasing compile times.

4 Related Work

4.1 Type Prediction for C++

Aigner and Hölzle [3] compared the execution time performance improvements due to elimination of virtual function calls via class hierarchy analysis and profilebased type prediction. Our work differs from theirs in that we compare three different static analysis techniques, and in that we demonstrate the ability of static analysis to reduce code size and reduce program complexity. We also use dynamic information to bound the performance of static analysis.

Type prediction has advantages and disadvantages compared with static analysis. Its advantages are that it resolves more calls, and does not rely on the typecorrectness of the program. Its disadvantages are that it requires the introduction of a run-time test; it requires profiling; and it is potentially dependent upon the input used during the profile.

Ultimately, we believe that a combination of static analysis with type prediction is likely to be the best solution.

In Aigner and Hölzle's study, excluding the trivial benchmarks deltablue and richards and weighting each program equally, Class Hierarchy Analysis resolved an average of 27% of the dynamic virtual function calls (and a median of 9%). They said they were surprised by the poor performance of CHA on their benchmarks, since others had found it to perform well. In our measurements, CHA resolved an average of 51% of the dynamic virtual calls, so it seems that there is considerable variation depending upon the benchmark suite. In fact, we got different results for the one large benchmark that
we had in common, ixx, due to a different input file and possibly a different version of the program.

Type prediction can always "resolve" more virtual calls than static analysis, because it precedes a direct call with a run-time test. Call sites resolved by static analysis do not need to perform this test, and one would therefore expect the execution time benefit from static resolution to be greater than that from type prediction. This trend is indeed evident in their execution time numbers: for only one of their benchmarks does type feedback provide more than a 3% speedup over Class Hierarchy Analysis. This is despite the fact that in all but one of the benchmarks, type prediction resolves a significantly larger number of virtual calls.

4.2 Alias Analysis for C++

The most precise, and also most expensive, proposed static method for resolving virtual function calls is to use interprocedural flow-sensitive alias analysis. Pande and Ryder [19, 18] have implemented an alias analysis algorithm for C++ based on Landi et al.'s algorithm for C [15]. This analysis is then used to drive virtual function elimination. They give preliminary results for a set of 19 benchmark programs, ranging in size from 31 to 968 lines of code.

In comparison with our RTA algorithm, which processes about 3300 lines of source code per second (on an 80 MHz PowerPC 601), the speed of their algorithm ranges from 0.4 to 55 lines of source code per second (on a Sparc-10). At this speed, alias analysis will not be practical in any normal compilation path.

We have obtained their benchmark suite; Figure 10 shows the performance of our static analysis algorithms on the 9 programs that we could execute (since their analysis is purely static, not all programs were actually executable). Of these 9, two are completely polymorphic (no resolution is possible), and two were all or almost all resolved by Rapid Type Analysis or Class Hierarchy Analysis. So for four out of nine, RTA does as well as alias analysis.

RTA resolved 33% of the virtual call sites in objects, compared to about 50% by alias analysis (for comparative data, see their paper [19]). For the remaining four (deriv1, deriv2, family, and office) fast static analysis did not resolve any virtual call sites, and significant fractions of the call sites were dynamically monomorphic. Alias analysis was able to resolve some of the virtual call sites in deriv1 and deriv2, and all of the virtual call sites in family and office. However, the latter two programs are contrived examples where aliases are deliberately introduced to objects created in the main routine. Because of the small size and unrealistic nature of the benchmarks used by Pande and Ryder, we hesitate to make any generalizations based on the results of our comparison. Two of our seven large benchmarks, sched and lcom, appear to be programs for which alias analysis could perform better than RTA. These programs make use of *disjointed polymorphism*, as discussed in Section 3.3.2.

Over all, our benchmarks and Pande and Ryder's indicate that for most programs, there is relatively little room for improvement by alias analysis over RTA. However, there are definitely cases where alias analysis will make a significant difference. The ideal solution would be to use RTA first, and only employ alias analysis when RTA fails to resolve a large number of monomorphic calls.

In a similar vein as Pande and Ryder, Carini et al. [7] have also devised an alias analysis algorithm for C++ based on an algorithm for C and Fortran [10, 5]. We are currently collaborating with them on an implementation of their algorithm within our analysis framework. This will allow a direct comparison of both the precision and the efficiency of alias analysis.

4.3 Other Work in C++

Porat et al. [21] implemented the Unique Name optimization in combination with type prediction in the IBM x1C compiler for AIX, and evaluated the results for 3 benchmark programs. Their two large benchmarks were identical to two of ours: taldict and lcom. They achieved a speedup of 1.25 on taldict and a speedup of 1.04 on lcom, using a combination of Unique Name and type prediction. Our estimates and experiments indicate that a significantly higher speedup is achievable for taldict using Rapid Type Analysis.

Calder and Grunwald [6] implemented the first virtual function resolution algorithm for C++. Their Unique Name algorithm (which might more accurately be called "Unique Signature") is very fast, since it only requires a linear scan over the method declarations in the program. Calder and Grunwald implemented Unique Name as a link-time analysis, and found it to be quite effective. With their benchmarks, it resolved anywhere from 2.9% to 70.3% of the virtual calls executed by the program. We found it to be not nearly so effective on our benchmarks, and it was significantly outperformed by Rapid Type Analysis.

Srivastava [22] developed an analysis technique with the sole object of eliminating unused procedures from C++ programs. He builds a graph starting at the root of the call graph. Virtual call sites are ignored; instead, when a constructor is reached, the referenced virtual methods of the corresponding class are added to the graph. His algorithm could also be used to resolve virtual function calls by eliminating uninstantiated classes from consideration and then using Class Hierarchy Analysis. His technique is less general than RTA because the resulting graph is not a true call graph, and can not be used as a basis for further optimization.

4.4 Other Related Work

Related work has been done in the context of other object-oriented languages like Smalltalk, SELF, Cecil, and Modula-3. Of those, Modula-3 is the most similar to C++.

Fernandez [13] implemented virtual function call elimination as part of her study on reducing the cost of opaque types in Modula-3. She essentially implemented Class Hierarchy Analysis, although only for the purpose of resolving virtual calls, and not for eliminating dead code.

Diwan et al. [12] have investigated a number of algorithms for Modula-3, including an interprocedural uni-directional flow-sensitive technique, and a "name-sensitive" technique.

For the benchmarks they studied, their more powerful techniques were of significant benefit for Modula-3, because they eliminated the NULL class as a possible target. However, when NULL is ignored (as it is in C++), in all but one case the more sophisticated analyses did no better than class hierarchy analysis. This is interesting because we found several cases in which Rapid Type Analysis was significantly better than Class Hierarchy Analysis – this may indicate that class instantiation information is more important than the flow-based information.

Because of the wide variation we have seen even among our C++ benchmarks, it seems unwise to extrapolate from Modula-3 results to C++. However, despite the difference between their and our algorithms, the basic conclusion is the same: that fast static analysis is very effective for statically typed object-oriented languages.

Dean et al. [11] studied virtual method call elimination for the pure object-oriented language Cecil, which includes support for multi-methods. They analyzed the class hierarchy as we do to determine the set of typecorrect targets of a virtual method call, and used this information to drive an intraprocedural flow analysis of the methods. Their method is not directly comparable to RTA: it uses more precise information within procedures, but performs no interprocedural analysis at all. Measured speedups for benchmarks of significant size were on the order of 25%, and code size reduction was also on the order of 25%.

There has been considerable work on type inference for dynamically typed languages [20, 8, 1, 17]. In a recent paper [2], Agesen and Hölzle showed that type inference can do as well or better than dynamic receiver prediction in the SELF compiler, and proceeded to extrapolate from these results to C++ by excluding dispatches for control structures and primitive types. However, C++ and SELF may not be sufficiently similar for such comparisons to be meaningful.

5 Conclusions

We have investigated the ability of three types of static analysis to improve C++ programs by resolving virtual function calls, reducing compiled code size, and reducing program complexity to improve both human and automated program understanding and analysis.

We have shown that Rapid Type Analysis is highly effective for all of these purposes, and is also very fast. This combination of effectiveness and speed make Rapid Type Analysis an excellent candidate for inclusion in production C++ compilers.

RTA resolved an average of 71% of the virtual function calls in our benchmarks, and ran at an average speed of 3300 non-blank source lines per second. CHA resolved an average of 51% and UN resolved an average of only 15% of the virtual calls. CHA and RTA were essentially identical for reducing code size; UN is not designed to find dead code. RTA was significantly better than CHA at removing virtual call targets.

Unique Name was shown to be relatively ineffective, and can therefore not be recommended. Both RTA and CHA were quite effective. In some cases there was little difference, in other cases RTA performed substantially better. Because the cost of RTA in both compile-time and implementation complexity is almost identical to that of CHA, RTA is clearly the best of the three algorithms.

We have also shown, using dynamic traces, that the best fast static analysis (RTA) often resolves all or almost all of the virtual function calls (in five out of the seven large benchmarks). For these programs, there is no advantage to be gained by using more expensive static analysis algorithms like flow-sensitive type analysis or alias analysis. Since these algorithms will invariably be at least one to two orders of magnitude more expensive than RTA, RTA should be used first to reduce the complexity of the program and to determine if there are significant numbers of virtual call sites left to resolve. In some cases, this will allow the expensive analysis to be skipped altogether.

Acknowledgements

We thank Michael Burke, Susan Graham, and Jim Larus for their support of our work; Harini Srinivasan and G. Ramalingam for their assistance with the development of the analyzer; Mark Wegman for his many helpful suggestions; Ravi Nair for the use of his *xtrace* system, the accompanying benchmarks, and for his technical assistance; Vance Waddle for his NARC graph display system; and Yong-Fong Lee and Mauricio Serrano for sharing their benchmark suite and their insights. We thank Gerald Aigner, Urs Hölzle, Brad Calder, and Dirk Grunwald for helpful discussions and explanations of their work.

We also thank Rob Cecco, Yee-Min Chee, Derek Inglis, Michael Karasick, Derek Lieber, Mark Mendell, Lee Nackman, Jamie Schmeiser, and the other Montana team members for their invaluable assistance with their prototype C++ compiler upon which our optimizer was built.

Finally, we thank those who provided feedback on earlier drafts of this paper: Michael Burke, Paul Carini, German Goldszmidt, Urs Hölzle, Michael Karasick, Harini Srinivasan and Kenny Zadeck.

References

- AGESEN, O. Constraint-based type inference and parametric polymorphism. In Proceedings of the First International Static Analysis Symposium (Namur, Belgium, Sept. 1994), B. Le Charlier, Ed., Springer-Verlag, pp. 78–100.
- [2] AGESEN, O., AND HÖLZLE, U. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (Austin, Tex., Oct. 1995), ACM Press, New York, N.Y., pp. 91-107.
- [3] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In Proceedings of the Tenth European Conference on Object-Oriented Programming – ECOOP'96 (Linz, Austria, July 1996), vol. 1098 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 142– 166.
- [4] BACON, D. F., WEGMAN, M., AND ZADECK, K. Rapid type analysis for C++. Tech. Rep. RC

number pending, IBM Thomas J. Watson Research Center, 1996.

- [5] BURKE, M., CARINI, P., CHOI, J.-D., AND HIND, M. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing* (Ithaca, N.Y., Aug. 1994), K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., vol. 892 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 234–250.
- [6] CALDER, B., AND GRUNWALD, D. Reducing indirect function call overhead in C++ programs. In Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages (Portland, Ore., Jan. 1994), ACM Press, New York, N.Y., pp. 397–408.
- [7] CARINI, P., HIND, M., AND SRINIVASAN, H. Type analysis algorithm for C++. Tech. Rep. RC 20267, IBM Thomas J. Watson Research Center, 1995.
- [8] CHAMBERS, C., AND UNGAR, D. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *LISP and Symbolic Computation* 4, 3 (July 1991), 283-310.
- [9] CHAMBERS, C., UNGAR, D., AND LEE, E. An efficient implementation of SELF, a dynamicallytyped object-oriented language based on prototypes. *LISP and Symbolic Computation* 4, 3 (July 1991), 243–281.
- [10] CHOI, J.-D., BURKE, M., AND CARINI, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages (Charleston, South Carolina, Jan. 1993), ACM Press, New York, N.Y., pp. 232–245.
- [11] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming - ECOOP'95* (Aarhus, Denmark, Aug. 1995),
 W. Olthoff, Ed., vol. 952 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 77-101.
- [12] DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. Simple and effective analysis of staticallytyped object-oriented programs. In *Proceedings of*

the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (San Jose, Calif., Oct. 1996), pp. 292–305.

- [13] FERNANDEZ, M. F. Simple and effective link-time optimization of Modula-3 programs. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (La Jolla, Calif., June 1995), ACM Press, New York, N.Y., pp. 103– 115.
- [14] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Proceedings of the European Conference on Object-Oriented Programming – ECOOP'91 (Geneva, Switzerland, July 1991), P. America, Ed., vol. 512 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 21–38.
- [15] LANDI, W., RYDER, B. G., AND ZHANG, S. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June 1993), ACM Press, New York, N.Y., pp. 56–67.
- [16] LEE, Y., AND SERRANO, M. J. Dynamic measurements of C++ program characteristics. Tech. Rep. STL TR 03.600, IBM Santa Teresa Laboratory, Jan. 1995.
- [17] OXHØJ, N., PALSBERG, J., AND SCHWARTZBACH, M. I. Making type inference practical. In Proceedings of the European Conference on Object-Oriented Programming – ECOOP'92 (Utrecht, Netherlands, June 1992), O. L. Madsen, Ed., vol. 615 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 329–349.
- [18] PANDE, H. D., AND RYDER, B. G. Static type determination for C++. In Proceedings of the 1994 USENIX C++ Conference (Cambridge, Mass., Apr. 1994), Usenix Association, Berkeley, Calif., pp. 85–97.
- [19] PANDE, H. D., AND RYDER, B. G. Data-flowbased virtual function resolution. In Proceedings of the Third International Static Analysis Symposium (1996), vol. 1145 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 238-254.
- [20] PLEVYAK, J., AND CHIEN, A. A. Precise concrete type inference for object-oriented languages.

In Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (Portland, OR, Oct. 1994), ACM Press, New York, N.Y., pp. 324-340.

- [21] PORAT, S., BERNSTEIN, D., FEDOROV, Y., RO-DRIGUE, J., AND YAHAV, E. Compiler optimizations of C++ virtual function calls. In Proceedings of the Second Conference on Object-Oriented Technologies and Systems (Toronto, Canada, June 1996), Usenix Association, pp. 3-14.
- [22] SRIVASTAVA, A. Unreachable procedures in objectoriented programming. ACM Letters on Programming Languages and Systems 1, 4 (December 1992), 355–364.
- [23] TIP, F., CHOI, J.-D., FIELD, J., AND RAMA-LINGAM, G. Slicing class hierarchies in C++. In Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (San Jose, Calif., Oct. 1996), pp. 179– 197.
- [24] UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HOLZLE, U. Object, message, and performance: how they coexist in Self. *Computer 25*, 10 (Oct. 1992), 53-64.







Figure 2: Static Distribution of Function Call Types

Figure 3: Classification of Dynamic Calls







Figure 4: Resolution of Possibly Live Static Callsites









Figure 7: Elimination of Functions



17



Figure 8: Elimination of Virtual Call Arcs by Static Analysis

Figure 9: Effectiveness of Type Prediction







Figure 10: Resolution of Static Callsites – Alias Analysis Benchmarks

Scalable Propagation-Based Call Graph Construction Algorithms

Frank Tip IBM T.J. Watson Research Center P.O. Box 704 Yorktown Heights, NY 10598 tip@watson.ibm.com

ABSTRACT

Propagation-based call graph construction algorithms have been studied intensively in the 1990s, and differ primarily in the number of sets that are used to approximate run-time values of expressions. In practice, algorithms such as RTA that use a single set for the whole program scale well. The scalability of algorithms such as 0-CFA that use one set per expression remains doubtful.

In this paper, we investigate the design space between RTA and 0-CFA. We have implemented various novel algorithms in the context of Jax, an application extractor for Java, and shown that they all scale to a 325,000-line program. A key property of these algorithms is that they do not analyze values on the run-time stack, which makes them efficient and easy to implement. Surprisingly, for detecting unreachable methods, the inexpensive RTA algorithm does almost as well as the seemingly more powerful algorithms. However, for determining call sites with a single target, one of our new algorithms obtains the current best tradeoff between speed and precision.

1. INTRODUCTION

A key task that is required by most approaches to wholeprogram optimization is the construction of a call graph approximation. Using a call graph, one can remove methods that are not reachable from the main method, replace dynamically dispatched method calls with direct method calls, inline methods calls for which there is a unique target, and perform more sophisticated optimizations such as interprocedural constant propagation, object inlining, and transformations of the class hierarchy. In the context of objectoriented languages with dynamic dispatch, the crucial step in constructing a call graph is to compute a conservative approximation of the set of methods that can be invoked by a given virtual (i.e., dynamically dispatched) method call.

In proceedings of OOPSLA'00, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 281–293, 2000.

Jens Palsberg Dept. of Computer Science Purdue University West Lafayette, IN 47907 palsberg@cs.purdue.edu

Call-graph construction algorithms have been studied intensively in the 1990s. While their original formulations use a variety of formalisms, most of them can be recast as setbased analyses. The common idea is to abstract an object into the name of its class, and to abstract a set of objects into the set of their classes. For any given call site e.m(), the goal is then to compute a set of class names S_e that approximates the run-time values of the receiver expression e. Once the sets S_e are determined for all expressions e, the class hierarchy can be examined to identify the methods that can be invoked.

Most call graph construction algorithms differ primarily in the *number* of sets that are used to approximate run-time values of expressions. Examples:

Number of sets used to approximate run-time values of expressions	Algorithm name
No sets	Class Hierarchy Analysis (CHA) [9, 10]
One set for the whole program	Rapid Type Analysis (RTA) [6, 5]
One set per expression	0-CFA (Control-Flow Analysis) [33, 17]
Several sets per expression	k-CFA, $k > 0$ [33, 17]

Intuitively, algorithms that use more sets compute more precise call graphs, but need more time and space to do the construction. In practice, the scalability of the algorithms at either end of the spectrum is fairly clear. The CHA and RTA algorithms at the low end of the range scale well and are widely used. The k-CFA algorithms (for k > 0) at the high end seem not to scale well at all [17]. The scalability of 0-CFA remains doubtful, mostly due to the large amounts of space required to represent the many different sets that arise. Recent work by Fähndrich et al. [14, 37] give grounds for optimism, although their recent results are obtained on a machine with 2,048 Megabytes of memory [37]. In the case of Java, another complicating factor for 0-CFA is that sets of class names need to be computed for locations on the runtime stack. Those locations are unnamed, and to facilitate 0-CFA, it seems necessary to first do a program transformation that names all the locations in some fashion, as done in various recent work [41, 38, 21]. Such transformations introduce both time and space overhead.

With the investigation of the scalability of 0-CFA still pending, our research focuses on the following questions:

- Are there interesting design points in the space between RTA and 0-CFA?
- Can we achieve better precision than RTA without analyzing values on the run-time stack?

We have implemented several novel algorithms in the context of Jax, an application extractor for Java, and shown that they all scale to a 325,000-line program. A key property of the algorithms is that they do not require simulation of the run-time stack, which makes them easy to implement and which helps efficiency. Our algorithms associate a single distinct set with each class, method, and/or field (but not each expression) in an application. Surprisingly, for detecting unreachable methods, the inexpensive RTA does almost as well as the seemingly more powerful algorithms. However, for determining call sites with a single target, one of our new algorithms obtains the current best tradeoff between speed and precision.

In summary, the results for the most precise of the new algorithms look as follows:

- The constructed call graphs tend to contain only slightly fewer method definitions (i.e., methods that have a body) when compared to RTA: up to 3.0% fewer method definitions, and 1.6% fewer method definitions on average, but in several cases significantly fewer edges (i.e., calling relationships between method definitions): up to 29.0% fewer edges, and 7.2% fewer edges on average.
- An in-depth study of the constructed call graphs revealed that the most precise of our algorithms uniquely resolves up to 26.3% of the virtual call sites that are deemed polymorphic by RTA (12.5% on average).
- Associating a distinct set of types with each *method* in a class has a significantly greater impact on precision than using a distinct set for each *field* in a class.
- The algorithms scale well: the running time is within an order of magnitude of a well-tuned RTA implementation in all cases. The most precise of our algorithms runs up to 8.3 times slower than RTA, and the correlation of this "slowdown factor" with program size appears to be weak (for the largest benchmark, our most expensive algorithm ran 5.0 times slower than RTA).
- The algorithms do not require exorbitant amounts of space. All our measurements were performed on a IBM ThinkPad 600E PC with 288 MB memory. None of our benchmarks required more than 200MB of heap space.

The remainder of this paper is organized as follows. Section 2 presents our new algorithms, and discusses their relation to several previous algorithms such as RTA. Section 3 discusses implementation issues. In Section 4, we compare the results computed by the new algorithms with those obtained with RTA. Section 5 presents related work. Finally, directions for future work are presented in Section 6.

2. THE ALGORITHMS

We will use a set-based framework to present both some existing and some new algorithms. This will enable easy comparison and help put our work in context. Figure 1 shows the relationships between four new algorithms (shown in a shaded area, and code-named CTA, MTA, FTA, and XTA) that will be presented shortly, and four well-known previous algorithms: RA (Reachability Analysis), CHA (Class Hierarchy Analysis), RTA (Rapid Type Analysis), and 0-CFA. The ordering from left to right corresponds to increased accuracy and increased cost. The increased cost stems from increased amounts of information used in resolving virtual method calls. The algorithms to the left of the new algorithms have been shown to scale well, whereas the scalability of 0-CFA remains doubtful. In Section 4, we will present results that demonstrate the scalability of the new algorithms.

2.1 **Previous Algorithms**

Since our new algorithms can be viewed as a natural "next step" with respect to previous work, we will first discuss some relevant previous algorithms. These algorithms progressively take more information into account when resolving virtual method calls.

2.1.1 Name-Based Resolution (RA)

We will begin by giving a set-constraint formulation of Reachability Analysis (RA), a simple algorithm for constructing call graphs that only takes into account the name of a method. (A slightly more advanced version of this algorithm relies on the equality of method *signatures* instead of method *names*.) Variations of RA have been presented in many places (see, e.g., [34]) and used in the context of tree-shakers for Lisp [16].

RA can be defined in terms of a set variable R (for "reachable methods") that ranges over sets of methods, and the following constraints, derived from the program text:

- 1. $main \in R$ (main denotes the main method in the program)
- 2. For each method M, each virtual call site e.m(...) occurring in M, and each method M' with name m: $(M \in R) \Rightarrow (M' \in R).$

Intuitively, the first constraint reads "the main method is reachable," and the second constraint reads "if a method is reachable, and a virtual method call e.m(...) occurs in its body, then every method with name m is also reachable." It is straightforward to show that there is a least set R that satisfies the constraints, and a solution procedure that computes that set. The reason for computing the least R that satisfies the constraints is that this maximizes the complement of R, i.e., the set of unreachable methods that can be removed safely.



Figure 1: Schematic overview of the algorithms studied in this paper, and their relationship to several previous algorithms.

2.1.2 Class Hierarchy Analysis (CHA)

We can extend the constraint system for the basic reachability analysis to also take class hierarchy information into account. The result is known as *class hierarchy analysis* (CHA) [9, 10]. We will use the notation StaticType(e) to denote the static type of the expression e, SubTypes(t) to denote the set of declared subtypes of type t, and the notation StaticLookup(C, m) to denote the definition (if any) of a method with name m that one finds when starting a static method lookup in the class C. Like RA, CHA uses just one set variable R ranging over sets of methods. The constraints:

- 1. $main \in R$ (main denotes the main method in the program)
- 2. For each method M, each virtual call site e.m(...) occurring in M, and each class $C \in SubTypes(StaticType(e))$ where StaticLookup(C, m) = M':

 $(M \in R) \ \Rightarrow \ (M' \in R).$

Intuitively, the second constraint reads: "if a method is reachable, and a virtual method call e.m(...) occurs in the body of that method, then every method with name m that is inherited by a subtype of the static type of e is also reachable."

2.1.3 Rapid Type Analysis (RTA)

We can further extend CHA to take class-instantiation information into account. The result is known as *rapid type analysis* (RTA) [6, 5]. RTA uses both a set variable R ranging over sets of methods, and a set variable S which ranges over sets of class names. The variable S approximates the set of classes for which objects are created during a run of the program. The constraints:

- 1. $main \in R$ (main denotes the main method in the program)
- 2. For each method M, each virtual call site e.m(...) occurring in M, and each class $C \in SubTypes(StaticType(e))$ where StaticLookup(C,m) = M':
 - $(M \in R) \land (C \in S) \implies (M' \in R).$

- For each method M, and for each "new C()" occurring in M:
 - $(M \in R) \ \Rightarrow \ (C \in S).$

Intuitively, the second constraint refines the corresponding constraint of CHA by insisting that $C \in S$, and the third constraint reads: "S contains the classes that are instantiated in a reachable method."

RTA is easy to implement, scales well, and has been shown to compute call graphs that are significantly more precise than those computed by CHA [6]. We are aware of several whole-program analysis systems that rely on RTA to compute call graphs (e.g., the Jax application extractor of [40].) In Section 4, we will use RTA as the baseline against which we compare the new call graph construction algorithms that we are about to present.

2.2 New Algorithms

The new algorithms use multiple set variables that range over sets of classes. We will associate these set variables with program entities such as classes, methods, and fields. The idea is that by giving each program entity a more precise "local" view of the types of objects available, call sites may be resolved more accurately.

2.2.1 Separate sets for methods and fields (XTA)

We will first present an algorithm that uses a distinct set variable S_M for each method M, and a distinct set variable S_x for each field x. We call this analysis XTA. We will use the notation ParamTypes(M) for the set of static types of the arguments of the method M (excluding method M's this pointer), and the notation ReturnType(M) for the static return type of M. We also extend the function $SubTypes(\cdot)$ to work on a set of types:

$$ubTypes(Y) = \bigcup_{y \in Y} SubTypes(y)$$

The following constraints define XTA:

S'

- 1. $main \in R$ (main denotes the main method in the program)
- 2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in$

SubTypes(StaticType(e)) where StaticLookup(C, m) = M':

$$(M \in R) \land (C \in S_M)$$

$$\Rightarrow \begin{cases} M' \in R \land \\ Sub Types(Param Types(M')) \cap S_M \subseteq S_{M'} \land \\ Sub Types(Return Type(M')) \cap S_{M'} \subseteq S_M \land \\ C \in S_{M'} \end{cases}$$

3. For each method M, and for each "new C()" occurring in M:

$$(M \in R) \Rightarrow C \in S_M$$

- 4. For each method M in which a read of a field x occurs: $(M \in R) \Rightarrow S_x \subseteq S_M$
- 5. For each method M in which a write of a field x occurs: $(M \in R) \ \Rightarrow$

$$(SubTypes(StaticType(x)) \cap S_M) \subseteq S_x$$

Intuitively, the second constraint refines the corresponding constraint of RTA by (i) insisting that objects of the target class C are available in the local set S_M associated with M, (ii) adding two inclusions that capture a flow of data from M to M', and from M' back to M, and (iii) stating that an object of type C (the "this" pointer) is available in M'. The third constraint refines the corresponding constraint of RTA by adding the class name C to just the set variable for the method M. The fourth constraint reflects a data flow from a field to a method body, and the fifth constraint reflects a data flow from a method body to a field, taking hierarchy information and creation point information into account.

2.2.2 Algorithms in the space between RTA and XTA There is a spectrum of analyses between RTA and XTA. We have experimented with the following ones:

- **CTA:** The algorithm CTA uses a distinct set variable S_C for each class C. Intuitively, the set variable S_C unifies the flow information for all methods and fields. The constraints for CTA can be obtained by adding the following constraints to the definition of XTA:
 - 1. If a class C defines a method M: $S_C = S_M$.
 - 2. If a class C defines a field x: $S_C = S_x$.
- MTA: The algorithm MTA uses a distinct set variable S_C for each class C, and a set variable S_x for every field x. Intuitively, the set variable S_C unifies the flow information for all methods (but not fields.) The constraints for MTA can be obtained by adding the following constraints to the definition of XTA:

1. If a class C defines a method M: $S_C = S_M$.

- **FTA:** The algorithm FTA uses a distinct set variable S_C for each class C, and a set variable S_M for every method M. Intuitively, the set variable S_C unifies the flow information for all fields (but not methods.) The constraints for MTA can be obtained by adding the following constraints to the definition of XTA:
 - 1. If a class C defines a field x: $S_C = S_x$.

Many other possibilities exist. For example, one could unify the sets associated with fields in the same class if they have the same type.

2.2.3 Summary

Let us now summarize the algorithms with which we have done experiments. For a given program, define:

\mathcal{C}	:	the	number	of	classes	in	the	program
---------------	---	-----	--------	----	---------	----	-----	---------

- \mathcal{M} : the number of methods in the program
- \mathcal{F} : the number of fields in the program.

In the following table, the first column gives the number of set variables used to approximate run-time values of expressions.

Number of sets	Algorithm
0	CHA
1	RTA
${\mathcal C}$	CTA
$\mathcal{C}+\mathcal{F}$	MTA
$\mathcal{C} + \mathcal{M}$	FTA
$\mathcal{M}+\mathcal{F}$	XTA

All of our new algorithms and also 0-CFA can be executed in $O(n^2 \times C)$ time, where *n* is the number of set variables [28]. We can view 0-CFA as an extension of XTA in the following way. Rather than using just one set variable for each method, 0-CFA uses one set variable for each argument and each expression that evaluates to an object, including references to objects on the run-time stack. The main problem for 0-CFA is that stack locations are unnamed in the Java virtual machine, so it seems necessary to first do a program transformation that names all the locations in some fashion, as done in various recent work [41, 38, 21].

The lattice that was shown previously in Figure 1 illustrates the relationships between the algorithms in terms of cost and accuracy. Section 5 discusses further how these algorithms compare to other algorithms.

3. IMPLEMENTATION ISSUES

We implemented several of the new algorithms of Section 2 in the context of Jax, an application extractor for Java [40]. Our implementation relies on "JikesBT" (IBM's Jikes Bytecode Toolkit)¹ for reading in the Java class files that constitute an application, and for creating an internal representation of the classes in which the string-based references of the class file format are represented by pointer references. Jax uses RTA for constructing call graphs, and our new algorithms reuse several important data structures that were previously designed for RTA. Since the algorithms of Section 2 are fairly simple, the amount of new code we had to write is only about 4000 lines.

The implementation performs the XTA algorithm in an iterative, propagation-based style. Three work-lists are associated with each program component (i.e., method or field)

¹Jikes Bytecode Toolkit is a publically available class library for manipulating Java class files. See www.alphaworks.ibm.com/tech/jikesbt.

that keep track of "processed" types that have been propagated onwards from the component to other components, "current" types that will be propagated onwards in the current iteration, and "new" types that are propagated to the component in the current iteration and that will be propagated onwards in the next iteration.

The FTA and MTA algorithms are implemented by using a shared set for all the methods and fields in a class, respectively. Note that in the case of MTA (FTA) propagations between different methods (fields) in the same class are not needed. However, once a type is propagated to a method (field) in class C, the other methods (fields) in Cstill have to be revisited because onward propagations from those methods (fields) may have to take place. Due to time constraints, we have not completed the implementation of the CTA algorithm yet.

We use a combination of array-based and hash-based data structures that allow efficient membership-test operations, element addition, and iteration through all elements. We found that it is important to make *all* of these operations very efficient. Since the propagation of elements is filtered by types of method parameters, method return types, and types of fields, it is very important to efficiently implement subtype-tests. We use an approach described in [42] that relies on associating two integers with each class, corresponding to a pre-order and a post-order traversal of the class hierarchy. Using this numbering scheme, the existence of a subclass-relationship between two classes can be determined in unit time by comparing the associated numbers.

Applying the algorithms to realistic Java applications forced us to address several pragmatic issues:

- **direct method calls.** Direct method calls can be modeled using simple set-inclusions between the sets associated with the callee and the caller.
- **arrays.** Arrays are modeled as classes with one instance field that represents all of its elements. A method m is assumed to read an element from array A if: (i) an object of type A is propagated to m, and (ii) m contains an **aaload** byte code instruction. Similarly, a method m is assumed to write to A-element if: (i) an object of type A is propagated to m, and (ii) m contains an **aaload** byte code instruction.
- exception handling. The use of exception handling may cause nontrivial flow of types between methods, since exception objects may skip several stack frames before being caught. Any approach for tracking this flow of types precisely is fraught with complexity, and—in our opinion— unlikely to be very worthwhile, since the number of types involved is likely to be small (only subtypes of java.lang.Throwable are involved), and the hierarchy of user-defined exception types is often not very large or complex. Therefore, we use a single, global set of types that represents the run-time type of all expressions in the entire program whose static type is a subtype of java.lang.Throwable, and use that set to resolve all method calls on exception objects.

stack examination. While conducting experiments, we

observed that better precision along with greater efficiency can be achieved by examining the instruction that follows a method call (or field read). If this instruction is of the form **checkcast** C, an exception is thrown unless the run-time type of the object returned by the method) is a subtype of C. In such cases, we can exclude from the types being propagated to the calling method any type that is not a subtype of C. Similarly, if the instruction that follows a method call is a stack **pop** operation, we can avoid propagating from the callee to the caller altogether. These situations occur frequently in the presence of polymorphic containers such as vectors and hash-tables.

- incomplete applications. While we have described how our algorithms construct call graphs for complete applications with a single entry point, any realistic implementation must deal with situations where applications extend classes in the standard libraries, call library methods, and override library methods that are invoked from outside the application. Our basic approach is to associate a single set of objects S_E with the "outside world" (i.e., all code outside the application). This set interacts with the other sets as follows:
 - If a method m calls a method m' outside the application, we propagate $(S_m \cap ParamTypes(m'))$ to S_E . For virtual methods, any types passed via the **this** pointer are also propagated to S_E .
 - Whenever a method m writes to a field f outside the application, we propagate (S_m ∩ Type(f)) to S_E. Read-accessing an external field causes similar flow in the opposite direction.
 - If a virtual method in the application overrides an external method m, we make the conservative assumption that the external code contains a call to m'. Our approach is to use the set S_E to determine the set of methods in the application that may be invoked by the dynamic dispatch mechanism. For each such method m', we use the parameter types and return types of m' to model the flow of objects between S_E and $S_{m'}$.

We have observed that the above scheme is in certain cases unnecessarily conservative, because objects passed to a library method do not always pollute the global set S_E . Based on these observations, our implementation incorporates two refinements to the above scheme:

- For calls to certain methods in the standard libraries, we know that propagation to the set S_E is unnecessary, because the objects passed to the method will not be the receiver of subsequent method calls. The constructor of class java.lang.Object is a prime example in this category. We have identified about 25 heavily-called library methods for which calls can be ignored.
- A separate set of objects S_C can be associated with an external class C in cases where the objects passed to methods in C only interact with the other external classes in limited ways. For example, we can associate a distinct set with class java.util.Vector. Care has to

benchmark	# classes	# methods	#fields (reference-typed)	# virtual call sites
Hanoi	44	379	232 (107)	285
Ice Browser	76	761	500 (253)	922
mBird	2,050	17,946	6739 (4284)	3,269
Cindy	468	4,449	3075 (1677)	5,085
CindyApplet	468	4,449	3075 (1677)	2,502
eSuite Sheet	588	5,590	4305 (1412)	4,459
eSuite Chart	733	8,302	5448 (2141)	8,074
javaFig 1.43	161	2,108	1526 (971)	3,482
BLOAT	282	2,677	1255 (541)	6,623
JAX 6.3	309	2,754	1252 (579)	3,836
javac	210	1,512	1107 (406)	3,621
Res. System	2,332	21,495	12487 (6334)	23,640

Table 1: Benchmark characteristics.

be taken, because objects may flow from this class to other external classes (e.g., due to a call to java.util.Vector.elements()). There are three other collection classes that we modeled similarly.

reflection and dynamic loading. Nearly all of our benchmarks use dynamic loading and reflection. Since it is impossible for a static analysis to determine which classes may be accessed using these mechanisms, we have to manually supply the analysis with information about where objects are created. Issues related to whole-program analysis in the presence of these mechanisms are discussed at length in [39].

4. **RESULTS**

For a range of benchmarks, we have measured five characteristics of the results of MTA, FTA, and XTA, with the results of RTA as a baseline:

- the number of types available per method,
- the number of reachable methods,
- the number of edges in the call graphs,
- the number monomorphic and polymorphic call sites, and
- the running times.

Of particular interest is the classification of call sites into monomorphic and polymorphic ones, and we provide a detailed study of how our algorithms improve on RTA. While the number of reachable methods decreases little, we have found significant reductions in the number of edges in the constructed call graphs for several of the benchmarks. More importantly, we found a significant increase in the number of monomorphic call sites when moving from RTA to, especially, XTA.

4.1 Benchmark characteristics

Table 1 lists the benchmark applications that we used to evaluate our algorithm, and provides a number of relevant statistics for each of them. These benchmarks cover a wide spectrum of programming styles and are publically available (except for *Mockingbird* and *Reservation System*).

Hanoi is an interactive applet version of the well-known "Towers of Hanoi" problem, and is shipped with Jax. ICE $Browser^2$ is a simple internet browser. Mockingbird (*mBird*) is a proprietary IBM tool for multi-language interoperability. It relies on, but uses only limited parts of, several large class libraries (including Swing, now part of JDK 1.2, and IBM's XML parser). $Cinderella^3$ is an interactive geometry tool used for education and self-study in schools and universities. *CindyApplet* is an applet that allows users to interactively solve geometry exercises that were created with Cinderella. It is contained in the same class file archive as Cinderella. Lotus eSuite Sheet and Lotus eSuite Chart are interactive spreadsheet and charting applets, which are examples shipped with Lotus' eSuite⁴ productivity suite (DevPack 1.5 version). $JavaFig^5$ (version 1.43 (22.02.99)) is a Java version of the xfig drawing program. $BLOAT^6$ is a byte-code optimizer developed at Purdue University. Version 6.3 of Jax itself was used as a benchmark. $javac^7$ is the SPEC JVM 98 version Sun's javac compiler. Our largest benchmark. Reservation System, is an interactive front-end for an airline, hotel, and car rental reservation system developed by an IBM customer, and consists of approximately 325,000 lines of Java source code.

Table 1 shows the number of classes, methods, and fields for each of the benchmarks. The table also shows the number of reference-typed fields in the application (i.e., fields whose type is a reference to a class), which is indicated between brackets in the "fields" column. Our system creates one

 $^{^{2}\}mathrm{See}$ www.icesoft.no.

 $^{^{3}\}mathrm{See}$ www.cinderella.de.

 $^{^{4}\}mathrm{See}$ www.esuite.lotus.com.

 $^{^5\}mathrm{See}$ tech-www.informatik.uni-hamburg.de/applets/javafig.

 $^{^6\}mathrm{See}$ www.cs.purdue.edu/homes/hosking/pjama.html.

⁷See www.specbench.org.

set for each reference-typed field that is accessed from a reached method. Hence, this number is a bound on the number of field-sets that are created. Table 1 also shows the number of virtual call sites in each benchmark. For reasons we will explain shortly, virtual calls to methods outside the application are excluded from this statistic.

4.2 Set sizes

Statistics such as the number of reached methods and the percentage of uniquely resolved method calls are the measures by which call graph construction algorithms are traditionally compared. Since such measures all depend on the number of types available in a method, it is interesting to examine the average set of types available in each method as a more "absolute" measure. Table 4.2 shows, for each of the algorithms we implemented, the total number of types (instantiated classes), the average number of types available in each method body, and the latter as a percentage of the former. In the case of RTA, which uses only one set, the average number of types.

Table 2 tells us several interesting things:

- There are only a very few cases where RTA determines a larger total number of types than the other algorithms.
- Using MTA, an average 57.8% of all types is available in each method, a roughly two-fold reduction over RTA (where all types are available in each method). FTA does much better than MTA and determines that, on average, only 15.6% of all types are available in each method. XTA is better still, with 12.3% of the types being available on average.

While these reductions in average set size are substantial, it seems that there is still room for improvement. For *Reservation System*, we compute that, on average, about 200 types are available in each method when XTA is used. This number seems high, considering that the average size of a method is in the order of 15-20 lines of source code.

4.3 Reached methods

Table 3 shows, for each of the benchmarks, the number of methods in the call graphs computed by RTA, MTA, FTA, and XTA. Also shown are the percentage reductions of MTA, FTA, and XTA relative to RTA.

These statistics do not include **abstract** methods, which do not have a body, do not call other methods, and which cannot be the target of a dynamic dispatch. The rationale for excluding **abstract** methods has to do with the following observation: In cases where a virtual method m is called, but where m cannot be the target of a dynamic dispatch, it is possible to remove the m's body and make m into an **abstract** method without affecting program behavior (in fact, this is one of the optimizations performed by Jax). Therefore, counting abstract methods as first-class citizens makes it impossible to distinguish between call graphs that contain the same set of method headers, but different sets of method implementations. One could of course provide detailed statistics that include the number of **abstract** methods as well as the number of non-**abstract** methods, but we do not consider this additional detail to be very worthwhile.

In summary, we find that:

- MTA computes call graphs with 0 to 2.3% fewer method definitions than RTA (0.6% on average),
- FTA computes call graphs with 0 to 2.7% fewer method definitions than RTA (1.4% on average), and
- XTA computes call graphs with 0 to 3.0% fewer method definitions than RTA (1.6% on average).

4.4 Call graph edges

The next measure we study is the number of *edges* in the computed call graphs. In determining this number, we count each direct call site (i.e., a call that does not involve a dynamic dispatch) as one, and each virtual call site as the number of "target" methods that may be invoked by a dynamic dispatch from that site. Multiple calls to the same method m within a method body are counted separately (although our analyses will treat each of these calls similarly).

Since we do not know whether or not classes outside the application have been instantiated, it is not possible to accurately determine the number of targets of calls to virtual methods outside the application. Therefore, in performing these measurements, any calls to methods outside the application are ignored.

The results are shown in Table 4. It is clear that several of the new algorithms do eliminate substantially more edges than RTA does. The results can be summarized as follows:

- MTA computes call graphs with 0 to 4.7% fewer edges than RTA (1.6% on average),
- FTA computes call graphs with 0.1% to 26.7% fewer edges than RTA (6.6% on average), and
- XTA computes call graphs with 0.3% to 29.0% fewer edges than RTA (7.2% on average).

4.5 Uniquely resolved call sites

One of the key goals in the optimization of object-oriented programs is to find "monomorphic" virtual call sites from which only a single method can be invoked. Such call sites can be transformed into direct calls, and subsequently inlined and optimized further.

Table 5 classifies the virtual call sites in each of the benchmark applications as "unreached" (i.e., occurring in an unreached method), "monomorphic" (i.e., having a single target), or "polymorphic" (i.e., having multiple targets). Calls to methods outside the application are ignored again, since we cannot accurately determine the number of targets in such cases. We can conclude the following from Table 5:

• The percentage of virtual call sites that is unreached varies significantly from one benchmark to another.

benchmark	RTA	MTA			FTA			ХТА		
	# types	# types	#types/meth	od (avg)	# types	#types/meth	od (avg)	#types	#types/meth	od (avg)
Hanoi	19	19	7.1	37.3%	19	3.8	20.0%	19	2.8	14.8%
Ice Browser	59	59	23.4	39.7%	59	7.1	12.0%	59	4.5	7.7%
mBird	178	178	90.0	50.6%	178	13.3	7.5%	178	11.6	6.5%
Cindy	238	237	153.2	64.6%	237	36.5	15.4%	237	28.3	11.9%
CindyApplet	105	104	64.6	62.1%	104	17.2	16.5%	104	13.9	13.4%
eSuite Sheet	174	174	96.0	55.2%	174	21.1	12.1%	174	13.8	7.9%
eSuite Chart	303	303	224.1	74.0%	303	48.6	16.0%	303	24.0	7.9%
javaFig 1.43	110	110	73.4	66.7%	110	21.2	19.3%	110	17.1	15.5%
BLOAT	209	209	163.8	78.4%	209	44.7	21.4%	209	42.4	20.3%
JAX 6.3	221	221	70.9	32.1%	221	10.0	4.5%	221	8.3	3.8%
javac	171	171	106.4	62.2%	171	39.9	23.3%	171	35.5	20.8%
Res. System	1,174	1174	833.8	71.0%	1172	228.2	19.5%	1172	200.0	17.1%
AVERAGE				57.8%			15.6%			12.3%

Table 2: Average set size per method for RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	(RTA-MTA)/RTA	(RTA-FTA)/RTA	(RTA-XTA)/RTA
Hanoi	183	179	178	178	2.2%	2.7%	2.7%
Ice Browser	644	644	643	643	0.0%	0.2%	0.2%
mBird	1,862	1,855	1,825	1,824	0.4%	2.0%	2.0%
Cindy	2,437	2,412	2,404	2,403	1.0%	1.4%	1.4%
CindyApplet	1,237	1,209	1,203	1,202	2.3%	2.7%	2.8%
eSuite Sheet	2,414	2,400	2,385	2,367	0.6%	1.2%	1.9%
eSuite Chart	4,428	4,419	4,313	4,296	0.2%	2.6%	3.0%
javaFig 1.43	1,441	1,435	1,413	1,411	0.4%	1.9%	2.1%
BLOAT	2,143	2,142	2,120	2,091	0.0%	1.1%	2.4%
JAX 6.3	1,900	1,894	1,894	1,892	0.3%	0.3%	0.4%
javac	1,366	1,366	1,366	1,366	0.0%	0.0%	0.0%
Res. System	11,232	11,227	11,204	11,201	0.0%	0.2%	0.3%
AVERAGE					0.6%	1.4%	1.6%

Table 3: Number of methods in the call graphs computed by RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	(RTA-MTA)/RTA	(RTA-FTA)/RTA	(RTA-XTA)/RTA
Hanoi	400	386	382	379	3.5%	4.5%	5.3%
Ice Browser	1,594	1,594	1,593	1,588	0.0%	0.1%	0.4%
mBird	8,061	8,036	7,772	7,760	0.3%	3.6%	3.7%
Cindy	15,457	14,729	11,331	10,967	4.7%	26.7%	29.0%
CindyApplet	5,223	4,990	4,399	4,347	4.5%	15.8%	16.8%
eSuite Sheet	7,171	7,149	7,093	7,071	0.3%	1.1%	1.4%
eSuite Chart	14,669	14,648	13,857	13,771	0.1%	5.5%	6.1%
javaFig 1.43	5,128	5,064	4,963	4,961	1.2%	3.2%	3.3%
BLOAT	19,384	18,772	16,704	16,672	3.2%	13.8%	14.0%
JAX 6.3	7,053	7,018	6,904	6,895	0.5%	2.1%	2.2%
javac	13,154	13,154	13,115	13,113	0.0%	0.3%	0.3%
Res. System	46,130	45,944	44,792	44,412	0.4%	2.9%	3.7%
AVERAGE					1.6%	6.6%	7.2%

Table 4: Number of edges in the call graphs computed by RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark		RTA			MTA			FTA			XTA	
	unreached	mono	poly									
Hanoi	34.0%	61.6%	4.4%	34.0%	62.3%	3.7%	34.0%	62.7%	3.3%	34.0%	62.7%	3.3%
Ice Browser	4.0%	91.4%	4.7%	4.0%	91.4%	4.7%	4.0%	91.4%	4.7%	4.0%	91.6%	4.5%
mBird	14.2%	73.4%	12.3%	14.2%	73.5%	12.2%	17.4%	70.8%	11.8%	17.4%	70.9%	11.7%
Cindy	49.3%	45.0%	5.7%	49.5%	45.0%	5.5%	49.4%	45.2%	5.4%	49.4%	45.5%	5.0%
CindyApplet	72.0%	24.6%	3.4%	72.1%	24.6%	3.3%	72.3%	24.4%	3.3%	72.3%	24.5%	3.2%
eSuite Sheet	28.1%	68.4%	3.5%	28.1%	68.4%	3.5%	28.1%	69.1%	2.8%	28.2%	69.1%	2.8%
eSuite Chart	13.3%	76.6%	10.1%	13.3%	76.6%	10.1%	15.7%	75.7%	8.7%	15.7%	76.0%	8.3%
javaFig 1.43	9.1%	87.1%	3.9%	9.1%	87.4%	3.6%	9.7%	87.2%	3.1%	9.7%	87.2%	3.1%
BLOAT	6.6%	82.4%	11.1%	6.6%	82.4%	11.1%	6.7%	82.5%	10.8%	7.0%	82.2%	10.8%
JAX 6.3	18.7%	75.9%	5.4%	18.9%	75.7%	5.4%	18.9%	76.6%	4.5%	18.9%	76.8%	4.3%
javac	3.0%	77.6%	19.4%	3.0%	77.6%	19.4%	3.0%	77.6%	19.4%	3.0%	77.7%	19.3%
Res. System	18.1%	72.0%	9.9%	18.1%	72.2%	9.7%	18.2%	73.1%	8.7%	18.2%	74.0%	7.9%
AVERAGE			7.8%			7.7%			7.2%			7.0%

Table 5: Classification of virtual call sites according to the RTA, MTA, FTA, and XTA algorithms, for each of the benchmarks.

For example, only 3.0% of the virtual call sites in javac are unreached, whereas 72.0% of the virtual call sites in CindyApplet are unreached.

- Restricting our attention to *reached* virtual call sites, we find that *all* of the algorithms classify a vast majority of the call sites as monomorphic. Specifically, we find that:
 - RTA classifies between 3.4% and 19.4% of all call sites as polymorphic (7.8% on average).
 - MTA classifies between 3.3% and 19.4% of all call sites as polymorphic (7.7% on average).
 - FTA classifies between 2.8% and 19.4% of all call sites as polymorphic (7.2% on average).
 - XTA classifies between 2.8% and 19.3% of all call sites as polymorphic (7.0% on average).

In summary, we can conclude that RTA does a very good job in classifying virtual call sites as monomorphic. For the benchmarks we study in this paper, only 7.8% of all virtual call sites are classified as polymorphic (on average), which leaves little room for improvement. XTA classifies an average of 7.0% of all virtual call sites as polymorphic.

4.6 Detailed comparison

Table 6 shows a detailed comparison of the call graphs constructed by RTA and MTA/FTA/XTA, for each of the benchmarks. Each call site in the RTA call graph is classified as one of the following:

- mono-to-unreached: virtual call sites that were resolved to a single target method in the RTA call graph, and that became unreached in the MTA/FTA/XTA call graphs (due to the fact that the method containing the call site in question became unreachable).
- mono-to-mono: virtual call sites that were resolved to a single target method in both the RTA and the MTA/FTA/XTA call graphs.
- **poly-to-unreached**: virtual call sites that were resolved to more than 1 target in the RTA call graph,

and that became unreached in the MTA/FTA/XTA call graphs.

- **poly-to-mono**: virtual call sites that were resolved to more than 1 target in the RTA call graph, but to a unique target in the MTA/FTA/XTA call graphs.
- **poly-to-poly**: virtual call sites that were resolved to more than 1 target in both the RTA and the MTA/FTA/XTA call graphs.

To determine how much more accurate the MTA/FTA/XTA algorithms are when compared to RTA in relative terms, we can observe the following:

- Any call sites determined to be unreachable by MTA/FTA/XTA have no impact on an application's performance. After all, they are never executed.
- Any call sites determined to be monomorphic by RTA will not be improved by a better algorithm (because they either stay monomorphic, or they become unreached).

Hence, what remains are the **poly-to-mono** and the **poly-to-poly** categories. The *ratio* between these categories reflects the *relative* improvement of MTA/FTA/XTA over RTA.

As an example, consider *Reservation System*, our largest benchmark. This application contains a total of 23,640 virtual call sites. If we subtract (i) all call sites that are determined to be monomorphic by RTA, and (ii) all call sites that are determined to be unreachable by XTA, we are left with 2,824 call sites that are determined to be polymorphic by RTA. XTA determines that 569 of these call sites are, in fact, monomorphic. Hence, XTA is capable of devirtualizing 569/2,824 = 20.1% of the call sites deemed polymorphic by RTA. If we apply this line of reasoning to Table 6, we find that:

• MTA finds a single target for up to 15.8% of the call sites deemed polymorphic by RTA (2.9% on average).

benchmark	mono->ι	inreached	mor	no->mono	p	oly->unreached		poly->mono		poly->poly
Hanoi	0	(MTA)	266	(MTA)	0	(MTA)	3	(MTA)	16	(MTA)
	0	(FTA)	266	(FTA)	0	(FTA)	5	(FTA)	14	(FTA)
	0	(XTA)	266	(XTA)	0	(XTA)	5	(XTA)	14	(XTA)
Ice Browser	0	(MTA)	877	(MTA)	0	(MTA)	0	(MTA)	45	(MTA)
	1	(FTA)	876	(FTA)	0	(FTA)	0	(FTA)	45	(FTA)
	1	(XTA)	876	(XTA)	0	(XTA)	2	(XTA)	43	(XTA)
mBird	0	(MTA)	2,807	(MTA)	0	(MTA)	4	(MTA)	458	(MTA)
	141	(FTA)	2,666	(FTA)	8	(FTA)	12	(FTA)	442	(FTA)
	141	(XTA)	2,666	(XTA)	8	(XTA)	16	(XTA)	438	(XTA)
Cindy	8	(MTA)	4,510	(MTA)	0	(MTA)	9	(MTA)	548	(MTA)
	11	(FTA)	4,507	(FTA)	0	(FTA)	24	(FTA)	543	(FTA)
	14	(XTA)	4,504	(XTA)	0	(XTA)	52	(XTA)	515	(XTA)
CindyApplet	8	(MTA)	2,185	(MTA)	3	(MTA)	12	(MTA)	294	(MTA)
	21	(FTA)	2,172	(FTA)	3	(FTA)	3	(FTA)	303	(FTA)
	25	(XTA)	2,168	(XTA)	3	(XTA)	14	(XTA)	292	(XTA)
eSuite Sheet	1	(MTA)	4,272	(MTA)	2	(MTA)	0	(MTA)	184	(MTA)
	1	(FTA)	4,272	(FTA)	2	(FTA)	45	(FTA)	139	(FTA)
	2	(XTA)	4,271	(XTA)	2	(XTA)	45	(XTA)	139	(XTA)
eSuite Chart	2	(MTA)	7,186	(MTA)	2	(MTA)	1	(MTA)	883	(MTA)
	209	(FTA)	6,979	(FTA)	32	(FTA)	100	(FTA)	754	(FTA)
	217	(XTA)	6,971	(XTA)	32	(XTA)	135	(XTA)	719	(XTA)
javaFig 1.43	1	(MTA)	3,333	(MTA)	0	(MTA)	12	(MTA)	136	(MTA)
	23	(FTA)	3,311	(FTA)	2	(FTA)	26	(FTA)	120	(FTA)
	25	(XTA)	3,309	(XTA)	2	(XTA)	26	(XTA)	120	(XTA)
BLOAT	0	(MTA)	5,838	(MTA)	0	(MTA)	2	(MTA)	783	(MTA)
	26	(FTA)	5,812	(FTA)	0	(FTA)	20	(FTA)	765	(FTA)
	58	(XTA)	5,780	(XTA)	0	(XTA)	20	(XTA)	765	(XTA)
JAX 6.3	11	(MTA)	3,571	(MTA)	0	(MTA)	2	(MTA)	252	(MTA)
	11	(FTA)	3,571	(FTA)	0	(FTA)	46	(FTA)	208	(FTA)
	11	(XTA)	3,571	(XTA)	0	(XTA)	55	(XTA)	199	(XTA)
javac	0	(MTA)	2,898	(MTA)	0	(MTA)	0	(MTA)	723	(MTA)
	0	(FTA)	2,898	(FTA)	0	(FTA)	0	(FTA)	723	(FTA)
	0	(XTA)	2,898	(XTA)	0	(XTA)	2	(XTA)	721	(XTA)
Res. System	4	(MTA)	20,797	(MTA)	2	(MTA)	48	(MTA)	2,789	(MTA)
	22	(FTA)	20,779	(FTA)	14	(FTA)	324	(FTA)	2,501	(FTA)
	23	(XTA)	20,778	(XTA)	15	(XTA)	569	(XTA)	2,255	(XTA)

Table 6: Detailed comparison of the call graphs constructed by RTA and by MTA/FTA/XTA for each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	MTA/RTA	FTA/RTA	XTA/RTA
Hanoi	0.2	1.1	1.0	1.2	5.5	5.0	6.0
Ice Browser	0.4	1.7	2.4	2.1	4.3	6.0	5.3
mBird	1.1	4.8	5.2	5.8	4.4	4.7	5.3
Cindy	2.0	10.8	8.7	9.3	5.4	4.4	4.7
CindyApplet	0.7	3.0	2.4	2.6	4.3	3.4	3.7
eSuite Sheet	1.5	5.4	6.4	6.9	3.6	4.3	4.6
eSuite Chart	7.0	33.0	24.3	19.9	4.7	3.5	2.8
javaFig 1.43	0.8	3.4	4.5	4.9	4.3	5.6	6.1
BLOAT	1.8	13.1	7.9	8.7	7.3	4.4	4.8
JAX 6.3	1.2	3.4	3.2	3.5	2.8	2.7	2.9
javac	1.2	12.1	8.9	10.0	10.1	7.4	8.3
Res. System	44.5	329.4	309.1	250.2	7.4	6.9	5.6
AVERAGE					5.3	4.9	5.0

Table 7: Running times (in seconds) of the RTA, XTA, and YTA algorithms on each of the benchmarks.

- FTA finds a single target for up to 26.3% of the call sites deemed polymorphic by RTA (9.9% on average), and
- XTA finds a single target for up to 26.3% of the call sites deemed polymorphic by RTA (12.5% on average).

4.7 **Running times**

Table 7 shows the running time for the RTA, MTA, FTA, and XTA algorithms on each of the benchmarks⁸. In summary, we found that the XTA algorithm is up to 8.3 times slower than RTA. The correlation between the slowdown factor and program size appears to be weak: XTA is only 5.0 times slower than RTA on *Reservation System*. Based on our experiences we believe that, on a large machine, our algorithms should have no problems with million-line programs.

Surprisingly, the MTA and FTA algorithms were in several instances somewhat slower than XTA. Due to time constraints, we have not been able to analyze the source of these slowdowns. A possible explanation is that the increased number of types available in a method results in additional work in resolving the call sites within that method. However, it seems unlikely that this would negate all the benefits from a decreased number of propagations between sets. We would expect an efficient implementation of MTA/FTA (e.g, using techniques by Fähndrich et al [13]) to be significantly more efficient than XTA.

4.8 Assessment

Our experiments have demonstrated that it is feasible to construct propagation-based call graph construction algorithms that use more than a single set of objects to approximate the run-time values of expressions. Regarding the precision of these algorithms, we have observed that:

- The algorithms are slightly more accurate than RTA in terms of the number of reached methods.
- In several cases, the algorithms are significantly more accurate than RTA in terms of the number of edges between the methods in the call graph.
- More importantly, the reduction in the number of edges is to a significant extent derived from call sites that are classified as polymorphic by RTA, but as monomorphic call sites by our new algorithms.

With respect to the number of sets one should use, we conclude that:

- Using a distinct set for each method in the program is useful, because it improves accuracy.
- Using a unified set to represent the fields in a class does not lead to a great loss of accuracy.

In light of the improved results of XTA over FTA in some cases, we consider the XTA algorithm the best choice. If heap space is at a premium, FTA offers reduced space consumption in exchange for a slight loss of precision. MTA seems a poor choice since it computes call graphs that have roughly the same precision as those computed by RTA, but it is more complex to implement. Although we do not have data for CTA, we know it is less accurate than MTA, and the same arguments can be made to argue why it is not a very good choice.

5. RELATED WORK

5.1 **Propagation-based algorithms**

The idea of doing a propagation-based program analysis with one set variable for each expression is well known. This so-called *monovariant* style of analysis can be done in $O(n^3)$ time where n is the number of expressions. When the goal is to construct a call graph approximation in objectoriented or functional languages, then that style of analysis is known as 0-CFA [33], and when the goal is to do pointsto analysis for C programs, then that style of analysis is often referred to as "Andersen's analysis" [3, 31]. 0-CFA has been implemented for a variety of languages, including dynamically-typed object-oriented languages [29, 28, 1], functional languages [33, 19], and statically-typed objectoriented languages, including Java [11, 38, 21]. The experience has been that the effectiveness of the approaches is language-dependent, and perhaps even programming-style dependent.

The idea of *polyvariance* is to associate more than one set variable with each expression, and thereby obtain better precision for each call site. Polyvariant analysis was pioneered by Sharir and Pnueli [32], and Jones and Muchnick [25]. In the 1990s the study of polyvariant analysis has been intensive. Well known are the k-CFA algorithms of Shivers [33], the poly-k-CFA of Jagannathan and Weeks [24], and the cartesian product algorithm of Agesen [1, 2]. A particularly simple polyvariant analysis was presented by Schmidt [30]. Frameworks for defining polyvariant analyses have been presented by Stefanescu and Zhou [36], Jagannathan and Weeks [23], and Nielson and Nielson [26]. Successful applications of polyvariant analysis include the optimizing compiler of Chambers et al [17], and of Hendren et al [12], and the partial evaluator of Consel [8]. As far as we know, these polyvariant approaches have not been tried on programs of 300,000 + lines of code.

5.2 Algorithms not based on propagation

Calder and Grunwald [7] investigated a particularly simple approach to inlining based on the *unique name* measure, that is, inlining in cases where there statically is a unique target for a given call site.

A variation of 0-CFA is the unification-based approach, also known as the equality-based approach, pioneered for call graph construction by [20], and later adapted to points-to analysis for C by Steensgaard [35]. A comparison of Andersen's analysis and Steensgaard's analysis has been presented by Shapiro and Horwitz [31]. The unification-based approach is cheaper and less precise than the 0-CFA-style approach.

⁸Measurements taken on an IBM ThinkPad 600E PC with a 300Mhz processor and 288MB of main memory. We used the Sun JDK 1.1.8 VM with the just-in-time compiler developed at the IBM Tokyo Research Laboratory [22]. None of the benchmarks required more than 200MB of heap space.

A broader comparison was given by Foster, Fähndrich, and Aiken [15]; they compared both polymorphic versus monomorphic and equality-based versus inclusion-based points-to analysis. Their main conclusion is that the monomorphic inclusion-based algorithm is a good choice because 1) it usually beats the polymorphic equality-based algorithm, 2) it is not much worse than the polymorphic inclusion-based algorithm, and 3) it is simple to implement because it avoids the complications of polymorphism.

An experimental comparison of RTA and a unification-based approach to call graph construction was carried out by De-Fouw, Grove, and Chambers [11]. Their paper presents a family of algorithms that blend propagation and unification, thereby in effect dynamically determining which set variables to unify based on how propagation proceeds. Members of the family include RTA, 0-CFA, and a number of algorithms with cost and precision in between. Our algorithms, by contrast, uses static criteria to decide which set variables are to be merged, and then performs the usual propagations between them. This is potentially a poorer choice, both for accuracy and analysis time, than the approach in [11]. Our static criterion for merging set variables stems from our desire to keep the algorithm simple, in particular by avoiding analysis of the run-time stack.

Ashley [4] also presented an algorithm that blends unification and propagation, in the setting of Scheme.

6. FUTURE WORK

A comparison of our algorithms, 0-CFA, and the variations presented by Sundaresan et al. [38] and Ishizaki et al. [21] seems to require a framework in which a program transformation names all stack locations. This would be a significant extension to our framework so it may be easier to do a comparison in the settings of [38, 21]. Questions that could be addressed by such a comparison include: 1) does 0-CFA use significantly more time and space than our algorithms for large benchmarks? 2) is the potential extra precision of 0-CFA worth the increased cost? and 3) when used in a compiler for devirtualization of monomorphic calls, does 0-CFA give significantly better speedups than our algorithms? Answers may well shed more light on which algorithm to choose. Perhaps 0-CFA needs to be a fair bit better than the other algorithms before it becomes tempting to implement the program transformation that enables 0-CFA for Java bytecodes.

While adding Hindley-Milner polymorphism seems not to be worthwhile [15], we have conducted some initial experiments with the use of data-polymorphism. The idea is well known: treat each distinct allocation site as a separate class, and keep the fields in these artificial classes distinct [27, 18]. Similarly, distinct sets may be used when methods are invoked on objects of the same type but allocated at different sites. Data-polymorphism has the potential of significantly increasing the cost (more elements have to be propagated, and the number of distinct sets may increase as well). However, accuracy may improve as well because unrelated instantiations of the same type are kept separate, thereby leading to a more precise analysis of their fields.

Acknowledgments

We are grateful to Aldo Eisma, David Grove, Tony Hosking, and especially Craig Chambers for useful discussions and feedback on drafts of this paper. The anonymous OOPSLA referees also provided many helpful suggestions.

Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265, and by IBM.

7. REFERENCES

- AGESEN, O. Constraint-based type inference and parametric polymorphism. Proceedings of the First International Static Analysis Symposium (SAS'94) (September 1994), 78–100. Springer-Verlag LNCS vol. 864.
- [2] AGESEN, O. Concrete Type Inference: Delivering Object-Oriented Applications. PhD thesis, Stanford University, December 1995. Appeared as Sun Microsystems Laboratories Technical Report SMLI TR-96-52.
- [3] ANDERSEN, L. O. Self-applicable C program specialization. In Proceedings of PEPM'92, Workshop on Partial Evaluation and Semantics-Based Program Manipulation (June 1992), pp. 54–61. (Technical Report YALEU/DCS/RR-909, Yale University).
- [4] ASHLEY, J. M. A practical and flexible flow analysis for higher-order languages. In Proceedings of POPL'96, 23nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1996), pp. 184–194.
- [5] BACON, D. F. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.
- [6] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96) (San Jose, CA, 1996), pp. 324–341. SIGPLAN Notices 31(10).
- [7] CALDER, B., AND GRUNWALD, D. Reducing indirect function call overhead in C++ programs. Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages (January 1994), 397–408.
- [8] CONSEL, C. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Proceedings of PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (1993), pp. 145–154.
- [9] DEAN, J., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. Tech. Rep. 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.
- [10] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference* on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.
- [11] DEFOUW, G., GROVE, D., AND CHAMBERS, C. Fast interprocedural class analysis. In Conference Record of the Twenty-Fifth ACM Symposium on Principles of Programming Languages (San Diego, CA, January 1998), pp. 222–236.
- [12] EMAMI, M., GHIYA, R., AND HENDREN, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of ACM* SIGPLAN 1994 Conference on Programming Language Design and Implementation (1994), pp. 242–256.

- [13] FÄHNDRICH, M., AND AIKEN, A. Program analysis using mixed term and set constraints. In *Proceedings of SAS'97*, *International Static Analysis Symposium* (1997), Springer-Verlag (*LNCS*), pp. 114–126.
- [14] FÄHNDRICH, M., FOSTER, J. S., SU, Z., AND AIKEN, A. Partial online cycle elimination in inclusion constraint graphs. In Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (1998), pp. 85–96.
- [15] FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of SAS 2000, 7th Static Analysis Symposium* (2000), J. Palsberg, Ed., pp. 175–198.
- [16] GOLDBERG, A., AND ROBSON, D. Smalltalk-80—The Language and its Implementation. Addison-Wesley, 1983.
- [17] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In Proceedings of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97) (Atlanta, GA, 1997), pp. 108–124. SIGPLAN Notices 32(10).
- [18] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In Proceedings of OOPSLA'97, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (1997), pp. 108–124. SIGPLAN Notices 32(10).
- [19] HEINTZE, N. Set-based analysis of ML programs. In Proceedings of ACM Conference on LISP and Functional Programming (1994), pp. 306–317.
- [20] HENGLEIN, F. Dynamic typing. In Proceedings of ESOP'92, European Symposium on Programming (1992), Springer-Verlag (LNCS 582), pp. 233–253.
- [21] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java just-in-time compiler. In Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00) (Minneapolis, Minnesota), 2000).
- [22] ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM SIGPLAN JavaGrande Conference* (San Francisco, CA, June 1999).
- [23] JAGANNATHAN, S., AND WEEKS, S. A unified treatment of flow analysis in higher-order languages. In Proceedings of POPL'95, 22nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1995), pp. 393–407.
- [24] JAGANNATHAN, S., AND WRIGHT, A. Effective flow analysis for avoiding run-time checks. In *Proceedings of SAS'95*, *International Static Analysis Symposium* (Glasgow, Scotland, September 1995), Springer-Verlag (*LNCS* 983).
- [25] JONES, N., AND MUCHNICK, S. A flexible approach to interprocedural data flow analysis of programs with recursive data structures. In *Ninth Symposium on Principles of Programming Languages* (1982), pp. 66–74.
- [26] NIELSON, F., AND NIELSON, H. R. Infinitary control flow analysis: A collecting semantics for closure analysis. In Proceedings of POPL'97, 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1997), pp. 332–345.
- [27] OXHØJ, N., PALSBERG, J., AND SCHWARTZBACH, M. I. Making type inference practical. In *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming* (Utrecht, The Netherlands, July 1992), Springer-Verlag (*LNCS* 615), pp. 329–349.

- [28] PALSBERG, J., AND SCHWARTZBACH, M. Object-Oriented Type Systems. John Wiley & Sons, 1993.
- [29] PALSBERG, J., AND SCHWARTZBACH, M. I. Object-oriented type inference. In Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications (Phoenix, Arizona, October 1991), pp. 146–161.
- [30] SCHMIDT, D. Natural-semantics-based abstract interpretation. In *Proceedings of SAS'95, International Static Analysis Symposium* (Glasgow, Scotland, September 1995), Springer-Verlag (*LNCS* 983).
- [31] SHAPIRO, M., AND HORWITZ, S. Fast and accurate flow-insensitive points-to analysis. In Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages (Paris, France, 1997), pp. 1–14.
- [32] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis, Theory and Applications*, S. Muchnick and N. Jones, Eds. 1981.
- [33] SHIVERS, O. Control-Flow Analysis of Higher-Order Languages. PhD thesis, CMU, May 1991. CMU-CS-91-145.
- [34] SRIVASTAVA, A. Unreachable procedures in object oriented programming. ACM Letters on Programming Languages and Systems 1, 4 (December 1992), 355–364.
- [35] STEENSGAARD, B. Points-to analysis in almost linear time. In Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages (St. Petersburg, FL, January 1996), pp. 32–41.
- [36] STEFANESCU, D., AND ZHOU, Y. An equational framework for flow analysis of higher-order functional programs. In *Proceedings of ACM Conference on LISP and Functional Programming* (1994), pp. 318–327.
- [37] SU, Z., FÄHNDRICH, M., AND AIKEN, A. Projection merging: Reducing redundancies in inclusion constraint graphs. In Proceedings of POPL'00, 27nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2000), pp. 81–95.
- [38] SUNDARESAN, V., HENDREN, L., RAZAFIMAHEFA, C., VALLÉE-RAI, R., LAM, P., GAGNON, E., AND GODIN, C. Practical virtual method call resolution for Java. In Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00) (Minneapolis, Minnesota), 2000).
- [39] SWEENEY, P. F., AND TIP, F. Extracting library-based object-oriented applications. In Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8) (November 2000). To appear. A previous version of this paper appeared as IBM Research Report RC 21596, November 1999.
- [40] TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. Practical experience with an application extractor for Java. In Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99) (Denver, CO), 1999), pp. 292–305. SIGPLAN Notices 34(10).
- [41] VALLÉ-RAI, R., GAGNON, E., HENDREN, L., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing java bytecode using the soot framework: Is it feasible? In Proceedings of CC'00, International Conference on Compiler Construction (2000), Springer-Verlag (LNCS).
- [42] VITEK, J., HORSPOOL, R. N., AND KRALL, A. Efficient type inclusion tests. In Proceedings of OOPSLA'97, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (1997), pp. 142–157. SIGPLAN Notices 32(10).

Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second¹

Nevin Heintze

Research, Agere Systems (formerly Lucent's Microelectronics Division) nch@agere.com Olivier Tardieu Ecole des Mines, Paris olivier.tardieu@mines.org

ABSTRACT

We describe the design and implementation of a system for very fast points-to analysis. On code bases of about a million lines of unpreprocessed C code, our system performs fieldbased Andersen-style points-to analysis in less than a second and uses less than 10MB of memory. Our two main contributions are a database-centric analysis architecture called compile-link-analyze (CLA), and a new algorithm for implementing dynamic transitive closure. Our points-to analysis system is built into a forward data-dependence analysis tool that is deployed within Lucent to help with consistent type modifications to large legacy C code bases.

1. INTRODUCTION

The motivation for our work is the following software maintenance/development problem: given a million+ lines of C code, and a proposed change of the form "change the type of this object (e.g. a variable or struct field) from type1 to type2", find all other objects whose type may need to be changed to ensure the "type consistency" of the code base. In particular, we wish to avoid data loss through implicit narrowing conversions. To solve this problem, we need a global data-dependence analysis that in effect performs a forward data-dependence analysis (Section 2 describes this analysis, and how it differs from other more standard dependence analyses in the literature.). A critical part of this dependence analysis is an adequate treatment of pointers: for assignments such as *p = x we need to determine what objects p could point to. This kind of aliasing analysis is commonly called points-to analysis in the literature [4]. The scalability of points-to analysis has been a subject of intensive study over the last few years [5, 8, 21, 11, 23]. However the feasibility of building interactive tools that employ some form of "sufficiently-accurate" pointer analysis on million line code-bases is still an open question.

The paper has two main contributions. The first is an archi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. *PLDI 2001 6*/01 Snowbird, Utah, USA tecture for analysis systems that utilizes ideas from indexed databases. We call this architecture compile-link-analyze (CLA), in analogy with the standard compilation process. This architecture provides a substrate on which we can build a variety of analyses (we use it to implement a number of different algorithms for Andersen-style points-to analysis, dependence analysis and a unification-style points-to analysis, all using a common database format for representing programs). It scales to large code bases and supports separate and/or parallel compilation of collections of source files. Also, its indexing structures support rapid dynamic loading of just those components of object files that are needed for a specific analysis, and moreover after reading a component we have the choice of keeping it in memory or discarding it and re-reading it if we even need it again (this is used to reduce the memory footprint of an analysis). We describe CLA in detail in Section 4, and discuss how it differs from other approaches in the literature, such as methods where separate files are locally analyzed in isolation and then the individual results are combined to analyze an entire codebase.

The second contribution is a new algorithm for implementing dynamic transitive closure (DTC). Previous algorithms in the literature for Andersen's analysis are based on a transitively closed constraint graph e.g. [4, 10, 11, 21, 23, 22]. In contrast, our algorithm is based on a pre-transitive graph i.e. we maintain the graph in a form that is not transitively closed. When information about a node is requested, we must perform a graph reachability computation (as opposed to just looking up the information at the node itself in the case of a transitively closed constraint graph). A direct implementation of the pre-transitive graph idea is impractical. We show how two optimizations - caching of reachability computations, and cycle elimination - yield an efficient algorithm. Cycle elimination has previously been employed in the context of transitively closed graph and shown to result in significant improvement [11], however in that work the cost of finding cycles is non-trivial and so completeness of cycle detection is sacrificed in order to contain its cost. However, in the pre-transitive setting, cycle detection is essentially free during graph reachability. We describe our algorithm in detail in Section 5.

Section 6 presents various measurements of the performance of our system. For the Lucent code bases for which our system is targeted, runtimes are typically less than a second (800MHz Pentium) and space utilization is about 10MB.

¹This is a substantially revised version of [16].

^{© 2001} ACM ISBN 1-58113-414-2/01/06...\$5.00

These code bases are in excess of a million lines of code (uncommented non-blank lines of source, before pre-processing). On gimp (a publicly available code base of about 440K lines), our system performs field-based Andersen-style pointsto analysis in about a second (800MHz Pentium) and uses about 12MB. We also present data to illustrate the space advantages of CLA.

2. MOTIVATING APPLICATION – DEPEN-DENCE ANALYSIS

Our points-to analysis system is built into a forward datadependence analysis tool that is deployed within Lucent to help with consistent type modifications to large legacy C code bases. The basic problem is as follows: suppose that the range of values to be stored in a variable must be increased to support additional system functionality. This may require changing the type of a variable, for example from short to int. To avoid data loss through implicit narrowing conversions, any objects that take values from the changed variable must also have their types appropriately altered. Consider the following program fragment.

```
short x, y, z, *p, v, w;
y = x;
z = y+1;
p = &v;
*p = z;
w = 1;
```

If the type of x is changed from short to int, then we may also have to change the types of y, z, v and probably p, but we do not need to change the type of w.

Given an object whose type must be changed (the *target*), we wish to find all other objects that can be assigned values from the specified object. This is a forward dependence problem, as opposed to backwards dependence used for example in program slicing [25]. Moreover it only involves data-dependencies, as opposed to both data-dependencies and control-dependencies which are needed in program slicing. Our analysis refines forward data-dependence analysis to reflect the importance of a dependency for the purposes of consistent type changes. The most important dependencies are those involving assignments such as x = y and z =y+1. On the other hand, an assignment such as z1 = !ycan be ignored, since changing the type of y has no effect on the range of values of z1, and so the type of z1 does not need to be changed. Assignments involving operations such as division and multiplication are less clear. We discuss this later in the section.

An important issue for the dependence analysis is how to treat structs. Consider the program fragment involving structs in Figure 1. If the target is the variable target, then u, w and s.x are all dependent objects. If the type of target is changed from short to int, then the types of u, w and s.x should also be changed. To effect this change to the type of s.x, we can either change the type of the x field of the struct S, or we can introduce a new struct type especially for s. The advantage of the former case is that we make minimal changes to the program. The disadvantage is that we also change the type of t.x, and this may not be

Table 1: Classification of operations.

Operations	Argument 1	Argument 2
+, -, 1, &, ^	Strong	Strong
*	Weak	Weak
% , >>, <<	Weak	None
unary: +, -	Strong	n/a
&&,	None	None
!	None	n/a

strictly necessary. However, in practice it is likely that if we have to change the type of the x field of s, then we will have to change the type of the x field of t. As a result, it is desirable to treat objects that refer to the same field in a uniform way. By "same field", we mean not just that the fields have the same name, but that they are the same field of the same struct type.

Since our ultimate use of dependence analysis is to help identify objects whose type must be changed, we are not just interested in the set of dependent objects. Rather, we need to give a user information about why one object is dependent on another. To this end, we computes the dependence chains, which identify paths of dependence between one object and another. In general there are many dependence paths between a pair of objects. Moreover, some paths are more important than others. Dependencies arising from direct assignments such as x=y are usually the most important; dependencies involving arithmetic operations x=y+1, x=y>>3, x=42%y, x=1<<y are increasingly less important. Our metric of importance is biased towards operations that are likely to preserve the shape and size of input data. Table 1 outlines a simple strong/weak/none classification that we have employed. Our analysis computes the most important path, and if there are several paths of the same importance, we compute the shortest path.

Large code bases often generate many dependent objects typically in the range 1K-100K. To help users sift through these dependent objects and determine if they are objects whose type must be changed, we prioritize them according to the importance of their underlying dependence chain. We also provide a collection of graphic user interface tools for browsing the tree of chains and inspecting the corresponding source code locations. In practice, there are often too many chains to inspect - a common scenario is that a central object that is not relevant to a code change becomes dependent (often due the context- or flow-insensitivity of the underlying analysis), and then everything that is dependent on this central object also becomes dependent. We address this issue with some additional domain knowledge: we allow the user to specify "non-targets", which are objects that the user knows are certainly not dependent on the target object. This has proven to be a very effective mechanism for focusing on the important dependencies.

3. ANDERSEN'S POINTS-TO ANALYSIS

We review Andersen's points-to analysis and introduce some definitions used in the rest of the paper. In the literature, there are two core approaches to points-to analysis, ignoring context-sensitivity and flow-sensitivity. The first approach is 1. short target; 2. struct S { short x; short y; }; 3. short u, *v, w; 4. struct S s, t; 5. v = &w;6. u = target;7. *v = u;8. s.x = w;w/short <eg1.c:3> \rightarrow u/short <eg1.c:7> \rightarrow target/short <eg1.c:6> where target/short <eg1.c:1> target/short <eg1.c:1> u/short <eg1.c:3> \rightarrow target/short <eg1.c:6> where target/short <eg1.c:1> S.x/short <eg1.c:2> \rightarrow w/short <eg1.c:6> where target/short <eg1.c:6> ...

Figure 1: A program fragment involving structs and its dependence results (the target is target).

unification-based [24]: an assignment such as x = y invokes a unification of the node for x and the node for y in the points-to graph. The algorithms for the unification-based approach typically involve union/find and have essentially linear-time complexity. The second approach is based on subset relationships: an assignment such as x = y gives rise to a subset constraint $x \supseteq y$ between the nodes x and y in the points-to graph [4]. The algorithms for the subset-based approach utilize some form of subtyping system, subset constraints or a form of dynamic transitive closure, and have cubic-time complexity.

The unification-based approach is faster and less accurate [22]. There has been considerable work on improving the performance of the subset-based approach [11, 23, 21], although the performance gap is still sizable (c.f. [23, 21] and [8]). As Das very recently observed "In spite of these efforts, Andersen's algorithm does not yet scale to programs beyond 500KLOC." [8] There has also been work on improving the accuracy of the unification-based approach by incorporating some of the directional features of the subset-based approach to produce a hybrid unification-based algorithm [8]: for a small increase in analysis time (and quadratic worst-case complexity), much of the additional accuracy of the subset-based approach can be recovered.

A Deductive Reachability Formulation

We use a context-insensitive, flow-insensitive version of the subset-based approach that is essentially the analysis due to Andersen [4]. One reason for this choice is the better accuracy of the subset-based approach over the unificationbased approach. Another reason is that users of our dependence analysis system must be able to inspect the dependence chains produced by our system (Section 2), and understand why they were produced. Subset-based approaches generate easier to understand results; unification-based approaches often introduce hard to understand "backwards" flows of information due to the use of equalities.

Previous presentations of Andersen's algorithm have used some form of non-standard type system. Our presentation uses a simple deductive reachability system. This style of analyses presentation was developed by McAllester [20]. It has also been used to describe control-flow analysis [18]. To simplify our presentation, we consider a tiny language consisting of just the operations \star and &. Expressions *e* have

$$\frac{x \longrightarrow \& y}{y \longrightarrow e} \quad (\text{if } \star x = e \text{ in } P) \tag{STAR-1}$$

$$\frac{x \longrightarrow \& y}{e \longrightarrow y} \quad (\text{if } e = \star x \text{ in } P) \tag{STAR-2}$$

$$\frac{1}{e_1 \longrightarrow e_2} \quad (\text{if } e_1 = e_2 \text{ in } P) \tag{ASSIGN}$$

$$\frac{e_1 \longrightarrow e_2 \quad e_2 \longrightarrow e_3}{e_1 \longrightarrow e_3}$$
(TRANS)

Figure 2: Deduction rules for aliasing analysis.

the form:

$$e ::= x | \star x | \& x$$

We shall assume that nested uses of \star and & are removed by a preprocessing phase. Programs are sequences of assignments of the form $e_1 = e_2$ where e_1 cannot be &x.

Given some program P, we construct deduction rules as specified in Figure 2. In the first rule, the side condition "if $\star x = e$ in P" indicates that there is an instance of this rule for each occurrence of an assignment of the form $\star x = e$ in P. The side conditions in the other rules are similarly interpreted. Intuitively, an edge $e_1 \longrightarrow e_2$ indicates that any object pointer that we can derive from e_2 is also derivable from e_1 . The first rule deals with expressions of the form $\star x$ on the left-hand-sides of assignments: it states that if there is a transition from x to & y, then add a transition from y to e, where e is the left-hand-side of the assignment. The second rule deals with expressions of the form $\star x$ on the right-hand-sides of assignments: it states that if there is a transition from x to & y, then add a transition from e to y where e is the right-hand-side of the assignment. The third rule adds a transition from e_1 to e_2 for all assignments $e_1 = e_2$ in the program, and finally, the fourth rule is just transitive closure. The core of our points-to analysis can now be stated as follows: x can point to y if we can derive $x \longrightarrow \& y$. Figure 3 contains an example program and shows how $y \to \&x$ can be derived.

Analysis of Full C

Extending this core analysis to full C presents a number of choices. Adding values such as integers is straightforward. It is also easy to deal with nested uses of \star and & through the addition of new temporary variables (we remark that

int w www.		
int at the	$z \rightarrow \& y$	(ASSIGN)
int **z;	there and from	(1007033)
z = &v:	~2 ~~7 &X	(ASSIGN)
*7 = bv.	$y \rightarrow \& x$	(from STAR-1)

Figure 3: Example program and application of deduction rules to show $y \rightarrow kx$.

considerable implementation effort is required to avoid introducing too many temporary variables). However, treating structs and unions is more complex. One possibility is, in effect, to ignore them: each declaration of a variable of struct or union type is treated as an unstructured memory location and any assignment to a field is viewed as an assignment to the entire chunk e.g. x.f is viewed as an assignment to x and the field component f is ignored. We call this the *field-independent* approach and examples include [10, 11, 22]. Another approach is to use a field-based treatment of structs such as that taken by Andersen [4]. In essence, the field-based approach collects information with each field of each struct, and so an assignment to x.f is viewed as an assignment to f and the base object x is ignored. (Note that two fields of different structs that happen to have the same name are treated as separate entities.) The following code illustrates the distinction between field-based and field-independent.

In the field-independent approach, the analysis determines that only p and q can point to &z. In the field-based approach, only p and r can point to &z. Hence, neither of these approaches strictly dominates the other in terms of accuracy. We note that while the works [10, 11, 22] are based on Andersen's algorithm [4], they in fact differ in their treatment of structs: they are field-independent whereas Andersen's algorithm is field-based¹. In Section 6, we show this choice has significant implications in practice, especially for large code bases. Our aliasing analysis uses the field-based approach, in large part because our dependence analysis is also field-based.

4. COMPILE-LINK-ANALYZE

A fundamental problem in program analysis is modularity: how do we analyze large code bases consisting of many source files? The simple approach of concatenating all of the source files into one file does not scale beyond a few thousand lines of code. Moreover, if we are to build interactive tools based on an analysis, then it is important to avoid reparsing/reprocessing the entire code base when changes are made to one or two files.

The most basic approach to this problem is to parse compilation units down to an intermediate representation, and then defer analysis to a hybrid link-analyze phase. For example, at the highest level of optimization, DEC's MIPS compiler treats the internal ucode files produced by the frontend as "object files", and then invokes a hybrid linker (uld) on the ucode files [9]. The uld "linker" simply concatenates the ucode files together into a single big ucode file and then performs analysis, optimization and code generation on this file. The advantage of this approach is it modularizes the parsing problem – we don't have to parse the entire program as one unit. Also, we can avoid re-parsing of the entire code base if one source file changes. However, it does not modularize the analysis problem – the analysis proceeds as if presented with the whole program in one file.

One common way to modularize the analysis problem is to analyze program components (at the level of functions or source files), and compute summary information that captures the results of these local analyses. Such summaries are then combined/linked together in a subsequent "globalanalysis" phase to generate results for the entire program. This idea is analogous to the construction of principle types in type inference systems. For example, assigning " $\alpha \rightarrow \alpha$ " to the identity function in a simply typed language is essentially a way of analyzing the identity function in a modular way. Uses of the identity function in other code fragments can utilize $\alpha \rightarrow \alpha$ as a summary of the behavior of the identity function, thus avoiding inspection of the original function. (Of course, full polymorphic typing goes well beyond simply analyzing code in a modular way, since it allows different type instantiations for different uses of a function akin to context-sensitive analysis - which is beyond the scope of the present discussion.)

This modular approach to analysis has a long history. According to folklore, one version of the MIPS compiler employed local analysis of separate files and then combined the local analysis results during a "linking" phase. The idea is also implicit in Aiken et. al.'s set-constraint type systems [3], and is much more explicit in Flanagan and Felleisen's componential analysis for set-based analysis [12]. Recently, the idea has also been applied to points-to analysis. Das [8] describes a hybrid unification-based points-to analysis with the following steps. First, each source file is parsed, and the assignment statements therein are used to construct a points-to graph with flow edges, which is simplified using a propagation step. The points-to graph so computed is then "serialized" and written to disk, along with a table that associates symbols and functions with nodes in the graph. The second phase reads in all of these (object) files, unifies nodes corresponding to the same symbol or function from different object files, and reapplies the propagation step to obtain global points-to information. In other words, the analysis algorithm is first applied to individual files and the internal state of the algorithm (which in this case is a pointsto graph, and symbol information) is frozen and written to a file. Then, all of these files are thawed, linked and the algorithm re-applied.

¹Strictly speaking, while Andersen's core algorithm is fieldbased, he assumes that a pre-processing phase has duplicated and renamed struct definitions so that structs whose values cannot flow together have distinct names (see Section 2.3.3 and 4.3.1 of [4]).

This means that the object files used are specific not just to a particular class of analysis (points-to analysis), but to a particular analysis algorithm (hybrid unification-based analysis), and arguably even to a particular implementation of that algorithm. The object files are designed with specific knowledge of the internal data-structures of an implementation in such a way that the object file captures sufficient information about the internal state of the implementation that this state can be reconstructed at a later stage.

The CLA Model

Our approach, which we call compile-link-analyze (CLA), also consists of a local computation, a linking and a global analysis phase. However, it represents a different set of tradeoffs from previous approaches, and redraws the boundaries of what kind of work is done in each phase. A key difference is that the first phase simply parses source files and extracts assignment statements - no actual analysis is performed - and the linking phase just links together the assignment statements. One advantage of the CLA architecture is that the first two phases remains unchanged for many different implementations of points-to analysis and even different kinds of analysis (we return to this point later). A follow-on advantage is that we can justify investing resources into optimizing the representation of the collections of assignments, because we can reuse this work in a number of different analysis implementations. In particular, we have developed a database-inspired representation of assignments and function definitions/calls/returns. This representation is compact and heavily indexed. The indexing allows relevant assignments for a specific variable to be identified in just one lookup step, and more generally, it supports a mode where the assignments needed to solve a particular analysis problem can be dynamically loaded from the database on demand.

More concretely, CLA consists of three phases. The *compile* phase parses source files, extracts assignments and function calls/returns/definitions (in what follows we just call these "assignments"), and writes an object file that is basically an indexed database structure of these basic program components. No analysis is performed yet. Complex assignments are broken down into primitive ones by introducing temporary variables. The elements of the database, which we call primitive assignments, involve variables and (typically) at most one operation.

The *link* phase merges all of the database files into one database, using the linking information present in the object files to link global symbols (the same global symbol may be referenced in many files). During this process we must recompute indexing information. The "executable" file produced has the same format as the object files, although its linking information is typically obsolete (and could be stripped).

The *analyze* phase performs the actual analysis: the linked object file is dynamically loaded on demand into the running analysis. Importantly, only those parts of the object file that are required are loaded. An additional benefit of the indexing structure of the object file is that when we have read information from the object file we can simply discard it and re-load it later if necessary (memory-mapped I/O is used to support efficient reading and re-reading of the object file). We use this feature in our implementation of Andersen's analysis to greatly reduce the memory footprint of the analysis. It allows us to maintain only a very small portion of the object file in memory.

An example source file and a partial sketch of its object file representation is given in Figure 4. These object files consist of a header section which provides an index to the remaining sections, followed by sections containing linking information, primitive assignments (including information about function calls/returns/definitions) and string information, as well as indexing information for identifying targets for the dependence analysis. The primitive assignments are contained in the dynamic section; it consists of a list of blocks, one for each object in the source program. Each block consists of information about the object (its name, type, source code location and other attributes), followed by a list of primitive assignments where this object is the source. For example, the block for z contains two primitive assignments, corresponding to the second and third assignments in the program (a very rough intuition is that whenever z changes. the primitive assignments in the block for z tell us what we must recompute).

As mentioned before, one of the goals of our work is to build infrastructure that can be used for a variety of different analysis implementations as well as different kinds of analysis. We have used our CLA infrastructure for a number of different subset-based points-to analysis implementations (including an implementation based on bit-vectors, as well as many variations of the graph-based points-to algorithm described later in this paper), and field-independent and field-based points-to analysis. The key point is that our object files do not depend on the internals of our implementation and so we can freely change the implementation details without changing the object file format. We have also used CLA infrastructure for implementing unification-based points-to analysis, and for the dependence analysis described in Section 2. Finally, we note that we can write pre-analysis optimizers as database to database transformers. In fact, we have experimented with context-sensitive analysis by writing a transformation that reads in databases and simulates context-sensitivity by controlled duplication of primitive assignments in the database - this requires no changes to code in the compile, link or analyze components of our system.

We now briefly sketch how the dependence and points-to analyses use object files. Returning to Figure 4, consider performing points-to analysis. The starting point for pointsto analysis is primitive assignments such as q = &y in the static section. Such an assignment says that y should be added to the points-to set for q. This means that the pointsto set for q is now non-empty, and so we must load all primitive assignments where q is the source. In this case, we load p = q, which imposes the constraint $p \supseteq q$. This is all we need to load for points-to analysis in this case. Now consider a dependence analysis. Suppose that the target of the dependence analysis is the variable z. We first look up "z" in the hashtable in the target section to find all variables in the object file whose name is "z" (strictly speaking, we find the object file offsets of all such variables). In this case we find just one variable. We build a data-structure to



Figure 4: Example program and sketch of its object file

say that this variable is a target of the dependence analysis. We then load the block for z, which contains the primitive assignments x = z and *p = z. Using the first assignment, we build a data-structure for x and then we load the block for x, which is empty. Using the second assignment, we find from the points-to analysis that p can point to &y, and so we build a data-structure for y and load the block for y, etc. In the end, we find that both x and y depend on z.

The compilation phase we have implemented includes more information in object files that we have sketched here. Our object files record information about the strength of dependencies (see Section 2), and also information about any operations involved in assignments. For example, corresponding to a program assignment x = y + z, we obtain two primitive assignments x = y and x = z in the database. Each would retain information about the "+" operation. Such information is critical for printing out informative dependence chains; it is also useful for other kinds of analysis that need to know about the underlying operations. We include sections that record information about constants in the program. To support advanced searches and experiment with context-sensitive analysis, we also include information for each local variable that identifies the function in which it is defined. We conjecture that our object file format can be used (or easily adapted) for any flow-insensitive analysis that computes properties about the values of variables i.e. any analysis that focuses entirely on the assignments of the program, and ignores control constructs. Examples include points-to analysis, dependence analysis, constant propagation, binding-time analysis and many variations of set-based analysis. One advantage of organizing object files using sections (much like COFF/ELF), is that new sections can be transparently added to object files in such a way that existing analysis systems do not need to be rewritten.

We conclude with a discussion of functions and function pointers. Function are handled by introducing standardized names for function arguments and returns. For example, corresponding to a function definition int f(x, y) { \dots return(z)}, we generate primitive assignments x = $f_1, y = f_2, f_{ret} = z$, where f_1, f_2, f_{ret} are respectively the standardized variables for the two arguments of f and f's return value. Similarly, corresponding to a call of the form $w = f(e_1, e_2)$, we generate primitive assignments $f_1 = e_1$, $f_2 = e_2$ and $w = f_{ret}$. These standardized names are treated as global objects, and are linked together, like other global objects, by the linker. The treatment of indirect function calls uses the same naming convention, however some of the linking of formal and actual parameters happens at analysis time. Specifically, corresponding to a function definition for g, there is an object file entry (in the block for g) that records the argument and return variables for g. Corresponding to an indirect call (*f)(x, y), we mark f as a function pointer as well as adding the primitive assignments $f_1 = x$, $f_2 = y$, etc. During analysis, if a function g is added to the points-to set for f (marked as a function pointer), then we load the record of argument and return variables for both f and g. Using this information, we add new assignments $g_1 = f_1$, $g_2 = f_2$ and $f_{ret} = g_{ret}$.

5. A GRAPH-BASED ALGORITHM FOR AN-DERSEN'S ANALYSIS

Scalability of Andersen's context-insensitive flow-insensitive points-to analysis has been a subject of much research over the last five years. One problem with Andersen's analysis is the "join-point" effect of context-insensitive flow-insensitive analysis: results from different execution paths can be joined together and distributed to the points-to sets of many variables. As a result, the points-to sets computed by the analysis can be of size O(n) where n is the size of the program; such growth is commonly encountered in large benchmarks. This can spell scalability disaster if all points-to sets are explicitly enumerated.

Aiken et. al. have addressed a variety of scaling issues for Andersen's analysis in a series of papers. Their work has included techniques for elimination of cycles in the inclusion graph [11], and projection merging to reduce redundancies in the inclusion graph [23]. All of these are in the context of a transitive-closure based algorithm, and their results show very substantial improvements over their base algorithm – with all optimizations enabled, they report analysis times of 1500s for the gimp benchmark on a SPARC Enterprise 5000 with 2GB [23].

Alternatively, context- and flow-sensitivity can be used to

reduce the effect of join-points. However the cost of these additional mechanisms can be large, and without other breakthroughs, they are unlikely to scale to millions of lines of code. Also, recent results suggest that this approach may be of little benefit for Andersen's analysis [13].

In principle, ideas from sub-transitive control-flow analysis [18] could also be applied to avoid propagation of the information from join-points. The basic idea of sub-transitive control-flow analysis is that the usual dynamic transitive closure formulation of control-flow analysis is redesigned so that the dynamic edge-adding rules are de-coupled from the transitive closure rules. This approach can lead to lineartime algorithms. However, it is currently only effective on bounded-type programs, an unreasonable restriction for C.

The main focus of our algorithm, much like that of the subtransitive approach, is on finding a way to avoid the cost of computing the full transitive closure of the inclusion graph. We begin by classifying the assignments used in the deductive reachability system given in Section 3 into three classes: (a) simple assignments, which have the form x = y, (b) base assignments, which have the form x = &y, and (c) complex assignments, which have the form x = &y or $\star x = y$. For simplity, we omit treatment of $\star x = \star y$ it can be split into $\star x = z$ and $z = \star y$ where z is a new variable. In what follows we refer to items of the form &x as lvals.

The central data-structure of our algorithm is a graph \mathcal{G} , which initially contains all information about the simple assignments and base assignments in the program. The nodes of \mathcal{G} are constructed as follows: for each variable x in the program, we introduce nodes n_x and $n_{\star x}$ (strictly speaking, we only need a node $n_{\star x}$ if there is a complex assignment of the form $y = \star x$). The initial edges of \mathcal{G} are constructed as follows: corresponding to each simple assignment x = y, there is an edge $n_x \to n_y$ from n_x to n_y . Corresponding to every node n in \mathcal{G} , there is a set of base elements, defined as follows:

 $baseElements(n_x) = \{y : x = \& y \text{ appears in } P\}$

The complex assignments, which are not represented in \mathcal{G} , are collected into a set \mathcal{C} . The algorithm proceeds by iterating through the complex assignments in \mathcal{C} and adding edges to \mathcal{G} based on the information currently in \mathcal{G} . At any point in the algorithm, \mathcal{G} represents what we explicitly know about the sets of lvals for each program variable. A major departure from previous work is that \mathcal{G} is maintained in *pretransitive form* i.e. we do not transitively close the graph. As a result, whenever we need to determine the current lvals of a specific variable, we must perform graph reachability: to find the set of lvals for variable x, we find the set of nodes reachable from n_x in zero or more steps, and compute the union of the baseElements sets for all of these nodes. We use the function getLvals (n_x) to denote this graph reachability computation for node n_x .

The process of iterating through the complex assignments in C and adding edges to G based on the information currently in G is detailed in Figure 5. Note that line 7 need only be executed once, rather than once for each iteration of the loop.

Before discussing the getLvals() function, we give some intuition on the computational tradeoffs involved in maintaining the constraint graph in pre-transitive form and computing lvals on demand. First, during its execution, the algorithm only requires the computation of lvals for some subset of the nodes in the graph. Now, of course, at the end of the algorithm, we may still have to compute all lvals for all graph nodes. However, in the presence of cycle-elimination (discussed shortly), it is typically much cheaper to compute all lvals for all nodes when the algorithm terminates than it is to do so during execution of the algorithm. Second, the pretransitive algorithm trades off traversal of edges versus flow of lvals along edges. More concretely, consider a complex assignment such as $\star x = y$, and suppose that the set of lvals for x includes $\&x_1$ and $\&x_2$. As a result of this complex assignment, we add edges from x_1 to y and x_2 to y. Now, in an algorithm based on transitive-closure, all lvals associated with y will flow back along the new edges inserted and from there back along any paths that end with x_1 or x_2 . In the pre-transitive graph, the edges are added and there is no flow of lvals. Instead, lvals are collected (when needed) by a traversal of edges. In the transitive closure case, there are O(n.E) transitive closure steps, where n is the average number of distinct lvals that flow along an edge, and E is the number of edges, versus O(E) steps per reachability computation This tradeoff favors the pre-transitive graph approach when E is small and n is large. (We remark that this is analysis is for intuition only; it is not a formal analysis, since neither the transitive closure step nor the reachability step are O(1) operations.)

We next describe getLvals(), which is the graph reachability component of the algorithm. A key part of the graph reachability algorithm is the treatment of cycles. Not only is cycle detection important for termination, but it has fundamental performance implications. The first argument of getLvals() is a graph node and the second is a list of elements that define the path we are currently exploring; toplevel calls have the form getLvals(n, nil). Each node in \mathcal{G} has a one-bit field onPath.

The function unifyNodes() merges two nodes. We implement node merging by introducing an optional skip field for each node. Two nodes n_1 and n_2 are then unified by setting $skip(n_1)$ to n_2 , and merging edge and baseElement information from n_1 into n_2 . Subsequently, whenever node n_1 is accessed, we follow its skip pointer. We use an incremental algorithm for updating graph edges to skip-nodes to their de-skipped counter-parts.

Cycle elimination was first used for points-to analysis by Fähndrich et. al [11]. In their work, the cost of finding cycles was non-trivial and so completeness of cycle detection was sacrificed in order to contain its cost. In contrast, cycle detection is essentially free in our setting during graph reachability computations. Moreover, we find almost all cycles – more precisely, we find all cycles in the parts of the graphs we traverse during graph reachability. In essence, we find the costly cycles – those that are not detected are in parts of the graph that we ignore. In other words, one of the benefits of our algorithm is that it finds more of the important cycles and it does so more cheaply.

```
getLvals(n, path) {
                                                              1.
                                                                      if (onPath(n)) { /* we have a cycle */
                                                              2.
      /* The Iteration Algorithm */
                                                                        foreach n' in path, unifyNode(n', n);
                                                              3.
1.
     do f
                                                              4.
                                                                        return(emptySet);
2.
       nochange = true;
                                                                      } else { /* explore new node n */
                                                              5.
З.
        for each complex assignment *x = y in C
                                                              6.
                                                                        onPath(n) = 1;
4.
          for each &z in getLvals(n_x)
                                                              7.
                                                                        lvals = emptySet;
5.
            add an edge n_z \rightarrow n_y to \mathcal{G};
                                                              8.
                                                                        path = cons(n, path);
6.
        for each complex assignment x = *y in C
                                                              10.
                                                                           lvals = union(lvals, baseElements(n);
          add an edge n_x \rightarrow n_{\star y};
7.
                                                                        foreach n' such that there is an edge from n to n'
                                                              11.
8.
          for each \&z in getLvals(n<sub>y</sub>)
                                                              12.
                                                                          lvals = union(lvals, getLvals(n, path));
9.
            add an edge n_{*y} \rightarrow n_z
                                                              13.
                                                                        onPath(n) = 0;
10.
     } until nochange
                                                              14.
                                                                        return(lvals);
                                                              15.
                                                                   }
```

Figure 5: The Pre-Transitive Graph Algorithm for Points-to Analysis

This completes the basic description of our algorithm. We conclude with a number of enhancements to this basic algorithm. First, and most important, is a caching of reachability computations. Each call to getLvals() first checks to see if the lvals have been computed for the node during the current iteration of the iteration algorithm; if so, then the previous lvals are returned, and if not, then they are recomputed (and stored in the node). Note that this means we might use "stale" information; however if the information is indeed stale, the nochange flag in the iteration algorithm will ensure we compute another iteration of the algorithm using fresh information. Second, the graph edges are maintained in both a hash table and a per-node list so that it is fast to determine whether an edge has been previously added and also to iterate through all of the outgoing edges from a node. Third, since many lval sets are identical, a mechanism is implemented to share common lvals set. Such sets are implemented as ordered lists, and are linked into a hash table, based on set size. When a new lval set is created, we check to see if it has been previously created. This table is flushed at the beginning of each pass through the complex assignments. Fourth, lines 4-5 and lines 8-9 of Figure 5 are changed so that instead of iterating over all lvals in $getLvals(n_y)$, we iterate over all nodes in $getLvalsNodes(n_y)$. Conceptually, the function getLvalsNodes() returns all of the de-skipped nodes corresponding to the lvals computed by getLvals(); however it can be implemented more efficiently.

From the viewpoint of performance, the two most significant elements of our algorithm are cycle elimination and the caching of reachability computations. We have observed a slow down by a factor in excess of > 50K for gimp (45,000s c.f. 0.8s user time) when both of these components of the algorithm are turned off.

6. **RESULTS**

Our analysis system is implemented using a mix of ML and C. The compile phase is implemented in ML using the ckit frontend[6]. The linker and the analyzer are implemented in C. Our implementation deals with full C including structs, unions, arrays and function call/return (including indirect calls). Support for many of these features is based on simple syntactic transformations in the compile phase. The field-based treatment of structs is implemented as follows: we generate a new variable for each field f of a struct definition, and then map each access of that field to the variable. Our treatment of arrays is index-independent (we essentially ignore the index component of sub expressions). The bench-

marks we use are described in Table 2. The first six benchmarks were obtained from the authors of [21], and the lines of code reported for these are the number of lines of nonblank, non-# lines in each case. We do not currently have accurate source line counts for these benchmarks. The seventh benchmark was obtained from the authors of [23]. The last benchmark is the Lucent code base that is the main target of our system (for proprietary reasons, we have not included all informations on this benchmark). For each benchmark, we also measure the size of the preprocessed code in bytes, the size of the object files produced by the analysis (compiler + linker, also in bytes), and the number of primitive assignments in the object files – the five kinds of assignments allowed in our intermediate language.

We remark that line counts are only a very rough guide to program size. Source code is misleading for many reasons. For instance, macro expansion can considerably increase the amount of work that must be performed by an analysis. Preprocessed code is misleading because many extraneous extern declarations are included as the result of generic system include files. Moreover, these system include files can vary considerably in size from system to system. AST node counts of preprocessed code are a better measure of complexity because they de-emphasize the effects of coding style; however there is no agreed upon notion of AST nodes, and AST nodes might still be inflated by unnecessary declarations generated by rampant include files. Counts of primitive assignments may be a more robust measure.

Results from these benchmarks are included in Table 3. These results measure analysis where (a) each static occurrence of a memory allocation primitive (malloc, calloc, etc.) is treated as a fresh location, and (b) we ignore constant strings. This is the default setup we use for points-to and dependence analysis. The first column of Table 3 represents the count of program objects (variables and fields) for which we compute non-empty pointer sets; it does not include any temporary variables introduced by the analysis. The second column represents the total sizes of the points-to sets for all program objects. The third and fourth columns give wall-clock time and user time in seconds respectively, as reported by /bin/time using a single processor of a two processor Pentium 800MHz machine with 2GB of memory running Linux². The fifth column represents space utiliza-

²Red Hat Linux release 6.2 (Piglet)

VA Linux release 6.2.3 07/28/00 b1.1 P2

Kernel 2.2.14-VA.5.1smp on a 2-processor i686.

	LOC	LOC	preproc.	object	program		a	ssignmen	ts	
	(source)	(preproc.)	size	size	variables	x = y	x = & y	*x = y	*x = *y	x = *y
nethack	-	44.1K	1.4MB	0.7MB	3856	9118	1115	30	34	105
burlap	-	$74.6 \mathrm{K}$	2.4MB	1.4MB	6859	14202	1049	1160	714	1897
vortex	-	170.3K	7.7MB	2.6MB	11395	24218	7458	353	231	1866
emacs	-	93.5K	40.2MB	2.6MB	12587	31345	3461	614	154	1029
povray	-	175.5K	68.1MB	3.1MB	12570	29565	4009	2431	1190	3085
gcc	-	199.8K	69.0MB	4.4MB	18749	62556	3434	1673	585	1467
gimp	440K	7486.7 K	201.6MB	27.2MB	131552	303810	25578	5943	2397	6428
lucent	1.3M	-	-	20.1MB	96509	270148	72355	1562	991	3989

Table 2: Benchmarks

tion in MB, obtained by summing static data and text sizes (reported by /bin/size), and dynamic allocation (as reported by malloc_stats()). We note that for the lucent benchmark - the target code base of our system - we see total wall-clock times of about half a second, and space utilization of under 10MB.

The last three columns explain why the space utilizations are so low: these columns respectively show the number of primitive assignments maintained "in-core", the number loaded during the analysis, and the total number of primitive assignments in the object file. Note that only primitive assignments relevant to aliasing analysis are loaded (e.g. nonpointer arithmetic assignments are usually ignored). Recall that once we have loaded a primitive assignment from an object file and used it, we can discard it, or keep it in memory for future use. Our discard strategy is: discard assignments x = y and x = & y, but maintain all others. These numbers demonstrate the effectiveness of the load-on-demand and load-and-throw-away strategies supported by the CLA architecture.

Table 4 studies the effect of changing the baseline system. The first group of columns represents the baseline and is just a repeat of information from Table 3. The second group shows the effect of changing the underlying treatment of structs from field-based to field-independent. We caution that these results are very preliminary, and should not be interpreted as a conclusive comparison of field-based and field-independent. In particular, there are a number of opportunities of optimization that appear to be especially important in the field-independent case that have not implemented in our current system. We expect that these optimizations could significantly close the time and space gap between the two approaches. However, it is clear that the choice between field-based and field-independent has significant implications in practice. Most points-to systems in the literature use the field-independent approach. Our results suggest that the field-based might in fact represent a better tradeoff. The question of the relative accuracy of the two approaches is open - even the metric for measuring their relative accuracy is open to debate.

We conclude by briefly discussing empirical results from related systems in the literature. Since early implementations of Andersen's analysis [22], much progress has been made [11, 23, 21]. Currently, the best results for Andersen's are analysis times of about 430 seconds for about 500K lines of code (using a single 195 MHz processor on a multi-processor SGI Origin machine with 1.5GB) [21]. The main limiting factor in these results is that space utilization (as measured by the amount of live data after GC) is 150MB and up – in fact the largest benchmark in [21] ran out of memory. Results from [23] report analysis times of 1500s for gimp (on a SPARC Enterprise 5000 with 2GB). We note that both of these implementations of Andersen's analysis employ a field-independent treatment of structs, and so these results are not directly comparable to ours (see the caveats above about the preliminary nature of results in Table 4).

Implementations of Steensgaard's algorithm are faster and use less memory. Das reports that Word97 (about 2.2 million lines of code) runs in about 60s on a 450MHz Intel Xeon running Windows NT [8]. Das also reports that modifications to Steensgaard's algorithm to improve accuracy yield analysis times of about 130s, and memory usage of "less than 200MB" for the same benchmark. We again note that Das uses a field-independent treatment of structs.

7. CONCLUSION

We have introduced CLA, a database-centric analysis architecture, and described how we have utilized it to implement a variety of high-performance analysis systems for pointsto analysis and dependence analysis. Central to the performance of these systems are CLA's indexing schemes and support for demand-driven loading of database components. We have also described a new algorithm for implementing dynamic transitive closure that is based on maintaining a pre-transitive graph, and computing reachability on demand using caching and cycle elimination techniques.

The original motivation for this work was dependence analysis to help identify potential narrowing bugs that may be introduced during type modifications to large legacy C code bases. The points-to analysis system described in this paper has been built into a forward data-dependence analysis tool that is deployed within Lucent. Our system has uncovered many serious new errors not found by code inspections and other tools.

Future work includes exploration of context-sensitivity, and a more accurate treatment of structs that goes beyond fieldbased and field-independent (e.g. modeling of the layout of C structs in memory[7], so that an expression x.f is treated as an offset "f" from some base object x)

Acknowledgements: Thanks to Satish Chandra and Jeff Foster for access to their respective systems and bench-

	pointer	points-to	points-to real user process				assignments			
	variables	relations	time	time	size	in core	loaded	in file		
nethack	1018	7K	0.03s	0.01s	5.2MB	114	5933	10402		
burlap	3332	201K	0.08s	0.03s	5.4MB	3201	12907	19022		
vortex	4359	392K	0.15s	0.11s	5.7MB	1792	15411	34126		
emacs	8246	11232K	0.54s	0.51s	6.0MB	1560	28445	36603		
povray	6126	141K	0.11s	0.09s	5.7MB	5886	27566	40280		
gcc	11289	123K	0.20s	0.17s	6.0MB	2732	53805	69715		
gimp	45091	15298K	1.05s	1.00s	12.1MB	8377	144534	344156		
lucent	22360	3865K	0.46s	0.38s	8.8MB	4281	101856	349045		

Table 3: Results

		field-ba	ased		field-independent (preliminary)			
	pointers	relations	utime	size	pointers	relations	utime	size
nethack	1018	7K	0.01s	5.2MB	1714	97K	0.03s	5.2MB
burlap	3332	201K	0.03s	5.4MB	2903	323K	0.21s	5.9MB
vortex	4359	392K	0.11s	5.7MB	4655	164K	0.09s	5.7MB
emacs	8246	11232K	0.51s	6.0MB	8314	14643K	1.05s	6.7MB
povray	6126	141K	0.09s	5.7MB	5759	1375K	0.39s	6.6MB
gcc	11289	123K	0.17s	6.0MB	10984	408K	0.65s	8.8MB
gimp	45091	15298K	1.00s	12.1MB	39888	79603K	30.12s	18.1MB
lucent	22360	3865K	0.46s	8.8MB	26085	19665K	137.20s	59.0MB

Table 4: Effect of a field-independent treatment of structs.

marks.

8. REFERENCES

- A. Aiken, M. Fähndrich, J. Foster, and Z. Su, "A Toolkit for Constructing Type- and Constraint-Based Program Analyses", *TIC'98*.
- [2] A. Aiken and E. Wimmers, "Solving Systems of Set Constraints", *LICS*, 1992.
- [3] A. Aiken and E. Wimmers, "Type Inclusion Constraints and Type Inference", *ICFP*, 1993.
- [4] L. Andersen, "Program Analysis and Specialization for the C Programming Language", PhD. thesis, DIKU report 94/19, 1994,
- [5] D. Atkinson and W. Griswold, "Effective Whole-Program Analysis in the Presence of Pointers", 1998 Symp. on the Foundations of Soft. Eng..
- [6] S. Chandra, N. Heintze, D. MacQueen, D. Oliva and M. Siff, "ckit: an extensible C frontend in ML", to be released as an SML/NJ library.
- [7] S. Chandra and T. Reps, "Physical Type Checking for C" PASTE, 1999.
- [8] M. Das, "Unification-Based Pointer Analysis with Directional Assignments" PLDI, 2000.
- [9] "Appendix D: Optimizing Techniques (MIPS-Based C Compiler)", Programmer's Guide: Digital UNIX Version 4.0, Digital Equipment Corporation, March 1996.
- [10] J. Foster, M. Fähndrich and A. Aiken, "Flow-Insensitive Points-to Analysis with Term and Set Constraints" U. of California, Berkeley, UCB//CSD97964, 1997.
- [11] M. Fähndrich, J. Foster, Z. Su and A. Aiken, "Partial Online Cycle Elimination in Inclusion Constraint Graphs" *PLDI*, 1998.
- [12] C. Flanagan and M. Felleisen, "Componential Set-Based Analysis" PLDI, 1997.

- [13] J. Foster, M. Fähndrich and A. Aiken, "Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C", SAS 2000.
- [14] N. Heintze, "Set Based Program Analysis", PhD thesis, Carnegie Mellon University, 1992.
- [15] N. Heintze, "Set-Based Analysis of ML Programs", LFP, 1994.
- [16] N. Heintze, "Analysis of Large Code Bases: The Compile-Link-Analyze Model" unpublished report, November 1999.
- [17] N. Heintze and J. Jaffar, "A decision procedure for a class of Herbrand set constraints" *LICS*, 1990.
- [18] N. Heintze and D. McAllester, "On the Cubic-Bottleneck of Subtyping and Flow Analysis" LICS, 1997.
- [19] "Programming Languages C", ISO/IEC 9899:1990, Internation Standard, 1990.
- [20] D. McAllester, "On the Complexity Analysis of Static Analysis", SAS, 1999.
- [21] A. Rountev and S. Chandra, "Off-line Variable Substitution for Scaling Points-to Analysis", PLDI, 2000.
- [22] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-To Analysis", POPL, 1997.
- [23] Z. Su, M. Fähndrich, and A. Aiken, "Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs", *POPL*, 2000.
- [24] B. Steensgaard, "Points-to Analysis in Almost Linear Time", POPL, 1996.
- [25] F. Tip, "Generation of Program Analysis Tools", Institute for Logic Language and Computation dissertation series, 1995-5, 1995.

Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams

John Whaley Monica S. Lam Computer Science Department Stanford University Stanford, CA 94305 {jwhaley, lam}@stanford.edu

ABSTRACT

This paper presents the first scalable context-sensitive, inclusionbased pointer alias analysis for Java programs. Our approach to context sensitivity is to create a clone of a method for every context of interest, and run a context-insensitive algorithm over the expanded call graph to get context-sensitive results. For precision, we generate a clone for every acyclic path through a program's call graph, treating methods in a strongly connected component as a single node. Normally, this formulation is hopelessly intractable as a call graph often has 10^{14} acyclic paths or more. We show that these exponential relations can be computed efficiently using binary decision diagrams (BDDs). Key to the scalability of the technique is a context numbering scheme that exposes the commonalities across contexts. We applied our algorithm to the most popular applications available on Sourceforge, and found that the largest programs, with hundreds of thousands of Java bytecodes, can be analyzed in under 20 minutes.

This paper shows that pointer analysis, and many other queries and algorithms, can be described succinctly and declaratively using Datalog, a logic programming language. We have developed a system called bddbddb that automatically translates Datalog programs into highly efficient BDD implementations. We used this approach to develop a variety of context-sensitive algorithms including side effect analysis, type analysis, and escape analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithms, Performance, Design, Experimentation, Languages

Keywords

context-sensitive, inclusion-based, pointer analysis, Java, scalable, cloning, binary decision diagrams, program analysis, Datalog, logic programming

Copyright 2004 ACM 1-58113-807-5/04/0006 ...\$5.00.

1. INTRODUCTION

Many applications of program analysis, such as program optimization, parallelization, error detection and program understanding, need pointer alias information. Scalable pointer analyses developed to date are imprecise because they are either *contextinsensitive*[3, 17, 19, 33] or *unification-based*[15, 16]. A contextinsensitive analysis does not distinguish between different calling contexts of a method and allows information from one caller to propagate erroneously to another caller of the same method. In unification-based approaches, pointers are assumed to be either unaliased or are pointing to the same set of locations[28]. In contrast, *inclusion-based* approaches are more efficient but also more expensive, as they allow two aliased pointers to point to overlapping but different sets of locations.

We have developed a context-sensitive and inclusion-based pointer alias analysis that scales to hundreds of thousands of Java bytecodes. The analysis is *field-sensitive*, meaning that it tracks the individual fields of individual pointers. Our analysis is mostly flow-insensitive, using flow sensitivity only in the analysis of local pointers in each function. The results of this analysis, as we show in this paper, can be easily used to answer users' queries and to build more advanced analyses and programming tools.

1.1 Cloning to Achieve Context Sensitivity

Our approach to context sensitivity is based on the notion of *cloning*. Cloning conceptually generates multiple instances of a method such that every distinct calling context invokes a different instance, thus preventing information from one context to flow to another. Cloning makes generating context-sensitive results algorithmically trivial: We can simply apply a *context-insensitive* algorithm to the cloned program to obtain *context-sensitive* results. Note that our analysis does not clone the code per se; it simply produces a separate answer for each clone.

The context of a method invocation is often distinguished by its *call path*, which is simply the call sites, or return addresses, on the invocation's call stack. In the case of a recursive program, there are an unbounded number of calling contexts. To limit the number of calling contexts, Shivers proposed the concept of k-CFA (Control Flow Analysis) whereby one remembers only the last k call sites[26]. Emami et al. suggested distinguishing contexts by their full call paths if they are acyclic. For cyclic paths, they suggested including each call site in recursive cycles only once[14]. Our approach also uses entire call paths to distinguish between contexts in programs without recursion. To handle recursion, call paths are *reduced* by eliminating all invocations whose callers and callees belong to the same strongly connected component in the call graph. These reduced call paths are used to identify contexts.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'04, June 9-11, 2004, Washington, DC, USA.

It was not obvious, at least to us at the beginning of this project, that a cloning-based approach would be feasible. The number of reduced call paths in a program grows exponentially with the number of methods, and a cloning-based approach must compute the result of every one of these contexts. Emami et al. have only reported context-sensitive points-to results on small programs[14]. Realistic programs have many contexts; for example, the megamek application has over 10^{14} contexts (see Section 6.1). The size of the final results alone appears to be prohibitive.

We show that we can scale a cloning-based points-to analysis by representing the context-sensitive relations using ordered binary decision diagrams (BDDs)[6]. BDDs, originally designed for hardware verification, have previously been used in a number of program analyses[2, 23, 38], and more recently for points-to analysis[3, 39]. We show that it is possible to compute context-sensitive pointsto results for over 10^{14} contexts.

In contrast, most context-sensitive pointer alias analyses developed to date are summary-based[15, 34, 37]. Parameterized summaries are created for each method and used in creating the summaries of its callers. It is not necessary to represent the results for the exponentially many contexts explicitly with this approach, because the result of a context can be computed independently using the summaries. However, to answer queries as simple as "which variables point to a certain object" would require all the results to be computed. The readers may be interested to know that, despite much effort, we tried but did not succeed in creating a scalable summary-based algorithm using BDDs.

1.2 Contributions

The contributions of this paper are not limited to just an algorithm for computing context-sensitive and inclusion-based points-to information. The methodology, specification language, representation, and tools we used in deriving our pointer analysis are applicable to creating many other algorithms. We demonstrate this by using the approach to create a variety of queries and algorithms.

Scalable cloning-based context-sensitive points-to analysis using BDDs. The algorithm we have developed is remarkably simple. We first create a cloned call graph where a clone is created for every distinct calling context. We then run a simple contextinsensitive algorithm over the cloned call graph to get contextsensitive results. We handle the large number of contexts by representing them in BDDs and using an encoding scheme that allows commonalities among similar contexts to be exploited. We improve the efficiency of the algorithm by using an automatic tool that searches for an effective variable ordering.

Datalog as a high-level language for BDD-based program analyses. Instead of writing our program analyses directly in terms of BDD operations, we store all program information and results as relations and express our analyses in Datalog, a logic programming language used in deductive databases[30]. Because Datalog is succinct and declarative, we can express points-to analyses and many other algorithms simply and intuitively in just a few Datalog rules.

We use Datalog because its set-based operation semantics matches the semantics of BDD operations well. To aid our algorithm research, we have developed a deductive database system called bddbddb (BDD Based Deductive DataBase) that automatically translates Datalog programs into BDD algorithms. We provide a high-level summary of the optimizations in this paper; the details are beyond the scope of this paper[35].

Our experience is that programs generated by bddbddb are faster than their manually optimized counterparts. More importantly, Datalog programs are orders-of-magnitude easier to write. They are so succinct and easy to understand that we use them to explain all our algorithms here directly. All the experimental results reported in this paper are obtained by running the BDD programs automatically generated by bddbddb.

Context-sensitive queries and other analyses. The contextsensitive points-to results, the simple cloning-based approach to context sensitivity, and the bddbddb system make it easy to write new analyses. We show some representative examples in each of the following categories:

- 1. *Simple queries*. The results from our context-sensitive pointer analysis provide a wealth of information of interest to programmers. We show how a few lines of Datalog can help programmers debug a memory leak and find potential security vulnerabilities.
- Algorithms using context-sensitive points-to results. We show how context-sensitive points-to results can be used to create advanced analyses. We include examples of a contextsensitive analysis to compute side effects (mod-ref) and an analysis to refine declared types of variables.
- 3. *Other context-sensitive algorithms.* Cloning can be used to trivially generate other kinds of context-sensitive results besides points-to relations. We illustrate this with a context-sensitive type analysis and a context-sensitive thread escape analysis. Whereas previous escape analyses require thousands of lines of code to implement[34], the algorithm here has only seven Datalog rules.

Experimental Results. We present the analysis time and memory usage of our analyses across 21 of the most popular Java applications on Sourceforge. Our context-sensitive pointer analysis can analyze even the largest of the programs in under 19 minutes. We also compare the precision of context-insensitive pointer analysis, context-sensitive pointer analysis and context-sensitive type analysis, and show the effects of merging versus cloning contexts.

1.3 Paper Organization

Here is an overview of the rest of the paper. Section 2 explains our methodology. Using Berndl's context-insensitive pointsto algorithm as an example, we explain how an analysis can be expressed in Datalog and, briefly, how bddbddb translates Datalog into efficient BDD implementations. Section 3 shows how we can easily extend the basic points-to algorithm to discover call graphs on the fly by adding a few Datalog rules. Section 4 presents our cloning-based approach and how we use it to compute context-sensitive points-to results. Section 5 shows the representative queries and algorithms built upon our points-to results and the cloning-based approach. Section 6 presents our experimental results. We report related work in Section 7 and conclude in Section 8.

2. FROM DATALOG TO BDDS

In this section, we start with a brief introduction to Datalog. We then show how Datalog can be used to describe the contextinsensitive points-to analysis due to Berndl et al. at a high level. We then describe how our bddbddb system translates a Datalog program into an efficient implementation using BDDs.

2.1 Datalog

We represent a program and all its analysis results as relations. Conceptually, a *relation* is a two-dimensional table. The columns are the *attributes*, each of which has a *domain* defining the set of possible attribute values. The rows are the tuples of attributes that share the relation. If tuple (x, y, z) is in relation A, we say that predicate A(x, y, z) is true.

A Datalog program consists of a set of rules, written in a Prologstyle notation, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule

$$D(w,z) \quad : - \quad A(w,x), B(x,y), C(y,z).$$

says that "D(w, z) is true if A(w, x), B(x, y), and C(y, z) are all true." Variables in the predicates can be replaced with constants, which are surrounded by double-quotes, or don't-cares, which are signified by underscores. Predicates on the right side of the rules can be inverted.

Datalog is more powerful than SQL, which is based on relational calculus, because Datalog predicates can be recursively defined[30]. If none of the predicates in a Datalog program is inverted, then there is a guaranteed minimal solution consisting of relations with the least number of tuples. Conversely, programs with inverted predicates may not have a unique minimal solution. Our bddbddb system accepts a subclass of Datalog programs, known as *stratified* programs[7], for which minimal solutions always exist. Informally, rules in such programs can be grouped into strata, each with a unique minimal solution, that can be solved in sequence.

2.2 Context-Insensitive Points-to Analysis

We now review Berndl et al.'s context-insensitive points-to analysis[3], while also introducing the Datalog notation. This algorithm assumes that a call graph, computed using simple class hierarchy analysis[13], is available a priori. Heap objects are named by their allocation sites. The algorithm finds the objects possibly pointed to by each variable and field of heap objects in the program. Shown in Algorithm 1 is the exact Datalog program, as fed to bddbddb, that implements Berndl's algorithm. To keep the first example simple, we defer the discussion of using types to improve precision until Section 2.3.

ALGORITHM 1. Context-insensitive points-to analysis with a precomputed call graph.

DOMAINS

V	262144	variable.map
Η	65536	heap.map
\mathbf{F}	16384	field.map

RELATIONS

$$\begin{array}{lll} \text{input} & vP_0 & (variable: V, heap: H) \\ \text{input} & store & (base: V, field: F, source: V) \\ \text{input} & load & (base: V, field: F, dest: V) \\ \text{input} & assign & (dest: V, source: V) \\ \text{output} & vP & (variable: V, heap: H) \\ \text{output} & hP & (base: H, field: F, target: H) \\ \end{array}$$

RULES

$$P(v,h) = -vP_0(v,h).$$
 (1)

$$vP(v_1,h)$$
 :- $assign(v_1,v_2), vP(v_2,h).$ (2)

$$P(n_1, j, h_2) := store(v_1, j, v_2), vP(v_1, h_1), vP(v_2, h_2).$$
(3)

$$vP(v_2, h_2) = - load(v_1, f, v_2), vP(v_1, h_1), hP(h_1, f, h_2).$$
(4)

A Datalog program has three sections: domains, relations, and rules. A *domain declaration* has a name, a size n, and an optional file name that provides a name for each element in the domain, internally represented as an ordinal number from 0 to n - 1. The latter

allows bddbddb to communicate with the users with meaningful names. A *relation declaration* has an optional keyword specifying whether it is an input or output relation, the name of the relation, and the name and domain of every attribute. A relation declared as neither input nor output is a temporary relation generated in the analysis but not written out. Finally, the rules follow the standard Datalog syntax. The rule numbers, introduced here for the sake of exposition, are not in the actual program.

We can express all information found in the intermediate representation of a program as relations. To avoid inundating readers with too many definitions all at once, we define the relations as they are used. The domains and relations used in Algorithm 1 are:

- V is the domain of variables. It represents all the allocation sites, formal parameters, return values, thrown exceptions, cast operations, and dereferences in the program. There is also a special *global* variable for use in accessing static variables.
- H is the domain of heap objects. Heap objects are named by the invocation sites of object creation methods. To increase precision, we also statically identify factory methods and treat them as object creation methods.
- F is the domain of field descriptors in the program. Field descriptors are used when loading from a field $(v_2 = v_1.f_i)$ or storing to a field $(v_1.f = v_2i)$. There is a special field descriptor to denote an array access.
- vP_0 : V × H is the initial variable points-to relation extracted from object allocation statements in the source program. $vP_0(v, h)$ means there is an invocation site h that assigns a newly allocate object to variable v.
- store: $V \times F \times V$ represents store statements. $store(v_1, f, v_2)$ says that there is a statement " $v_1 \cdot f = v_2$;" in the program.
- *load*: $V \times F \times V$ represents load statements. *load*(v_1 , f, v_2) says that there is a statement " $v_2 = v_1$. f;" in the program.
- assign: $V \times V$ is the assignments relation due to passing of arguments and return values. $assign(v_1, v_2)$ means that variable v_1 includes the points-to set of variable v_2 . Although we do not cover return values here, they work in an analogous manner.
- vP: V × H is the output variable points-to relation. vP(v, h) means that variable v can point to heap object h.
- hP: H × F × H is the output heap points-to relation. $hP(h_1, f, h_2)$ means that field f of heap object h_1 can point to heap object h_2 .

Note that local variables and their assignments are factored away using a flow-sensitive analysis[33]. The *assign* relation is derived by using a precomputed call graph. The sizes of the domains are determined by the number of variables, heap objects, and field descriptors in the input program.

Rule (1) incorporates the initial variable points-to relations into vP. Rule (2) finds the transitive closure over inclusion edges. If v_1 includes v_2 and variable v_2 can point to object h, then v_1 can also point to h. Rule (3) models the effect of store instructions on heap objects. Given a statement " v_1 .f = v_2 ;", if v_1 can point to h_1 and v_2 can point to h_2 , then h_1 .f can point to h_2 . Rule (4) resolves load instructions. Given a statement " v_2 = v_1 .f;", if v_1 can point to h_1 and h_1 .f can point to h_2 , then v_2 can point to h_2 . Applying these rules until the results converge finds all the possible context-insensitive points-to relations in the program.

2.3 Improving Points-to Analysis with Types

Because Java is type-safe, variables can only point to objects of *assignable* types. *Assignability* is similar to the subtype relation,

ALGORITHM 2. Context-insensitive points-to analysis with type filtering.

DOMAINS

Domains from Algorithm 1, plus:

4096 type.map

RELATIONS

т

Relations from Algorithm 1, plus:

RULES

ı

$$vPfilter(v,h) :- vT(v,t_v), hT(h,t_h), aT(t_v,t_h).$$
 (5)

$$P(v,h) = -vP_0(v,h).$$
 (6)

$$vP(v_1, h)$$
 :- $assign(v_1, v_2), vP(v_2, h),$
 $vPfilter(v_1, h).$ (7)

$$vP(v_2, h_2) \qquad : - \quad load(v_1, f, v_2), vP(v_1, h_1), \\ hP(h_1, f, h_2), vPfilter(v_2, h_2). \tag{9}$$

with allowances for interfaces, null values, and arrays[22]. By dropping targets of unassignable types in assignments and load statements, we can eliminate many impossible points-to relations that result from the imprecision of the analysis.¹

Adding type filtering to Algorithm 1 is simple in Datalog. We add a new domain to represent types and new relations to represent assignability as well as type declarations of variables and heap objects. We compute the type filter and modify the rules in Algorithm 1 to filter out unsafe assignments and load operations.

T is the domain of type descriptors (i.e. classes) in the program.

- vT: V × T represents the declared types of variables. vT(v,t) means that variable v is declared with type t.
- hT: H × T represents the types of objects created at a particular creation site. In Java, the type created by a new instruction is usually known statically.² hT(h,t) means that the object created at *h* has type *t*.
- aT: T × T is the relation of assignable types. $aT(t_1, t_2)$ means that type t_2 is assignable to type t_1 .
- $vPfilter: V \times H$ is the type filter relation. vPfilter(v, h) means that it is type-safe to assign heap object h to variable v.

Rule (5) in Algorithm 2 defines the vPfilter relation: It is typesafe to assign heap object h of type t_h to variable v of type t_v if t_v is assignable from t_h . Rules (6) and (8) are the same as Rules (1) and (3) in Algorithm 1. Rules (7) and (9) are analogous to Rules (2) and (4), with the additional constraint that only points-to relations that match the type filter are inserted.

2.4 Translating Datalog into Efficient BDD Implementations

We first describe how Datalog rules can be translated into operators from relational algebra such as "join" and "project", then show how to translate these operations into BDD operations.

2.4.1 Query Resolution

We can find the solution to an unstratified query, or a stratum of a stratified query, simply by applying the inference rules repeatedly until none of the output relations change. We can apply a Datalog rule by performing a series of relational natural join, project and rename operations. A natural join operation combines rows from two relations if the rows share the same value for a common attribute. A project operation removes an attribute from a relation. A rename operation changes the name of an attribute to another one.

For example, the application of Rule (2) can be implemented as:

 t_1 = rename(vP, variable, source); t_2 = project(join($assign, t_1$), source); $vP = vP \cup$ rename($t_2, dest, variable$);

We first rename the attribute in relation vP from variable to source so that it can be joined with relation assign to create a new points-to relation. The attribute dest of the resulting relation is changed to variable so that the tuples can be added to the vP tuples accumulated thus far.

The bddbddb system uses the three following optimizations to speed up query resolution.

Attributes naming. Since the names of the attributes must match when two relations are joined, the choice of attribute names can affect the costs of rename operations. Since the renaming cost is highly sensitive to how the relations are implemented, the bddbddb system takes the representation into account when minimizing the renaming cost.

Rule application order. A rule needs to be applied only if the input relations have changed. bddbddb optimizes the ordering of the rules by analyzing the dependences between the rules. For example, Rule 1 in Algorithm 1 does not depend on any of the other rules and can be applied only once at the beginning of the query resolution.

Incrementalization. We only need to re-apply a rule on those combinations of tuples that we have not seen before. Such a technique is known as incrementalization in the BDD literature and semi-naïve fixpoint evaluation in the database literature[1]. Our system also identifies loop-invariant relations to avoid unnecessary difference and rename operations. Shown below is the result of incrementalizing the repeated application of Rule (2):

$$\begin{split} d &= vP;\\ \text{repeat}\\ t_1 &= \text{rename}(d, variable, source);\\ t_2 &= \text{project}(\text{join}(assign, t_1), source);\\ d' &= \text{rename}(t_2, dest, variable);\\ d &= d' - vP;\\ vP &= vP \cup d;\\ \text{until } d &== \emptyset; \end{split}$$

2.4.2 Relational Algebra in BDD

We now explain how BDDs work and how they can be used to implement relations and relational operations. BDDs (Binary Decision Diagrams) were originally invented for hardware verification to efficiently store a large number of states that share many commonalities[6]. They are an efficient representation of boolean

¹We could similarly perform type filtering on stores into heap objects. However, because all stores must go through variables, such a type filter would only catch one extra case — when the base object is a null constant.

²The type of a created object may not be known precisely if, for example, the object is returned by a native method or reflection is used. Such types are modeled conservatively as all possible types.
functions. A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes, representing the constants one and zero. Each non-terminal node in the DAG represents an input variable and has exactly two outgoing edges: a high edge and a low edge. The high edge represents the case where the input variable for the node is true, and the low outgoing edge represents the case where the input variable is false. On any path in the DAG from the root to a terminal node, the value of the function on the truth values on the input variables in the path is given by the value of the terminal node. To evaluate a BDD for a specific input, one simply starts at the root node and, for each node, follows the high edge if the input variable is true, and the low edge if the input variable is false. The value of the terminal node that we reach is the value of the BDD for that input.

The variant of BDDs that we use are called *ordered binary decision diagrams*, or OBDDs[6]. "Ordered" refers to the constraint that on all paths through the graph the variables respect a given linear order. In addition, OBDDs are *maximally reduced* meaning that nodes with the same variable name and low and high successors are collapsed as one, and nodes with identical low and high successors are bypassed. Thus, the more commonalities there are in the paths leading to the terminals, the more compact the OBDDs are. Accordingly, the amount of the sharing and the size of the representation depends greatly on the ordering of the variables.

We can use BDDs to represent relations as follows. Each element d in an n-element domain D is represented as an integer between 0 and n-1 using $log_2(n)$ bits. A relation $R: D_1 \times \ldots \times D_n$ is represented as a boolean function $f: D_1 \times \ldots \times D_n \rightarrow \{0, 1\}$ such that $(d_1, \ldots, d_n) \in R$ iff $f(d_1, \ldots, d_n) = 1$, and $(d_1, \ldots, d_n) \notin R$ iff $f(d_1, \ldots, d_n) = 0$.

A number of highly-optimized BDD packages are available[21, 27]; the operations they provide can be used directly to implement relational operations efficiently. For example, the "replace" operation in BDD has the same semantics as the "rename" operation; the "relprod" operation in BDD finds the natural join between two relations and projects away the common domains.

Let us now use a concrete example to illustrate the significance of variable ordering. Suppose relation R_1 contains tuples $(1, 1), (2, 1), \ldots, (100, 1)$ and relation R_2 contains tuples $(1, 2), (2, 2), \ldots, (100, 2)$. If in the variable order the bits for the first attribute come before the bits for the second, the BDD will need to represent the sequence $1, \ldots, 100$ separately for each relation. However, if instead the bits for the second attribute come first, the BDD can share the representation for the sequence $1, \ldots, 100$ between R_1 and R_2 . Unfortunately, the problem of finding the best variable ordering is NP-complete[5]. Our bddbddb system automatically explores different alternatives empirically to find an effective ordering[35].

3. CALL GRAPH DISCOVERY

The call graph generated using class hierarchy analysis can have many spurious call targets, which can lead to many spurious pointsto relations[19]. We can get more precise results by creating the call graph on the fly using points-to relations. As the algorithm generates points-to results, they are used to identify the receiver types of the methods invoked and to bind calls to target methods; and as call graph edges are discovered, we use them to find more points-to relations. The algorithm converges when no new call targets and no new pointer relations are found.

Modifying Algorithm 2 to discover call graphs on the fly is simple. Instead of an input *assign* relation computed from a given call graph, we derive it from method invocation statements and points-to relations. ALGORITHM 3. Context-insensitive points-to analysis that computes call graph on the fly.

DOMAINS

Domains from Algorithm 2, plus:

32768	invoke.map
4096	name.map
16384	method.map
256	
	$32768 \\ 4096 \\ 16384 \\ 256$

RELATIONS

Relations from Algorithm 2, with the modification that *assign* is now a computed relation, plus:

input	cha	(type: T, name: N, target: M)
input	actual	(invoke : I, param : Z, var : V)
input	formal	(method : M, param : Z, var : V)
input	IE_0	(invoke : I, target : M)
input	mI	(method : M, invoke : I)
output	IE	(invoke : I, target : M)

RULES

Rules from Algorithm 2, plus:

$$\begin{split} IE(i,m) &: - IE_0(i,m). \quad (10) \\ IE(i,m_2) &: - mI(m_1,i,n), actual(i,0,v), \\ vP(v,h), hT(h,t), cha(t,n,m_2). (11) \\ assign(v_1,v_2) &: - IE(i,m), formal(m,z,v_1), \\ actual(i,z,v_2). \quad (12) \end{split}$$

- I is the domain of invocation sites in the program. An invocation site is a method invocation of the form $r = p_0.m(p_1 \dots p_k)$. Note that $H \subseteq I$.
- N is the domain of method names used in invocations. In an invocation $r = p_0.n(p_1 \dots p_k)$, n is the method name.
- M is the domain of implemented methods in the program. It does not include abstract or interface methods.
- Z is the domain used for numbering parameters.
- *cha*: $T \times N \times M$ encodes virtual method dispatch information from the class hierarchy. cha(t, n, m) means that m is the target of dispatching the method name n on type t.
- *actual*: $I \times Z \times V$ encodes the actual parameters for invocation sites. *actual*(*i*, *z*, *v*) means that *v* is passed as parameter number *z* at invocation site *i*.
- *formal*: $M \times Z \times V$ encodes formal parameters for methods. *formal*(m, z, v) means that formal parameter z of method m is represented by variable v.
- IE_0 : I × M are the initial invocation edges. They record the invocation edges whose targets are statically bound. In Java, some calls are static or non-virtual. Additionally, local type analysis combined with analysis of the class hierarchy allows us to determine that some calls have a single target[13]. $IE_0(i, m)$ means that invocation site *i* can be analyzed statically to call method *m*.
- mI: M × I × N represents invocation sites. mI(m, i, n) means that method m contains an invocation site i with virtual method name n. Non-virtual invocation sites are given a special null method name, which does not appear in the cha relation.
- *IE*: I × M is an output relation encoding all invocation edges. *IE*(*i*, *m*) means that invocation site *i* calls method *m*.

The rules in Algorithm 3 compute the *assign* relation used in Algorithm 2. Rules (10) and (11) find the invocation edges, with the former handling statically bound targets and the latter handling virtual calls. Rule (11) matches invocation sites with the type of the "this" pointer and the class hierarchy information to find the possible target methods. If an invocation site i with method name n is invoked on variable v, and v can point to h and h has type t, and invoking n on type t leads to method m, then m is a possible target of invocation i.

Rule (12) handles parameter passing.³ If invocation site *i* has a target method *m*, variable v_2 is passed as argument number *z*, and the formal parameter *z* of method *m* is v_1 , then the points-to set of v_1 includes the points-to set of v_2 . Return values are handled in a likewise manner, only the inclusion relation is in the opposite direction. We see that as the discovery of more variable points-to (vP) can create more invocation edges (IE), which in turn can create more assignments (*assign*) and more points-to relations. The algorithm converges when all the relations stabilize.

4. CONTEXT SENSITIVE POINTS-TO

A context-insensitive or *monomorphic* analysis produces just one set of results for each method regardless how many ways a method may be invoked. This leads to imprecision because information from different calling contexts must be merged, so information along one calling context can propagate to other calling contexts. A context-sensitive or *polymorphic* analysis avoids this imprecision by allowing different contexts to have different results.

We can make a context-sensitive version of a context-insensitive analysis as follows. We make a clone of a method for each path through the call graph, linking each call site to its own unique clone. We then run the original context-insensitive analysis over the exploded call graph. However, this technique can require an exponential (and in the presence of cycles, potentially unbounded) number of clones to be created.

It has been observed that different contexts of the same method often have many similarities. For example, parameters to the same method often have the same types or similar aliases. This observation led to the concept of partial transfer functions (PTF), where summaries for each input pattern are created on the fly as they are discovered[36, 37]. However, PTFs are notoriously difficult to implement and get correct, as the programmer must explicitly calculate the input patterns and manage the summaries. Furthermore, the technique has not been shown to scale to very large programs.

Our approach is to allow the exponential explosion to occur and rely on the underlying BDD representation to find and exploit the commonalities across contexts. BDDs can express large sets of redundant data in an efficient manner. Contexts with identical information will automatically be shared at the data structure level. Furthermore, because BDDs operate down at the bit level, it can even exploit commonalities between contexts with different information. BDD operations operate on entire relations at a time, rather than one tuple at a time. Thus, the cost of BDD operations depends on the size and shape of the BDD relations, which depends greatly on the variable ordering, rather than the number of tuples in a relation. Also, due to caching in BDD packages, identical subproblems only have to be computed once. Thus, with the right variable ordering, the results for all contexts can be computed very efficiently.

4.1 Numbering Call Paths

A call path is a sequence of invocation edges $(i_1, m_1), (i_2, m_2), \ldots$, such that i_1 is an invocation site in



Figure 1: Example of path numbering. The graph on the left is the original graph. Nodes M_2 and M_3 are in a cycle and therefore are placed in one equivalence class. Each edge is marked with path numbers at the source and target of the edge. The graph on the right is the graph with all of the paths expanded.

Call paths	Reduced call paths
reaching M_6	reaching M_6
$a(cd)^*eh$	aeh
$b(dc)^*deh$	beh
$a(cd)^*cfh$	afh
$b(dc)^*fh$	bfh
$a(cd)^{*}cgi$	agi
$b(dc)^*gi$	bgi

Figure 2: The six contexts of function M_6 in Example 1

an entry method, typically main⁴, and i_k is an invocation site in method m_{k-1} for all k > 1.

For programs without recursion, every call path to a method defines a context for that method. To handle recursive programs, which have an unbounded number of call paths, we first find the strongly connected components (SCCs) in a call graph. By eliminating all method invocations whose caller and callee belong to the same SCC from the call paths, we get a finite set of *reduced call paths*. Each reduced call path to an SCC defines a context for the methods in the SCC. Thus, information from different paths leading to the SCCs are kept separate, but the methods within the SCC invoked with the same incoming call path are analyzed contextinsensitively.

EXAMPLE 1. Figure 1(a) shows a small call graph with just six methods and a set of invocation edges. Each invocation edge has a name, being one of a through i; its source is labeled by the context number of the caller and its sink by the context number of the callee. The numbers will be explained in Example 2. Methods M_2 and M_3 belong to a strongly connected component, so invocations along edges c and d are eliminated in the computation of reduced call graphs. While there are infinitely many call paths reaching method M_6 , there are only six reduced call paths reaching M_6 , as shown in Figure 2. Thus M_6 has six clones, one for each reduced call path.

Under this definition of context sensitivity, large programs can have many contexts. For example, pmd from our test programs has 1971 methods and 10^{23} contexts! In the BDD representation, we give each reduced call path reaching a method a distinct *context*

 $^{^{3}}$ We also match thread objects to their corresponding run() methods, even though the edges do not explicitly appear in the call graph.

⁴Other "entry" methods in typical programs are static class initializers, object finalizers, and thread run methods.

number. It is important to find a context numbering scheme that allows the BDDs to share commonalities across contexts. Algorithm 4 shows one such scheme.

ALGORITHM 4. Generating context-sensitive invocation edges from a call graph.

INPUT: A call multigraph.

OUTPUT: Context-sensitive invocation edges $IE_c: C \times I \times C \times M$, where C is the domain of context numbers. $IE_c(c, i, c_m, m)$ means that invocation site *i* in context *c* calls method *m* in context c_m .

METHOD:

- 1. A method with *n* clones will be given numbers $1, \ldots, n$. Nodes with no predecessors are given a singleton context numbered 1.
- Find strongly connected components in the input call graph. The *i*th clone of a method always calls the *i*th clone of another method belonging to the same component.
- Collapse all methods in a strongly connected component to a single node to get an acyclic reduced graph.
- 4. For each node n in the reduced graph in topological order,

Set the counts of contexts created, c, to 0.

For each incoming edge,

If the predecessor of the edge p has k contexts, create k clones of node n, Add tuple (i, p, i+c, n) to $I\!E_{\rm c}$, for $1\leq i\leq k$, c=c+k.

EXAMPLE 2. We now show the results of applying Algorithm 4 to Example 1. M_1 , the root node, is given context number 1. We shall visit the invocation edges from left to right. Nodes M_2 and M_3 , being members of a strongly connected component, are represented as one node. The strongly connected component is reached by two edges from M_1 . Since M_1 has only one context, we create two clones, one reached by each edge. For method M_4 , the predecessor on each of the two incoming edges has two contexts, thus M_4 has four clones. Method M_5 has two clones, one for each clone that invokes M_5 . Finally, method M_6 has six clones: Clones 1-4 of method M_4 invoke clones 1-4 and clones 1-2 of method M_5 call clones 5-6, respectively. The cloned graph is shown in Figure 1(b).

The numbering scheme used in Algorithm 4 plays up the strengths of BDDs. Each method is assigned a contiguous range of contexts, which can be represented efficiently in BDDs. The contexts of callees can be computed simply by adding a constant to the contexts of the callers; this operation is also cheap in BDDs. Because the information for contexts that share common tail sequences are likely to be similar, this numbering allows the BDD data structure to share effectively across common contexts. For example, the sequentially-numbered clones 1 and 2 of M_6 both have a common tail sequence eh. Because of this, the contexts are likely to be similar and therefore the BDD can take advantage of the redundancies.

To optimize the creation of the cloned invocation graph, we have defined a new primitive that creates a BDD representation of contiguous ranges of numbers in O(k) operations, where k is the number of bits in the domain. In essence, the algorithm creates one BDD to represent numbers below the upper bound, and one to represent numbers above the lower bound, and computes the conjunction of these two BDDs.

4.2 Context-Sensitive Pointer Analysis with a Pre-computed Call Graph

We are now ready to present our context-sensitive pointer analysis. We assume the presence of a pre-computed call graph created, for example, by using a context-insensitive points-to analysis (Algorithm 3). We apply Algorithm 4 to the call graph to generate the context-sensitive invocation edges IE_c . Once that is created, we can simply apply a context-insensitive points-to analysis on the exploded call graph to get context-sensitive results. We keep the results separate for each clone by adding a context number to methods, variables, invocation sites, points-to relations, etc.

ALGORITHM 5. Context-sensitive points-to analysis with a precomputed call graph.

DOMAINS

Domains from Algorithm 2, plus:

RELATIONS

Relations from Algorithm 2, plus:

input	$IE_{\rm C}$	(caller : C, invoke : I, callee : C, tgt : M)
	$assign_{c}$	$(dest_c : C, dest : V, src_c : C, src : V)$
output	vP_{c}	$(context \cdot C \ variable \cdot V \ bean \cdot H)$

RULES

vPfilter(v, h)	: –	$vT(v,t_v), hT(h,t_h), aT(t_v,t_h).$	(13)
$vP_{\rm C}(c,v,h)$: –	$vP_0(v,h), IE_{C}(c,h,\underline{\ },\underline{\ }).$	(14)
$vP_{C}(c_1,v_1,h)$: –	$assign_{\rm C}(c_1, v_1, c_2, v_2), \ vP_{\rm C}(c_2, v_2, h), vP filter(v_1, h).$	(15)
$hP(h_1, f, h_2)$: –	$store(v_1, f, v_2), \\ vP_{\rm C}(c, v_1, h_1), vP_{\rm C}(c, v_2, h_2).$	(16)

$$vP_{c}(c, v_{2}, h_{2}) := load(v_{1}, f, v_{2}), vP_{c}(c, v_{1}, h_{1}), hP(h_{1}, f, h_{2}), vPfilter(v_{2}, h_{2}).$$
(17)

 $assign_{C}(c_1, v_1, c_2, v_2)$

$$- IE_{c}(c_{2}, i, c_{1}, m), formal(m, z, v_{1}),$$

$$actual(i, z, v_{2}).$$
(18)

- C is the domain of context numbers. Our BDD library uses signed 64-bit integers to represent domains, so the size is limited to 2^{63} .
- $IE_c: C \times I \times C \times M$ is the set of context-sensitive invocation edges. $IE_c(c, i, c_m, m)$ means that invocation site *i* in context *c* calls method *m* in context c_m . This relation is computed using Algorithm 4.
- $assign_{c}: C \times V \times C \times V$ is the context-sensitive version of the assign relation. $assign_{c}(c_{1}, v_{1}, c_{2}, v_{2})$ means variable v_{1} in context c_{1} includes the points-to set of variable v_{2} in context v_{2} due to parameter passing. Again, return values are handled analogously.
- vP_{c} : C × V × H is the context-sensitive version of the variable points-to relation (vP). $vP_{c}(c, v, h)$ means variable v in context c can point to heap object h.

Rule (18) interprets the context-sensitive invocation edges to find the bindings between actual and formal parameters. The rest of the rules are the context-sensitive counterparts to those found in Algorithm 2.

Algorithm 5 takes advantage of a pre-computed call graph to create an efficient context numbering scheme for the contexts. We can compute the call graph on the fly while enjoying the benefit of the numbering scheme by numbering all the *possible contexts* with a conservative call graph, and delaying the generation of the invocation edges only if warranted by the points-to results. We can reduce the iterations necessary by exploiting the fact that many of the invocation sites of a call graph created by a context-insensitive analysis have single targets. Such an algorithm has an execution time similar to Algorithm 5, but is of primarily academic interest as the call graph rarely improves due to the extra precision from contextsensitive points-to information.

5. QUERIES AND OTHER ANALYSES

The algorithms in sections 2, 3 and 4 generate vast amounts of results in the form of relations. Using the same declarative programming interface, we can conveniently query the results and extract exactly the information we are interested in. This section shows a variety of queries and analyses that make use of pointer information and context sensitivity.

5.1 Debugging a Memory Leak

Memory leaks can occur in Java when a reference to an object remains even after it will no longer be used. One common approach of debugging memory leaks is to use a dynamic tool that locates the allocation sites of memory-consuming objects. Suppose that, upon reviewing the information, the programmer thinks objects allocated in line 57 in file a . java should have been freed. He may wish to know which objects may be holding pointers to the leaked objects, and which operations may have stored the pointers. He can consult the static analysis results by supplying the queries:

$$\begin{array}{lll} whoPointsTo57(h,f) & :- & hP(h,f,``a.java:57").\\ whoDunnit(c,v_1,f,v_2) & :- & store(v_1,f,v_2),\\ & & vP_c(c,v_2,``a.java:57"). \end{array}$$

The first query finds the objects and their fields that may point to objects allocated at "a.java:57"; the second finds the store instructions, and the contexts under which they are executed, that create the references.

5.2 Finding a Security Vulnerability

The Java Cryptography Extension (JCE) is a library of cryptographic algorithms[29]. Misuse of the JCE API can lead to security vulnerabilities and a false sense of security. For example, many operations in the JCE use a secret key that must be supplied by the programmer. It is important that secret keys be cleared after they are used so they cannot be recovered by attackers with access to memory. Since String objects are immutable and cannot be cleared, secret keys should not be stored in String objects but in an array of characters or bytes instead.

To guard against misuse, the function that accepts the secret key, PBEKeySpec.init(), only allows arrays of characters or bytes as input. However, a programmer not versed in security issues may have stored the key in a String object and then use a routine in the String class to convert it to an array of characters. We can write a query to audit programs for the presence of such idioms. Let Mret(m, v) be an input relation specifying that variable v is the return value of method m. We define a relation fromString(h) which indicates if the object h was directly derived from a String. Specifically, it records the objects that are returned by a call to a method in the String class. An invocation i to method PBEKeySpec.init() is a vulnerability if the first argument points to an object derived from a String.

$$fromString(h) :- cha("String", _, m), Mret(m, v), \\ vP_c(_, v, h).$$

$$vuln(c,i) :- IE(i, "PBEKeySpec.init()"), actual(i, 1, v), vP_c(c, v, h), fromString(h).$$

Notice that this query does not only find cases where the object derived from a String is immediately supplied to PBEKeySpec.init(). This query will also identify cases where the object has passed through many variables and heap objects.

5.3 Type Refinement

Libraries are written to handle the most general types of objects possible, and their full generality is typically not used in many applications. By analyzing the actual types of objects used in an application, we can *refine* the types of the variables and object fields. Type refinement can be used to reduce overheads in cast operations, resolve virtual method calls, and gain better understanding of the program.

We say that variable v can be legally declared as t, written varSuperTypes(v, t), if t is a supertype of the types of all the objects v can point to. The type of a variable is refinable if the variable can be declared to have a more precise type. To compute the super types of v, we first find varExactTypes(v, t), the types of objects pointed to by v. We then intersect the supertypes of all the exact types to get the desired solution; we do so in Datalog by finding the complement of the union of the complement of the exact types.

$$\begin{aligned} varExactTypes(v,t) &: - vP_c(_,v,h), hT(h,t). \\ notVarType(v,t) &: - varExactTypes(v,t_v), \neg aT(t,t_v). \\ varSuperTypes(v,t) &: - \neg notVarType(v,t). \\ refinable(v,t_c) &: - vT(v,t_d), varSuperTypes(v,t_c), \\ & aT(t_d,t_c), t_d \neq t_c. \end{aligned}$$

The above shows a context-insensitive type refinement query. We find, for each variable, the type to which it can be refined regardless of the context. Even if the end result is context-insensitive, it is more precise to take advantage of the context-sensitive points-to results available to determine the exact types, as shown in the first rule. In Section 6.3, we compare the accuracy of this context-insensitive query with a context-sensitive version.

5.4 Context-Sensitive Mod-Ref Analysis

Mod-ref analysis is used to determine what fields of what objects may be modified or referenced by a statement or call site[18]. We can use the context-sensitive points-to results to solve a contextsensitive version of this query. We define mV(m, v) to mean that vis a local variable in m. The mV_c^* relation specifies the set of variables and contexts of methods that are transitively reachable from a method. $mV_c^*(c_1, m, c_2, v)$ means that calling method m with context c_1 can transitively call a method with local variable v under context c_2 .

$$mV_{\rm C}^*(c,m,c,v) \qquad :- mV(m,v). \\ mV_{\rm C}^*(c_1,m_1,c_3,v_3) \qquad :- mI(m_1,i), IE_{\rm C}(c_1,i,c_2,m_2), \\ mV_{\rm C}^*(c_2,m_2,c_3,v_3).$$

The first rule simply says that a method m in context c can reach its local variable. The second rule says that if method m_1 in context c_1 calls method m_2 in context c_2 , then m_1 in context c_1 can also reach all variables reached by method m_2 in context c_2 .

We can now define the mod and ref set of a method as follows:

$$\begin{array}{lll} mod(c,m,h,f) &: - & mV_{c}^{*}(c,m,c_{v},v), \\ & & store(v,f,_), vP_{c}(c_{v},v,h). \\ ref(c,m,h,f) &: - & mV_{c}^{*}(c,m,c_{v},v), \\ & & load(v,f,_), vP_{c}(c_{v},v,h). \end{array}$$

The first rule says that if method m in context c can reach a variable v in context c_v , and if there is a store through that variable to field f of object h, then m in context c can modify field f of object h. The second rule for defining the ref relations is analogous.

5.5 Context-Sensitive Type Analysis

Our cloning technique can be applied to add context sensitivity to other context-insensitive algorithms. The example we show here is the type inference of variables and fields. By not distinguishing between instances of heap objects, this analysis does not generate results as precise as those extracted from running the complete context-sensitive pointer analysis as discussed in Section 5.3, but is much faster.

The basic type analysis is similar to 0-CFA[26]. Each variable and field in the program has a set of concrete types that it can refer to. The sets are propagated through calls, returns, loads, and stores. By using the path numbering scheme in Algorithm 4, we can convert this basic analysis into one which is context-sensitive in essence, making the analysis into a k-CFA analysis where k is the depth of the call graph and recursive cycles are collapsed.

ALGORITHM 6. Context-sensitive type analysis.

DOMAINS

Domains from Algorithm 5

RELATIONS

Relations from Algorithm 5, plus:

RULES

$$vTfilter(v,t) \quad : - \quad vT(v,t_v), aT(t_v,t).$$
(19)

$$vT_{\rm C}(c,v,t) = -vP_0(v,h), IE_{\rm C}(c,h,-,-), hT(h,t).$$
 (20)

$$vT_{c}(c_{v_{1}}, v_{1}, t) :- assign_{c}(c_{v_{1}}, v_{1}, c_{v_{2}}, v_{2}), vT_{c}(c_{v_{2}}, v_{2}, t), vT_{filter}(v_{1}, t).$$
(21)

$$fT(f,t)$$
 :- $store(_, f, v_2), vT_c(_, v_2, t).$ (22)

$$vT_{c}(\underline{\ },v,t) \qquad :- \quad load(\underline{\ },f,v), fT(f,t), \\ vTfilter(v,t).$$
(23)

 $assign_{\rm C}(c_1, v_1, c_2, v_2)$

:
$$-IE_{c}(c_{2}, i, c_{1}, m), formal(m, z, v_{1}), actual(i, z, v_{2}).$$
 (24)

- vT_c : C × V × T is the context-sensitive variable type relation. $vT_c(c, v, t)$ means that variable v in context c_v can refer to an object of type t. This is the analogue of vP_c in the pointsto analysis.
- $fT: F \times T$ is the field type relation. fT(f, t) means that field f can point to an object of type t.
- vTfilter: V × T is the type filter relation. vTfilter(v, t) means that it is type-safe to assign an object of type t to variable v.

Rule (20) initializes the vT_c relation based on the initial local points-to information contained in vP_0 , combining it with hT to get the type and IE_c to get the context numbers. Rule (21) does transitive closure on the vT_c relation, filtering with vTfilter to enforce type safety. Rules (22) and (23) handle stores and loads, respectively. They differ from their counterparts in the pointer analysis in that they do not use the base object, only the field. Rule (24) models the effects of parameter passing in a context-sensitive manner.

5.6 Thread Escape Analysis

Our last example is a thread escape analysis, which determines if objects created by one thread may be used by another. The results of the analysis can be used for optimizations such as synchronization elimination and allocating objects in thread-local heaps, as well as for understanding programs and checking for possible race conditions due to missing synchronizations[8, 34]. This example illustrates how we can vary context sensitivity to fit the needs of the analysis.

We say that an object allocated by a thread has *escaped* if it may be *accessed* by another thread. This notion is stronger than most other formulations where an object is said to escape if it can be *reached* by another thread[8, 34].

Java threads, being subclasses of java.lang.Thread, are identified by their creation sites. In the special case where a thread creation can execute only once, a thread can simply be named by the creation site. The thread that exists at virtual machine startup is an example of a thread that can only be created once. A creation site reached via different call paths or embedded in loops or recursive cycles may generate multiple threads. To distinguish between thread instances created at the same site, we create two thread contexts to represent two separate thread instances. If an object created by one instance is not accessed by its clone, then it is not accessed by any other instances created by the same call site. This scheme creates at most twice as many contexts as there are thread creation sites.

We clone the thread run() method, one for each thread context, and place these clones on the list of entry methods to be analyzed. Methods (transitively) invoked by a context's run() method all inherit the same context. A clone of a method not only has its own cloned variables, but also its own cloned object creation sites. In this way, objects created by separate threads are distinct from each other. We run a points-to analysis over this slightly expanded call graph; an object created in a thread context escapes if it is accessed by variables in another thread context.

ALGORITHM 7. Thread-sensitive pointer analysis.

DOMAINS

Domains from Algorithm 5

RELATIONS

Relations from Algorithm 2, plus:

$$\begin{array}{lll} \text{input} & H_{\mathrm{T}} & (c:\mathrm{C},heap:\mathrm{H}) \\ \text{input} & vP_{0\mathrm{T}} & (cv:\mathrm{C},variable:\mathrm{V},ch:\mathrm{C},heap:\mathrm{H}) \\ \text{output} & vP_{\mathrm{T}} & (cv:\mathrm{C},variable:\mathrm{V},ch:\mathrm{C},heap:\mathrm{H}) \\ \text{output} & hP_{\mathrm{T}} & (cb:\mathrm{C},base:\mathrm{H},field:\mathrm{F},ct:\mathrm{C},target:\mathrm{H}) \\ \end{array}$$

RULES

$$vPfilter(v,h)$$
 : $-vT(v,t_v), hT(h,t_h), aT(t_v,t_h).$ (25)

$$vP_{\mathrm{T}}(c_1, v, c_2, h) = :-vP_{0\mathrm{T}}(c_1, v, c_2, h).$$
 (26)

$$vP_{\rm T}(c,v,c,h)$$
 : $-vP_0(v,h), H_{\rm T}(c,h).$ (27)

$$vP_{T}(c_{2}, v_{1}, c_{h}, h) = :-assign(v_{1}, v_{2}), vP_{T}(c_{2}, v_{2}, c_{h}, h), vP filter(v_{1}, h).$$
(28)

$$hP_{\mathsf{T}}(c_1, h_1, f, c_2, h_2): -store(v_1, f, v_2), vP_{\mathsf{T}}(c, v_1, c_1, h_1), vP_{\mathsf{T}}(c, v_2, c_2, h_2).$$
(29)

$$vP_{\mathsf{T}}(c, v_2, c_2, h_2) = -load(v_1, f, v_2), vP_{\mathsf{T}}(c, v_1, c_1, h_1), hP_{\mathsf{T}}(c_1, h_1, f, c_2, h_2), vP filter(v_2, h_2).$$
(30)

- $H_{\rm T}$: C × H encodes the non-thread objects created by a thread. $H_{\rm T}(c,h)$ means that a thread with context c may execute non-thread allocation site h; in other words, there is a call path from the run() method in context c to allocation site h.
- vP_{0T} : C × V × C × H is the set of initial inter-thread points-to relations. This includes the points-to relations for thread creation sites and for the global object. $vP_{0T}(c_1, v, c_2, h)$ means that thread c_1 has an thread allocation site h, and v points to the newly created thread context c_2 . (There are usually two contexts assigned to each allocation site). All global objects across all contexts are given the same context.
- vP_{T} : C × V × C × H is the thread-sensitive version of the variable points-to relation vP_{c} . $vP_{T}(c_{1}, v, c_{2}, h)$ means variable v in context c_{1} can point to heap object h created under context c_{2} .
- hP_{T} : C × H × F × C × H is the thread-sensitive version of the heap points-to relation hP. $hP_{T}(c_1, h_1, f, c_2, h_2)$ means that field f of heap object h_1 created under context c_1 can point to heap object h_2 created under context c_2 .

Rule (26) incorporates the initial points-to relations for thread creation sites. Rule (27) incorporates the points-to information for non-thread creation sites, which have the context numbers of threads that can reach the method. The other rules are analogous to those of the context-sensitive pointer analysis, with an additional context attribute for the heap objects.

From the analysis results, we can easily determine which objects have escaped. An object h created by thread context c has escaped, written *escaped*(c, h), if it is accessed by a different context c_v . Complications involving unknown code, such as native methods, could also be handled using this technique.

$$escaped(c,h) := vP_{T}(c_{v}, _, c, h), c_{v} \neq c.$$

Conversely, an object h created by context c is captured, written captured(c, h), if it has not escaped. Any captured object can be allocated on a thread-local heap.

$$captured(c,h) := vP_{T}(c,v,c,h), \neg escaped(c,h).$$

We can also use escape analysis to eliminate unnecessary synchronizations. We define a relation syncs(v) indicating if the program contains a synchronization operation performed on variable v. A synchronization for variable v under context c is necessary, written neededSyncs(c, v), if syncs(v) and v can point to an escaped object.

Notice that *neededSyncs* is context-sensitive. Thus, we can distinguish when a synchronization is necessary only for certain threads, and generate specialized versions of methods for those threads.

6. EXPERIMENTAL RESULTS

In this section, we present some experimental results of using bddbddb on the Datalog algorithms presented in this paper. We describe our testing methodology and benchmarks, present the analysis times, evaluate the results of the analyses, and provide some insight on our experience of developing these analyses and the bddbddb tool.

6.1 Methodology

The input to bddbddb is more or less the Datalog programs exactly as they are presented in this paper. (We added a few rules to handle return values and threads, and added annotations for the physical domain assignments of input relations.) The input relations were generated with the Joeq compiler infrastructure[32]. The entire bddbddb implementation is only 2500 lines of code. bd-dbddb uses the JavaBDD library[31], an open-source library based on the BuDDy library[21]. The entire system is available as open-source[35], and we hope that others will find it useful.

All experiments were performed on a 2.2GHz Pentium 4 with Sun JDK 1.4.2_04 running on Fedora Linux Core 1. For the contextinsensitive and context-sensitive experiments, respectively: we used initial BDD table sizes of 4M and 12M; the tables could grow by 1M and 3M after each garbage collection; the BDD operation cache sizes were 1M and 3M.

To test the scalability and applicability of the algorithm, we applied our technique to 21 of the most popular Java projects on Sourceforge as of November 2003. We simply walked down the list of 100% Java projects sorted by activity, selecting the ones that would compile directly as standalone applications. They are all real applications with tens of thousands of users each. As far as we know, these are the largest benchmarks ever reported for any context-sensitive Java pointer analysis. As a point of comparison, the largest benchmark in the specjvm suite, javac, would rank only 13th in our list.

For each application, we chose an applicable main() method as the entry point to the application. We included all class initializers, thread run methods, and finalizers. We ignored null constants in the analysis—every points-to set is automatically assumed to include null. Exception objects of the same type were merged. We treated reflection and native methods as returning unknown objects. Some native methods and special fields were modeled explicitly.

A short description of each of the benchmarks is included in Figure 3, along with their vital statistics. The number of classes, methods, and bytecodes were those discovered by the context-insensitive on-the-fly call graph construction algorithm, so they include only the reachable parts of the program and the class library.

The number of context-sensitive (C.S.) paths is for the most part correlated to the number of methods in the program, with the exception of pmd. pmd has an astounding 5×10^{23} paths in the call graph, which requires 79 bits to represent. pmd has different characteristics because it contains code generated by the parser generator JavaCC. Many machine-generated methods call the same class library routines, leading to a particularly egregious exponential blowup. The JavaBDD library only supports physical domains up to 63 bits; contexts numbered beyond 2^{63} were merged into a single context. The large number of paths also caused the algorithm to require many more rule applications to reach a fixpoint solution.

6.2 Analysis Times

We measured the analysis times and memory usage for each of the algorithms presented in this paper (Figure 4). The algorithm with call graph discovery, in each iteration, computes a call graph based on the points-to relations from the previous iteration. The number of iterations taken for that algorithm is also included here.

All timings reported are wall-clock times from a cold start, and include the various overheads for Java garbage collection, BDD garbage collection, growing the node table, etc. The memory numbers reported are the sizes of the peak number of live BDD nodes during the course of the algorithm. We measured peak BDD memory usage by setting the initial table size and maximum table size increase to 1MB, and only allowed the table to grow if the node table was more than 99% full after a garbage collection.⁵

⁵To avoid garbage collections, it is recommended to use more memory. Our timing runs use the default setting of 80%.

Name	Description	Classes	Methods	Bytecodes	Vars	Allocs	C.S. Paths
freetts	speech synthesis system	215	723	48K	8K	3K	4×10^4
nfcchat	scalable, distributed chat client	283	993	61K	11K	3K	8×10^{6}
jetty	HTTP Server and Servlet container	309	1160	66K	12K	3K	9×10^5
openwfe	java workflow engine	337	1215	74K	14K	4K	3×10^{6}
joone	Java neural net framework	375	1531	92K	17K	4K	1×10^7
jboss	J2EE application server	348	1554	104K	17K	4K	3×10^8
jbossdep	J2EE deployer	431	1924	119K	21K	5K	4×10^8
sshdaemon	SSH daemon	485	2053	115K	24K	5K	4×10^9
pmd	Java source code analyzer	394	1971	140K	19K	4K	5×10^{23}
azureus	Java bittorrent client	498	2714	167K	24K	5K	2×10^{9}
freenet	anonymous peer-to-peer file sharing system	667	3200	210K	38K	8K	2×10^7
sshterm	SSH terminal	808	4059	241K	42K	8K	5×10^{11}
jgraph	mathematical graph-theory objects and algorithms	1041	5753	337K	59K	10K	1×10^{11}
umldot	makes UML class diagrams from Java code	1189	6505	362K	65K	11K	3×10^{14}
jbidwatch	auction site bidding, sniping, and tracking tool	1474	8262	489K	90K	16K	7×10^{13}
columba	graphical email client with internationalization	2020	10574	572K	111K	19K	1×10^{13}
gantt	plan projects using Gantt charts	1834	10487	597K	117K	20K	1×10^{13}
jxplorer	ldap browser	1927	10702	645K	133K	22K	2×10^9
jedit	programmer's text editor	1788	10934	667K	124K	20K	6×10^7
megamek	networked BattleTech game	1265	8970	668K	123K	21K	4×10^{14}
gruntspud	graphical CVS client	2277	12846	687K	145K	24K	2×10^9

Figure 3: Information about the benchmarks we used to test our analyses.

The context-insensitive analyses (Algorithms 1 and 2) are remarkably fast; the type-filtering version was able to complete in under 45 seconds on all benchmarks. It is interesting to notice that introducing type filtering actually improved the analysis time and memory usage. Along with being more accurate, the points-to sets are much smaller in the type-filtered version, leading to faster analysis times.

For Algorithm 3, the call graph discovery sometimes took over 40 iterations to complete, but it was very effective in reducing the size of the call graph as compared to CHA[19]. The complexity of the call graph discovery algorithm seems to vary with the number of virtual call sites that need resolving—jedit and megamek have many methods declared as final, but jxplorer has none, leading to more call targets to resolve and longer analysis times.

The analysis times and memory usages of our context-sensitive points-to analysis (Algorithm 5) were, on the whole, very reasonable. It can analyze most of the small and medium size benchmarks in a few minutes, and it successfully finishes analyzing even the largest benchmarks in under 19 minutes. This is rather remarkable considering that the context-sensitive formulation is solving up to 10^{14} times as many relations as the context-insensitive version! Our scheme of numbering the contexts consecutively allows the BDD to efficiently represent the similarities between calling contexts. The analysis times are most directly correlated to the number of paths in the call graph. From the experimental data presented here, it appears that the analysis time of the context-sensitive algorithm scales approximately with $O(\lg^2 n)$ where n is the number of paths in the call graph; more experiments are necessary to determine if this trend persists across more programs.

The context-sensitive type analysis (Algorithm 6) is, as expected, quite a bit faster and less memory-intensive than the contextsensitive points-to analysis. Even though it uses the same number of contexts, it is an order of magnitude faster than the context-sensitive points-to analysis. This is because in the type analysis the number of objects that can be pointed to is much smaller, which greatly increases sharing in the BDD. The thread-sensitive pointer analysis (Algorithm 7) has analysis times and memory usages that are roughly comparable to those of the context-insensitive pointer analysis, even though it includes thread context information. This is because the number of thread creation sites is relatively small, and we use at most two contexts per thread.

6.3 Evaluation of Results

An in-depth analysis of the accuracy of the analyses with respect to each of the queries in Section 5 is beyond the scope of this paper. Instead, we show the results of two specific queries: thread escape analysis (Section 5.6) and type refinement (Section 5.3).

The results of the escape analysis are shown in Figure 5. The first two columns give the number of captured and escaped object creation sites, respectively. The next two columns give the number of unneeded and needed synchronization operations. The single-threaded benchmarks have only one escaped object: the global object from which static variables are accessed. In the multi-threaded benchmarks, the analysis is effective in finding 30-50% of the allocation sites to be captured, and 15-30% of the synchronization operations to be unnecessary. These are static numbers; to fully evaluate the results would require dynamic execution counts, which is outside of the scope of this paper.

The results of the type refinement query are shown in Figure 6. We tested the query across six different analysis variations. From left to right, they are context-insensitive pointer analysis without and with type filtering, context-sensitive pointer analysis and context-sensitive type analysis with the context projected away, and context-sensitive pointer and type analysis on the fully cloned graph. Projecting away the context in a context-sensitive analysis makes the result context-insensitive; however, it can still be more precise than context-insensitive analysis because of the extra precision at the intermediate steps of the analysis. We measured the percentages of variables that can point to multiple types and variables whose types can be refined.

Including the type filtering makes the algorithm strictly more precise. Likewise, the context-sensitive pointer analysis is strictly more precise than both the context-insensitive pointer analysis and the context-sensitive type analysis. We can see this trend in the results. As the precision increases, the percentage of multi-typed variables drops and the percentage of refinable variables increases. The context-insensitive pointer analysis and the context-sensitive type analysis are not directly comparable; in some cases the point-

	Context-insensitive					pointers			Context-sensitive				-sensitive
Name	no typ	e filter	with ty	pe filter	with	ı cg diso	covery	pointer	analysis	type a	analysis	pointe	r analysis
	time	mem	time	mem	iter	time	mem	time	mem	time	mem	time	mem
freetts	1	3	1	3	20	2	4	1	6	1	6	1	4
nfcchat	1	4	1	4	23	4	6	2	12	2	12	1	6
jetty	1	5	1	5	22	4	7	3	12	2	10	1	6
openwfe	1	5	1	6	23	5	8	4	14	2	14	1	7
joone	2	7	1	7	24	7	10	4	18	3	18	1	9
jboss	2	7	2	7	30	8	10	7	24	4	22	2	9
jbossdep	2	9	2	9	26	7	12	9	30	5	26	3	11
sshdaemon	2	9	2	10	26	13	14	12	34	6	28	3	13
pmd	1	7	1	7	33	9	10	297	111	19	36	1	9
azureus	2	10	2	10	29	13	15	9	32	6	30	2	12
freenet	7	16	5	16	40	41	23	21	38	10	32	6	21
sshterm	8	17	5	17	31	37	25	50	86	18	60	7	23
jgraph	17	27	11	25	42	78	37	119	134	33	20	13	35
umldot	17	30	11	29	34	97	43	457	304	63	130	16	41
jbidwatch	31	43	20	40	32	149	58	580	394	68	140	25	56
columba	43	55	27	49	42	273	73	807	400	123	178	38	72
gantt	41	59	26	51	39	261	76	1122	632	113	174	34	71
jxplorer	57	68	39	60	41	390	88	337	198	78	118	51	83
jedit	61	61	38	54	37	278	80	113	108	60	82	50	76
megamek	40	57	26	51	34	201	76	1101	600	100	224	34	73
gruntspud	66	76	41	67	35	389	99	312	202	86	130	58	95

Figure 4: Analysis times and peak memory usages for each of the benchmarks and analyses. Time is in seconds and memory is in megabytes.

ers are more precise, in other cases the context-sensitive types are more precise.

When we do not project away the context, the context-sensitive results are remarkably precise—the percentage of multi-typed variables is never greater than 1% for the pointer analysis and 2% for the type analysis. Projecting away the context loses much of the benefit of context sensitivity, but is still noticeably more precise than using a context-insensitive analysis.

	heap o	bjects	sync ope	rations
Name	captured	escaped	$\neg needed$	needed
freetts	2349	1	43	0
nfcchat	1845	2369	52	46
jetty	2059	2408	47	89
openwfe	3275	1	57	0
joone	1640	1908	34	75
jboss	3455	2836	112	105
jbossdep	1838	2298	32	94
sshdaemon	12822	22669	468	1244
pmd	3428	1	47	0
azureus	8131	9183	226	229
freenet	5078	9737	167	309
sshterm	16118	24483	767	3642
jgraph	25588	48356	1078	5124
umldot	38930	69332	2146	8785
jbidwatch	97234	143384	2243	11438
columba	111578	174329	3334	18223
gantt	106814	156752	2377	11037
jxplorer	188192	376927	4127	18904
jedit	446896	593847	7132	36832
megamek	179221	353096	3846	22326
oruntenud	248426	497971	5902	25568

Figure 5: Results of escape analysis.

6.4 Experience

All the experimental results reported here are generated using bddbddb. At the early stages of our research, we hand-coded every points-to analysis using BDD operations directly and spent a considerable amount of time tuning their performance. Our contextnumbering scheme is the reason why the analysis would finish at all on even small programs. Every one of the optimizations described in Section 2.4 was first carried out manually. After considerable effort, megamek still took over three hours to analyze, and jxplorer did not complete at all. The incrementalization was very difficult to get correct, and we found a subtle bug months after the implementation was completed. We did not incrementalize the outermost loops as it would have been too tedious and error-prone. It was also difficult to experiment with different rule application orders.

To get even better performance, and more importantly to make it easier to develop new queries and analyses, we created bddbddb. We automated and extended the optimizations we have used in our manual implementation, and implemented a few new ones to empirically choose the best BDD library parameters. The end result is that code generated by bddbddb outperforms our manually tuned context-sensitive pointer analysis by as much as an order of magnitude. Even better, we could use bddbddb to quickly and painlessly develop new analyses that are highly efficient, such as the type analysis in Section 5.5 and the thread escape analysis in Section 5.6.

7. RELATED WORK

This paper describes a scalable cloning-based points-to analysis that is context-sensitive, field-sensitive, inclusion-based and implemented using BDDs. Our program analyses, expressed in Datalog, are translated by bddbddb into a BDD implementation automatically. We also presented example queries using our system to check for vulnerabilities, infer types, and find objects that escape a thread. Due to space constraints, we can only describe work that is very closely related to ours.

Scalable pointer analyses. Most of the scalable algorithms proposed are context-insensitive and flow-insensitive. The first scalable pointer analysis proposed was a unification-based algorithm due to Steensgaard[28]. Das et al. extended the unification-based approach to include "one-level-flow"[10] and one level of context sensitivity[11]. Subsequently, a number of inclusion-based algorithms have been shown to scale to large programs[3, 17, 19, 33].

A number of context-sensitive but flow-insensitive analyses have been developed recently[15, 16]. The C pointer analysis due to

	Cont	text-inser	nsitive po	inters	Proj	jected con	text-sens	itive		Context-	sensitive	
Name	no typ	e filter	with ty	pe filter	pointer	analysis	type a	inalysis	pointer	analysis	type a	inalysis
	multi	refine	multi	refine	multi	refine	multi	refine	multi	refine	multi	refine
freetts	5.1	41.1	2.3	41.6	2.0	41.9	2.5	41.3	0.1	44.4	0.3	44.0
nfcchat	12.4	36.4	8.6	37.0	8.2	37.4	8.6	36.9	0.1	45.9	0.7	45.3
jetty	12.6	36.2	7.7	37.1	7.3	37.4	7.7	37.1	0.1	45.4	0.6	44.8
openwfe	12.1	36.9	7.0	37.7	6.6	38.0	7.0	37.6	0.1	45.5	0.5	44.8
joone	11.9	37.5	6.8	38.1	6.4	38.4	6.7	38.1	0.1	45.8	0.5	45.0
jboss	13.4	37.8	7.9	38.7	7.4	39.3	7.8	38.7	0.1	47.3	0.7	46.4
jbossdep	10.2	40.3	7.4	39.5	7.0	40.0	7.5	39.4	0.2	47.6	0.8	46.6
sshdaemon	10.7	39.3	6.0	40.3	5.8	40.5	5.9	40.4	0.1	46.8	0.6	46.1
pmd	9.6	42.3	6.2	43.1	5.9	43.4	6.2	43.1	0.1	52.1	0.6	48.1
azureus	10.0	43.6	6.1	44.1	6.0	44.3	6.2	44.1	0.1	50.8	0.9	49.7
freenet	12.1	39.1	6.3	40.0	5.9	40.5	6.3	40.1	0.1	47.0	0.8	46.0
sshterm	14.7	40.8	8.9	42.0	8.5	42.5	9.0	42.1	0.6	51.3	1.6	49.9
jgraph	16.1	43.2	9.6	45.1	9.3	45.4	9.7	45.2	0.7	54.7	1.9	53.2
umldot	15.7	42.3	9.4	43.6	9.0	43.9	9.4	43.6	0.6	53.0	2.0	51.2
jbidwatch	14.9	42.3	8.6	43.4	8.2	43.7	8.6	43.4	0.6	52.0	1.7	50.5
columba	15.7	42.3	9.0	43.7	8.6	44.1	8.9	43.9	0.6	52.4	1.8	51.0
gantt	15.0	43.4	8.2	44.7	7.9	45.0	8.2	44.7	0.5	53.0	1.7	51.4
jxplorer	15.2	43.1	7.9	44.3	7.7	44.6	8.0	44.4	0.5	52.5	1.6	50.8
jedit	15.4	43.6	8.1	44.7	7.9	44.9	8.1	44.7	0.6	53.1	1.6	51.5
megamek	13.3	44.6	7.1	45.1	6.8	45.3	7.2	45.2	0.5	53.3	1.4	51.6
gruntspud	15.4	44.0	7.7	45.5	7.5	45.7	7.8	45.5	0.5	53.6	1.4	52.1

Figure 6: Results of the type refinement query. Numbers are percentages. Columns labeled multi and refine refer to multi-type variables and refinable-type variables, respectively.

Fähndrich et al.[15] has been demonstrated to work on a 200K-line gcc program. Unlike ours, their algorithm is unification-based and field-independent, meaning that fields in a structure are modeled as having the same location. Their context-sensitive analysis discovers targets of function pointers on-the-fly. Our algorithm first computes the call graph using a context-insensitive pointer alias analysis; there are significantly more indirect calls in Java programs, the target of our technique, due to virtual method invocations. Their algorithm uses CFL-reachability queries to implement context sensitivity[24]. Instead of computing context-sensitive results and represent them in a form convenient for further analysis.

Other context-sensitive pointer analysis. Some of the earlier attempts of context-sensitive analysis are flow-sensitive[14, 18, 34, 37]. Our analysis is similar to the work by Emami et al. in that they also compute context-sensitive points-to results directly for all the different contexts. Their analysis is flow-sensitive; ours uses flow sensitivity only in summarizing each method intraprocedurally. While our technique treats all members of a strongly connected component in a call graph as one unit, their technique only ignores subsequent invocations in recursive cycles. On the other hand, their technique has only been demonstrated to work for programs under 3000 lines.

As discussed in Section 1, using summaries is another common approach to context sensitivity. It is difficult to compute a compact summary if a fully flow-sensitive result is desired. One solution is to use the concept of partial transfer functions, which create summaries for observed calling contexts[36, 37]. The same summary can be reused by multiple contexts that share the same relevant alias patterns. This technique has been shown to handle C programs up to 20,000 lines.

One solution is to allow only weak updates[34]; that is, a write to a variable only adds a value to the contents of the variable without removing the previously held value. This greatly reduces the power of a flow-sensitive analysis. This approach has been used to handle programs up to 70,000 lines of code. However, on larger programs the representation still becomes too large to deal with. Because the goal of the prior work was escape analysis, it was not necessary to maintain precise points-to relations for locations that escape, so the algorithm achieved scalability by collapsing escaped nodes.

BDD-based pointer analysis. BDDs have recently been used in a number of program analyses such as predicate abstraction[2], shape analysis[23, 38], and, in particular, points-to analysis[3, 39]. Zhu proposed a summary-based context-sensitive points-to analysis for C programs, and reported preliminary experimental results on C programs with less than 5000 lines[39]. Berndl et al. showed that BDDs can be used to compute context-insensitive inclusionbased points-to results for large Java programs efficiently. In the same conference this paper is presented, Zhu and Calman describe a cloning-based context-sensitive analysis for C pointers, assuming that only the safe C subset is used. The largest program reported in their experiment has about 25,000 lines and 3×10^8 contexts[40].

High-level languages and tools for program analysis. The use of Datalog and other logic programming languages has previously been proposed for describing program analyses[12, 25, 30]. Our bddbddb system implements Datalog using BDDs[35] and has been used to compute context-sensitive points-to results and other advanced analyses. Other examples of systems that translate program analyses and queries written in logic programming languages into implementations using BDDs include Toupie[9] and Croco-Pat[4]. Jedd is a Java language extension that provides a relational algebra abstraction over BDDs[20].

8. CONCLUSION

This paper shows that, by using BDDs, it is possible to obtain efficient implementations of context-sensitive analyses using an extremely simple technique: We clone all the methods in a call graph, one per context of interest, and simply apply a context-insensitive analysis over the cloned graph to get context-sensitive results. By numbering similar contexts contiguously, the BDD is able to handle the exponential blowup of contexts by exploiting their commonalities. We showed that this approach can be applied to type inference, thread escape analysis and even fully context-sensitive pointsto analysis on large programs.

This paper shows that we can create efficient BDD-based analyses easily. By keeping data and analysis results as relations, we can express queries and analyses in terms of Datalog. The bddbddb system we have developed automatically converts Datalog programs into BDD implementations that are even more efficient than those we have painstakingly hand-tuned.

Context-sensitive pointer analysis is the cornerstone of deep program analysis for modern programming languages. By combining (1) context-sensitive points-to results, (2) a simple approach to context sensitivity, and (3) a simple logic-programming based query framework, we believe we have made it much easier to create advanced program analyses.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0086160 and an NSF Graduate Student Fellowship. We thank our anonymous referees for their helpful comments.

9. **REFERENCES**

- I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query optimization. *Journal of Logic Programming*, 4(3):259–262, Sept. 1987.
- [2] T. Ball and S. K. Rajamani. A symbolic model checker for boolean programs. In *Proceedings of the SPIN 2000 Workshop on Model Checking of Software*, pages 113–130, Aug. 2000.
- [3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.
- [4] D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, Nov. 2003.
- [5] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, Sept. 1996.
- [6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
- [7] A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.
- [8] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference* on Object-Oriented Programming Systems, Languages, and Applications, pages 1–19, Nov. 1999.
- [9] M.-M. Corsini, K. Musumbu, A. Rauzy, and B. L. Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In *Proceedings of the International Symposium on Programming Language Implementation* and Logic Programming, pages 75–91, Aug. 1993.
- [10] M. Das. Unification-based pointer analysis with directional assignments. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 35–46, June 2000.
- [11] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Static Analysis Symposium*, pages 260–278, July 2001.
- [12] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126, May 1996.
- [13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the* 9th European Conference on Object-Oriented Programming, pages 77–101, Aug. 1995.
- [14] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [15] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the*

SIGPLAN Conference on Programming Language Design and Implementation, pages 253–263, June 2000.

- [16] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7th International Static Analysis Symposium*, pages 175–198, Apr. 2000.
- [17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.
- [18] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.
- [19] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In Proceedings of the 12th International Conference on Compiler Construction, pages 153–169, April 2003.
- [20] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June 2004.
- [21] J. Lind-Nielsen. BuDDy, a binary decision diagram package. http://www.itu.dk/research/buddy/.
- [22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- [23] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Proceedings of the 9th International Static Analysis Symposium*, pages 196–212, Sept. 2002.
- [24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages*, pages 49–61, Jan. 1995.
- [25] T. W. Reps. Demand Interprocedural Program Analysis Using Logic Databases, pages 163–196. Kluwer, 1994.
- [26] O. Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, May 1991.
- [27] F. Somenzi. CUDD: CU decision diagram package release, 1998.
- [28] B. Steensgaard. Points-to analysis in almost linear time. In Symposium on Principles of Programming Languages, pages 32–41, Jan. 1996.
- [29] Java cryptography extension (JCE). http://java.sun.com/products/jce, 2003.
- [30] J. D. Ullman. Principles of Database and Knowledge-Base Systems. Computer Science Press, Rockville, Md., volume II edition, 1989.
- [31] J. Whaley. JavaBDD library. http://javabdd.sourceforge.net.
- [32] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators, pages 58–66, June 2003.
- [33] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, Sept. 2002.
- [34] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 187–206, Nov. 1999.
- [35] J. Whaley, C. Unkel, and M. S. Lam. A BDD-based deductive database for program analysis. http://suif.stanford.edu/bddbddb, 2004.
- [36] R. P. Wilson. Efficient, context-sensitive pointer analysis for C programs. PhD thesis, Stanford University, Dec. 1997.
- [37] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN Conference* on *Programming Language Design and Implementation*, pages 1–12, June 1995.
- [38] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proceedings of the 9th International Static Analysis Symposium*, pages 69–84, Sept. 2002.
- [39] J. Zhu. Symbolic pointer analysis. In Proceedings of the International Conference in Computer-Aided Design, pages 150–157, Nov. 2002.
- [40] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, June 2004.

Type-Safe Method Inlining

Neal Glew^a Jens Palsberg^b

^aIntel Corporation Santa Clara, CA 95054 aglew@acm.org ^bUCLA Computer Science Department 4531K Boelter Hall, Los Angeles, CA 90095 palsberg@ucla.edu

Abstract

In a typed language such as Java, inlining of virtual methods does not always preserve typability. The best known solution to this problem is to insert type casts, which may hurt performance. This paper presents a solution that never hurts performance. The solution is based on a transformation that modifies static type annotations and changes some virtual calls into static calls, which can then be safely inlined. The transformation is parameterised by a flow analysis, and for any analysis that satisfies certain conditions, the transformation is correct and idempotent. The paper presents the transformation, the conditions on the flow analysis, and proves the correctness properties in the context of a variant of Featherweight Java.

Key words: Types, objects, inlining

1 Introduction

1.1 Background

Behavior-preserving program transformations can change the design or even the language of the programs they transform. They are key to several parts of the software engineering process. Compilers, for example, transform programs for efficiency and to translate from high-level languages to machine code. Software engineers also transform programs to improve the design, maintain the program, or evolve the program towards new goals. In the context of objectoriented software engineering, the whole area of refactoring [1] employs program transformations to achieve its goals of evolving software and in particular the design of the software. As such efforts become increasingly ambitious, so does the amount of detail that must be attended to and the importance of doing the transformations correctly. Hence there is an increasing interest in tool support for program transformation. Such tool support is challenging because of new languages and new language features.

Some of today's popular programming languages, including Java [2] and C++ [3], have static type systems. Java also has a bytecode language with a notion of type soundness, based on the concept of bytecode verification [4]. Transformations of programs in these languages must produce programs in these languages themselves, and in particular produce programs that still type check—a property called typability preservation. In a similar sense, a number of recent compilers use typed intermediate languages (*e.g.*, [5–8]) to obtain debugging and optimisation benefits [5,9]. Such compilers also require transformations that are typability preserving when translating from one typed representation to another.

One recent example of a typability-preserving program transformation in the context of software engineering was given by Tip, Kiezun, and Baumer [10]. They provided tool support for a transformation known as *Extract Inter-face/Superclass* for redirecting the access to a class via a newly created interface. This refactoring involves the updating of the types of variables, method parameters, method return types, and field types to make use of the newly added interface.

1.2 The Problem

This paper is concerned with a particular transformation, namely that of inlining of dynamic method calls in a statically-typed object-oriented language. Method inlining is standard in industrial-strength Java virtual machines, see for example [11]. Method inlining is also useful for software engineering and program maintenance; it is a standard refactoring operation that is supported by interactive development environments such as IntelliJ (see http://www.intellij.com). In Java, all method calls are dynamic; they are also known as virtual calls. For comparison, C++ has both dynamic and static calls. While there has been substantial previous work on method inlining (*e.g.*, [12,13]), the known approaches are either for an untyped language, or have to rely on adding type casts or extra types. For example, consider the following well-typed Java program. Note here that from the perspective of preserving type correctness, there is no major difference between working with Java source code and working with Java bytecode.

Both of the method calls x.m() and ((B)new C()).m() have a unique target method that is a small code fragment, so it makes sense to inline these calls.

In both cases, a compiler could inline by taking the body of **m** and replacing **this** with the actual receiver expression to get:

х	=	x.f;			does	\mathtt{not}	type	check
х	=	((B)new	C()).f		does	not	type	check

These two assignments do not type check. The reason is that while this in class C has static type C, both x and (B)new C() have static type B. Hence, both x.f and ((B)new C()).f will yield the compile-time error that there is no f-field in either x or (B)new C().

The problem can be solved by inserting type casts. In their Java compiler, Wright et al. [7] insert type casts (in the form of a typecase expression) of this in all translated method bodies. Applying this idea to our example program produces the following declaration for method m in class C:

B m() {
 return ((C)this).f;
}

After inlining, the two assignments type check:

х	=	((C)x).f;		11	type	checks
х	=	((C)((B)new	C())).f;	11	type	checks

A different approach was taken by Gagnon et al. [14] who first compile Java to a representation of Java bytecode in which variables do not have types, then do the optimizations on that representation, and finally infer types to regain static type annotations. Gagnon et al. use a style of type inference that combines a flow-based style [15] with the types of methods that came from the Java bytecode. Their results show that this works well for a substantial suite of benchmark programs. In general, however, their algorithm for type inference may fail, and in such cases they revert to inserting type casts.

A related approach, which does not require type casts at all, is to add new types to the program. Knoblock and Rehof [16] demonstrated how to add types in a way such that type inference will succeed for all verifiable Java bytecode programs.

In general, inserting type casts may hurt performance, and adding new types may not be acceptable. Since these type casts and extra types are not added in the untyped setting, they are there just for the purposes of satisfying the type system. It is intellectually unsatisfying that we cannot just use the untyped techniques. Until now, it has remained an open problem to devise a scheme for supporting typability-preserving method inlining in a way that does not require the insertion of type casts or extra types. This paper solves the problem.

1.3 Our Approach

The core of the problem is an instance of what we call *type rot*. Perfectly fine type annotations somehow "rot" during a step of method inlining. Before the transformation, the program type checks, but after the transformation, the *same* type annotations are suddenly no good. This observation leads us to the following insight:

Insight 1: For method inlining, transforming statements and expressions is insufficient; we must also transform the type annotations and type casts.

For the code snippet in Section 1.2, the type of x is B, even though the more precise type C could also be used. Similarly, the cast to B could as well be a cast to C. Thus, we can transform the types in a way that preserves well-typedness:

```
C x = new C(); // the type of x has been changed to C
x = x.m();
x = ((C)new C()).m(); // the type cast has been changed to C
```

Inlining then produces the following well-typed code snippet:

С	х	= new C();			
х	=	x.f;	//	type	checks
х	=	((C)new C()).f;	//	type	checks

Notice that for the example, the type transformation does not change the behavior of the program, and, hence, performance is not affected.

To ensure that our type transformation preserves well-typedness, it is designed carefully in the following way. Each type annotation and type case is either left unchanged or is changed to a subtype. For the example, such type transformation is sufficient to enable type-safe method inlining. However, this is not always the case. In Section 5, we present an example with a method call that has a unique receiver but where no type transformation can enable type-safe method inlining.

The centerpiece of our approach is showing that type transformation can be automated and can enable a large number of inlinings.

1.4 Our Result

We present an approach to typability-preserving method inlining that never hurts performance and does not require the insertion of type casts or new types. Our approach has three components:

- (1) **Type transformation:** We change some type annotations and type casts to be more precise.
- (2) **Devirtualisation:** If a dynamic call has a unique target, then it can be devirtualised, that is, changed to a static call.
- (3) **Inlining:** We inline the static calls.

Notice that we view devirtualisation and inlining as two separate components. As in previous work, a dynamic method call $\mathbf{e}.\mathbf{m}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$ can be transformed to a static call $\mathbf{e}.\mathbf{D}::\mathbf{m}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$ if all the objects that \mathbf{e} could evaluate to are instances of classes that inherit \mathbf{m} from a fixed class D. (The expression $\mathbf{e}.\mathbf{D}::\mathbf{m}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$ invokes D's version of \mathbf{m} on \mathbf{e} with arguments \mathbf{e}_1 through \mathbf{e}_n .) The static call $\mathbf{e}.\mathbf{D}::\mathbf{m}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$ can be inlined to $\mathbf{e}'\{\mathtt{this},\mathtt{x}_1,\ldots,\mathtt{x}_n:=\mathbf{e},\mathbf{e}_1,\ldots,\mathbf{e}_n\}$ where D has method \mathbf{m} with body \mathbf{e}' and parameters \mathbf{x}_1 through \mathbf{x}_n . Inlining of a static call is nothing other than applying a nonstandard reduction rule at compile time, and it is straightforward to show that the rule is typability preserving.

So, the main difficulty is to do type transformation and devirtualisation in a typability-preserving manner. For both of them, a compiler needs information that can drive the transformations. In the case of devirtualisation, a standard approach is *flow-directed* devirtualisation. The idea is to use a static program analysis, known as flow analysis, which approximates the results of evaluating expressions. For each expression, it determines a set of classes such that every possible result of evaluating the expression is an instance of one of those classes. Based on such information, a compiler can easily determine whether a dynamic call has a unique target.

The set of classes computed by a flow analysis is easily transformed into a Java type. In particular, the least upper bound of the set, if it exists, can replace the old type annotation. If a least upper bound does not exists, such as for multiple extensions of interfaces, we can fall back to the old static type. We call this *flow-directed type transformation*.

For our example program in Section 1.2, the best flow set for both receiver expressions in the program is $\{C\}$, and the least upper bound for this set is C. Therefore the program transforms into the one shown in Section 1.3; it type checks.

Given that we can do both type transformation and devirtualisation in a flowdirected manner, we are led to our second key insight:

Insight 2: Type transformation and devirtualisation should be done together.

The idea is to do a single flow analysis and then do both of the type transformation and the devirtualisation based on the *same* flow information. While, in theory, one can imagine the use of two flow analyses, it makes sense for a compiler to do just one.

Not all flow analyses enable type-safe method inlining in the style above. However, we give sufficient conditions on a flow analysis for ensuring correctness. We will present the transformation, the conditions on the flow analysis, and prove the correctness properties; all in the context of a variant of Featherweight Java. It is straightforward to extend our approach to full Java. (Students at Purdue have implemented our ideas for the full Java Virtual Machine.)

Because the flow analysis is used for flow-directed type transformation, it is crucial to align the flow analysis with the type system. There are several aspects of Java's type system that lead to unusual conditions on the flow analysis. One example is Java's lack of a bottom type, leading to a nonemptyness condition on flow sets. Another example is that the Java type system allows the use of subtyping in some places but not in others. One of the insights from previous work on aligning flow analysis and type systems [17–20,16] is that subset constraints correspond to subtyping, while equality constraints correspond to "no nontrivial subtyping," that is, types are related only if they are equal. The consequence is that a flow analyses must satisfy subset constraints in some places and equality constraints in others (see [21,22] for examples of subset constraints and [23] for an example of equality constraints and another example of the mixed use of subset and equality constraints).

Note that our transformation is based on a flow analysis which is a wholeprogram analysis. Hence, our transformation is a whole-program transformation. By making suitable conservative assumptions it could be used to transform separate program fragments. How effective this might be is beyond the scope of this paper. More recent work [24] has extended our ideas to a dynamic class loading environment with a just-in-time compiler. Note also that our flow analyses are context insensitive and we leave context-sensitive flow analyses to future work.

The following section presents our variant of Featherweight Java, Section 3 presents the constraints flow analyses must satisfy, Section 4 presents the program transformation, and Section 5 discusses some examples. The proofs of the correctness theorems are presented in three appendices.

2 The Language

We formalise our results in Featherweight Java [25] (FJ) extended with a static call construct, a language we call FJS. The language and its presentation follow the original FJ paper as closely as possible.

As in FJ, an FJS program is a list of class definitions and an expression to be evaluated. Each class definition is in a stylised form. Every class extends another; top-level classes extend Object. Every class has exactly one constructor. This constructor has one parameter for each of the fields of the class, with the same names and in the same order. It first calls the superclass constructor with the parameters that correspond to the superclass's fields. Then it uses the remaining parameters to initialise the fields declared in the class. Constructors are the only place where **super** or **=** appear in an FJS program. The receiver of a field access or method call is always explicit; **this** is used to refer to an object's fields and methods. FJS is functional, so a method body consists just of a return statement with an expression and there is no **void** type. There are just six forms of expressions: variables, field access, object constructors, dynamic casts, dynamic method call, and static method call. Although FJS does not have **super**, static method call can be used to call a superclass's methods. The remainder of this section formalises the language.

2.1 Syntax and Semantics

The syntax of FJS is:

P ::=
$$(\overline{CD}, e)$$

CD ::= class C extends C { \overline{C} $\overline{f^{\ell}}$; K \overline{M} }

The metavariables A, B, C, D, and E range over class names; f and g range over field names; m ranges over method names; x ranges over variables; d and e range over expressions; M ranges over method definitions; K ranges over constructors; CD ranges over class definitions; and P ranges over programs. Object is a class name, but no program may give it a definition; this is a variable, but no program may use it as a parameter. The over bar notation denotes sequences, so $\overline{\mathbf{f}}$ abbreviates \mathbf{f}_1 , \dots , \mathbf{f}_n . This notation also denotes pairs of sequences in an obvious way— $\overline{C} \quad \overline{f^{\ell}}$ abbreviates $C_1 \quad f_1^{\ell_1}, \ldots, C_n \quad f_n^{\ell_n}, \quad \overline{C} \quad \overline{f^{\ell}}$; abbreviates $C_1 \quad f_1^{\ell_1}; \quad \cdots; \quad C_n \quad f_n^{\ell_n};$, and this. $\overline{f}=\overline{f}$ abbreviates this. $f_1=f_1$; ...; this. $f_n=f_n$. The empty sequence is \bullet , and comma concatenates sequences. Sequences of class definitions, field declarations, method definitions, and parameter declarations may not contain duplicate names. We abuse notation and consider a sequence of class definitions to also be a mapping from class names to class definitions, and write $\overline{CD}(C)$ to mean the definition of C under the map corresponding to \overline{CD} . Any class name C except Objectappearing in a program must be given a definition by that program, and the extends clauses of a program must be acyclic.

Class definition class C extends D { $\overline{E} \ \overline{f^{\ell}}$; K \overline{M} } declares class C to be a subclass of D. In addition to the fields of its superclass, C has fields $\overline{f^{\ell}}$ of types \overline{E} . K is the constructor for the class, and it has the stylised form described above. \overline{M} are the methods declared by C, they may be new methods or may override those of D. C also inherits all methods of D that it does not override. Method declaration C m($\overline{D} \ \overline{x^{\ell}}$) {return^{ℓ}e;} declares a method m with return type C, with parameters $\overline{x^{\ell}}$ of types \overline{D} , and that when invoked evaluates expression e and returns it as the result of the call.

As mentioned above, there are six forms of expression: variables \mathbf{x} , field selection $\mathbf{e}.\mathbf{f}^{\ell}$, object constructors $\mathbf{new}^{\ell} \ C(\overline{\mathbf{e}})$, casts $(C)^{\ell}\mathbf{e}$, dynamic method calls $\mathbf{e}.\mathbf{m}(\overline{\mathbf{d}})^{\ell}$, and static method calls $\mathbf{e}.C::\mathbf{m}(\overline{\mathbf{d}})^{\ell}$. The latter invokes C's version of method \mathbf{m} on object \mathbf{e} , which should be in C or one of its subclasses.

Metavariable ℓ ranges over a set of labels. Such a label is used, for example, in C f^{ℓ} to label the field f. Notice that there is a label associated with all expressions, fields, method returns, and formal arguments; these labels are assumed to be unique. For a program P, labels(P) denotes the set of labels used in P. To simplify the technical definitions later, all the field names and argument names must be distinct. Furthermore, the label on any variable occurrence must be the same as the label on its declaration, and any two occurrences of this in a class must have the same label. Any well-typed program can easily

Field Lookup:

$$\overline{fields(\overline{\mathtt{CD}},\mathtt{Object})} = \bullet \tag{1}$$

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } C_0 \{\overline{D}_2 \ \overline{f^\ell}; K \ \overline{M}\} \quad fields(\overline{CD}, C_0) = \overline{D}_1 \ \overline{g}}{fields(\overline{CD}, C) = \overline{D}_1 \ \overline{g}, \overline{D}_2 \ \overline{f}}$$
(2)

Method Type Lookup:

$$\overline{CD}(C) = class \ C \text{ extends } C_0 \ \{\overline{D} \ f^{\ell_1}; \ K \ \overline{M}\} \\
\underline{B_0 \ m(\overline{B} \ \overline{x^{\ell_2}}) \ \{\text{return}^{\ell}e;\} \in \overline{M} \\
\underline{mtype(\overline{CD}, C, m) = \overline{B} \to B_0}$$
(3)

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } C_0 \{\overline{D} \ \overline{f^{\ell}}; K \ \overline{M}\} \text{ m not defined in } \overline{M} }{mtype(\overline{CD}, C, m) = mtype(\overline{CD}, C_0, m)}$$
(4)

Method Body Lookup:

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } C_0 \{\overline{D} \ \overline{f^{\ell_1}}; K \overline{M}\}}{B_0 \ \mathfrak{m}(\overline{B} \ \overline{x^{\ell_2}}) \{\text{return}^{\ell} \mathbf{e};\} \in \overline{M}}$$

$$\frac{1}{mbody(\overline{CD}, C, \mathfrak{m}) = (\ell, \overline{x}, \mathbf{e})}$$
(5)

Class of Method Lookup:

$$\frac{\overline{CD}(C) = class \ C \ extends \ C_0 \ \{\overline{D} \ \overline{f^{\ell_1}}; \ K \ \overline{M}\}}{B_0 \ m(\overline{B} \ \overline{x^{\ell_2}}) \ \{return^{\ell}e;\} \in \overline{M}}$$
(7)

$$\frac{\overline{\text{CD}}(\text{C}) = \text{class C extends } C_0 \{\overline{\text{D}} \ \overline{f^{\ell_1}}; \text{ K } \overline{\text{M}}\} \text{ m not defined in } \overline{\text{M}}}{impl(\overline{\text{CD}}, \text{C}, \text{m}) = impl(\overline{\text{CD}}, \text{C}_0, \text{m})}$$
(8)

Valid Method Overriding:

$$\frac{mtype(\overline{CD}, D, m) = \overline{E} \to E_0 \text{ implies } \overline{C} = \overline{E} \text{ and } C_0 = E_0}{can-declare(\overline{CD}, D, m, \overline{C} \to C_0)}$$
(9)

Fig. 1. Auxiliary Definitions

be transformed to satisfy these conditions. Function *lab* maps an expression, a field name, or an argument name to its label.

Some auxiliary definitions that are used in the rest of the paper appear in Figure 1. Unlike FJ, we do not make the list of class declarations global, but have them appear explicitly as parameters to functions, predicates, and rules. Function $fields(\overline{CD}, C)$ returns a list of C's fields and their types; $mtype(\overline{CD}, C, m)$

$$\frac{fields(\overline{CD}, C) = \overline{D} \ \overline{f}}{(\overline{CD}, X\langle \operatorname{new}^{\ell_1} \ C(\overline{e}) . f_i^{\ell_2} \rangle) \mapsto (\overline{CD}, X\langle e_i \rangle)}$$
(10)

$$\frac{\overline{\mathrm{CD}} \vdash \mathrm{C} <: \mathrm{D}}{(\overline{\mathrm{CD}}, \mathrm{X} \langle (\mathrm{D})^{\ell_1} \mathrm{new}^{\ell_2} \ \mathrm{C}(\overline{\mathrm{e}}) \rangle) \mapsto (\overline{\mathrm{CD}}, \mathrm{X} \langle \mathrm{new}^{\ell_2} \ \mathrm{C}(\overline{\mathrm{e}}) \rangle)}$$
(11)

$$mbody(\overline{CD}, C, m) = (\ell, \overline{x}, e_0)$$
 (12)

$$(\overline{CD}, X\langle \operatorname{new}^{\ell_1} C(\overline{e}) . \mathfrak{m}(\overline{d})^{\ell_2} \rangle) \mapsto (\overline{CD}, X\langle e_0\{\operatorname{this}, \overline{x} := \operatorname{new}^{\ell_1} C(\overline{e}), \overline{d}\} \rangle) \xrightarrow{(12)} mbodu(\overline{CD}, D, \mathfrak{m}) = (\ell, \overline{x}, e_0)$$

$$\overline{(\overline{CD}, X\langle \operatorname{new}^{\ell_1} C(\overline{e}) . D : :m(\overline{d})^{\ell_2} \rangle)} \mapsto (\overline{CD}, X\langle e_0 \{ \operatorname{this}, \overline{x} := \operatorname{new}^{\ell_1} C(\overline{e}), \overline{d} \} \rangle)$$
(13)

Fig. 2. Operational Semantics

returns the type of method m in class C, this type has the form $\overline{D} \rightarrow D_0$ where D_0 is the return type and \overline{D} are the argument types; $mbody(\overline{CD}, C, m)$ returns the body of method m in class C, this has the form $(\ell, \overline{\mathbf{x}}, \mathbf{e})$ where ℓ is the label of the return statement, e is the expression to evaluate, and $\overline{\mathbf{x}}$ are the parameter names; impl(CD, C, m) returns the class from which class C inherits method m (this might be C itself if C declares m), this has the form D::m where D is the class. Predicate $can-declare(\overline{CD}, D, m, \overline{C} \to C_0)$ is true when method m of type $\overline{C} \to C_0$ may be declared in a subclass of D. It checks that if D declares or inherits m then it has the same type, as required by Java's type system. The more general rule with contravariant argument types and covariant result types could be used, and the results of this paper would still hold (the definition of acceptable flow would change slightly). Notice that the definition of $can-declare(\overline{CD}, D, m, \overline{C} \rightarrow C_0)$ uses an implication rather than, say, a conjunction. This is because the definition captures *both* the case where no method m was declared in D or a superclass of D and the case where a method m was indeed declared in D or a superclass of D.

The operational semantics of the language appear in Figure 2. Metavariable X ranges over evaluation contexts, which are expressions with exactly one hole; $X\langle e \rangle$ denotes the expression formed by replacing the hole in X by the expression e. Unlike FJ, in addition to making the list of class declarations explicit in the rules we make the evaluation context explicit as well.

Because the language is functional and each class has exactly one constructor of a particular form, the values of the language, which are all objects, can be represented using object constructors $\mathbf{new}^{\ell} \ \mathbf{C}(\overline{\mathbf{e}})$. Field access reduces to the appropriate element of $\overline{\mathbf{e}}$. The cast $(\mathbf{C})^{\ell_1}\mathbf{new}^{\ell_2} \ \mathbf{D}(\overline{\mathbf{e}})$ reduces to the object \mathbf{new}^{ℓ_2} $\mathbf{D}(\overline{\mathbf{e}})$ if \mathbf{D} is a subclass of \mathbf{C} . If \mathbf{D} is not a subclass of \mathbf{C} then the cast is irreducible representing that the cast fails as a checked run-time error. The method call $\mathbf{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}}) \cdot \mathbf{m}(\overline{\mathbf{d}})^{\ell_2}$ reduces to the method body \mathbf{e}_0 with the actual parameters $\overline{\mathbf{d}}$ substituted for the formal parameters $\overline{\mathbf{x}}$ and the object $\mathbf{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}})$ substituted for this. The method body and formal parameters are obtained by looking up the method \mathbf{m} in class \mathbf{C} , $mbody(\overline{\mathbf{CD}}, \mathbf{C}, \mathbf{m}) = (\ell, \overline{\mathbf{x}}, \mathbf{e}_0)$. Static method call $\operatorname{new}^{\ell_1} C(\overline{e}) . D : :m(\overline{d})^{\ell}$ reduces similarly except that the method is looked up in D not C. Note that this method lookup can be done at compile time and a static method call can be implemented as a direct call rather than an indirect call through a method table.

An irreducible expression is *stuck* if it is of the form $X\langle e.f^{\ell} \rangle$, $X\langle e.m(\overline{d})^{\ell} \rangle$, or $X\langle e.D::m(\overline{d})^{\ell} \rangle$. The type system prevents stuck expressions from occurring during execution of a program (see Theorem 1). Irreducible expressions that are not stuck are of the form $v::=new^{\ell} C(\overline{v})$ or $X\langle (C)^{\ell_1}new^{\ell_2} D(\overline{e}) \rangle$ where D is not a subclass of C; the former represents normal termination with a fully evaluated object, the latter represents a failed cast.

2.2 Type System

The type system consists of the following judgements:

Judgement	Meaning
$\overline{\texttt{CD}} \vdash \texttt{C} <: \texttt{D}$	C is a subtype of D
$\overline{\mathtt{CD}};\Gamma\vdash \mathtt{e}\in\mathtt{C}$	${\bf e}$ is well formed and of type ${\bf C}$
$\overline{\mathtt{CD}} \vdash \mathtt{M} \ \mathrm{OK} \ \mathrm{in} \ \mathtt{C}$	${\tt M}$ is well formed in class ${\tt C}$
$\overline{\texttt{CD}} \vdash \texttt{CD} \ \mathrm{OK}$	CD is well formed
$\vdash \mathtt{P} \in \mathtt{C}$	${\tt P}$ is well formed and of type ${\tt C}$

A typing context Γ has the form $\overline{\mathbf{x}}:\overline{\mathbf{C}}$ where there are no duplicate variable names. The only types are the names of classes, and such a type includes all instances of that class and its subclasses. The rules appear in Figure 3. The bar notation denotes sequences of typing judgements, so $\overline{\mathbf{CD}}$; $\Gamma \vdash \overline{\mathbf{e}} \in \overline{\mathbf{C}}$ abbreviates $\overline{\mathbf{CD}}$; $\Gamma \vdash \mathbf{e}_1 \in \mathbf{C}_1, \ldots, \overline{\mathbf{CD}}$; $\Gamma \vdash \mathbf{e}_n \in \mathbf{C}_n$.

The rules for constructors and method call check that each actual parameter has a subtype of the corresponding formal parameter. The typing rule for dynamic method call looks up the type of the method in the class of the receiver. The typing rule for static method call $e.D::m(\overline{d})^{\ell}$ requires that ehas some subtype of D and looks up the type of the method in D. As in FJ, the typing rules allow *stupid casts*, such as $(C)^{\ell_1}new^{\ell_2} D(\overline{e})$ where D is not a subclass of C and the cast will always fail. Allowing stupid casts is needed to prove type preservation. Unlike FJ, FJS has only one rule for casts, which just requires the expression being cast to have some type. This rule is equivalent to FJ's three rules except that it does not issue stupid-cast warnings. The type system is sound, that is, well-typed programs never get stuck. This fact is stated in the following theorem, which can be proved by standard Subtyping:

$$\overline{\overline{\text{CD}}} \vdash \text{C} <: \text{C} \tag{14}$$

$$\frac{\overline{\text{CD}} \vdash \text{C} <: \text{D}}{\overline{\text{CD}} \vdash \text{C} <: \text{E}}$$
(15)

$$\frac{\overline{\text{CD}}(\text{C}) = \text{class C extends D} \{\ldots\}}{\overline{\text{CD}} \vdash \text{C} <: \text{D}}$$
(16)

Expression Typing:

$$\overline{\overline{\text{CD}}; \Gamma \vdash \mathbf{x}^{\ell} \in \Gamma(\mathbf{x})} \tag{17}$$

$$\frac{\overline{\text{CD}}; \Gamma \vdash \mathbf{e}_0 \in \mathbf{C}_0 \quad fields(\overline{\text{CD}}, \mathbf{C}_0) = \overline{\mathbf{C}} \ \overline{\mathbf{f}}}{\overline{\text{CD}}; \Gamma \vdash \mathbf{e}_0.\mathbf{f}_i^{\ell} \in \mathbf{C}_i}$$
(18)

$$\frac{fields(\overline{CD}, C) = \overline{D} \ \overline{f} \quad \overline{CD}; \Gamma \vdash \overline{e} \in \overline{E} \quad \overline{CD} \vdash \overline{E} <: \overline{D}}{\overline{CD}; \Gamma \vdash new^{\ell} \ C(\overline{e}) \in C}$$
(19)

$$\frac{\overline{\mathrm{CD}}; \Gamma \vdash \mathbf{e}_0 \in \mathrm{D}}{\overline{\mathrm{CD}}; \Gamma \vdash (\mathrm{C})^{\ell} \mathbf{e}_0 \in \mathrm{C}}$$
(20)

$$\begin{array}{ccc} \overline{\operatorname{CD}}; \Gamma \vdash \mathbf{e}_0 \in \operatorname{C}_0 & mtype(\overline{\operatorname{CD}}, \operatorname{C}_0, \mathbf{m}) = \overline{\operatorname{B}} \to \operatorname{C} & \overline{\operatorname{CD}}; \Gamma \vdash \overline{\mathbf{e}} \in \overline{\operatorname{E}} & \overline{\operatorname{CD}} \vdash \overline{\operatorname{E}} <: \overline{\operatorname{B}} \\ \hline \overline{\operatorname{CD}}; \Gamma \vdash \mathbf{e}_0. \mathbf{m}(\overline{\mathbf{e}})^\ell \in \operatorname{C} \end{array} \end{array}$$

$$\overline{CD}; \Gamma \vdash \mathbf{e}_0 \in C_0 \qquad \overline{CD} \vdash C_0 <: D$$

$$mtype(\overline{CD}, D, m) = \overline{B} \to C$$

$$\overline{CD}; \Gamma \vdash \overline{\mathbf{e}} \in \overline{E} \qquad \overline{CD} \vdash \overline{E} <: \overline{B}$$

$$\overline{\overline{CD}}; \Gamma \vdash \mathbf{e}_0 . D :: m(\overline{\mathbf{e}})^{\ell} \in C$$
(22)

Method Typing:

$$\frac{\overline{CD}; \text{this}: D, \overline{x}: \overline{C} \vdash e_0 \in E_0 \qquad \overline{CD} \vdash E_0 <: C_0}{\frac{\overline{CD}(C) = \text{class } D \text{ extends } D_0 \{\ldots\} \qquad can-declare(\overline{CD}, D_0, m, \overline{C} \to C_0)}{\overline{CD} \vdash C_0 \ m(\overline{C} \ \overline{x^{\ell}}) \text{ {return}}^{\ell} e_0; \text{ } OK \text{ in } D}$$
(23)

Class Typing:

$$\overline{CD} \vdash \overline{M} \text{ OK in } C$$

$$fields(\overline{CD}, C_0) = \overline{D}_1 \ \overline{g}$$

$$K = C(\overline{D}_1 \ \overline{g}, \ \overline{D}_2 \ \overline{f}) \ \{\text{super}(\overline{g}); \ \text{this}.\overline{f} = \overline{f}; \}$$

$$\overline{CD} \vdash \text{class } C \text{ extends } C_0 \ \{\overline{D}_2 \ \overline{f^\ell}; \ K \ \overline{M}\} \text{ OK}$$

$$(24)$$

Program Typing:

$$\frac{\overline{\text{CD}} \vdash \overline{\text{CD}} \text{ OK} \quad \overline{\text{CD}}; \bullet \vdash e \in C}{\vdash (\overline{\text{CD}}, e) \in C}$$
(25)

Fig. 3. Typing Rules

methods [26, 27, 25].

Theorem 1 (Type Soundness) $If \vdash P \in C$ then P does not reduce to a program with a stuck expression.

The rules are syntax directed, with the exception of the rules for subtyping. So, disregarding the details of how subtyping judgments are derived, for any program there is exactly one derivation possible. Thus for a program P and any ℓ , which can be the label of a field, method parameter, method return, or expression appearing in P, there is a uniquely determined static type for the program point labeled ℓ , written $static-type(\ell, P)$.

3 Flow Analysis

A flow analysis approximates the results of evaluating expressions. In our setting, flow information for an expression is a set of classes such that the expression will evaluate to an instance of one of those classes.

For a program P, classes(P) denotes the set of class names declared in P, flow(P) is the powerset of classes(P); elements of flow(P) are called flows. The set subclasses(P, C) is the set of subclasses of C (including C). Flow information for P is a member of flow-information(P) = $labels(P) \rightarrow flow(P)$ —it associates a flow with each expression, field, method parameter, and method return. Metavariables S and T range over flow(P), φ ranges over flow-information(P). We order flow-information(P) such that $\varphi_1 \leq \varphi_2$ if and only if $\varphi_1(\ell) \subseteq \varphi_2(\ell)$ for every $\ell \in labels(P)$. In flow-information(P), the least element is $\lambda \ell. \emptyset$ and the greatest element is $\lambda \ell. classes(P)$.

Some members of flow-information(P) are not valid approximations of the results of evaluating expressions in P and do not support our program transformation. The flow analyses with the desired properties are the ones that are both *acceptable* and *type respecting*. (The term "type respecting" was coined by Jagannathan et al. [28].) Intuitively, an acceptable analysis contains sets that are *big* enough, in that it correctly approximates the results of evaluating expressions. A type-respecting analysis contains sets that are *small* enough, in that it is at least as precise as the static type system, that is, each flow only contains classes that are subclasses of the corresponding static type. For a program P, we define:

 $acceptable(P) = \{ \varphi \in flow\text{-}information(P) \mid \\ \varphi \text{ satisfies the conditions listed in Figure 4} \}$

$$\begin{split} type\text{-respecting}(\mathsf{P}) &= \{ \begin{array}{l} \varphi \in \textit{flow-information}(\mathsf{P}) & | \\ & \forall \ell \in \textit{labels}(\mathsf{P}) : \\ & \varphi(\ell) \subseteq \textit{subclasses}(\mathsf{P},\textit{static-type}(\ell,\mathsf{P})) \end{array} \} \\ \textit{flow-analysis}(\mathsf{P}) &= \textit{acceptable}(\mathsf{P}) \cap \textit{type-respecting}(\mathsf{P}). \end{split}$$

The conditions in Figure 4 for a flow analysis to be acceptable are somewhat unusual. The design of those conditions is influenced by the way the program transformation will use flow information to change type annotations: for a program point with label ℓ , the transformation uses the least upper bound of $\varphi(\ell)$, written $\sqcup \varphi(\ell)$, as the new type annotation. With that in mind, here is a closer look at the rules in Figure 4.

Rules (26)-(37) are related to one way of specifying 0-CFA [15,22,19]. The unusual aspect of them is that they are a mixture of subset constraints [22] and equality constraints [19]. If the sole purpose were to approximate the results of evaluating expressions, then all of the equality constraints can be relaxed to be subset constraints; the result would be 0-CFA. The reason for using equality constraints in some cases is to align the flow analysis with the type system. The type system does not have a general subsumption rule that allows subtyping to be used everywhere. Rather, in the type rules in Figure 3, subtyping is used in four places: Rule (19) for new-expressions, Rule (21) for calls, Rule (22) for static calls, and Rule (23) for method typing. In each case, there is a subset constraint in the corresponding rule for acceptable flow analyses in Figure 4: Rule (27) for new-expressions, Rule (30) for dynamic method calls, Rule (33) for static method calls, and Rule (37) for method typing. In contrast, Rule (18) for field selection requires the type of the field to equal the type of the field-selection expression; this is matched by the equality constraint in Rule (26). A similar comment applies to Rules (31) and (34). If there were a general subsumption rule, then that would allow subtyping to be used in the three mentioned cases where it is not allowed in the current definition of Java.

Rules (29), (32), and (35) have no direct counterparts in the type system and are needed to ensure that the flow analysis approximates the results of evaluating expressions. Specifically, Rule (29) models that a type cast to C only can be an object of C or a subclass of C, while Rule (32) and Rule (35) model that a receiver expression will become the this-object in the body of the called method. In Rule (32), the intersection with subclasses(P, D) ensures that the flow set for the this variable will only contain elements of subclasses(P, D). This intersection is correct, because objects of other classes will have a different implementation of m than D::m, so will not flow to ℓ'' ; it is needed because Rule (23) requires this to have type D. In contrast, such an intersection is not needed in Rule (35) because Rule (22) guarantees that the receiver expression has a type that is a subtype of the class that defines the called method. Rule (36) is rather conservative: it says that the this object always can be an object of the class in which this occurs. The rule is needed because of Rule (23) for method typing, which asserts that this has type C. Finally, Rules (38) and (39) ensure that the signature of a method and the signature of an overriding method are the same. First, notice that Rule (40) ensures that least upper bounds are of a nonempty set. This rule is not needed only to ensure that least upper bounds exist, and is not a rule in most flow analyses.

A variant of Class Hierarchy Analysis [12] (CHA) can be defined as follows:

 $CHA(P) = \lambda \ell.subclasses(P, static-type(\ell, P)).$

It is straightforward to show that CHA(P) is the coarsest flow analysis of P, as stated in the following theorem.

Theorem 2 (CHA) CHA(P) is the greatest element of flow-analysis(P).

Note that flow-analysis(P) does not have a least element. This is due to Rule (40) that requires all flows to be nonempty. Without it, flow-analysis(P) always has a least element. This is because constraints of the forms used in Rules (26)–(39) always have a least solution [15]. The least solution can be found in $O(n^3)$ time where n is the size of the program from which the constraints were generated.

If Java had a bottom type, then this type could be used as the least upper bound of the empty set and Rule (40) would not be needed. Furthermore, flow-analysis(P) would be a meet semilattice with both a greatest and least element. However, Java does not have a bottom type, so we have kept this constraint.

The property of being a flow analysis is preserved during computation, as stated in the following theorem, which is proved in Appendix A. (Palsberg [22] proved a similar result for the λ -calculus.)

Theorem 3 (Flow Preservation) If $\varphi \in flow-analysis(P_1)$ and $P_1 \mapsto P_2$, then $\varphi \in flow-analysis(P_2)$ (technically φ restricted to the labels of P_2).

It is straightforward to compute CHA(P). However, since CHA(P) is the greatest element of flow-analysis(P), it is the most conservative choice of flow analysis and will lead to the least number of inlinings. This raises the question of whether other polynomial-time algorithms could do better. The main difficulty is that flow-analysis(P) does not have a least element, so there is not a unique best choice of flow analysis that improves on CHA(P).

To illustrate that indeed there is a better polynomial time algorithm, we now define a flow analysis with mixed constraints and nonempty sets; the analysis

- for each $e.f^{\ell}$ in P: $\varphi(lab(f)) = \varphi(\ell)$ (26)
- for each new^{ℓ} C(\overline{e}) in P, where $fields(\overline{CD}, C) = \overline{D}$ \overline{f} :

$$\varphi(lab(\overline{\mathbf{e}})) \subseteq \varphi(lab(\overline{\mathbf{f}})) \tag{27}$$

$$\mathbf{C} \in \varphi(\ell) \tag{28}$$

• for each (C)^{ℓ}e in P:

$$\varphi(lab(\mathbf{e})) \cap subclasses(\mathbf{P}, \mathbf{C}) \subseteq \varphi(\ell) \tag{29}$$

for each e.m(d)^ℓ in P and each class C in P, where *mbody*(CD, C, m) = (ℓ', x̄, e'), impl(CD, C, m) = D::m, and ℓ'' is the label for D's this occurrences:

$$C \in \varphi(lab(\mathbf{e})) \Rightarrow \varphi(lab(\overline{\mathbf{d}})) \subseteq \varphi(lab(\overline{\mathbf{x}}))$$
(30)

$$\mathbf{C} \in \varphi(lab(\mathbf{e})) \Rightarrow \varphi(\ell') = \varphi(\ell) \tag{31}$$

$$C \in \varphi(lab(\mathbf{e})) \Rightarrow \varphi(lab(\mathbf{e})) \cap subclasses(\mathbf{P}, \mathbf{D}) \subseteq \varphi(\ell'')$$
(32)

for each e.C::m(d)^ℓ in P, where mbody(CD, C, m) = (ℓ', x, e'), impl(CD, C, m) = D::m, and ℓ'' is the label for D's this occurrences

$$\varphi(lab(\overline{\mathbf{d}})) \subseteq \varphi(lab(\overline{\mathbf{x}})) \tag{33}$$

$$\varphi(\ell') = \varphi(\ell) \tag{34}$$

$$\varphi(lab(\mathbf{e})) \subseteq \varphi(\ell'') \tag{35}$$

• for each class C in P, where ℓ is the label for C's this occurrences:

$$\mathbf{C} \in \varphi(\ell) \tag{36}$$

• for each method in P, with body {return ℓe_0 ;}

$$\varphi(lab(\mathbf{e}_0)) \subseteq \varphi(\ell) \tag{37}$$

• for each method name m declared or inherited in class C of P, if

$$\label{eq:cd} \begin{split} \overline{\text{CD}}(\text{C}) &= \text{class C extends } \text{C}_0 \ \{\overline{\text{D}} \ \overline{\text{f}}; \ \text{K} \ \overline{\text{M}}\} \\ mbody(\overline{\text{CD}}, \text{C}, \text{m}) &= (\ell_1, \overline{\textbf{x}}_1, \textbf{e}_1) \\ mbody(\overline{\text{CD}}, \text{C}_0, \text{m}) &= (\ell_2, \overline{\textbf{x}}_2, \textbf{e}_2) \end{split}$$

then

$$\varphi(lab(\overline{\mathbf{x}}_1)) = \varphi(lab(\overline{\mathbf{x}}_2)) \tag{38}$$

$$\varphi(\ell_1) = \varphi(\ell_2) \tag{39}$$

• for each $\ell \in labels(\mathbf{P})$: $\varphi(\ell) \neq \emptyset$ (40)

Fig. 4. Requirements for an acceptable flow analyses φ of a program $P = (\overline{CD}, e)$.

is called MN(P) (for Mixed and Nonempty). First, the notion of *lifting* a flow analysis is:

$$\begin{split} \textit{lift}(\varphi) \, : \, \textit{flow-information}(\mathsf{P}) &\to \textit{flow-information}(\mathsf{P}) \\ \textit{lift}(\varphi) \, = \, \lambda\ell. \left\{ \begin{array}{cc} \varphi(\ell) & \text{if } \sqcup \varphi(\ell) \text{ exists} \\ \{ \ \textit{static-type}(\ell,\mathsf{P}) \ \} & \text{otherwise} \end{array} \right. \end{split}$$

Notice that $lift(\varphi)(\ell) \neq \emptyset$ for all $\ell \in labels(P)$. For Featherweight Java and FJS, $\Box \varphi(\ell)$ exists for all nonempty sets $\varphi(\ell)$. The full Java type system enables multiple subtyping among interfaces which can lead to nonempty flows without a least upper bound.

Second, the definition of MN(P) is:

 φ_0 = the least flow analysis satisfying Rules (26)–(39) MN(P) = the least flow analysis greater than $lift(\varphi_0)$ satisfying Rules (26)–(39)

This algorithm is polynomial time: φ_0 takes polynomial time to compute using the technique of Fähndrich and Aiken [23], which intuitively is a fixed-point computation with $\lambda \ell.\emptyset$ as the starting point. Lifting clearly takes polynomial time, and MN(P) takes polynomial time to compute by using the Fähndrich-Aiken algorithm again, but this time with $lift(\varphi_0)$ as the starting point for the fixed-point computation. This two-step procedure makes $\sqcup MN(P)(\ell)$ equal to $static-type(\ell, P)$ for any $\ell \in labels(P)$ such that $\varphi_0(\ell) = \emptyset$. Thus the program transformation based on MN(P) will not change the type annotation for the program points labeled by such ℓ .

It is left to future work to settle whether there is a way to avoid both Rule (40) and the two-step procedure of MN(), by somehow mapping the empty flow set to a legal Java type.

There might be worthwhile elements of flow-analysis(P) other than CHA(P) and MN(P). Any element of flow-analysis(P) can be used as an argument to the program transformation, which we present next.

4 Program Transformation

The program transformation is parameterised by a flow analysis, and it operates on program fragments and type environments in a compositional fashion. It transforms each program fragment into a similar program fragment with the same label, and it transforms each type environment into a type environment which defines the same variables. The changes made are that:

- it changes some dynamic method calls to static method calls,
- it changes the type annotations, and
- it changes the classes used in type casts.

In each case, the change is made on the basis of the supplied flow analysis. Specifically, (1) a dynamic call is changed to a static call when the flow analysis determines that there is a unique target method and (2) a type annotation and the class in a type cast are changed to the least upper bound of the classes in the corresponding flow. Taking the least upper bound of the classes in a flow is justified as follows. The transformation is restricted to flow analyses that are acceptable and type respecting, which means the following. First, all flows are nonempty. Second, nonempty sets of classes admit least upper bounds because FJS is a single-inheritance language. Third, the type-respecting property implies that the new types (that is, the least upper bounds) can only be more refined than the old ones.

Transformation	Meaning
$\llbracket \mathtt{P} \rrbracket_{\varphi}$	the transformation of ${\tt P}$ using φ
$\llbracket \mathtt{CD} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}$	the transformation of CD using φ and $\overline{\tt CD}$
$\llbracket \mathtt{K} \rrbracket_{\varphi}$	the transformation of K using φ
$\llbracket M \rrbracket_{\varphi}^{\overline{CD}}$	the transformation of \mathtt{M} using φ and $\overline{\mathtt{CD}}$
$\llbracket \mathbf{e} \rrbracket_{arphi}^{\overline{\mathtt{CD}}}$	the transformation of ${\tt e}$ using φ and $\overline{\tt CD}$
$\llbracket \Gamma \rrbracket_{\varphi}$	the transformation of Γ using φ

The transformation consists of the following cases:

The definition of the transformation appears in Figure 5.

We now present four correctness theorems: the transformation preserves typability, the transformation is operationally correct, a flow analysis of the original program is also a flow analysis of the transformed program, and the transformation is idempotent. First our main result, which is proved in Appendix B.

Theorem 4 (Typability Preservation) Suppose $\varphi \in flow-analysis(P)$ and $P = (\overline{CD}, e)$. If $\vdash P \in C$ then $\vdash [\![P]\!]_{\varphi} \in \sqcup \varphi(lab(e))$.

The transformation is also operationally correct, in that the transformed program simulates the original program step for step and vice versa, as stated in

$$\begin{split} \left[(\overline{\text{CD}}, e) \right]_{\varphi} &= \left(\left[\overline{\text{CD}} \right]_{\varphi}^{\overline{\text{CD}}}, \left[e \right]_{\varphi}^{\overline{\text{CD}}} \right) \\ \left[\text{class C extends C}_{0} \left\{ \overline{\text{D}} \ \overline{\text{f}^{\ell}}; \ \text{K} \ \overline{\text{M}} \right\} \right]_{\varphi}^{\overline{\text{CD}}} &= \text{class C extends C}_{0} \\ \left\{ \Box \varphi(\overline{\ell}) \ \overline{\text{f}^{\ell}}; \ \left[\text{K} \right]_{\varphi} \ \left[\overline{\text{M}} \right]_{\varphi}^{\overline{\text{CD}}} \right\} \\ \left[\mathbb{C}(\overline{\text{D}} \ \overline{\text{f}}) \ \left\{ \text{super}(\overline{\text{f1}}); \ \text{this.} \overline{\text{f2}} = \overline{\text{f2}} \right\} \right]_{\varphi} &= \mathbb{C}(\Box \varphi(lab(\overline{\text{f}})) \ \overline{\text{f}}) \\ \left\{ \text{super}(\overline{\text{f1}}); \ \text{this.} \overline{\text{f2}} = \overline{\text{f2}} \right\} \\ \left[\mathbb{D} \ \text{m}(\overline{\text{E}} \ \overline{\text{x}^{\ell}}) \ \left\{ \text{return}^{\ell} e; \right\} \right]_{\varphi}^{\overline{\text{CD}}} &= \Box \varphi(\ell) \ \text{m}(\Box \varphi(\overline{\ell}) \ \overline{\text{x}^{\ell}}) \\ \left\{ \text{return}^{\ell} \left[e \right]_{\varphi}^{\overline{\text{CD}}}; \right\} \end{split}$$

$$\begin{split} & \left[\mathbf{x}^{\ell} \right]_{\varphi}^{\overline{\text{CD}}} &= \mathbf{x}^{\ell} \\ & \left[\mathbf{e}.\mathbf{f}^{\ell} \right]_{\varphi}^{\overline{\text{CD}}} &= \left[\mathbf{e} \right]_{\varphi}^{\overline{\text{CD}}} . \mathbf{f}^{\ell} \\ & \left[\mathbf{n}\mathbf{e}\mathbf{w}^{\ell} \ \mathbf{D}(\overline{\mathbf{e}}) \right]_{\varphi}^{\overline{\text{CD}}} &= \mathbf{n}\mathbf{e}\mathbf{w}^{\ell} \ \mathbf{D}(\left[\overline{\mathbf{e}} \right]_{\varphi}^{\overline{\text{CD}}}) \\ & \left[(\mathbf{D})^{\ell} \mathbf{e} \right]_{\varphi}^{\overline{\text{CD}}} &= \left(\sqcup \varphi(\ell) \right)^{\ell} \left[\mathbf{e} \right]_{\varphi}^{\overline{\text{CD}}} \\ & \left[\mathbf{e}.\mathbf{m}(\overline{\mathbf{d}})^{\ell} \right]_{\varphi}^{\overline{\text{CD}}} &= \left[(\sqcup \varphi(\ell))^{\ell} . \mathbf{m}(\left[\overline{\mathbf{d}} \right]_{\varphi}^{\overline{\text{CD}}} \right)^{\ell} \\ & \text{where } \forall \mathbf{E} \in \varphi(lab(\mathbf{e})) : impl(\overline{\text{CD}}, \mathbf{E}, \mathbf{m}) = \mathbf{D} : : \mathbf{m} \\ & \left[\mathbf{e}.\mathbf{m}(\overline{\mathbf{d}})^{\ell} \right]_{\varphi}^{\overline{\text{CD}}} &= \left[\left[\mathbf{e} \right]_{\varphi}^{\overline{\text{CD}}} . \mathbf{m}(\left[\overline{\mathbf{d}} \right]_{\varphi}^{\overline{\text{CD}}} \right)^{\ell} \\ & \text{otherwise} \\ & \left[\mathbf{e}.\mathbf{D} : : \mathbf{m}(\overline{\mathbf{d}})^{\ell} \right]_{\varphi}^{\overline{\text{CD}}} &= \left[\left[\mathbf{e} \right]_{\varphi}^{\overline{\text{CD}}} . \mathbf{D} : : \mathbf{m}(\left[\overline{\mathbf{d}} \right]_{\varphi}^{\overline{\text{CD}}} \right)^{\ell} \\ & \left[\left[\mathbf{x}_{1} : \mathbf{C}_{1}, \dots, \mathbf{x}_{n} : \mathbf{C}_{n} \right]_{\varphi} &= \mathbf{x}_{1} : \sqcup \varphi(lab(\mathbf{x}_{1})), \dots, \\ & \mathbf{x}_{n} : \sqcup \varphi(lab(\mathbf{x}_{n})) \end{array} \end{split}$$

Fig. 5. The Transformation of Dynamic to Static Dispatch

the following theorem, which is proved in Appendix C. Operational correctness for a multistep computation follows from Theorems 3 and 5.

Theorem 5 (Operational Correctness) If $\varphi \in \text{flow-analysis}(P_1)$ then: $P_1 \mapsto P_2 \text{ if and only if } [\![P_1]\!]_{\varphi} \mapsto [\![P_2]\!]_{\varphi}.$

It is straightforward to prove that a flow analysis of a program is also a flow analysis of the transformed program, as stated in the following theorem.

Theorem 6 (Analysis Preservation) If $\varphi \in flow-analysis(\mathbb{P})$, then $\varphi \in flow-analysis([[\mathbb{P}]]_{\varphi})$.

```
class A {
                      void m(Q arg) {
                                              class Q {
A x = new A();
                                                  void p() {...}
                         arg.p();
B y = new B();
                      }
                                              }
                  }
                                              class S extends Q {
x.m(new Q());
                  class B extends A {
                                                  void p() {...}
y.m(new S());
                                              }
                      void m(Q arg) {...}
                  }
```

Fig. 6. Example program

Given a flow analysis, it is sufficient to apply the transformation only once: applying the transformation again with the *same* flow analysis will not lead to any further change. We can state this as the following idempotence property of the transformation, which is straightforward to prove.

Theorem 7 (Idempotence) If $\varphi \in flow\text{-information}(\mathbb{P})$ then $\llbracket \llbracket \mathbb{P} \rrbracket_{\varphi} \rrbracket_{\varphi} = \llbracket \mathbb{P} \rrbracket_{\varphi}$.

5 Examples

We first present an example that illustrates the difference between CHA(P), ∂ -CFA(P), and our analysis MN(P). Consider the example program in Figure 6. There are four classes: A, B, Q, and S, where B extends A and where S extends Q. Notice that A has a method m, and that B also has a method m that overrides the one from A. Similarly Q has a method p, and S also has a method p that overrides the one from Q. Among these methods, only the body of A.m is of interest here, while the bodies of the other methods are left unspecified. To the left of the classes are four lines of code that should be seen as being part of some other class. The first two lines are declarations of fields x and y, and the last two lines are method calls.

Now consider which of the method calls in the program will be inlined based on CHA(P), 0-CFA(P), and our analysis MN(P).

First consider CHA(P). In the first method call x.m(new Q()), the static type of x is A. Since A has a subclass B, $CHA(P)(lab(x)) = \{A, B\}$. Note that A and B have different implementations of m, so CHA(P) does not lead to inlining of the method call x.m(new Q()).

By the way, for the example program, Bacon and Sweeney's Rapid Type Analysis (RTA) [29,30] gives the same result as CHA. RTA is similar to CHA except that its flow sets contain only classes that are actually instantiated in the program [31]. In the example program, objects are created from all four classes, so there is no difference between RTA and CHA in this case. A x = new A();class A { Qy; class Q { Q m() { if (...) { void p() {...} // infinite y = new S();} // recursion! } else { class S extends Q { return this.m(); y = x.m();void p() {...} } } } } y.p();

Fig. 7. Example that shows empty flow sets

In the second method call y.m(new S()), the static type of y is B. Since B has no subclasses, $CHA(P)(lab(y)) = \{B\}$. Clearly, B has just one implementation of m, so CHA(P) leads to inlining of the method call x.m(new Q()).

In the third method call $\arg.p()$, the static type of \arg is Q. Since Q has a subclass S, $CHA(P)(lab(\arg)) = \{Q, S\}$. Note that Q and S have different implementations of p, so CHA(P) will not lead to inlining of the method call $\arg.p()$.

Second consider 0-CFA(P). The initialisations of x and y show that an A-object flows to x and that a B-object flows to y. So, in the first method call x.m(new Q()), 0- $CFA(P)(lab(x)) = \{A\}$, and in the second method call y.m(new S()), 0- $CFA(P)(lab(y)) = \{B\}$, and thus 0-CFA(P) leads to inlining of both those method calls. In the third method call arg.p(), the receiver is arg, and the only value that flows to arg is the Q-object from the call site x.m(new Q()). So, 0-CFA(P) also leads to inlining of the third method call.

Third consider our analysis MN(P), which, by construction, is at least as precise as CHA(P) and at most as precise as 0-CFA(P). For the first two method calls, it is straightforward to see that MN(P) gives the same results as 0-CFA(P) for **x** and **y**, Hence, MN(P) leads to inlining of both those method calls.

However, 0-CFA(P) and MN(P) differ on the third method call $\arg.p()$. Here, Rule (38) forces the flow sets for \arg in A.m to be the same as for \arg in B.m. The second call site y.m(new S()) shows that an S-object flows to \arg in B.m. So, the unified flow set for both \arg in A.m and \arg in B.m is {Q, S}, and hence the call site is not inlined.

In conclusion, CHA(P) leads to the inlining of one call site, MN(P) leads to the inlining of two call sites, while ∂ -CFA(P) leads to the inlining of three call sites.

As a final example, consider the program in Figure 7.

In this example, class S extends Q and both have a p method. Notice that variable y is initialised either with an instance of S or with the result of

new A().m(), which never returns. Clearly in any run of this program only an instance of S will reach the call site y.p(), so it is safe to inline S::p at this point. Since A::m does not return, ∂ -CFA(P)(lab(A::m)) = Ø, so in turn, ∂ -CFA(P)(lab(y)) = {S}. Thus ∂ -CFA(P) leads to inlining of the call site. However, our analysis will return $MN(P)(lab(A::m)) = \{Q,S\}$, so in turn, $MN(P)(lab(y)) = \{Q,S\}$. Thus MN(P) does not lead to inlining of the call site.

Observe that, in both examples, our analysis did not inline some call sites because of constraints imposed by the type system. In the first example, the constraints were due to Java's invariant subtyping for method parameters and returns. In the second example, the constraints were due to Java's lack of a bottom type. Based on the result of Palsberg and O'Keefe [17], one can imagine a more expressive type system for Java that would admit θ -CFA(P) as a type-preserving analysis, and that would allow all of the inlining discussed above. In particular, such a type system would likely have a bottom type and likely have depth subtyping for methods.

6 Conclusion

We have shown how to inline methods while preserving typability in a singleinheritance language without resorting to the insertion of type casts or new types. Our approach is based on flow analysis, and we found it tricky to get the requirements for the flow analysis right. During the process of proving correctness, we discovered the need for flow constraints that would not usually be used in a flow analysis, e.g., Rule (36). The requirement that all flow sets must be nonempty is unusual, and it entails that there is no unique best analysis that satisfies the requirements. While CHA and our own MN analysis satisfy the requirements, more work is needed to investigate alternatives.

Concerning the effectiveness of the various analyses, our analysis MN is always at least as precise as CHA and at most as precise as 0-CFA. This is because MN imposes fewer flow constraints than CHA and more flow constraints than 0-CFA.

One can evaluate the practical effectiveness of an inlining strategy in at least two ways: the static count of inlined calls and the run-time count of inlined calls. With regards to the static count of inlined calls, CHA is an excellent baseline. Tip and Palsberg [31] showed that RTA (the variant of CHA discussed earlier) inlines 92.2% of all virtual call sites in a large suite of Java programs. Thus, even though one can try better analyses, the room for improvement is just the remaining 7.8% of the virtual call sites. Tip and Palsberg did experiment with other analyses than RTA, but even their most powerful analysis inlined just 93.0% of all virtual call sites. Thus, even though MN is better than CHA and RTA, we conjecture that there will be a small difference in the number of call sites that will be inlined. The important property of MNis that it guarantees that inlining can be done without the insertion of type casts.

In practice, the run-time count of inlined calls is more important than the static count. For example, the inlining of a method call in an inner loop can have a major impact on the run-time performance. So, even though an analysis might inline less than one percent more of the statically-counted call sites, some of those calls might occur in frequently executed code and therefore be important to inline. Sundaresan et al. [32] investigated an analysis called VTA which is more powerful that CHA, and they found that their analysis leads to nontrivial improvements in run-time performance compared to CHA. They conclude: "some of the extra calls sites found by VTA could be important ones for inlining."

One idea for future work is to do *two* flow analyses of a program: one which is typability preserving, and one that is more powerful but not necessarily typability preserving. The difference between the two sets of call sites suggested for inlining can then be inlined *with* type casts, in the style of Wright et al. [7]. In this way, the performance penalty of the type casts is only payed when deemed necessary.

Another idea, due to Ralf Laemmel, is to change the type system such that 0-CFA would lead to a type preserving transformation, perhaps with inspiration from the equivalence result of Palsberg and O'Keefe [17].

Acknowledgements. We thank the 84 students who took Palsberg's graduate course on programming languages in 1999-2001 and tried, as a homework, to find and implement a solution to the problem of type-safe method inlining for a subset of Java. Only 10 of the implementations seemed not to have errors, leading to the realisation that the problem is considerably harder than flow-directed inlining for an untyped language.

A preliminary version of this paper was presented at ECOOP'02, European Conference on Object-Oriented Programming 2002. We thank Ralf Laemmel, Mayur Naik, and the anonymous referees for helpful comments on a draft of the paper. We also thank Ben Titzer for finding a mistake in an earlier version of the flow analysis.

Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

A Proof of Theorem 3

First, observe that $labels(P_2) \subseteq labels(P_1)$ so φ (restricted to $labels(P_2)$) is a flow analysis of P₂. Let P_i = (\overline{CD} , X(e_i)) where e_1 and e_2 are as in the rules of Figure 2. Since P₁ and P₂ differ only in e_1 and e_2 , φ satisfies the conditions for acceptability and type respecting for P₂ except for the conditions on e_2 and its subexpressions that are not subexpressions of e_1 , and on the expression that e_1 appears immediately within. By inspection of the rules, the last condition will hold if $\varphi(lab(e_2)) \subseteq \varphi(lab(e_1))$. Thus, we need just to show the latter and that φ satisfies the conditions for e_2 and its subexpressions that are not subexpressions of e_1 . Consider the various cases for the reduction rule.

field selection: In this case $\mathbf{e}_1 = \mathbf{new}^{\ell_1} \ \mathsf{C}(\overline{\mathbf{e}}) \cdot \mathbf{f}_i^{\ell_2}, \mathbf{e}_2 = \mathbf{e}_i$, and $fields(\overline{\mathsf{CD}}, \mathsf{C}) = \overline{\mathsf{D}} \ \overline{\mathbf{f}}$. Thus, \mathbf{e}_2 is a subexpression of \mathbf{e}_1 . By Rule 27, $\varphi(lab(\mathbf{e}_i)) \subseteq \varphi(lab(\mathbf{f}_i))$; by Rule 26, $\varphi(lab(\mathbf{f}_i)) = \varphi(\ell_2)$. By transitivity, $\varphi(lab(\mathbf{e}_2)) = \varphi(lab(\mathbf{e}_i)) \subseteq \varphi(\ell_2) = \varphi(lab(\mathbf{e}_1))$, as required.

cast: In this case $\mathbf{e}_1 = (\mathbf{D})^{\ell_1} \operatorname{new}^{\ell_2} C(\overline{\mathbf{e}})$, $\mathbf{e}_2 = \operatorname{new}^{\ell_2} C(\overline{\mathbf{e}})$, and $\overline{CD} \vdash C <: D$. Thus, \mathbf{e}_2 is a subexpression of \mathbf{e}_1 . By the type respecting property and Rule 19, $\varphi(\ell_2) \subseteq subclasses(\mathbf{P}_1, \mathbf{C})$. Since $\overline{CD} \vdash C <: D$, $subclasses(\mathbf{P}_1, \mathbf{C}) \subseteq subclasses(\mathbf{P}_2, \mathbf{D})$, so:

$$\varphi(\ell_2) \cap subclasses(\mathsf{P}_1, \mathsf{D}) = \varphi(\ell_2) \tag{A.1}$$

By Rule 29, $\varphi(\ell_2) \cap subclasses(\mathsf{P}_1, \mathsf{D}) \subseteq \varphi(\ell_1)$. Thus by (A.1), $\varphi(\ell_2) \subseteq \varphi(\ell_1)$, as required.

dynamic method call: In this case both $\mathbf{e}_1 = \mathbf{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}}) \cdot \mathbf{m}(\overline{\mathbf{d}})^{\ell_2}$ and $\mathbf{e}_2 = \mathbf{e}_0\{\mathtt{this}, \overline{\mathbf{x}} := \mathtt{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}}), \overline{\mathbf{d}}\}$ where $mbody(\overline{CD}, \mathbf{C}, \mathbf{m}) = (\ell', \overline{\mathbf{x}}, \mathbf{e}_0)$. By Rule 28, $\mathbf{C} \in \varphi(\ell_2)$. So by Rule 30, $\varphi(lab(\overline{\mathbf{d}})) \subseteq \varphi(lab(\overline{\mathbf{x}}))$, and by Rule 32,

$$\varphi(\ell_1) \cap subclasses(\mathsf{D},\mathsf{P}) \subseteq \varphi(\ell'')$$

where $impl(\overline{CD}, C, m) = D::m$ and ℓ'' is the label for D's this occurances. Since φ satisfies the conditions for type respecting, and the static type of ℓ_1 must be C by Rule 19, $\varphi(\ell_1) \subseteq subclasses(C, P)$. By the rules, $\overline{CD} \vdash C <:$ D, so $subclasses(C, P) \subseteq subclasses(D, P)$. Thus $\varphi(\ell_1) \cap subclasses(D, P) = \varphi(\ell_1)$ and $\varphi(\ell_1) \subseteq \varphi(\ell'')$. By Lemma 8 (below), φ satisfies the conditions for acceptability and type respecting for \mathbf{e}_2 and all its subexpressions. By Rule 31, $\varphi(\ell') = \varphi(\ell_2)$; by Rule 37, $\varphi(lab(\mathbf{e}_0)) \subseteq \varphi(\ell')$. Also by Lemma 8, $\varphi(lab(\mathbf{e}_2)) \subseteq \varphi(lab(\mathbf{e}_0))$. Thus $\varphi(lab(\mathbf{e}_2)) \subseteq \varphi(\ell') = \varphi(\ell_2) = \varphi(lab(\mathbf{e}_1))$, as required.

static method call: In this case both $\mathbf{e}_1 = \mathbf{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}}) . \mathbf{D} :: \mathbf{m}(\overline{\mathbf{d}})^{\ell_2}$ and $\mathbf{e}_2 = \mathbf{e}_0\{\mathtt{this}, \overline{\mathbf{x}} := \mathbf{new}^{\ell_1} \ \mathbf{C}(\overline{\mathbf{e}}), \overline{\mathbf{d}}\}$ where $mbody(\overline{\mathbf{CD}}, \mathbf{D}, \mathbf{m}) = (\ell', \overline{\mathbf{x}}, \mathbf{e})$. By Rule 33, $\varphi(lab(\overline{\mathbf{d}})) \subseteq \varphi(lab(\overline{\mathbf{x}}))$, and by Rule 35, $\varphi(\ell_1) \subseteq \varphi(lab(\mathtt{this}))$. By Lemma 8 (below), φ satisfies the conditions for acceptability and type respecting for \mathbf{e}_2 and all its subexpressions. By Rule 34, $\varphi(\ell') = \varphi(\ell_2)$, and by Rule 37,

 $\varphi(lab(\mathbf{e}_0)) \subseteq \varphi(\ell')$. Also by Lemma 8, $\varphi(lab(\mathbf{e}_2)) \subseteq \varphi(lab(\mathbf{e}_0))$. Thus

$$\varphi(lab(\mathbf{e}_2)) \subseteq \varphi(lab(\mathbf{e}_0)) \subseteq \varphi(\ell') = \varphi(\ell_2) = \varphi(lab(\mathbf{e}_1))$$

as required.

Lemma 8 If φ satisfies the conditions for acceptability and type respecting for all labels in \mathbf{e} and $\overline{\mathbf{d}}$ and if $\varphi(lab(\overline{\mathbf{d}})) \subseteq \varphi(lab(\overline{\mathbf{x}}))$ then φ satisfies the conditions for acceptability and type respecting for all labels in $\mathbf{e}\{\overline{\mathbf{x}} := \overline{\mathbf{d}}\}$ and $\varphi(lab(\mathbf{e}\{\overline{\mathbf{x}} := \overline{\mathbf{d}}\})) \subseteq \varphi(lab(\mathbf{e})).$

Proof. Straightforward.

B Proof of Theorem 4

Theorem 4 follows immediately from Rule (25), and from Lemma 9 and Lemma 11, as stated and proved below.

Lemma 9 Suppose $\varphi \in acceptable(P) \cap type-respecting(P)$, and $P = (\overline{CD}, e_0)$. If $\overline{CD} \vdash \overline{CD} OK$, then $[[\overline{CD}]]^{\overline{CD}}_{\varphi} \vdash [[\overline{CD}]]^{\overline{CD}}_{\varphi} OK$.

Proof. Immediate from Lemmas 10 and 14 (below), using Rule (24).

Lemma 10 Suppose $\varphi \in acceptable(P) \cap type\text{-respecting}(P)$, and $P = (\overline{CD}, e_0)$. If $\overline{CD} \vdash \overline{M} \ OK \ in \ C$, then $[\overline{CD}]_{\varphi}^{\overline{CD}} \vdash [\overline{M}]_{\varphi}^{\overline{CD}} \ OK \ in \ C$.

Proof. Straightforward from Lemmas 11, 12, 15, and 16 (below), using Rules (17), (23), (36), and (37).

Lemma 11 Suppose $\varphi \in acceptable(P) \cap type\text{-respecting}(P)$, and $P = (\overline{CD}, e_0)$. If \overline{CD} ; $\Gamma \vdash e \in D$, then $[\overline{CD}]_{\varphi}^{\overline{CD}}$; $[\Gamma]_{\varphi} \vdash [e]_{\varphi}^{\overline{CD}} \in \sqcup \varphi(lab(e))$.

Proof. We proceed by induction on the structure of the derivation of \overline{CD} ; $\Gamma \vdash e \in D$. There are six cases, depending on which one of Rules (17)–(22) was the last one used to derive \overline{CD} ; $\Gamma \vdash e \in D$.

- (17) e ≡ x^ℓ. We have [[x^ℓ]]^{CD}_φ = x^ℓ and [[Γ]]_φ(x) = ⊔φ(ℓ), so we can derive, using Rule (17), [[CD]]^{CD}_φ; [[Γ]]_φ ⊢ [[e]]^{CD}_φ ∈ ⊔φ(ℓ), as desired.
 (18) e ≡ e₀.f_i^ℓ. We have CD; Γ ⊢ e₀ ∈ C₀ and fields(CD, C₀) = C f, where f_i
- (18) $\mathbf{e} \equiv \mathbf{e}_0 \cdot \mathbf{f}_i^{\ell}$. We have \overline{CD} ; $\Gamma \vdash \mathbf{e}_0 \in C_0$ and $fields(\overline{CD}, C_0) = \overline{C} \ \overline{\mathbf{f}}$, where \mathbf{f}_i occurs in $\overline{\mathbf{f}}$. From the induction hypothesis we have $[\overline{CD}]_{\varphi}^{\overline{CD}}$; $[\Gamma]_{\varphi} \vdash [\mathbf{e}_0]_{\varphi}^{\overline{CD}} \in \Box\varphi(lab(\mathbf{e}_0))$. From $\varphi \in type\text{-respecting}(\mathbf{P})$ we have $\overline{CD} \vdash \Box\varphi(lab(\mathbf{e}_0)) <: \mathbf{C}_0$ so $fields(\overline{CD}, \Box\varphi(lab(\mathbf{e}_0))) = \overline{\mathbf{D}} \ \overline{\mathbf{g}} \ \overline{\mathbf{C}} \ \overline{\mathbf{f}}$. Hence, from Lemma 14 (below), we have $fields([\overline{CD}]_{\varphi}^{\overline{CD}}, \Box\varphi(lab(\mathbf{e}_0))) = \Box\varphi(lab(\overline{\mathbf{g}})) \ \overline{\mathbf{g}} \ \Box\varphi(lab(\overline{\mathbf{f}})) \ \overline{\mathbf{f}}$. From Rule (26) we

have $\varphi(lab(\mathbf{f}_{\mathbf{i}})) = \varphi(\ell)$, so $\sqcup \varphi(lab(\mathbf{f}_{\mathbf{i}})) = \sqcup \varphi(\ell)$, so we can derive, using Rule (18), that $\llbracket \overline{\mathbb{CD}} \rrbracket_{\varphi}^{\overline{\mathbb{CD}}}; \llbracket \Gamma \rrbracket_{\varphi} \vdash \llbracket \mathbf{e} \rrbracket_{\varphi}^{\overline{\mathbb{CD}}} \in \sqcup \varphi(\ell)$, as desired. • (19) $\mathbf{e} \equiv \mathbf{new}^{\ell} \ \mathbb{C}(\overline{\mathbf{e}})$. We have $fields(\overline{\mathbb{CD}}, \mathbb{C}) = \overline{\mathbb{D}} \ \overline{\mathbf{f}}$, and $\overline{\mathbb{CD}}; \Gamma \vdash \overline{\mathbf{e}} \in \overline{\mathbb{E}}$,

- (19) e ≡ new^ℓ C(ē). We have fields(CD, C) = D f, and CD; Γ ⊢ ē ∈ E, and CD ⊢ E <: D. From Lemma 14 (below) we have fields([CD]]^{CD}_φ, C) = ⊔φ(lab(f)) f. From the induction hypothesis we have [CD]]^{CD}_φ; [Γ]]_φ ⊢ [ē]]^{CD}_φ ∈ ⊔φ(lab(ē)). From Rule (27), we have φ(lab(ē)) ⊆ φ(lab(f)), so from Lemma 16 (below) we have CD ⊢ ⊔φ(lab(ē)) <: ⊔φ(lab(f)), and so from Lemma 15 (below) we have [CD]]^{CD}_φ ⊢ ⊔φ(lab(ē)) <: ⊔φ(lab(f)). Finally we have from Rule (28) that C ∈ φ(ℓ), and since new^ℓ C(ē) has static type C and φ ∈ type-respecting(P), we have ⊔φ(ℓ) = C. We conclude, using Rule (19), that we have [CD]^{CD}_φ: [Γ]]_φ ⊢ new^ℓ C([ē]]^{CD}_φ) ∈ C.
 (20) e ≡ (C)^ℓe₀. We have CD; Γ ⊢ e₀ ∈ D. From the induction hypothesis
- (20) $\mathbf{e} \equiv (\mathbf{C})^{\ell} \mathbf{e}_{0}$. We have $\overline{\mathbf{CD}}; \Gamma \vdash \mathbf{e}_{0} \in \mathbf{D}$. From the induction hypothesis we have $[\![\overline{\mathbf{CD}}]\!]_{\varphi}^{\overline{\mathbf{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathbf{e}_{0}]\!]_{\varphi}^{\overline{\mathbf{CD}}} \in \sqcup \varphi(lab(\mathbf{e}_{0}))$. From Rule (20) we have that we can derive $[\![\overline{\mathbf{CD}}]\!]_{\varphi}^{\overline{\mathbf{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash (\sqcup \varphi(\ell))^{\ell} [\![\mathbf{e}_{0}]\!]_{\varphi}^{\overline{\mathbf{CD}}} \in \sqcup \varphi(\ell)$.
- (21) e ≡ e₀.m(ē)^ℓ. There are two cases. First, assume that there is a class D in P such that ∀E ∈ φ(lab(e₀)) : impl(CD, E, m) = D::m. We have

$$\overline{CD}; \Gamma \vdash \mathbf{e}_0 \in \mathbf{C}_0 \tag{B.1}$$

$$mtype(\overline{CD}, C_0, m) = \overline{D} \to C$$
(B.2)

$$\overline{\mathsf{CD}}; \Gamma \vdash \overline{\mathbf{e}} \in \overline{\mathsf{C}} \tag{B.3}$$

From (B.1), (B.3), and the induction hypothesis, we have

$$\llbracket \overline{\text{CD}} \rrbracket_{\varphi}^{\overline{\text{CD}}}; \llbracket \Gamma \rrbracket_{\varphi} \vdash \llbracket \mathbf{e}_{0} \rrbracket_{\varphi}^{\overline{\text{CD}}} \in \sqcup \varphi(lab(\mathbf{e}_{0}))$$
(B.4)

$$\llbracket \overline{\mathsf{CD}} \rrbracket_{\varphi}^{\mathsf{CD}}; \llbracket \Gamma \rrbracket_{\varphi} \vdash \llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\mathsf{CD}} \in \sqcup \varphi(lab(\overline{\mathbf{e}}))$$
(B.5)

We have $\overline{CD} \vdash \sqcup \varphi(lab(e_0)) \leq D$, so from Lemma 15 (below), we have

$$\llbracket \overline{\text{CD}} \rrbracket_{\varphi}^{\overline{\text{CD}}} \vdash \sqcup \varphi(lab(\mathbf{e}_0)) <: \mathbf{D},$$
(B.6)

and together with (B.2), we have $mtype(\overline{CD}, D, m) = \overline{D} \to C$. Suppose also $mbody(\overline{CD}, D, m) = (\ell', \overline{x}, e')$. From Lemma 13 (below) we have

$$mtype(\llbracket \overline{CD} \rrbracket_{\varphi}^{\overline{CD}}, D, \mathfrak{m}) = (\sqcup \varphi(lab(\overline{\mathfrak{x}}))) \to (\sqcup \varphi(\ell')).$$
(B.7)

From Rule (40) we have $\varphi(lab(\mathbf{e}_0)) \neq \emptyset$, so suppose $\mathbf{E}_0 \in \varphi(lab(\mathbf{e}_0))$. Suppose also $mbody(\overline{CD}, \mathbf{E}_0, \mathbf{m}) = (\ell'', \overline{\mathbf{x}''}, \mathbf{e}'')$, From Rules (30)–(31) we have

$$\begin{aligned} \varphi(lab(\overline{\mathbf{e}})) &\subseteq \varphi(lab(\overline{\mathbf{x}''})) \\ \varphi(\ell'') &= \varphi(\ell). \end{aligned}$$

Finally, from Rules (38)–(39) we have

$$\begin{aligned} \varphi(lab(\overline{\mathbf{x}})) &\subseteq \varphi(lab(\overline{\mathbf{x}''})) \\ \varphi(\ell') &= \varphi(\ell''), \end{aligned}$$
$$\varphi(lab(\overline{\mathbf{e}})) \subseteq \varphi(lab(\overline{\mathbf{x}})) \tag{B.8}$$

$$\varphi(\ell') = \varphi(\ell). \tag{B.9}$$

Thus, from Rule (22), and from (B.4)–(B.9), we have that we can derive

$$[\![\overline{\mathbf{CD}}]\!]_{\varphi}^{\overline{\mathbf{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathbf{e_0}]\!]_{\varphi}^{\overline{\mathbf{CD}}}.\mathtt{D}::\mathtt{m}([\![\overline{\mathbf{e}}]\!]_{\varphi}^{\overline{\mathbf{CD}}})^{\ell} \in \sqcup \varphi(\ell)$$

as desired.

Second, suppose we have the "otherwise" case from the definition of the transformation of a method call. The proof of this case is similar to the first case, and we omit the details.

• (22) $\mathbf{e} \equiv \mathbf{e}_0 \cdot \mathbf{D} :: \mathbf{m}(\overline{\mathbf{e}})^{\ell}$. The proof of this case is similar to the previous case, and we omit the details.

Lemma 12 Suppose $\varphi \in acceptable(P) \cap type\text{-respecting}(P)$, and $P = (\overline{CD}, e_0)$. If can-declare($\overline{CD}, D, m, \overline{C} \to C_0$), $\overline{CD} \vdash C <: D$, and $mbody(\overline{CD}, C, m) = (\ell, \overline{x}, e)$, then

$$can-declare(\llbracket \overline{\mathtt{CD}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}, \mathtt{D}, \mathtt{m}, (\sqcup \varphi(lab(\overline{\mathtt{x}}))) \to (\sqcup \varphi(\ell))).$$

Proof. Immediate from Lemma 13 (below), using Rule (9), (14)–(16).

Lemma 13 Suppose $\varphi \in acceptable(P) \cap type\text{-respecting}(P)$, and $P = (\overline{CD}, e_0)$. If $mtype(\overline{CD}, D, m) = \overline{D} \rightarrow D_0$, $\overline{CD} \vdash C <: D$, and $mbody(\overline{CD}, C, m) = (\ell, \overline{x}, e)$, then

$$mtype(\llbracket\overline{\mathtt{CD}}\rrbracket_{\varphi}^{\overline{\mathtt{CD}}},\mathtt{D},\mathtt{m}) = (\sqcup\varphi(lab(\overline{\mathtt{x}}))) \to (\sqcup\varphi(\ell)).$$

Proof. Straightforward, using the rules Rules (3)-(6), (14)-(16), (38)-(39).

Lemma 14 Suppose $\varphi \in acceptable(P) \cap type\text{-respecting}(P)$, and $P = (\overline{CD}, e_0)$. If fields $(\overline{CD}, D) = \overline{D} \ \overline{g}$, then fields $([\overline{CD}]_{\varphi}^{\overline{CD}}, D) = \sqcup \varphi(lab(\overline{g})) \ \overline{g}$.

Proof. Straightforward, by induction on the structure of the derivation of the judgment $fields(\overline{CD}, D) = \overline{D} \overline{g}$, using Rules (1)–(2).

Lemma 15 If $\overline{CD} \vdash C \leq :D$, then $\llbracket \overline{CD} \rrbracket_{\varphi}^{\overline{CD}} \vdash C \leq :D$.

Proof. Immediate from the definition of subtyping, that is, Rules (14)-(16).

Lemma 16 Suppose $S, T \in flow(\mathbb{P}) \setminus \emptyset$. If $S \subseteq T$, then $\overline{\mathbb{CD}} \vdash \sqcup S \prec : \sqcup T$.

Proof. Immediate from the observation that the subtyping order forms a tree and therefore admits least upper bounds of nonempty sets.

 \mathbf{SO}

C Proof of Theorem 5

 (\Rightarrow) Let $P_1 = (\overline{CD}, X\langle e_1 \rangle)$ and $P_2 = (\overline{CD}, X\langle e_2 \rangle)$ where e_1 and e_2 are given by one of the rules in Figure 2. Clearly $[\![P_i]\!]_{\varphi} = ([\![\overline{CD}]\!]_{\varphi}^{\overline{CD}}, [\![X]\!]_{\varphi}^{\overline{CD}} \langle [\![e_i]\!]_{\varphi}^{\overline{CD}} \rangle)$, so it remains to show that $[\![e_1]\!]_{\varphi}^{\overline{CD}} \mapsto [\![e_2]\!]_{\varphi}^{\overline{CD}}$. The interesting cases are when e_1 is a cast and when e_1 is a dynamic method call that is transformed to a static method call.

• Case 1, $e_1 = (D)^{\ell_1} new^{\ell_2} C(\overline{e})$: In this case

$$\mathbf{e}_2 = \mathbf{n}\mathbf{e}\mathbf{w}^{\ell_2} \ \mathsf{C}(\overline{\mathbf{e}}) \tag{C.1}$$

$$\llbracket \mathbf{e}_1 \rrbracket_{\varphi}^{\mathsf{CD}} = (\sqcup \varphi(\ell_1))^{\ell_1} \mathsf{new}^{\ell_2} \ \mathsf{C}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\mathsf{CD}})$$
(C.2)

$$\overline{CD} \vdash C <: D \tag{C.3}$$

By Rule 28, $C \in \varphi(\ell_2)$; by Rule 29, $\varphi(\ell_2) \cap subclasses(P_1, D) \subseteq \varphi(\ell_1)$. Thus $C \in \varphi(\ell_1)$, so $\overline{CD} \vdash C <: \sqcup \varphi(\ell_1)$. By the reduction rules,

$$\llbracket \mathbf{e}_1 \rrbracket_{\varphi}^{\overline{\mathbf{CD}}} \mapsto \mathbf{n} \mathbf{e} \mathbf{w}^{\ell_2} \ \mathsf{D}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{\mathbf{CD}}}) = \llbracket \mathbf{e}_2 \rrbracket_{\varphi}^{\overline{\mathbf{CD}}}$$
(C.4)

• Case 2, $e_1 = \text{new}^{\ell_1} C(\overline{e}) . m(\overline{d})^{\ell_2}$ and $\llbracket e_1 \rrbracket_{\varphi}^{\overline{CD}} = \text{new}^{\ell_1} C(\llbracket \overline{e} \rrbracket_{\varphi}^{\overline{CD}}) . D : :m(\llbracket \overline{d} \rrbracket_{\varphi}^{\overline{CD}})^{\ell_2}$: In this case

$$\forall \mathbf{E} \in \varphi(\ell_1) : impl(\overline{\mathbf{CD}}, \mathbf{E}, \mathbf{m}) = \mathbf{D} : :\mathbf{m}$$
(C.5)

$$\mathbf{e}_2 = \mathbf{e}_0\{\texttt{this}, \overline{\mathbf{x}} := \texttt{new}^{\ell_1} \ \texttt{C}(\overline{\mathbf{e}}), \overline{\mathsf{d}}\}$$
(C.6)

where

$$mbody(\overline{CD}, C, m) = (\ell, \overline{x}, e_0)$$
 (C.7)

By Rule 28, $C \in \varphi(\ell_1)$. By (C.5), $impl(\overline{CD}, C, m) = D::m$. By inspecting Rules 5–8 and (C.7)

$$mbody(\overline{CD}, D, m) = (\ell, \overline{x}, e_0)$$
 (C.8)

Thus:

$$\begin{split} \llbracket \mathbf{e}_1 \rrbracket_{\varphi}^{\overline{\texttt{CD}}} &= \texttt{new}^{\ell_1} \ \texttt{C}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{\texttt{CD}}}) . \texttt{D} :: \texttt{m}(\llbracket \overline{\mathbf{d}} \rrbracket_{\varphi}^{\overline{\texttt{CD}}})^{\ell_2} \\ &\mapsto \llbracket \mathbf{e}_0 \rrbracket_{\varphi}^{\overline{\texttt{CD}}} \{\texttt{this}, \overline{\mathbf{x}} := \texttt{new}^{\ell_1} \ \texttt{C}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{\texttt{CD}}}), \llbracket \overline{\mathbf{d}} \rrbracket_{\varphi}^{\overline{\texttt{CD}}} \} \\ &= \llbracket \mathbf{e}_2 \rrbracket_{\varphi}^{\overline{\texttt{CD}}} \end{split}$$

(\Leftarrow) If $\llbracket P_1 \rrbracket_{\varphi}$ takes any step then it is easy to see that P_1 has the form $(\overline{CD}, X \langle e_1 \rangle)$ and that $\llbracket P_1 \rrbracket_{\varphi} \mapsto (\llbracket \overline{CD} \rrbracket_{\varphi}^{\overline{CD}}, \llbracket X \rrbracket_{\varphi}^{\overline{CD}} \langle e' \rangle)$ for $\llbracket e_1 \rrbracket_{\varphi}^{\overline{CD}}$ and e' as in the rules in Figure 2. It remains to show that $e_1 \mapsto e_2$ and $\llbracket e_2 \rrbracket_{\varphi}^{\overline{CD}} = e'$ for some e_2 . The interesting cases are when e_1 is a cast and when e_1 is a dynamic method call that is transformed to a static method call.

• Case 1, $e_1 = (D)^{\ell_1} new^{\ell_2} C(\overline{e})$: In this case

$$\llbracket \mathbf{e}_1 \rrbracket_{\varphi}^{\overline{\mathrm{CD}}} = (\sqcup \varphi(\ell_1))^{\ell_1} \mathrm{new}^{\ell_2} \ \mathsf{C}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{\mathrm{CD}}})$$
(C.9)

$$\mathbf{e}' = \mathbf{n}\mathbf{e}\mathbf{w}^{\ell_2} \ \mathsf{C}(\left[\!\left[\overline{\mathbf{e}}\right]\!\right]_{\varphi}^{\mathsf{CD}}) \tag{C.10}$$

$$\overline{\mathsf{CD}} \vdash \mathsf{C} <: \sqcup \varphi(\ell_1) \tag{C.11}$$

Since φ is type respecting and Rule 20, $\overline{CD} \vdash \Box \varphi(\ell_1) \lt$: D. By transitivity of subtyping, $\overline{CD} \vdash C <: D$. Then $\mathbf{e}_1 \mapsto \mathbf{e}_2$ and $\llbracket \mathbf{e}_2 \rrbracket_{\varphi}^{\overline{CD}} = \mathbf{e}'$ if \mathbf{e}_2 is $\mathsf{new}^{\ell_2} \ C(\overline{\mathbf{e}})$. • Case 2, $\mathbf{e}_1 = \mathsf{new}^{\ell_1} \ C(\overline{\mathbf{e}}) \cdot \mathfrak{m}(\overline{\mathbf{d}})^{\ell_2}$ and $\llbracket \mathbf{e}_1 \rrbracket_{\varphi}^{\overline{CD}} = \mathsf{new}^{\ell_1} \ C(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{CD}}) \cdot D :: \mathfrak{m}(\llbracket \overline{\mathbf{d}} \rrbracket_{\varphi}^{\overline{CD}})^{\ell_2}$:

In this case

$$\forall \mathbf{E} \in \varphi(\ell_1) : impl(\overline{\mathtt{CD}}, \mathbf{E}, \mathbf{m}) = \mathtt{D} :: \mathtt{m}$$

$$\mathbf{e}' = \llbracket \mathbf{e}_0 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} \{\mathtt{this}, \overline{\mathbf{x}} := \mathtt{new}^{\ell_1} \ \mathtt{C}(\llbracket \overline{\mathbf{e}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}), \llbracket \overline{\mathtt{d}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} \} \mathbf{13}$$

where

$$mbody(\overline{CD}, D, m) = (\ell, \overline{x}, e_0)$$
 (C.14)

By Rule 28, $C \in \varphi(\ell_1)$. By (C.12), $impl(\overline{CD}, C, m) = D::m$. By inspection of Rules 5-8 and (C.14)

$$mbody(\overline{CD}, C, m) = (\ell, \overline{x}, e_0)$$
 (C.15)

Then $\mathbf{e}_1 \mapsto \mathbf{e}_2$ and $\llbracket \mathbf{e}_2 \rrbracket_{\varphi}^{\overline{\mathsf{CD}}} = \mathbf{e}'$ if \mathbf{e}_2 is

$$\mathbf{e}_0\{\texttt{this}, \overline{\mathbf{x}} := \texttt{new}^{\ell_1} \ \mathsf{C}(\overline{\mathbf{e}}), \overline{\mathbf{d}}\}$$
(C.16)

References

- [1] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
- [2] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996.
- [3] M. A. Ellis, B. Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990.
- [4] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996.
- [5] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, P. Lee, The TIL/ML compiler: Performance and safety through types, in: ACM SIGPLAN Workshop on Compiler Support for System Software, Tucson, AZ, USA, 1996.
- [6] G. Morrisett, D. Walker, K. Crary, N. Glew, From System F to typed assembly language, ACM Transations on Programming Languages and Systems 21 (3) (1999) 528–569.

- [7] A. Wright, S. Jagannathan, C. Ungureanu, A. Hertzmann, Compiling Java to a typed lambda-calculus: A preliminary report, in: ACM Workshop on Types in Compilation, Kyoto, Japan, 1998, pp. 9–27.
- [8] N. Glew, An efficient class and object encoding, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 311–324.
- [9] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee, TIL: A typedirected optimizing compiler for ML, in: 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, USA, 1996, pp. 181–192.
- [10] F. Tip, A. Kiezun, D. Bäumer, Refactoring for generalization using type constraints, manuscript (2003).
- [11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, T. Nakatani, A study of devirtualization techniques for a Java just-in-time compiler, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 294–310.
- [12] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: W. Olthoff (Ed.), Ninth European Conference on Object-Oriented Programming (ECOOP), Vol. 952 of Lecture Notes in Computer Science, Springer-Verlag, Aarhus, Denmark, 1995, pp. 77– 101.
- [13] D. Detlefs, O. Agesen, Inlining of virtual methods, in: Thirteenth European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag (LNCS 1628), 1999, pp. 258–278.
- [14] E. M. Gagnon, L. J. Hendren, G. Marceau, Efficient inference of static types for Java bytecode, in: Seventh International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS* 1824), 2000, pp. 199–219.
- [15] J. Palsberg, M. I. Schwartzbach, Object-Oriented Type Systems, John Wiley & Sons, 1994.
- [16] T. Knoblock, J. Rehof, Type elaboration and subtype completion for Java Bytecode, ACM Transations on Programming Languages and Systems 23 (2) (2001) 243–272.
- [17] J. Palsberg, P. M. O'Keefe, A type system equivalent to flow analysis, ACM Transations on Programming Languages and Systems 17 (4) (1995) 576–599, preliminary version in Proceedings of POPL'95.
- [18] N. Heintze, Control-flow analysis and type systems, in: Second International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS* 983), Glasgow, Scotland, 1995, pp. 189–206.
- [19] J. Palsberg, Equality-based flow analysis versus recursive types, ACM Transations on Programming Languages and Systems 20 (6) (1998) 1251–1264.

- [20] J. Palsberg, C. Pavlopoulou, From polyvariant flow information to intersection and union types, Journal of Functional Programming 11 (3) (2001) 263–317, preliminary version in Proceedings of POPL'98.
- [21] L. O. Andersen, Self-applicable C program specialization, in: 1992 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), 1992, pp. 54–61, (Technical Report YALEU/DCS/RR-909, Yale University).
- [22] J. Palsberg, Closure analysis in constraint form, ACM Transations on Programming Languages and Systems 17 (1) (1995) 47–62, preliminary version in Proceedings of CAAP'94.
- [23] M. Fähndrich, A. Aiken, Program analysis using mixed term and set constraints, in: Fourth International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS*), 1997, pp. 114–126.
- [24] N. Glew, J. Palsberg, Method inlining, dynamic class loading, and type soundness, submitted for publication (2003).
- [25] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transations on Programming Languages and Systems 23 (3) (2001) 396–450, first appeared in OOPSLA 1999.
- [26] F. Nielson, The typed lambda-calculus with first-class processes, in: Proceedings of PARLE'89, 1989, pp. 357–373.
- [27] A. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.
- [28] S. Jagannathan, A. Wright, S. Weeks, Type-directed flow analysis for typed intermediate languages, in: Fourth International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS*), 1997, pp. 232–249.
- [29] D. F. Bacon, P. F. Sweeney, Fast static analysis of C++ virtual function calls, in: Eleventh Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), San Jose, CA, 1996, pp. 324–341, SIGPLAN Notices 31(10).
- [30] D. F. Bacon, Fast and effective optimization of statically typed object-oriented languages, Ph.D. thesis, Computer Science Division, University of California, Berkeley, report No. UCB/CSD-98-1017 (Dec. 1997).
- [31] F. Tip, J. Palsberg, Scalable propagation-based call graph construction algorithms, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 281–293.
- [32] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, C. Godin, Practical virtual method call resolution for Java, in: Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00), Minneapolis, Minnesota), 2000, pp. 264–280.

benchmark	prog. size	SBA		new alg. ("all" sites		ll"sites)
parameter	(lines)	work units	time (sec)	build	close	querying
10	42	1594	0.028	0.008	0.001	0.000
20	82	9964	0.100	0.016	0.001	0.001
40	162	71104	0.422	0.029	0.002	0.003
80	322	538984	3.494	0.082	0.003	0.012
160	642	4201144	27.115	0.184	0.006	0.062

Table 1:

			linear-time algorithm				
benchmark	prog. size	SBA	total	bu	ild	clo	ose
	(lines)	(time)	time	time	nodes	time	nodes
life	150	0.201	0.083	0.069	1429	0.013	564
lexgen	1180	1.090	0.368	0.217	3624	0.150	2651

Table 2:

Our algorithm could potentially be combined with the standard cubic-time CFA algorithm to obtain a hybrid algorithm that terminates for arbitrary programs but is linear for bounded-type programs. Another area of future work involves extending the analysis to make use of evaluationorder information.

References

- A. Aho, J. Hopcroft, and J. Ullmann, "Time and tape complexity of pushdown automaton languages", *Information and Control*, vol. 13, no. 3, pp. 186-206, 1968.
- [2] A. Bondorf and J. Jorgensen, "Efficient analysis for realistic off-line partial evaluation", *Journal of Functional Programming*, Vol. 3, No. 3, July 1993.
- [3] S. Debray and T. Proebsting, "Interprocedural Control Flow Analysis of First Order Programs with Tail Call Optimization", draft, May 1996. (http://www.cs.arizona.edu/people/ debray/papers/cfa.ps)
- [4] N. Heintze, "Set-Based Analysis of ML Programs", ACM Conference on Lisp and Functional Programming, pp 306-317, 1994.
- [5] N. Heintze, "Control-Flow Analysis and Type Systems", *Static Analysis Symposium*, 1995, pp 189-206.
- [6] N. Heintze and D. McAllester, "On the Cubic-Bottleneck of Subtyping and Flow Analysis" *IEEE Symposium on Logic in Computer Science*, 1997, to appear.
- [7] F. Henglein, "Type Inference with Polymorphic Recursion", *Transactions on Program*-

ming Languages and Systems, Vol. 15, No. 2, pp. 253-289, 1993.

- [8] N. Jones, "Flow Analysis of Lambda Expressions", Symp. on Functional Languages and Computer Architecture, pp. 66-74, 1981.
- [9] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in Abstract Interpretation of Declarative Languages, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.
- [10] D. McAllester, "Inferring Recursive Data Types", draft manuscript, July 1996.
- [11] C. Mossin, "Control Flow Analysis for Higher-Order Typed Programs", draft Ph.D. thesis, DIKU, University of Copenhagen, December, 1996.
- [12] E. Melski and T. Reps, "Interconvertibility of Set Constraints and Context-Free Language Reachability", PEPM'97, to appear.
- [13] R. Milner, M. Tofte and R. Harper, "The Definition of Standard ML", MIT Press, 1990.
- [14] R. Neal, "The computational complexity of taxonomic inference", unpublished manuscript, 18 pages, 1989, (ftp://ftp.cs.utoronto.ca/pub/radford/ taxc.ps.Z).
- [15] J. Palsberg and P. O'Keefe, "A Type System Equivalent to Flow Analysis", POPL-95, pp. 367-378, 1995.
- [16] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference" *Information and Computation*, Vol. 118, No. 1, pp. 128-141, April 1995.
- [17] O. Shivers, "Control Flow Analysis in Scheme", Proc. 1988 ACM Conf. on Programming Language Design and Implementation, Atlanta, pp. 164-174, June 1988.

We compare the performance of the lineartime algorithm with an implementation of setbased analysis (SBA) [4], run in monovariant mode (a generalization of the standard CFA algorithm). All results are for a 150 MHz MIPS R4400 processor with 512 MBytes; all timings are user time in seconds and represent the fastest of 10 runs of the benchmark. Each benchmark consists of analyzing the example program and writing out the control flow information for all nontrivial applications (i.e. applications of the form $(e_1 e_2)$ where e_1 is not a function identifier or an abstraction).

The first set of results are for a parameterized benchmark that illustrates the cubic behavior of the standard CFA algorithm. The benchmark of size 1 consists of:

```
fun fs x = x
fun bs x = x
fun f1 x = x
fun b1 x = x
val x1 = b1(fs f1)
val y1 = (bs b1) f1
```

and the benchmark of size n consists of the first two lines of the above code and n copies of the last four lines, with f1, b1, x1 and y1 appropriately renamed. The following table presents results for a variety of sizes of this benchmark. The cubictime behavior of SBA is clear (the table not only give runtimes, but also a measure of the units of work involved, since cache effects and optimizations in the implementation of SBA itself mean that timings are somewhat misleading). The last three columns describe the behavior of our new algorithm: the first two of them give the results for the linear-time LC algorithm, and the last one shows the quadratic cost of querying all nontrivial applications (there are O(n) of them and each has cost O(n)). Note that the graph building phase (which consists of a simple linear-time pass over the program text) appears to be slightly non-linear (e.g. $0.029 \times 2 \neq 0.082$). This may be due to cache effects, timing inaccuracies, or to inefficiencies introduced in the implementation of the prototype (e.g. lists are currently used to represent some aspects of the graph's structure).

Next, we give the results from two standard SML benchmarks, the life program, and the lexer generator. These results indicate that our preliminary implementation of the linear-time algorithm is 2.5 - 3 times faster than SBA. Perhaps more significant is the number of nodes generated by the linear-time algorithm. The number of nodes in the "build" phase of the analysis is essentially the same as the number of syntax nodes in the program. The key quantity is the number of nodes added during the "close" phase of the algorithm: this gives a measure of the number of times rules such as CLOSE-DOM and CLOSE-RAN are applied. The results suggest that the number of nodes added in the close phase is typically no more than the number of nodes in the build phase, although more benchmarks are clearly needed.

We remark that the timing results probably overstate the cost of the linear-time algorithm. Additional measurements have shown that the cost of the analysis time for the linear-algorithm is now dominated by the cost of just traversing the intermediate representation: for the lexgen example, this cost accounted for up to 198 ms out of the total 368 ms for the benchmark, and for life it was 65 ms out of 83 ms.

11 Conclusion

We have introduced the notion of a sub-transitive control-flow graph: this is a graph whose transitive closure represents control-flow information. The key advantage of this graph is that we can develop O(n) algorithms (where n measures the size of the graph) for many control-flow consuming applications.

Our main result is a linear-time algorithm for bounded-type programs that builds a subtransitive control-flow graph whose transitive closure gives exactly the results of the standard (cubic-time) CFA algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. However, we argue that the "all calls from all call-sites" view of control-flow analysis is unsuitable for investigating the complexity of analyses. In particular, we show that by directly using the sub-transitive graph (instead of using the quadratic sized "all calls from all call-sites" representation), we can develop linear-time algorithms for many CFAconsuming applications. Examples include effects analysis and k-limited CFA. We leave the question of the generality of this approach to future work. In particular, are there "natural" CFAconsuming applications that require the entire "all calls from all call-sites" information, or can we adapt our techniques to all CFA-consuming applications?

We note that linear-time algorithms for *other* forms of control-flow analysis have previously been proposed. In effect, these algorithms replace containment by unification in the definition of control-flow information, and as a result compute information that is strictly less accurate that standard CFA. Our paper shows that this loss of information is not necessary to obtain linear-time algorithms.

For a let-bound function, we can first analyze the function using the above technique to obtain a simplified graph fragment representing the analysis of the function; then, we make copies of this graph fragment for each place the function is used. In practice, we rarely want to blindly duplicate graph fragments for all functions in this way (in general, this duplication could lead to an exponential control-flow analysis). Rather, we focus on functions where polyvariance pays off. For example, we could look at the types of the program and determine which functions are polymorphic. Alternatively, we could first perform a simple monovariant analysis, and then use that information to control a subsequent polyvariant analysis (see e.g. [4]).

Note that we could force our polyvariant algorithm to be linear-time by restricting polyvariance so that there is some global bound on the number of times each graph fragment is effectively duplicated (if one graph fragment is duplicated inside another graph fragment, then any duplication of the enclosing graph fragment must be counted as duplication of the enclosed fragment).

8 Linear-time Effects Analysis

Suppose that we want to use CFA information to drive an effects analysis. We could, for example, run the standard CFA algorithm, build the list of functions that can be called from each call-site, and then iterate over this information to determine which expressions have side-effects. For simplicity, assume that all side-effecting primitives are fully applied and that the language consists only of applications and abstractions. We start by marking applications of side-effecting primitives as side-effecting. Then, we mark an application $(e_1 \ e_2)$ as side-effecting if either e_1 or e_2 are marked side-effecting or if the CFA says that e_1 could be a side-effecting function. Note that this analysis has complexity at least quadratic in the program size, because it uses a representation of control-flow information that is quadratic in program size.

Using our algorithm, we can obtain a procedure that computes exactly the same effects information as the process just described, but runs in linear-time. The basic idea is that we color all applications that involve side-effecting operations with red, and then propagate coloring as follows: (a) a node $(e_1 \ e_2)$ is colored red if either e_1, e_2 or $ran(e_1)$ are red; (b) a node ran(e) is colored red if there is an edge $ran(e) \rightarrow e'$ and e' is red. Rule (a) corresponds to the condition used in the previous paragraph; rule (b) is a (limited) form of transitive closure for coloring. This is clearly a linear-time procedure (it is just a graph reachability problem).

9 Linear-time k-limited CFA

In many applications of CFA, we are only interested in knowing information about call sites where a small number of functions can be called (e.g. one, two or three functions). If more functions than that may be called, then we might not be interested in knowing the exact details of the functions, because the optimization we have in mind may be intractable or inappropriate. Examples of these kinds of applications include inlining and specialization.

We can use our algorithm to obtain a lineartime procedure for computing this information as follows. First, we annotate with each node with a value that is either a small set or the token "many". We start by annotating nodes corresponding to functions with the singleton set containing just that function, and all other nodes with the empty set. Then, we propagate information back along edges: if a node n has edges to nodes with sets S_1, \ldots, S_j , then we update the annotation at n with $S_1 \cup \cdots \cup S_i$ if this is a small set (size $\leq k$), and "many" otherwise. Each update can be done in constant time, each node can be updated at most a constant number of times. and hence if we only propagate changes, we can obtain a linear-time algorithm for computing klimited CFA.

10 Experimental Results

We present some preliminary results from a prototype implementation of the linear-time algorithm. This prototype implements the basic linear-time CFA algorithm, with extensions for datatypes and records; however certain aspects of ML have not yet been implemented, and as a result the benchmarks we have been able to run are limited.

Our implementation is essentially a naive implementation of the algorithm described in this paper. A number of improvements could be made (such as taking advantage of the many nodes that have only one outgoing edge). We expect considerable improvement in the performance of the prototype as we better understand how the algorithm behaves in practice. It is also likely that we can exploit a number of graph implementation techniques. shall only be interested in the case where $\tau(n)$ is a datatype).

We now define two node congruences (by congruence, we mean that if $n_1 \equiv n_2$ then $dom(n_1) \equiv$ $dom(n_2)$, etc.). The first congruence, \equiv_1 , is defined to be the least congruence such that $n_1 \equiv_1 n_2$ whenever $\tau(n_1) = \tau(n_2)$ and both are datatypes. The second congruence differs from the first in that only nodes that have the same base node and involve a de-constructor are considered equivalent. To this end, observe that any node can be written in the form $\alpha(n)$ where n is a basic node and α is a sequence of *dom. ran.* de-constructors, etc. We say that n is the base node of $\alpha(n)$. Now, define \equiv_2 , to be the least congruence such that $n_1 \equiv_2 n_2$ whenever (a) $\tau(n_1) = \tau(n_2)$ and both are datatypes and (b) n_1 and n_2 both have the same base node and involve a de-constructor. This second congruence is finer than the first, and it leads to a strictly more accurate analysis. To illustrate these constructions, suppose that we have the program fragment cons(2, cons(1, nil)). Let *e* denote this expression, and let e' denote the sub-expression cons(1, nil). Using \equiv_2 , we generate the following edges (we include some of the rules that may be generated by the closure rules):

$$\begin{array}{ccc} e & \longrightarrow cons(2, e') \\ e' & \longrightarrow cons(1, nil) \\ car(e) & \longrightarrow 2 \\ cdr(e) & \longrightarrow e' \\ car(e') & \longrightarrow 1 \\ cdr(e') & \longrightarrow nil \\ car(cdr(e)) & \longrightarrow car(e') & (CAR-CLOSE) \\ cdr(e) & \longrightarrow cdr(e') & (CDR-CLOSE) \end{array}$$

noting that in the last line, $cdr(e) \equiv_2 cdr(cdr(e))$. Now, if we use \equiv_1 instead of \equiv_2 , then e, e', cdr(e), cdr(e') would all be in the same equivalence class, and so, for example, there would be edges to both 1 and 2 from car(e).

For bounded type programs, \equiv_1 generates O(n) congruence classes, and this leads to a linear-time analysis algorithm. In contrast, for bounded type programs, \equiv_2 generates up to $O(n^2)$ congruence classes, and hence leads to a quadratic-time analysis algorithm. However, if in addition to bounded types, we assume that nesting levels of datatypes are bounded, then \equiv_2 gives a linear-time algorithm. We define nesting levels of datatypes as follows: label a datatype definition that does not mention other datatypes with 0, and label any other datatype definition with the maximum of the labels of all datatypes it uses, plus 1.

We are currently investigating the tradeoffs

between these two approaches. In particular, how much more accurate is the second approach? Note that the first approach is akin to statically fixing the set of allowed functions that can appear in a particular slot in a data-constructor; the second approach allows more dynamic behavior. We also plan to investigate whether the bounded nesting-level assumption for datatypes is realistic.

7 Polyvariance

So far, we have described a monovariant algorithm. We now describe polyvariant extensions to our algorithm that are analogous to letpolymorphism. Consider a program P. At a very naive level, consider just let-expanding P and analyzing the resulting (probably very large) program. Our intent is to develop an analysis whose end result is equivalent to doing a monomorphic analysis of the let-expanded P, without doing the explicit let-expansion. Instead, we analyze the function once, and build a summary of the analysis of its code body. The resulting parameterized and simplified graph can then be instantiated (copied) at the points of the function where it is mentioned, much like polymorphic type inference in ML.

One of the strengths of our algorithm is that the simplification/parameterization steps can be easily carried out — they correspond to graph reachability and simplification steps. To illustrate the basic issues, let e be the code fragment $\lambda^l x.((\lambda^{l'} y.x) nil)$. If we look at e in isolation, the LC' rules introduce edges:

$$\begin{array}{ccc} ran(\lambda^{l'}y.x) & \longrightarrow x \\ y & \longrightarrow & dom(\lambda^{l'}y.x) \\ dom(\lambda^{l'}y.x) & \longrightarrow & nil \\ ((\lambda^{l}y.x) & nil) & \longrightarrow & ran(\lambda^{l'}y.x) \\ ran(e) & \longrightarrow & ((\lambda^{l}y.x) & nil) \\ x & \longrightarrow & dom(e) \end{array}$$

To simplify/parameterize this graph fragment, we first isolate the critical nodes, which are those nodes that may be used by surrounding program text, in this case $ran(\lambda^{l'}x.((\lambda^l y.x) nil))$ and $dom(\lambda^{l'}x.((\lambda^l y.x) nil))$. Next, we do a graph reachability from these two nodes (and here we must generalize reachable so that if n is reachable, then so is dom(n) and ran(n)). Any non-reachable nodes (such as nil) can now be removed. Finally, we can compress the graph and remove intermediate nodes (such as x). In this case, we are left with just $ran(e) \rightarrow dom(e)$.

bounded number of edges for bounded type size polymorphically typed programs, it therefore suffices to show that the number of distinct positions in the type trees of the monotypes in the let-expanded version of the program. This is immediate, since the sizes of the monotypes in the let-expanded program are bounded by some constant k, and hence there is at most a total of 2^k positions in these types.

We have thus established that our algorithm runs in linear-time on bounded-type (in the sense of McAllester) polymorphic programs. Note that although we have addressed programs with polymorphic types, the algorithm itself is still monovariant (context insensitive). Making the algorithm polyvariant is a separate issue, and is considered in Section 7.

6 CFA for ML

Thus far, we have worked in the context of a simple version of the lambda calculus (with just abstraction and application). We now extend the algorithm to recursion, records and (recursive) datatypes. First consider fix: since we have worked with simply typed lambda terms with only abstraction and application, there is no recursion. To address this, consider adding a construct letrec $f = \lambda^l x.e_1$ in e_2 . It is simple to extend the linear-time algorithm for this construct: for each instance of this construct, we add transitions:

letrec
$$f = \lambda^l x.e_1$$
 in $e_2 \longrightarrow e_2$
 $f \longrightarrow \lambda^l x.e_1$

Next consider records. Suppose we add constructs (e_1, \ldots, e_n) and $proj_j$, j = 1..n, for record creation and accessing. We extend the algorithm by adding $proj_j$ as a node operator (i.e. it has the same status as *dom* and *ran* in the last section, and has its own closure rule, similar to CLOSE-RAN). Then, for each expression (e_1, \ldots, e_n) , we add transitions $proj_j((e_1, \ldots, e_n)) \longrightarrow e_j$, j = 1..n, and each program expression $proj_j(e)$ is treated as a node. If the program has bounded types, then only a bounded number of nodes need be considered, and so the extended algorithm is linear-time. (Note, however, that for programs with records, bounded-order and bounded-arity no longer implies bounded type size.)

Next consider recursive data types. One possibility is to ignore recursive data types: whenever a function is put in a recursive data structure and then extracted, we lose all information about the function (i.e. we obtain the set of all abstraction labels). The rational for this is that functions are rarely put in recursive data structures, and so we can obtain most of the important information about control-flow in a program by ignoring recursive data types.

However, many generalized forms of CFA track data-constructors in the same way as they track functions: the advantage here is that not only do they give better control-flow information, but they also give information about the shape of first-order values. We consider a similar approach. First, we extend the node operators dom and ran with additional operators just as we did for records. Specifically, we add one operator ("de-constructor") for each argument of each data-constructor": for an n-ary constructor c, we would add $c_{(1)}^{-1}, \ldots, c_{(n)}^{-1}$. Then we add appropriate (demand-driven) closures rules for deconstructors. Finally, for each expression e of form $c(e_1, \ldots, e_n)$, we add transitions $c_{(j)}^{-1}(e) \longrightarrow e_j, j = 1..n$.

Unfortunately, as formulated, we have no way of bounding the size of the nodes we must consider. In fact, the monadic monotone closure problem⁸ can be mapped into this analysis problem, and Neal [14] has shown that monadic monotone closure is essentially as hard as the 2NPDA acceptability problem, a well-studied problem for which the best known algorithm is $O(n^3)$ and has not been improved since Aho, Hopcroft and Ullmann's early work [1] in 1968. Melski and Reps [12] have recently obtained a similar result in their work on set-based analysis and context-free reachability.

To reduce this complexity, we consider two alternatives, both of which reduce the accuracy of the analysis so that it is less accurate than, for example, mono-variant set-based analysis. The basic idea is to use a finite node congruence that bounds the number of nodes that are considered (the algorithm considers only one node from each congruence class) at the expense of reducing analysis accuracy. First, note that each node can be associated with a type. In particular, each node of the form $c_{(i)}^{-1}(n)$ can be associated with the type of the i^{th} argument of the constructor c. Let $\tau(n)$ denote the type thus associated with n (we

⁷For simplicity, we view an ML datatype declaration as a definition of a collection of of multi-arity dataconstructors.

⁸This is a generalization of transitive closure that includes two (or more) monotone node functions f and g such that if n is a node, then f(n) and g(n) are nodes, and if $n \to n'$ then $f(n) \to f(n')$ and $g(n) \to g(n')$. Given a set of edges between nodes, and two nodes n and n' appearing in this set, does $n \to n'$ follow from the standard transitive closure rule and the additional f and g rules?

Algorithm 1

Input: A program P, label l and a program sub-expression e. Output: Is $l \in L(e)$?

1. Apply LC' to P.

2. Use graph reachability to determine whether l is reachable from e.

Algorithm 2

- Input: A program P and a program sub-expression e.
- Output: L(e)
- 1. Apply LC' to P.
- 2. Use graph reachability to find all nodes reachable from e.
- 3. Output the labels of abstractions in these reachable nodes.

We can also obtain an $O(n^2)$ algorithm for computing all label sets (i.e. complete CFA information) by repeatedly applying Algorithm 2 to all program sub-expressions.

The key part of our new algorithm is the use of the type tree at each program node to limit the number of edges that must be added during the analysis. Proposition 4 bounds the number of nodes using the maximum size type trees that can appear at any program node. This provides a rather loose bound. We could obtain tighter bounds by observing that the work done by the algorithm at each node is proportional to the type tree at that node. Hence the total work done by the algorithm is proportional to the sum of the type tree sizes at each node. In other words, it is proportional to $k_{ave} \cdot |P|$, where k_{ave} is the average size of the type trees at each node in the program. One of the principal concerns of our implementation was the size of this constant: would it be prohibitive? Early results indicate that this is not the case: the constant is quite small, typically around 2 or 3.

Note that the algorithm itself does not actually look at the types during its execution. Rather, the types are used only to establish termination (and the linear-time complexity bounds in the bounded-type case). In other words, our algorithm only needs to know that the (appropriate) types exist – it does not need to know what they are. This simplifies implementation — we do not need to transmit the types from type inference to our algorithm.

5 Polymorphic Types

Thus far we have considered monomorphic programs and we have assumed a bound k on the size of the monotypes in a program. Now consider polymorphically typed programs (in the sense of ML) and suppose our expression language is extended with an appropriate let construct. There are at least two notions of bounded size polymorphically typed programs. McAllester [10] defines that a polymorphically typed program P has bounded type size if there is some constant k such that the tree-size of the monotypes of each expression in the let-expansion of P all have size $\leq k$. Alternatively, motivated by Henglein's "ML programs with small types" [7], we can define that a polymorphically typed program P has bounded type size if there is some constant k such that the types (including polytypes) of expressions in Pall have tree-size bounded by < k.

Unfortunately, the two definitions are not equivalent⁶. We shall use McAllester's definition. Suppose that we have a polymorphically typed programs (according to McAllester's definition). For monotyped program, we bound the running of the algorithm by using the monotypes of expressions to provide a template for the nodes that need to be considered during the edgeadding phase. The situation is similar for polymorphically typed programs, except that we use the induced collection of monotypes in the letexpansion of a program to provide a collection of templates for bounding the behavior of our algorithm. Note, again, that our algorithm does not actually need to have the types (and in particular, our algorithm does not need to construct the let-expansion of the program!); we just use their existence to prove termination. For example, in the program

fun id x = xval y = ((id id) id) 1

the induced monotypes for id are $int \rightarrow int$, $(int \rightarrow int) \rightarrow (int \rightarrow int)$ and $((int \rightarrow int) \rightarrow (int \rightarrow int)) \rightarrow ((int \rightarrow int) \rightarrow (int \rightarrow int))$. In essence, we can set up a correspondence between each node *n* added during the edge-adding step and a position in some type tree for the based node of *n* (recall that nodes in the edges-adding phase are built from basic nodes by applying $dom(\cdot)$ and $ran(\cdot)$).

To show that our algorithm adds only a

⁶Consider the program consisting of *n* functions where the first function f_0 is just the identity function, and f_{i+1} is defined to be $\lambda x. (f_i \ f_i) x$. This program has bounded type using Henglein's definition, but the monotypes in the let-expansion of the program have exponential tree size.

edge-adding phase (which adds basic edges) and a transitive closure phase. However, note that the CLOSE-DOM and CLOSE-RAN rules are open ended: given any edge $n \to n'$ (added by one of the other rules), the CLOSE-RAN rule says that we can add all edges of the form $ran^k(n) \to ran^k(n')$ for all $k \geq 1$, and similarly for the CLOSE-DOM rule.

To control the application of the CLOSE-DOM and CLOSE-RAN rules, we make them demand driven, as follows:

$$\frac{n_1 \longrightarrow n_2 \qquad n \longrightarrow dom(n_2)}{dom(n_2) \longrightarrow dom(n_1)} (\text{CLOSE-DOM}')$$

$$\frac{n_1 \longrightarrow n_2 \qquad n \longrightarrow ran(n_1)}{ran(n_1) \longrightarrow ran(n_2)} \quad (\text{CLOSE-RAN}')$$

This means that CLOSE-DOM' can only applied if there is a transition whose right-hand-side could immediately match with the left-hand-side of the added transition i.e. if it is "needed". Similarly for CLOSE-RAN'. Call this new system LC' (that is, LC' consists of ABS-1, ABS-2, APP-1, APP-2, CLOSE-DOM' and CLOSE-RAN'). LC' is equivalent to LC in the following sense:

Proposition 2 For all programs P, and expressions e and $\lambda^{l} x.e$ appearing in P:

- There exist nodes n_i such that $e \longrightarrow n_1 \longrightarrow$ $\dots \longrightarrow n_k \longrightarrow \lambda^l x \cdot e \text{ in } LC \text{ iff}$
- there exist nodes n'_i such that $e \longrightarrow n'_1 \longrightarrow \cdots \longrightarrow n'_{k'} \longrightarrow \lambda^l x \cdot e$ in LC'.

This modification of LC into LC' improves the termination properties of the system (discussed further in the next section). It also introduces an element of demand-driven/incremental behavior. For example, suppose that we have a function $\lambda^{l} x \cdot x$. The rules introduce edges $ran(\lambda^l x.x) \rightarrow x$ and $x \rightarrow dom(\lambda^l x.x)$. At this stage, these are the only edges involving these nodes. Eventually, if and when the function is used, we may invoke the CLOSE-DOM' and CLOSE-RAN' rules. For example, if the entire program is $((\lambda^{l}x.x \ \lambda^{l'}y.y) \ e)$, then the CLOSE-DOM' and CLOSE-RAN' rules will add $ran(ran(\lambda^l x \cdot x)) \rightarrow$ ran(x) and $ran(x) \rightarrow ran(dom(\lambda^l x.x))$, amongst others. In other words, we only explore the parts of the "type" of an expression that are actually needed.

4 Termination

The LC' system can be viewed as an algorithm: given a program P, add all of the edges specified by the rules (this is best represented as a graph). However, this procedure does not terminate in general. We now show that the algorithm:

- terminates for typed programs (either simply typed or polymorphically typed).
- is fast (linear) for bounded-type programs (either simply typed or polymorphically typed).

In essence, we shall use the types as a template for the nodes that need to be considered. To illustrate this, suppose e is a program expression with (non-polymorphic) type $(\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \rightarrow \tau_4$, then we need to consider six new nodes, one for each proper subexpression of the type: dom(e), ran(e), dom(dom(e)), ran(dom(e)), dom(ran(e))and ran(ran(e)). In general, the number of new nodes that must be added corresponds to the size of the type trees of program nodes. We show how this idea can be applied to programs with polymorphic types in the next section (Section 5); for the moment we shall consider programs with monotypes.

Bounding the number of nodes guarantees termination, because the inference rules ABS-1, ABS-2, APP-1, APP-2 generate a fixed (programdependent) number of rules, and CLOSE-DOM' and CLOSE-RAN' can add at most one new rule for each rule/node pair. In the case of programs with k-bounded types, the size of these type trees is bounded by k, and hence we can obtain a linear bound on the number of rules that must be added. (Note that, in general, the tree-size of a program can be exponential in program size for programs that exhibit this behavior, our proposed algorithm would be a poor choice compared with the standard cubic-time algorithm. For untyped (or recursively typed programs), there is no bound, and our algorithm may not terminate.)

To make the behavior of our algorithm on bounded-type program more precise, fix on some constant k, and define \mathcal{P}_k to be the class of monotyped programs whose whose types are bounded by k. In system LC' we have:

Proposition 3 There exists a constant c_k such that when LC' is applied to a program P in \mathcal{P}_k , LC' constructs at most $c_k \cdot |P|$ edges where |P| denotes the size of program P.

Hence, we obtain the following linear-time algorithms for bounded-type monotyped programs: tially the same steps as the algorithm based on the previous control-flow definition and is also $O(n^3)$ (in fact, the close correspondence between the two algorithms can be used to provide an easy proof of equivalence of the two definitions).

Observe that in this algorithm, the transitive closure computation in intertwined with the addition of new edges. However, this does not need to be the case; this is a key insight of our algorithm. To show this, we define a new transition system. First, define a set of nodes n by the following grammar:

$$n ::= e \mid dom(n) \mid ran(n)$$

Intuitively, dom(n) represents the "domain" of the node n, and ran(n) represents the "range" of the node n. If n corresponds to an abstraction, then dom(n) is simply the argument (bound variable) of the abstraction; otherwise, dom(n) denotes the collection of arguments of the abstractions represented by n. Similarly, if n corresponds to an abstraction, then ran(n) is simply the result (body) of the abstraction; otherwise, dom(n)denotes the collection of results of abstractions represented by n.

Now, define a transition system between nodes n as follows:

$$\overline{x \longrightarrow dom(\lambda^l x.e)} \quad (\text{if } \lambda^l x.e \text{ in } P) \qquad (\text{ABS-1})$$

$$\overline{ran(\lambda^l x.e) \longrightarrow e} \quad (\text{if } \lambda^l x.e \text{ in } P) \qquad (\text{ABS-2})$$

$$\overline{dom(e_1) \longrightarrow e_2} \quad (\text{if } (e_1 \ e_2) \text{ in } P) \quad (\text{APP-1})$$

$$\overline{(e_1 \ e_2) \longrightarrow ran(e_1)} \quad (\text{if } (e_1 \ e_2) \text{ in } P) \quad (APP-2)$$

$$\frac{n_1 \longrightarrow n_2}{dom(n_2) \longrightarrow dom(n_1)}$$
(CLOSE-DOM)

$$\frac{n_1 \longrightarrow n_2}{ran(n_1) \longrightarrow ran(n_2)}$$
(CLOSE-RAN)

Call this system LC (for "linear closure"); it forms the core part of our linear-time CFA algorithm. To illustrate LC, consider the program $(\lambda^{l}x.(x \ x) \ (\lambda^{l'}x'.x'))$ used earlier in this section. Applying the first four rules leads to:

$$x \xrightarrow{ABS-1} dom(\lambda^{l}x(x x)) \quad (1)$$

$$ran(\lambda^{l} x.(x \ x)) \xrightarrow{\text{ABS-2}} (x \ x)$$
(2)
$$x' \xrightarrow{\text{ABS-1}} dom(\lambda^{l'} x'.x')$$
(3)

$$ran(\lambda^{l'} x' x') \xrightarrow{\text{ABS-2}} x' \tag{4}$$

$$lom(\lambda^{l}x.(x \ x)) \xrightarrow{\text{APP-1}} \lambda^{l'}x'.x'$$
(5)

$$(\lambda^{l}x.(x\ x)\ (\lambda^{l'}x'.x')) \xrightarrow{\text{APP-2}} ran(\lambda^{l}x.(x\ x)) \quad (6$$

$$dom(x) \xrightarrow{\text{APP-1}} x \tag{7}$$

$$(x \ x) \xrightarrow{\text{APP-2}} ran(x)$$
 (8)

where the subscripts on the arrows indicate which rule is employed. Applying the last two rules leads to the following transitions (among others).

 $\begin{array}{ccc} dom(\lambda^{l'}x'.x') &\longrightarrow & dom(dom(\lambda^{l}x.(x\ x))) & (9) \\ ran(dom(\lambda^{l}x.(x\ x))) &\longrightarrow & ran(\lambda^{l'}x'.x') & (10) \\ & dom(dom(\lambda^{l}x.(x\ x))) &\longrightarrow & dom(x) & (11) \\ & ran(x) &\longrightarrow & ran(dom(\lambda^{l}x.(x\ x))) & (12) \end{array}$

(9) and (11) are by CLOSE-DOM; (10) and (12) are by CLOSE-RAN. Combining these transitions, we can derive $\lambda^{l'}x'.x'$ from $(\lambda^{l}x.(x\ x)\ (\lambda^{l'}x'.x'))$:

$$\begin{array}{cccc} (\lambda^{l}x.(x\ x)\ (\lambda^{l'}x'.x')) & \longrightarrow ran(\lambda^{l}x.(x\ x)) & \text{by} & (6) \\ & \longrightarrow (x\ x) & \text{by} & (2) \\ & \longrightarrow ran(x) & \text{by} & (8) \\ & \longrightarrow ran(dom(\lambda^{l}x.(x\ x)))) & \text{by} & (12) \\ & \longrightarrow ran(\lambda^{l'}x'.x') & \text{by} & (10) \\ & \longrightarrow x' & \text{by} & (4) \\ & \longrightarrow dom(\lambda^{l'}x'.x') & \text{by} & (3) \\ & \longrightarrow dom(dom(\lambda^{l}x.(x\ x)))) & \text{by} & (9) \\ & \longrightarrow dom(x) & \text{by} & (11) \\ & \longrightarrow x & \text{by} & (7) \\ & \longrightarrow dom(\lambda^{l}x.(x\ x)) & \text{by} & (1) \\ & \longrightarrow \lambda^{l'}x'.x' & \text{by} & (5) \end{array}$$

Compare this to DTC in which the transition $\lambda^{l'}x'.x' \longrightarrow (\lambda^{l}x.(x \ x) \ (\lambda^{l'}x'.x'))$ was added by the deduction rules. In other words, what was a single transition step in DTC has become a multistep transition in LC. This relationship holds in general: the transitive closure of LC corresponds to DTC in the following sense:

Proposition 1 For all programs P, and expressions e and $\lambda^l x.e$ appearing in P, $e \longrightarrow \lambda^l x.e$ in DTC iff for some nodes $n_i, e \longrightarrow n_1 \longrightarrow \cdots \longrightarrow n_k \longrightarrow \lambda^l x.e$ in LC.

What we have achieved, then, is a factorization of the algorithm into two separate parts: an all at once. Since each label set contains up to n labels, this represents $O(n^2)$ information, and takes $O(n^2)$ time just to output. So a linear time algorithm for "compute all label sets!" is out of the question. However, in compiler applications we rarely want to know the control-flow information for every node in the program. Instead we usually only want to know the functions that can be called from a (relatively few) specific call sites. Alternatively, we may want to know whether only one function can be called from a particular site (e.g. for inlining or specialization applications). More generally, we may not be interested in specific control-flow information, but rather we may need to know about control-flow to answer other questions such as "is this expression side-effect free?". We address the complexity of these such questions in Sections 8 and 9; for now, we consider four basic control-flow questions:

- Given a label l and an expression occurrence
 e, is l ∈ L(e)?
- Given an expression occurrence e, compute L(e).
- Given a label l, compute all expression occurrences e such that l ∈ L(e).
- Compute all label sets.

The following table compares the complexity of our algorithm with the standard algorithm on these questions, for bounded-type programs.

Problem	Std Alg.	New Alg.
Is $l \in L(e)$?	$O(n^3)$	O(n)
L(e)	$O(n^3)$	O(n)
$\{e: l \in L(e)\}$	$O(n^3)$	O(n)
All Label Sets	$O(n^3)$	$O(n^2)$

3 The Algorithm

We begin by reformulating the definition of standard control-flow information as a transition system between program nodes. This reformulation makes explicit the connection with transitive closure. For convenience we assume that programs are renamed to ensure that bound variables are distinct. Now, for a program P, construct the following deduction rules:

$$\overline{\lambda^l x \cdot e \longrightarrow \lambda^l x \cdot e} \tag{ABS}$$

$$\frac{e_1 \longrightarrow \lambda^l x \cdot e}{x \longrightarrow e_2} \quad (\text{if } (e_1 \ e_2) \text{ in } P) \qquad (\text{APP-1})$$

$$\frac{e_1 \longrightarrow \lambda^l x.e}{(e_1 \ e_2) \longrightarrow e} \quad (\text{if } (e_1 \ e_2) \text{ in } P) \qquad (\text{APP-2})$$

$$\frac{e_1 \longrightarrow e_2}{e_1 \longrightarrow e_3} \qquad (\text{TRANS})$$

where the condition on the second and third rules "if $(e_1 \ e_2)$ in P" indicates that there is instance of the respective rule for each occurrence of a term of the form $(e_1 \ e_2)$ in P. Intuitively, an edge $e_1 \longrightarrow e_2$ indicates that anything (e.g. an abstraction) we can derive from e_2 is also derivable from e_1 . In the case where e_2 is itself an abstraction, this edge says that e_2 is one possible "value" for e_1 . The first rule is a boot-strapping rule that says that any abstraction leads to itself. The second and third rules deal with application and respectively say that if there is a transition from the operator of an application to an abstraction, then add a transition from the bound variable of the abstraction to the operand of the application (rule APP-1) and also add a transition from the entire application to the body of the abstraction (rule APP-2). The final rule is transitivity (we note that it is sufficient to restrict this rule to the case where e_3 is an abstraction). Standard control-flow analysis can now be redefined as follows: given a program expression e, find all abstractions $\lambda^l x.e$ such that $e \to \lambda^l x.e$ is derivable from the above rules.

For example, consider $(\lambda^{l}x.(x \ x) \ (\lambda^{l'}x'.x'))$. The above rules lead to:

$$\lambda^{l} x \cdot (x \ x) \longrightarrow \lambda^{l} x \cdot (x \ x) \quad (ABS) \tag{1}$$
$$\lambda^{l'} x' \cdot x' \longrightarrow \lambda^{l'} x' \cdot x' \quad (ABS) \tag{2}$$

$$\begin{array}{cccc} r & \longrightarrow & \lambda & r & \vdots & r & \text{(ABS)} & (2) \\ r & \longrightarrow & \lambda^{l'} r' r' & (\text{APP-1}) & (3) \end{array}$$

$$(\lambda^{l} x.(x \ x) \ \lambda^{l'} x'.x') \longrightarrow (x,x) \qquad (APP-2) \qquad (4)$$

$$x' \longrightarrow \lambda^{i} x' \cdot x' \quad (\text{APP-1}) \quad (5)$$

$$\begin{array}{ccc} (x \ x) \longrightarrow x & \text{(APP-2)} & (6) \\ (\lambda^l x . (x \ x)) \ \lambda^{l'} x' . x') \longrightarrow \lambda^{l'} x' . x' & \text{(TRANS: 4, 5, 6)} \end{array}$$

The last rule follows from two applications of transitivity on (4), (6) and (5) (in that order). In effect, the four deduction rules ABS, APP-1, APP-2 and TRANS define a dynamic transitive closure problem: ABS sets up some initial edges, TRANS is transitive closure, and APP-1 and APP-2 add new basic edges as the transitive closure proceeds. We refer to this system as DTC because of its close connection to dynamic transitive closure. Clearly we can use DTC as the basis of an iterative fixed-point algorithm for control-flow analysis: we start with the empty set of transitions and add a transition $e \longrightarrow e'$ if it follows from one of the above rules in the context of the set of transitions obtained thus far. This algorithm performs essentiations and the performance of the set of transitions obtained thus far.

most important, the overhead of the new algorithm (i.e. the "size of the constant") appears to be small.

In general terms, what we establish in this paper is a connection between control-flow analysis and graph reachability. Recent work by Melski and Reps [12] has show a similar kind of connection between (a limited class of) set constraints and context-free reachability. While the differences are significant (our work deals with reachability in a standard graph, whereas Melski/Reps deals with S-path reachability in a *labeled* graph), the use of graphical constructs in analysis is becoming increasingly common. We also note that very recent (and independent) work by Mossin [11] investigates ideas similar to those in this paper (in particular, the use of types to bound control-flow graph construction).

2 Control-Flow Analysis

We define standard control-flow analysis on a variant of the λ -calculus with labeled abstractions. Expressions *e* are defined by⁴:

$$e ::= x \mid \lambda^l x \cdot e \mid (e_1 \cdot e_2)$$

where x is a variable and l is a label. Such labels allow us to trace a program's control flow. A program is a closed term in which each abstraction has a unique label. Program evaluation is β -reduction, appropriately adapted to labeled expressions. Note that reduction preserves labels on abstractions: e.g. a single step of β -reduction rewrites $(\lambda^l x.(x \ x)) \ (\lambda^{l'} y.y))$ into $(\lambda^{l'} y.y) \ (\lambda^{l'} y.y)$. There are no restrictions on the reduction order: the β -reduction can be applied at any subterm at any time.

The purpose of control-flow analysis is to associate a set of labels L(e) with each sub-expression e of the program⁵ such that if e reduces to an abstraction labeled l during execution of the program, then $l \in L(e)$. In other words, control-flow analysis gives a conservative approximation of the abstractions that can be encountered at each expression during execution of a program. ysis as the least such association of label sets that satisfies the following two conditions:

- for any abstraction $\lambda^{l} x.e$, we have $l \in L(\lambda^{l} x.e)$, and
- for any application $(e_1 \ e_2)$, if $l \in L(e_1)$ and l labels the abstraction $\lambda^l x.e$, then
 - L(x) ⊇ L(e₂) for each occurrence of x bound by the abstraction l, and
 L((e₁ e₂)) ⊃ L(e).

The standard algorithm for computing this analysis is essentially a least fixed-point computation: each label set is initially the empty set, and then the algorithm repeatedly picks a sub-expression at which one of the conditions is not satisfied, and minimally adds new labels to label sets to locally satisfy the condition. This process is guaranteed to terminate because there are only a finite number of labels in a program, and the label sets are monotonically increasing. The complexity of this algorithm is $O(n^3)$ where n is the size (number of syntax nodes) of the input program: informally, at most n labels may be added to the n label sets, and each of these n^2 possible additions may involve up to O(n) work.

To give some intuition about how this kind of behavior can arise in practice, consider the following program fragment:

fun f x = (f x1) (f x2)

Using the above algorithm, the label set collected for x is the union of the label sets collected for x1 and x2. Since the number of calls to function f can linearly increase with program size, the information collected for x can grow linearly - in effect. x acts like a "join" point, combining information from diverse parts of a program. If x is applied in the body of f, then we must perform work proportional to the information collected for \mathbf{x} at this application site. Worse, if \mathbf{x} is returned then all of the information joined by \mathbf{x} can flow back to the call sites of the function f. This joinpoint behavior is independent of type size and is observed in practice, particularly for extensions of standard CFA that deal with data constructors (for which library functions such as append and map become common join-points). Although it rarely leads to cubic behavior, it can have a significant impact on analysis running time and space.

Note that the standard CFA algorithm computes the label sets for each program expression

We can now define standard control-flow anal-

⁴When discussing control-flow of ML typable programs, we shall assume the addition of a fix construct and a let construct. However, we describe the core of our algorithm using just this simple version of the λ -calculus.

⁵Strictly speaking, the association is with each occurrence of a sub-expression: different occurrences can have different label sets.

The usual algorithm for standard CFA has $O(n^3)$ time complexity and $O(n^2)$ space complexity, where *n* measures the length of the input program. The conventional wisdom is that this algorithm cannot be improved because the algorithm is essentially performing a dynamic transitive closure and the best known algorithm for dynamic transitive closure differs from the usual transitive closure problem because new basic edges may be added during the algorithm's execution; as a result the usual techniques for obtaining sub-cubic algorithms for transitive closure cannot be applied in the dynamic case.

The apparent cubic complexity of standard CFA has been a barrier to the use of CFA in compilers. In fact, although a number of prototypes analyzers have been built, standard CFA has yet to find its way into a widely used compiler. Many implementors have instead used simpler but less accurate algorithms. For example, Bondorf and Jorgensen [2] employ an equality-based algorithm for CFA because "the equality-based flow analysis can be done in almost-linear time whereas an inclusion-based analysis is expected to be at least cubic." Another common choice is to use very simple approximations of the control-flow graph based on known function applications. We remark that, in practice, the standard CFA algorithm (and its derivatives) rarely exhibit cubic behavior, but they are often non-linear.

Recent work [6] indicates that it is unlikely we can improve on the cubic-time complexity of standard control-flow analysis, because it is as hard as the 2-NPDA acceptability problem. (Melski and Reps [12] have recently shown a similar kind of "cubic-hardness" result for first-order set-based analysis with data-constructors.) In this paper we focus on bounded-type programs. For monotyped (or simply typed) programs³, we simply bound the tree-size of a program's types by some constant. Equivalently, we could bound a program's order and arity, where arity is defined so that currying increases argument count rather than order; for example, the usual curried version of integer map with type $(Int \rightarrow Int) \rightarrow Int \ list \rightarrow Int \ list$ has arity 2 and order 2.

Bounded-type programs capture the intuition that functions rarely have more than, say, 20 arguments or have order greater than 3, and almost never at the same time (while this holds for most hand-written programs, the case is rather less clear for automatically generated programs). This class of programs has been particularly useful for understanding the observed linear behavior of type inference for ML [13]; see [7, 10]. However, it cannot be used to control the complexity of the standard CFA algorithm, which is cubic even when type size is bounded. Section 10 illustrates this with an example.

The main result of this paper is a linear-time algorithm for bounded-type programs that builds a directed graph whose transitive closure gives exactly the results of the standard (cubic-time) CFA algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. More importantly, for many applications that consume (standard) control-flow information, we can adapt our algorithm to perform the necessary control-flow analysis and postprocessing of the control-flow information all in linear-time. We illustrate this by giving lineartime algorithms for:

- effects analysis: find the side-effecting expressions in a program.
- k-limited CFA: for each call-site, list the functions if there are only a few of them (≤ k) and otherwise output "many".
- called-once analysis: identify all functions called from only one call-site.

Our algorithm is simple, incremental, demanddriven and easily adapted to polyvariant usage. Early experimental evidence suggests that this new algorithm is significantly faster than the standard algorithm.

Section 2 gives the definition of the standard control-flow problem. In Section 3, we present the linear-time algorithm. The main insight of our algorithm is a decoupling of the transitive closure and edge addition aspects of the standard CFA algorithm (the standard CFA algorithm can be viewed as transitive closure intertwined with edge addition due to newly discovered function applications). Section 4 uses types to show termination. Sections 5 shows termination for polymorphically typed programs. Section 6 extends the basic algorithm to records and recursive datatypes (the treatment of recursive datatypes is somewhat less accurate than in other approaches such as set-based analysis [4]). Section 7 considers polyvariance. Sections 8 and 9 show how our algorithm can be used to provide linear-time algorithms for some CFA-consuming applications. In Section 10, we provide some empirical evidence that our new algorithm has practical significance. Preliminary results suggest that the new algorithm is significantly faster than the standard algorithm for ML programs. We also provide a comparison on some programs that exhibit worst-case cubic behavior. Perhaps

 $^{^3 \}rm We$ discuss the notion of bounded-type for polymorphically typed programs (in the sense of ML) in Section 5.

Linear-time Subtransitive Control Flow Analysis

Nevin Heintze*

David McAllester[†]

Abstract

We present a linear-time algorithm for boundedtype programs that builds a directed graph whose transitive closure gives exactly the results of the standard (cubic-time) Control-Flow Analysis (CFA) algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. More importantly, it can be used to give linear-time algorithms for CFAconsuming applications such as:

- effects analysis: find the side-effecting expressions in a program.
- k-limited CFA: for each call-site, list the functions if there are only a few of them (≤ k) and otherwise output "many".
- called-once analysis: identify all functions called from only one call-site.

1 Introduction

The control-flow graph of a program plays a central role in compilation – it identifies the block and loop structure in a program, a prerequisite for many code optimizations. For first-order languages, this graph can be directly constructed from a program because information about flow of control from one point to another is explicit in the program (we remark that in the context of tail-call optimization, some extra work may be needed; see [3]).

For languages with higher-order functions, the situation is very different: the flow of control from one point to another is not readily apparent from program text because a function can be passed around as data and subsequently called from anywhere in the program. This limits compiler optimization. One way to address this problem is to perform *control-flow analysis* (CFA) to determine (an approximation) of the functions that may be called from each call site in a program [8, 9, 17, 16]. (In fact, some form of CFA is used in most forms of analyses for higher-order languages.)

Many different control-flow analyses have been developed (often independently), with variations in:

- 1. the treatment of context: does the analysis take into account the calling context of a function (polyvariant¹ treatment) or does it fold all activations of a function together (monovariant² treatment)?
- 2. the treatment of dead-code: does the analysis take into account which pieces of a program can actually be called? Does it take into account reduction order?
- 3. the treatment of data-constructors: what happens when a function is stored in a list and then later extracted from the list – is the identity of a function traced through recursive data-structures?

Despite these variations, a fundamental notion of CFA has emerged – the monovariant form of CFA defined over the pure lambda calculus. We call this analysis *standard CFA* (see Section 2 for a definition). Most other forms of CFA can be viewed as modification or extensions of standard CFA. Moreover, a number of connections have been established between standard CFA and a variety of type inference problems [15, 5].

^{*}Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, nch@bell-labs.com.

[†]AT&T Labs, 600 Mountain Ave, Murray Hill, NJ 07974, dmac@research.att.com.

¹A number of different terminologies have developed for this concept: a polyvariant analyses is often called "context-sensitive", and sometimes "polymorphic".

²A monovariant analysis is often called "contextinsensitive", and sometimes "monomorphic".

A Type System Equivalent to Flow Analysis^{*}

Jens Palsberg^{\dagger} Patrick O'Keefe^{\ddagger}

Abstract

Flow-based safety analysis of higher-order languages has been studied by Shivers, and Palsberg and Schwartzbach. Open until now is the problem of finding a type system that accepts exactly the same programs as safety analysis.

In this paper we prove that Amadio and Cardelli's type system with subtyping and recursive types accepts the same programs as a certain safety analysis. The proof involves mappings from types to flow information and back. As a result, we obtain an inference algorithm for the type system, thereby solving an open problem.

1 Introduction

1.1 Background

Many program analyses for higher-order languages are based on *flow analysis*, also known as *closure analysis*. Examples include the binding-time analyses for Scheme in the partial evaluators Schism [5] and Similix [3]. Such analyses have the advantage that they can be applied to *untyped* languages. This is in contrast to more traditional abstract interpretations which use types when defining the abstract domains.

Recently, it has become popular to define program analyses for typed languages by annotating the types with information about program behavior [10, 2]. This has lead to clear specifications of a range of analyses, and often such an analysis can be efficiently computed by a straightforward extension of a known type inference algorithm.

The precision of a type-based analysis depends on the expressiveness of the underlying type system. Similarly, the precision of a flow-based analysis depends on the expressiveness of the underlying flow analysis. In this paper we address an instance of the following fundamental question:

Fundamental question. What type-based analysis computes the same information as a given flow-based analysis?

^{*}ACM Transactions on Programming Languages and Systems, $17(4){:}576{-}599,$ July 1995. Preliminary version in Proc. POPL'95.

[†]Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark. E-mail: palsberg@daimi.aau.dk.

[‡]151 Coolidge Avenue #211, Watertown, MA 02172, USA. E-mail: pmo@world.std.com.

We consider the case of flow-based *safety* analysis, that is, an analysis which collects type information from for example constants and applications of primitive operations. Such an analysis was first presented in 1991 by Shivers [18] who called it type *recovery*. Later, Palsberg and Schwartzbach [12, 15] proved that on the basis of the collected information, one can define a predicate which accepts only programs which cannot go wrong. They called this *safety* analysis. They also proved that their safety analysis accepts more programs than simple type inference.

In this paper, we consider the following instance of the above question:

Which type system accepts the same programs as safety analysis?

The particular safety analysis we consider is defined in Section 3. It is based on a flow analysis which in the terminology of Shivers [17] is a 0CFA, that is, a 0-level control-flow analysis. Intuitively, it is a flow analysis which for each function merges all environment information.

Many program analyses are based on 0CFA-style analyses, see for example [16, 22, 7]. Our thesis is that the type system that answers the specific question will in many cases also be the answer to the fundamental question.

Flow-based analyses have the reputation of fitting poorly together with separate compilation because they deal with program points. In contrast, traditional type systems such as that of ML fit well together with separate compilation because one can compute a principal type for each subterm. Our hope is that the type system that answers the specific question above will lead to a better understanding of how to create program analyses that are both modular and have the power of flow-based analyses.

1.2 Our result

We prove that a natural type system with subtyping and recursive types accepts the same programs as safety analysis. The proof involves mappings from types to flow information and back.

The type system has been studied by Amadio and Cardelli [1], and an $O(n^2)$ algorithm for deciding the subtyping relation has been presented by Kozen, Palsberg, Schwartzbach, [9]. Open until now is the question of type inference. As a corollary of our result we get a type inference algorithm which works by first doing safety analysis and then mapping the flow information to types.

The set of types can be presented by the following grammar:

 $t ::= t_1 \to t_2 \mid \mathsf{Int} \mid v \mid \mu v.t \mid \top \mid \bot$

The type system contains the following components: the binary function type constructor \rightarrow , the constant type Int, the possibility for creating recursive types, and two more constant types \top , and \perp . Moreover, there is a subtype relation, written \leq . In contrast, safety analysis uses an abstract domain containing sets of syntactic occurrences of abstractions and the constant Int.

In slogan-form, our result reads:

Flow analysis + Safety checks = Simple types + Recursive types + \top + \perp + Subtyping

Each component of the type system captures a facet of flow analysis:

- The function type constructor → corresponds to a set of abstractions. Intuitively, a function type is less concrete than a set of abstractions. Indeed, the other components of the type system are essential to make it accept the same programs as the safety analysis.
- The constant **Int** is used for the same purpose in both systems. For simplicity, we do not consider other base types, or product and sum constructors, etc. Such constructs can be handled by techniques that are similar to the ones we will present.
- Recursive types are needed in order that safety analysis accepts all programs that do not contain constants.
- The constant ⊤ corresponds to the largest possible set of flow information. This type is needed for variables which can hold both a function and a base value. Intuitively, a program with such a variable should be type incorrect. However, the flow-based analysis may detect that this variable is only passed around but never actually used. For the type system to have that capability, ⊤ is required.
- The constant ⊥ corresponds to the empty set of flow information. This type is needed for variables which are used both as a function and as a base value. Intuitively, a program that uses a variable in both these ways should be type incorrect. However, the flow-based analysis may detect that this part of the program will never be executed. For the type system to have that capability, ⊥ is required.
- Subtyping is needed to capture flow of information. Intuitively, if information flows from A to B, then the type of A will be a subtype of the type of B.

Palsberg and Schwartzbach [15, 12] proved that the system without \perp accepts at most as many programs as safety analysis. In this paper we present the type system which accepts exactly the same programs as safety analysis. This may be seen as a natural culmination of the previous results.

1.3 Examples

Our example language is a λ -calculus, generated by the following grammar:

 $E ::= x \mid \lambda x.E \mid E_1E_2 \mid \mathbf{0} \mid \mathsf{succ} \ E$

Programs that yield a run-time error include (0 x), $\operatorname{succ}(\lambda x.x)$, and $(\operatorname{succ} 0)(x)$, because 0 is not a function, succ cannot be applied to functions, and $(\operatorname{succ} 0)$ is not a function. These programs are not typable and they are rejected by safety analysis. Some programs can be typed in the type system without the use of \bot and \top , for example

 $\lambda x.xx : \mu \alpha.\alpha \to \alpha$,

where E: t means "E has type t". Some programs require the use of \top , for example

$$(\lambda f.(\lambda x.fI)(f\mathbf{0}))I : \top,$$

where $I = \lambda x.x$. Note that \top is the only type of $(\lambda f.(\lambda x.fI)(f\mathbf{0}))I$ because f has to be assigned the type $\top \to \top$. Some programs require the use of \bot , for example

$$\lambda x.x(\operatorname{succ} x) : \bot \to t \text{ for any } t.$$

Both type inference and safety analysis can be phrased as solving a system of constraints, derived from the program text. The definitions of such constraint systems will be given in Sections 2.3 and 3. We will now present the constraint systems for the last of the above examples. For notational convenience, we give each of the two occurrences of x a label so that the λ -term reads $\lambda x.x_1(\operatorname{succ} x_2)$. For brevity, let $E = \lambda x.x_1(\operatorname{succ} x_2)$. The constraint system for type inference of E looks as follows:

$$\begin{array}{rcl} x \to \llbracket x_1(\operatorname{succ} \, x_2) \rrbracket &\leq & \llbracket E \rrbracket \\ \llbracket x_1 \rrbracket &\leq & \llbracket \operatorname{succ} \, x_2 \rrbracket \to \llbracket x_1(\operatorname{succ} \, x_2) \rrbracket \\ & x &\leq & \llbracket x_1 \rrbracket \\ & x &\leq & \llbracket x_2 \rrbracket \\ & \operatorname{Int} &\leq & \llbracket \operatorname{succ} \, x_2 \rrbracket \\ & \llbracket x_2 \rrbracket &\leq & \operatorname{Int} \end{array}$$

Here, the symbols x, $[x_1]$, $[x_2]$, $[succ x_2]$, $[x_1(succ x_2)]$, [E] are type variables. Intuitively, the type variable x is associated with the bound variable x, and the other type variables of the form $[\ldots]$ are associated with particular occurrences of subterms. Solving this constraint system yields that the possible types for the λ -term $\lambda x.x(succ x)$ are \top and $\bot \to t$ for any type t. Among these, $\bot \to \bot$ is a least type. In general, however, such a constraint system need not have a least solution. This reflects that in Amadio and Cardelli's type system, a typable term need not have a least type. For example, the term $\lambda x.x$ have both of the types $\bot \to \bot$ and $\top \to \top$, and these types are incomparable minimal types of $\lambda x.x$. Thus, before type checking and separately compiling a module, we may want to explicitly annotate the module boundary with the types we are interested in. It remains open, however, if modular program analyses can based on Amadio and Cardelli's type system.

The constraint system for safety analysis of E looks as follows:

$$\begin{array}{rcl} \{E\} &\subseteq & \llbracket E \rrbracket \\ & \llbracket x_1 \rrbracket &\subseteq & \{E\} \\ & x &\subseteq & \llbracket x_1 \rrbracket \\ & x &\subseteq & \llbracket x_2 \rrbracket \\ \{E\} \subseteq \llbracket x_1 \rrbracket &\Rightarrow & \llbracket \operatorname{succ} x_2 \rrbracket \subseteq x \\ \{E\} \subseteq \llbracket x_1 \rrbracket &\Rightarrow & \llbracket x_1 (\operatorname{succ} x_2) \rrbracket \subseteq & \llbracket x_1 (\operatorname{succ} x_2) \rrbracket \\ & \left\{\operatorname{Int}\right\} &\subseteq & \llbracket \operatorname{succ} x_2 \rrbracket \\ & \llbracket x_2 \rrbracket &\subseteq & \left\{\operatorname{Int}\right\} \end{array}$$

This constraint system uses the same variables as the one above, but now the type variables range over finite sets of occurrences of abstractions and the constant Int.

If such a constraint system is solvable, then it has a least solution. This particular constraint system is indeed solvable, and the least solution is the mapping φ , where

$$\begin{split} \varphi(\llbracket E \rrbracket) &= \{E\}\\ \varphi(\llbracket \mathsf{succ} \ x_2 \rrbracket) &= \{\mathsf{Int}\}\\ \varphi(\llbracket x_1(\mathsf{succ} \ x_2) \rrbracket) &= \varphi(x) = \varphi(\llbracket x_1 \rrbracket) = \varphi(\llbracket x_2 \rrbracket) = \emptyset \end{split}$$

This example will be treated in much further detail in Section 5.1.

In the following two sections we present the type system and the safety analysis, and in Section 4 we prove that they accept the same programs. In Section 5 we present two examples, in Section 6 we discuss various extensions, and in Section 7 we outline directions for further work. The reader is encouraged to refer to the examples while reading the other sections.

2 The type system

2.1 Types

We now define the notions of type, term, and term automaton. The idea is that a type is represented by a term which in turn is represented by a term automaton.

Definition 1 Let $\Sigma = \{ \rightarrow, \mathsf{Int}, \bot, \top \}$ be the ranked alphabet where \rightarrow is binary and Int, \bot, \top are nullary. A *type* is a regular tree over Σ . A *path* from the root of such a tree is a string over $\{0, 1\}$, where 0 indicates "left subtree" and 1 indicates "right subtree".

Definition 2 We represent a type by a *term*, that is, a partial function

 $t: \{0,1\}^* \to \Sigma$

with domain $\mathcal{D}(t)$ where t maps each path from the root of the type to the symbol at the end of the path. The set of all such terms is denoted T_{Σ} .

Following [9], we finitely represent a term by a so-called *term automaton*, as follows.

Definition 3 A term automaton over Σ is a tuple

 $\mathcal{M} = (Q, \Sigma, q_0, \delta, \ell)$

where:

- Q is a finite set of *states*,
- $q_0 \in Q$ is the start state,
- $\delta: Q \times \{0,1\} \to Q$ is a partial function called the *transition function*, and

• $\ell: Q \to \Sigma$ is a (total) labeling function,

such that for any state $q \in Q$, if $\ell(q) \in \{\rightarrow\}$ then

 $\{i \mid \delta(q, i) \text{ is defined}\} = \{0, 1\}$

and if $\ell(q) \in \{\mathsf{Int}, \bot, \top\}$ then

 $\{i \mid \delta(q, i) \text{ is defined}\} = \emptyset$.

The partial function δ extends naturally to a partial function

$$\widehat{\delta}: Q \times \{0,1\}^* \to Q$$

inductively as follows:

$$\begin{array}{rcl} \delta(q,\epsilon) &=& q\\ \widehat{\delta}(q,\alpha i) &=& \delta(\widehat{\delta}(q,\alpha),i) \;, \; \text{for} \; i \in \{0,1\}. \end{array}$$

The term represented by \mathcal{M} is the term

$$t_{\mathcal{M}} = \lambda \alpha . \ell(\delta(q_0, \alpha))$$

Intuitively, $t_{\mathcal{M}}(\alpha)$ is determined by starting in the start state q_0 and scanning the input α , following transitions of \mathcal{M} as far as possible. If it is not possible to scan all of α because some *i*-transition along the way does not exist, then $t_{\mathcal{M}}(\alpha)$ is undefined. If on the other hand \mathcal{M} scans the entire input α and ends up in state q, then $t_{\mathcal{M}}(\alpha) = \ell(q)$.

For example, consider the type



which can be understood as a representation of $\mu v.(v \to \bot)$. We represent this type by the term t where the domain of t is the infinite regular set $0^* + 0^*1$ and where $t(0^n) = \to$ and $t(0^n 1) = \bot$ for all $n \ge 0$. The corresponding term automaton is



Thus, infinite paths in a type yield cycles in the corresponding term automaton.

Types are ordered by the subtype relation \leq , as follows.

Definition 4 The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of 0's in α . The parity of α is denoted $\pi \alpha$. A string α is said to be *even* if $\pi \alpha = 0$ and *odd* if $\pi \alpha = 1$. Let \leq_0 be the partial order on Σ given by

$$\begin{array}{ll} \bot \leq_0 \to & \text{and} & \to \leq_0 \top \text{ and} \\ \bot \leq_0 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_0 \top \end{array}$$

and let \leq_1 be its reverse

$$\begin{array}{ll} \top \leq_1 \to & \text{and} & \to \leq_1 \bot \text{ and} \\ \top \leq_1 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_1 \bot \end{array}$$

For $s, t \in T_{\Sigma}$, define $s \leq t$ if $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$.

Kozen, Palsberg, and Schwartzbach [9] showed that the relation \leq is equivalent to the order defined by Amadio and Cardelli [1]. The relation \leq is a partial order, and if $s \to t \leq s' \to t'$, then $s' \leq s$ and $t \leq t'$ [1, 9].

2.2 Type rules

If E is a λ -term, t is a type, and A is a type environment, *i.e.* a partial function assigning types to variables, then the judgement

 $A \vdash E : t$

means that E has the type t in the environment A. Formally, this holds when the judgement is derivable using the following six rules:

$$A \vdash \mathbf{0} : \mathsf{Int} \tag{1}$$

$$A \vdash E : \mathsf{Int}$$
 (2)

$$A \vdash \mathsf{succ} \ E : \mathsf{Int}$$
 (2)

 $A \vdash x : t \pmod{A(x) = t}$ (3)

$$\frac{A[x \leftarrow s] \vdash E:t}{A \vdash \lambda x.E: s \to t} \tag{4}$$

$$\frac{A \vdash E : s \to t \quad A \vdash F : s}{A \vdash EF : t} \tag{5}$$

$$\frac{A \vdash E : s \quad s \le t}{A \vdash E : t} \tag{6}$$

The first five rules are the usual rules for simple types and the last rule is the rule of *subsumption*.

The type system has the subject reduction property, that is, if $A \vdash E : t$ is derivable and E β -reduces to E', then $A \vdash E' : t$ is derivable. This is proved by straightforward induction on the structure of the derivation of $A \vdash E : t$.

2.3 Constraints

Given a λ -term E, the type inference problem can be rephrased in terms of solving a system of type constraints. Assume that E has been α -converted so that all bound variables are distinct. Let X_E be the set of λ -variables x occurring in E, and let Y_E be a set of variables disjoint from X_E consisting of one variable $\llbracket F \rrbracket$ for each occurrence of a subterm F of E. (The notation $\llbracket F \rrbracket$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant.) We generate the following system of inequalities over $X_E \cup Y_E$. Each inequality is of the form $W \leq W'$ where W is of the forms V, lnt, or $(V \to V')_{\lambda x.F}$, and where W' is of the form V, lnt, or $(V \to V')_{GH}$, for $V, V' \in X_E \cup Y_E$.

• for every occurrence in E of a subterm of the form **0**, the inequality

Int \leq $\llbracket 0 \rrbracket$;

• for every occurrence in E of a subterm of the form succ F, the two inequalities

$$\begin{array}{rcl} \mathsf{Int} & \leq & [\![\mathsf{succ} \ F]\!] \\ [\![F]\!] & \leq & \mathsf{Int} \end{array}; \end{array}$$

• for every occurrence in E of a subterm of the form $\lambda x.F$, the inequality

 $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq \llbracket \lambda x.F \rrbracket;$

• for every occurrence in E of a subterm of the form GH, the inequality

 $\llbracket G \rrbracket \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH} ;$

• for every occurrence in E of a λ -variable x, the inequality

$$x \leq \llbracket x \rrbracket .$$

The subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

Denote by T(E) the system of constraints generated from E in this fashion. For every λ -term E, let $\mathsf{Tmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to T_{Σ} . The function $\psi \in \mathsf{Tmap}(E)$ is a solution of T(E), if it is a solution of each constraint in T(E). Specifically, for $V, V', V'' \in X_E \cup Y_E$, and occurrences of subterms $\lambda x.F$ and GH in E:

The constraint:	has solution ψ if:
$Int \leq V$	$Int \leq \psi(V)$
$V \leq Int$	$\psi(V) \leq Int$
$(V \to V')_{\lambda x.F} \le V''$	$\psi(V) \to \psi(V') \le \psi(V'')$
$V \le (V' \to V'')_{GH}$	$\psi(V) \le \psi(V') \to \psi(V'')$
$V \leq V'$	$\psi(V) \le \psi(V')$

The solutions of T(E) correspond to the possible type annotations of E in a sense made precise by Theorem 5.

Let A be a type environment assigning a type to each λ -variable occurring freely in E. If ψ is a function assigning a type to each variable in $X_E \cup Y_E$, we say that ψ extends A if A and ψ agree on the domain of A.

Theorem 5 The judgement $A \vdash E$: t is derivable if and only if there exists a solution ψ of T(E) extending A such that $\psi(\llbracket E \rrbracket) = t$. In particular, if E is closed, then E is typable with type t if and only if there exists a solution ψ of T(E) such that $\psi(\llbracket E \rrbracket) = t$.

Proof. Similar to the proof of Theorem 2.1 in the journal version of [8], in outline as follows. Given a solution of the constraint system, it is straightforward to construct a derivation of $A \vdash E:t$. Conversely, observe that if $A \vdash E:t$ is derivable, then there exists a derivation of $A \vdash E:t$ such that each use of one of the ordinary rules is followed by exactly one use of the subsumption rule. The approach in for example [21, 12] then gives a set of inequalities of the desired form.

3 The safety analysis

Following [15, 12], we will use a flow analysis as a basis for a safety analysis. Given a λ -term E, assume that E has been α -converted so that all bound variables are distinct. The set $\mathsf{Abs}(E)$ is the set of occurrences of subterms of E of the form $\lambda x.F$. The set $\mathsf{Cl}(E)$ is the powerset of $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$. Safety analysis of a λ -term E can be phrased as solving the following system of constraints over $X_E \cup Y_E$ where type variables range over $\mathsf{Cl}(E)$.

• For every occurrence in E of a subterm of the form 0, the constraint

 $\{\mathsf{Int}\} \subseteq \llbracket 0 \rrbracket;$

• for every occurrence in E of a subterm of the form succ F, the two constraints

$$\begin{aligned} \{ \mathsf{Int} \} &\subseteq \quad \llbracket \mathsf{succ} \ F \rrbracket \\ & \llbracket F \rrbracket \quad \subseteq \quad \{ \mathsf{Int} \} \end{aligned}$$

where the latter provides a safety check;

• for every occurrence in E of a subterm of the form $\lambda x.F$, the constraint

 $(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket \lambda x.F \rrbracket ;$

• for every occurrence in E of a subterm of the form GH, the constraint

 $\llbracket G \rrbracket \subseteq (\mathsf{Abs}(E))_{GH} ;$

which provides a safety check;

• for every occurrence in E of a λ -variable x, the constraint

 $x \subseteq \llbracket x \rrbracket;$

• for every occurrence in E of a subterm of the form $\lambda x.F$, and for every occurrence in E of a subterm of the form GH, the constraints

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket H \rrbracket \subseteq x (\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket F \rrbracket \subseteq \llbracket G H \rrbracket$$

Again, the subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

The constraints in the fourth and sixth items reflect some of the significant differences between the type system and the safety analysis. Intuitively, a constraint of the form $\llbracket G \rrbracket \subseteq$ $(Abs(E))_{GH}$ ensures that G does not evaluate to an integer. This is by the type constraints ensured by the constraint $\llbracket G \rrbracket \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$ because the type Int is not a subtype of any function type. The constraints of the forms

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket H \rrbracket \subseteq x (\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket F \rrbracket \subseteq \llbracket G H \rrbracket$$

creates a connection between the caller GH and the potential callee $\lambda x.F$. Intuitively, if G evaluates to $\lambda x.F$, then the argument H is bound to x, and the result of evaluating the body F becomes the result of whole application GH.

Denote by C(E) the system of constraints generated from E in this fashion. For every λ -term E, let $\mathsf{Cmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to $\mathsf{Cl}(E)$. The function $\varphi \in \mathsf{Cmap}(E)$ is a solution of C(E), if it is a solution of each constraint in C(E). Specifically, for $V, V', V'' \in X_E \cup Y_E$, and occurrences of subterms $\lambda x.F$ and GH in E:

The constraint:	has solution φ if:
$\{Int\}\subseteq V$	$\{Int\}\subseteq \varphi(V)$
$V \subseteq \{Int\}$	$\varphi(V) \subseteq \{Int\}$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq V$	$\{\lambda x.F\} \subseteq \varphi(V)$
$V \subseteq (Abs(E))_{GH}$	$\varphi(V) \subseteq Abs(E)$
$V \subseteq V'$	$\varphi(V) \subseteq \varphi(V')$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq V \Rightarrow V' \subseteq V''$	$\{\lambda x.F\} \subseteq \varphi(V) \Rightarrow \varphi(V') \subseteq \varphi(V'')$

Solutions are ordered by variable-wise set inclusion. See [15, 14] for a cubic time algorithm that given E computes the least solution of C(E) or decides that none exists. See [11] for a proof technique that enables a proof of the following subject reduction property. If $E \beta$ -reduces to E', and C(E) is solvable, then C(E') is also solvable.

4 Equivalence

4.1 Deductive Closures

We now introduce two auxiliary constraint systems called $\overline{C}(E)$ and $\overline{T}(E)$. They may be thought of as "deductive closures" of C(E) and T(E). We then show that they are isomorphic (Theorem 9). For examples of deductive closures, see Section 5.

Definition 6 For every λ -term E, define $\overline{C}(E)$ to be the smallest set such that:

- The non-conditional constraints of C(E) are members of $\overline{C}(E)$.
- If a constraint $c \Rightarrow K$ is in C(E) and c is in $\overline{C}(E)$, then K is in $\overline{C}(E)$.

• For $s \in X_E \cup Y_E$, if $r \subseteq s$ and $s \subseteq t$ both are in $\overline{C}(E)$, then $r \subseteq t$ is in $\overline{C}(E)$.

Notice that every constraint in $\overline{C}(E)$ is of the form $W \subseteq W'$, where W is of the forms V, $\{\mathsf{Int}\}$, or $(\{\lambda x.F\})_{\lambda x.F}$, and where W' is of the forms V, $\{\mathsf{Int}\}$, or $(\mathsf{Abs}(E))_{GH}$, for $V \in X_E \cup Y_E$.

For every λ -term E, define also the series $C_n(E)$, for $n \ge 0$, of subsets of $\overline{C}(E)$.

- $C_0(E)$ is the set of non-conditional constraint in C(E).
- For $n \ge 0$, $C_{2n+2}(E)$ is the smallest set such that $C_{2n+2}(E) \supseteq C_{2n+1}(E)$ and such that if a constraint $c \Rightarrow K$ is in C(E) and c is in $C_{2n+1}(E)$, then K is in $C_{2n+2}(E)$.
- For $n \ge 0$, $C_{2n+1}(E)$ is the smallest set such that $C_{2n+1}(E) \supseteq C_{2n}(E)$ and such that for $s \in X_E \cup Y_E$, if $r \le s$ and $s \le t$ both are in $C_{2n}(E)$, then $r \le t$ is in $C_{2n+1}(E)$.

Notice that $C_i(E) \subseteq C_j(E)$ for $0 \le i \le j$. Clearly, there exists $N \ge 0$ such that for all $n \ge N$, $C_n(E) = \overline{C}(E)$.

Definition 7 For every λ -term E, define $\overline{T}(E)$ to be the smallest set such that:

- $T(E) \subseteq \overline{T}(E)$.
- If $(s \to t)_{\lambda x.F} \leq (s' \to t')_{GH}$ is in $\overline{T}(E)$, then $s' \leq s$ and $t \leq t'$ are in $\overline{T}(E)$.
- For $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $\overline{T}(E)$, then $r \leq t$ is in $\overline{T}(E)$.

Notice that every constraint in $\overline{T}(E)$ is of the form $W \leq W'$ where W is of the forms V, Int, or $(V \to V')_{\lambda x.F}$, and where W' is of the form V, Int, or $(V \to V')_{GH}$, for $V, V' \in X_E \cup Y_E$. For every λ -term E, define also the series $T_n(E)$, for $n \geq 0$, of subsets of $\overline{T}(E)$.

- $T_0(E) = T(E)$.
- For $n \ge 0$, $T_{2n+2}(E)$ is the smallest set such that $T_{2n+2}(E) \supseteq T_{2n+1}(E)$ and such that if $(s \to t)_{\lambda x.F} \le (s' \to t')_{GH}$ is in $T_{2n+1}(E)$, then $s' \le s$ and $t \le t'$ are in $T_{2n+2}(E)$.
- For $n \ge 0$, $T_{2n+1}(E)$ is the smallest set such that $T_{2n+1}(E) \supseteq T_{2n}(E)$ and such that for $s \in X_E \cup Y_E$, if $r \le s$ and $s \le t$ both are in $T_{2n}(E)$, then $r \le t$ is in $T_{2n+1}(E)$.

Notice that $T_i(E) \subseteq T_j(E)$ for $0 \le i \le j$. Clearly, there exists $N \ge 0$ such that for all $n \ge N$, $T_n(E) = \overline{T}(E)$.

We will now present the definition of two functions \mathcal{I} and \mathcal{J} , one from $\overline{C}(E)$ to $\overline{T}(E)$ and one from $\overline{T}(E)$ to $\overline{C}(E)$. After the definition we prove that they are well-defined and each others inverses.

Definition 8 The functions

 $\mathcal{I}: \overline{C}(E) \to \overline{T}(E) \\ \mathcal{J}: \overline{T}(E) \to \overline{C}(E)$

are defined as follows.

$$\mathcal{I}(W \subseteq W') = (\mathcal{L}_{\mathcal{I}}(W) \le \mathcal{L}_{\mathcal{I}}(W'))$$

$$\mathcal{J}(W \le W') = (\mathcal{L}_{\mathcal{J}}(W) \subseteq \mathcal{L}_{\mathcal{J}}(W'))$$

where the functions $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}_{\mathcal{J}}$ are:

$$\mathcal{L}_{\mathcal{I}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \mathsf{Int} & \text{if } W = \{\mathsf{Int}\} \\ (x \to \llbracket F \rrbracket)_{\lambda x.F} & \text{if } W = (\{\lambda x.F\})_{\lambda x.F} \\ (\llbracket H \rrbracket \to \llbracket GH \rrbracket)_{GH} & \text{if } W = (\mathsf{Abs}(E))_{GH} \end{cases}$$
$$\mathcal{L}_{\mathcal{J}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \{\mathsf{Int}\} & \text{if } W = \mathsf{Int} \\ (\{\lambda x.F\})_{\lambda x.F} & \text{if } W = (x \to \llbracket F \rrbracket)_{\lambda x.F} \\ (\mathsf{Abs}(E))_{GH} & \text{if } W = (\llbracket H \rrbracket \to \llbracket GH \rrbracket)_{GH} \end{cases}$$

Theorem 9 The sets $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic, and \mathcal{I} and \mathcal{J} are bijections and each others inverses.

Proof. If \mathcal{I} and \mathcal{J} are well-defined, then clearly they are inverses of each other and thus bijections, so $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic.

First we show that \mathcal{I} is well-defined, that is, \mathcal{I} maps each element of $\overline{C}(E)$ to an element of $\overline{T}(E)$. It is sufficient to prove that for $n \geq 0$, \mathcal{I} maps each element of $C_n(E)$ to an element of $\overline{T}(E)$. We proceed by induction on n. In the base case, consider the constraints of $C_0(E)$, that is, the non-conditional constraints of C(E) and observe that for those we have:

$C_0(E)$	$T_0(E)$
$\{Int\}\subseteq \llbracket 0 rbracket$	$Int \leq \llbracket 0 \rrbracket$
$\{Int\} \subseteq \llbracket succ \ F \rrbracket$	$Int \leq \llbracket succ \ F \rrbracket$
$[\![F]\!] \subseteq \{Int\}$	$\llbracket F rbracket \leq Int$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket \lambda x.F \rrbracket$	$(x \to \llbracket F \rrbracket)_{\lambda x.F} \le \llbracket \lambda x.F \rrbracket$
$\llbracket G \rrbracket \subseteq (Abs(E))_{GH}$	$\llbracket G \rrbracket \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$
$x \subseteq [\![x]\!]$	$x \leq [\![x]\!]$

It follows that the lemma holds in the base case.

In the induction step, consider first $C_{2n+2}(E)$ for some $n \ge 0$. Suppose

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket H \rrbracket \subseteq x (\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket F \rrbracket \subseteq \llbracket GH \rrbracket$$

are in C(E) and suppose $(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket$ is in $C_{2n+1}(E)$. By the induction hypothesis, $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq \llbracket G \rrbracket$ is in $\overline{T}(E)$. Moreover, $\llbracket G \rrbracket \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$ is in T(E) and thus also in $\overline{T}(E)$. Hence, $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$ is in $\overline{T}(E)$, so also $\llbracket H \rrbracket \leq x$ and $\llbracket F \rrbracket \leq \llbracket G H \rrbracket$ are in $\overline{T}(E)$. Consider then $C_{2n+1}(E)$ for some $n \geq 0$. Suppose $r \subseteq s$ and $s \subseteq t$ are in $C_{2n}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(s)$ and $\mathcal{L}_{\mathcal{I}}(s) \leq \mathcal{L}_{\mathcal{I}}(t)$ are in $\overline{T}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{I}}(s) = s$, so $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(t)$ is in $\overline{T}(E)$.

Then we show that \mathcal{J} is well-defined, that is, \mathcal{J} maps each element of T(E) to an element of $\overline{C}(E)$. It is sufficient to prove that for $n \geq 0$, \mathcal{J} maps each element of $T_n(E)$ to an element of $\overline{C}(E)$. We proceed by induction on n. In the base case, consider the constraints of $T_0(E)$, that is, the constraints of T(E). Using the same table as above we observe that \mathcal{J} is well-defined on all these constraints.

In the induction step, consider first $T_{2n+2}(E)$ for some $n \geq 0$. Suppose $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq (\llbracket H \rrbracket \to \llbracket GH \rrbracket)_{GH}$ is in $T_{2n+1}(E)$. It is sufficient to prove that $\mathcal{L}_{\mathcal{J}}(\llbracket H \rrbracket) \subseteq \mathcal{L}_{\mathcal{J}}(x)$ and $\mathcal{L}_{\mathcal{J}}(\llbracket F \rrbracket) \subseteq \mathcal{L}_{\mathcal{J}}(\llbracket GH \rrbracket)$ are in $\overline{C}(E)$, or equivalently, that $\llbracket H \rrbracket \subseteq x$ and $\llbracket F \rrbracket \subseteq \llbracket GH \rrbracket$ are in $\overline{C}(E)$. In C(E) we have

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket H \rrbracket \subseteq x (\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket \implies \llbracket F \rrbracket \subseteq \llbracket G H \rrbracket$$

If $(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket$ is in $\overline{C}(E)$, then $\llbracket H \rrbracket \subseteq x$ and $\llbracket F \rrbracket \subseteq \llbracket G H \rrbracket$ are in $\overline{C}(E)$. To see that $(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket$ is in $\overline{C}(E)$, notice that in T(E), $(x \to \llbracket F \rrbracket)_{\lambda x.F}$ occurs only in the constraint $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq \llbracket \lambda x.F \rrbracket$, and $(\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$ occurs only in the constraint $\llbracket G \rrbracket \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$. Since $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq (\llbracket H \rrbracket \to \llbracket G H \rrbracket)_{GH}$ is in $T_{2n+1}(E)$ we get that also $\llbracket \lambda x.F \rrbracket \leq \llbracket G \rrbracket$ is in $T_{2n+1}(E)$. Hence, $(x \to \llbracket F \rrbracket)_{\lambda x.F} \leq \llbracket G \rrbracket$ is in $T_{2n+1}(E)$, so by the induction hypothesis, $(\{\lambda x.F\})_{\lambda x.F} \subseteq \llbracket G \rrbracket$ is in $\overline{C}(E)$.

Consider then $T_{2n+1}(E)$ for some $n \geq 0$. Suppose $r \leq s$ and $s \leq t$ are in $T_{2n}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(s)$ and $\mathcal{L}_{\mathcal{J}}(s) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ are in $\overline{C}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{J}}(s) = s$, so $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ is in $\overline{C}(E)$. \Box

4.2 The equivalence proof

The following construction is the key to mapping flow information to types.

Definition 10 For every λ -term $E, \varphi \in \mathsf{Cmap}(E)$, and $q_0 \in \mathsf{Cl}(E)$, define the term automaton $\mathcal{A}(E, \varphi, q_0)$ as follows:

$$\mathcal{A}(E,\varphi,q_0) = (\mathsf{Cl}(E), \Sigma, q_0, \delta, \ell)$$

where:

- $\delta(\{\lambda x_1.E_1,\ldots,\lambda x_n.E_n\},0) = \bigcap_{i=1}^n \varphi(x_i)$ for n > 0
- $\delta(\{\lambda x_1.E_1,\ldots,\lambda x_n.E_n\},1) = \bigcup_{i=1}^n \varphi[\![E_i]\!]$ for n > 0

•
$$\ell(q) = \begin{cases} \bot & \text{if } q = \emptyset \\ \mathsf{Int} & \text{if } q = \{\mathsf{Int}\} \\ \to & \text{if } q \subseteq \mathsf{Abs}(E) \land q \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

Lemma 11 Suppose $\varphi \in \mathsf{Cmap}(E)$ and $S_1, S_2 \in \mathsf{Cl}(E)$. If $S_1 \subseteq S_2$, then $t_{\mathcal{A}(E,\varphi,S_1)} \leq t_{\mathcal{A}(E,\varphi,S_2)}$.

Proof. Define the orderings \subseteq_0 , \subseteq_1 on $\mathsf{Cl}(E)$ such that \subseteq_0 equals \subseteq and \subseteq_1 equals \supseteq . The desired conclusion follows immediately from the property that if $\alpha \in \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_1)}) \cap \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_2)})$, then $\hat{\delta}(S_1, \alpha) \subseteq_{\pi\alpha} \hat{\delta}(S_2, \alpha)$. This property is proved by straightforward induction on the length of α .

We can now prove that the type system and the safety analysis accept the same programs.

Theorem 12 For every λ -term E, the following seven conditions are equivalent:

- 1. C(E) is solvable.
- 2. T(E) is solvable.
- 3. $\overline{T}(E)$ is solvable.
- 4. $\overline{C}(E)$ is solvable.
- 5. $\overline{C}(E)$ does not contain constraints of the forms {Int} $\subseteq Abs(E)$ or { $\lambda x.F$ } \subseteq {Int}.
- 6. $\overline{T}(E)$ does not contain constraints of the forms $\operatorname{Int} \leq V \to V'$ or $V \to V' \leq \operatorname{Int}$, where $V, V' \in X_E \cup Y_E$.
- 7. The function

 $\lambda V \{ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E) \}$

is the least solution of C(E).

Proof. Given a λ -term E, notice that by the isomorphism of Theorem 9, (5) \Leftrightarrow (6). To show the remaining equivalences, we proceed by proving the implications:

 $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4) \Rightarrow (5) \Rightarrow (7) \Rightarrow (1)$

To prove $(1) \Rightarrow (2)$, suppose C(E) has solution $\varphi \in \mathsf{Cmap}(E)$. Let f be the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and define $\psi \in \mathsf{Tmap}(E)$ by $\psi = f \circ \varphi$. We will show that T(E) has solution ψ . We consider each of the constraints in turn. The cases of the constraints generated from subterms of the forms 0, succ E, x are immediate, by using Lemma 11. Consider then $\lambda x.F$ and the constraint $x \to [\![F]\!] \leq [\![\lambda x.F]\!]$. By the definition of f and Lemma 11 we get

$$\psi(x) \to \psi(\llbracket F \rrbracket) = f(\{\lambda x.F\}) \le \psi(\llbracket \lambda x.F \rrbracket) .$$

Consider then GH and the constraint $\llbracket G \rrbracket \leq \llbracket H \rrbracket \to \llbracket GH \rrbracket$. We know that $\varphi(\llbracket G \rrbracket) \subseteq \mathsf{Abs}(E)$ so there are two cases. Suppose first that $\varphi(\llbracket G \rrbracket) = \emptyset$. We then have $\psi(\llbracket G \rrbracket) = \bot \leq \psi(\llbracket H \rrbracket) \to \psi(\llbracket GH \rrbracket)$. Consider then the case where $\varphi(\llbracket G \rrbracket) = \{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}$, for n > 0. We then have that $\varphi(\llbracket H \rrbracket) \subseteq \varphi(\llbracket x_i \rrbracket)$ and $\varphi(\llbracket E_i \rrbracket) \subseteq \varphi(\llbracket GH \rrbracket)$ for $i \in \{1, \ldots, n\}$. Thus, $\varphi(\llbracket H \rrbracket) \subseteq \bigcap_{i=1}^n \varphi(\llbracket x_i \rrbracket)$ and $\bigcup_{i=1}^n \varphi(\llbracket E_i \rrbracket) \subseteq \varphi(\llbracket GH \rrbracket)$. So, by Lemma 11,

$$\begin{split} \psi(\llbracket G \rrbracket) &= f(\varphi(\llbracket G \rrbracket)) \\ &= f(\bigcap_{i=1}^{n} \varphi(\llbracket x_{i} \rrbracket)) \to f(\bigcup_{i=1}^{n} \varphi(\llbracket E_{i} \rrbracket)) \\ &\leq f(\varphi(\llbracket H \rrbracket)) \to f(\varphi(\llbracket GH \rrbracket)) \\ &= \psi(\llbracket H \rrbracket) \to \psi(\llbracket GH \rrbracket) \end{split}$$

To prove $(2) \Rightarrow (3)$, suppose T(E) has solution $\psi \in \mathsf{Tmap}(E)$. It is sufficient to show that $\overline{T}(E)$ has solution ψ , and this can be proved by straightforward induction on the construction of $\overline{T}(E)$.

To prove (3) \Rightarrow (4), suppose $\overline{T}(E)$ has solution $\psi \in \mathsf{Tmap}(E)$. Define $\overline{\varphi} \in \mathsf{Cmap}(E)$ as follows:

$$\overline{\varphi}(V) = \begin{cases} \emptyset & \text{if } (\psi(V))(\epsilon) = \bot \\ \{ \mathsf{Int} \} & \text{if } (\psi(V))(\epsilon) = \mathsf{Int} \\ \mathsf{Abs}(E) & \text{if } (\psi(V))(\epsilon) = \to \\ \mathsf{Abs}(E) \cup \{ \mathsf{Int} \} & \text{if } (\psi(V))(\epsilon) = \top \end{cases}$$

We will show that $\overline{C}(E)$ has solution $\overline{\varphi}$. To see this, let $W \subseteq Z$ be a constraint in $\overline{C}(E)$. If it is of the forms $\{\operatorname{Int}\} \subseteq \{\operatorname{Int}\}$ or $\{\lambda x.F\} \subseteq \operatorname{Abs}(E)$, then it is solvable by all functions, including $\overline{\varphi}$. For the remaining cases, notice that by Theorem 9, $\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(Z)$ is in $\overline{T}(E)$ and thus it has solution ψ . This means that $W \subseteq Z$ cannot be of the forms $\{\operatorname{Int}\} \subseteq \operatorname{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\operatorname{Int}\}$. Suppose then that $W \subseteq Z$ is of one of the remaining forms, that is, $\{\operatorname{Int}\} \subseteq V$, $V \subseteq V', V \subseteq \{\operatorname{Int}\}, \{\lambda x.F\} \subseteq V, V \subseteq \operatorname{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first of them, the others are similar. For a constraint of the form $\{\operatorname{Int}\} \subseteq V$, it follows that $\operatorname{Int} \leq V$ is in $\overline{T}(E)$. Since $\overline{T}(E)$ has solution ψ we get that $(\psi(V))(\epsilon) \in \{\operatorname{Int}, \top\}$. Thus, $\overline{\varphi}(V)$ is either $\{\operatorname{Int}\}$ or $\operatorname{Abs}(E) \cup \{\operatorname{Int}\}$, and hence $\{\operatorname{Int}\} \subseteq V$ has solution $\overline{\varphi}$.

To prove (4) \Rightarrow (5), observe that constraints of the forms {Int} $\subseteq Abs(E)$ or { $\lambda x.F$ } \subseteq {Int} are not solvable.

To prove $(5) \Rightarrow (7)$, suppose $\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Define

 $\varphi' = \lambda V.\{ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E) \}$

We proceed in four steps, as follows.

• First we show that φ' is a solution of $\overline{C}(E)$. We consider in turn each of the seven possible forms of constraints in $\overline{C}(E)$. Constraints of the forms $\{\mathsf{Int}\} \subseteq \{\mathsf{Int}\}$ and $\{\lambda x.F\} \subseteq$ $\mathsf{Abs}(E)$ have any solution, including φ' . We are thus left with constraints of the forms $\{\mathsf{Int}\} \subseteq V, V \subseteq V', V \subseteq \{\mathsf{Int}\}, \{\lambda x.F\} \subseteq V, V \subseteq \mathsf{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first three, since case four is similar to case one and since case five is similar to case three. For a constraint of the form $\{\mathsf{Int}\} \subseteq V$, notice that $\mathsf{Int} \in \varphi'(V)$, so the constraint has solution φ' . For a constraint of the form $V \subseteq V'$, suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq V'$ is also in $\overline{C}(E)$. It follows that $k \in \varphi'(V')$. For a constraint of the form $V \subseteq \{\mathsf{Int}\}$, suppose it does not have solution φ' . Hence, there exist $k \in \varphi'(V)$ such that $k \neq \mathsf{Int}$. It follows that the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq \{\mathsf{Int}\}$ is also in $\overline{C}(E)$, a contradiction.

- Next we show that φ' is the least solution of $\overline{C}(E)$. To do this, let φ be any solution of $\overline{C}(E)$ and suppose $V \in X_E \cup Y_E$. It is sufficient to prove that $\varphi'(V) \subseteq \varphi(V)$. Suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$. Since φ is a solution of $\overline{C}(E)$, $k \in \varphi(V)$.
- Next we show that φ' is a solution of C(E). Consider first the non-conditional constraints of C(E). Since these constraints are also members of $\overline{C}(E)$, they have solution φ' . Consider then $\{\lambda x.F\} \subseteq V \Rightarrow K$ in C(E) and suppose $\{\lambda x.F\} \subseteq V$ has solution φ' . Then by the definition of φ' , we have that $\{\lambda x.F\} \subseteq V$ is in $\overline{C}(E)$, so also K is in $\overline{C}(E)$, and hence K has solution φ' .
- Finally we show that φ' is the least solution of C(E). To do this, let φ be any solution of C(E). Then φ is also a solution of $\overline{C}(E)$, as can be proved by straightforward induction on the construction of $\overline{C}(E)$. Since φ' is the least solution of $\overline{C}(E)$, φ' is smaller than or equal to φ .

To prove $(7) \Rightarrow (1)$, simply notice that since C(E) has a solution, it is solvable.

Corollary 13 The type system accepts the same programs as the safety analysis.

The equivalence proof is illustrated in Section 5.

4.3 Algorithms

As corollaries of Theorem 12 we get two cubic time algorithms. Given a λ -term E, first observe that both $\overline{C}(E)$ and $\overline{T}(E)$ can be computed in time $O(n^3)$ where n is the size of E. We can then easily answer the following two questions:

- Question (safety): Is E accepted by safety analysis? Algorithm: Check that $\overline{C}(E)$ does not contain constraint of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}.$
- Question (type inference): Is E typable? If so, what is an annotation of it? Algorithm: Use the safety checking algorithm. If E turns out to be typable, we get an annotation by first calculating the two functions

$$\varphi' = \lambda V.\{ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E) \}$$

and

 $f = \lambda S.t_{\mathcal{A}(E,\varphi,S)}$

and then forming the composition

 $\psi = f \circ \varphi' \; .$

This function ψ is a solution of T(E).

The question of type inference has been open until now. In contrast, it is well-known that flow analysis in the style discussed in this paper can be computed in time $O(n^3)$.

It remains open to define a more direct $O(n^3)$ time type inference algorithm, that is, one that does not use the reduction to the safety checking problem.

5 Examples

We will illustrate the proof of equivalence with two examples. The λ -terms that will be treated are $\lambda x.x(\operatorname{succ} x)$, which was also discussed in Section 1, and $(\lambda x.xx)(\lambda y.y)$.

5.1 $\lambda x.x(\operatorname{succ} x)$

As in Section 1, we give each of the two occurrences of x a label so that the λ -term reads $\lambda x.x_1(\operatorname{succ} x_2)$. For brevity, let $E = \lambda x.x_1(\operatorname{succ} x_2)$. Notice that $\operatorname{Abs}(E) = \{E\}$. As stated in Section 1, C(E) looks as follows:

$$\begin{array}{rcl} \{E\} & \subseteq & \llbracket E \rrbracket \\ & \llbracket x_1 \rrbracket & \subseteq & \{E\} \\ & x & \subseteq & \llbracket x_1 \rrbracket \\ & x & \subseteq & \llbracket x_2 \rrbracket \\ \{E\} \subseteq \llbracket x_1 \rrbracket & \Rightarrow & \llbracket \operatorname{succ} x_2 \rrbracket \subseteq x \\ \{E\} \subseteq \llbracket x_1 \rrbracket & \Rightarrow & \llbracket x_1(\operatorname{succ} x_2) \rrbracket \subseteq & \llbracket x_1(\operatorname{succ} x_2) \rrbracket \\ & \left\{\operatorname{Int}\right\} & \subseteq & \llbracket \operatorname{succ} x_2 \rrbracket \\ & \llbracket x_2 \rrbracket & \subseteq & \left\{\operatorname{Int}\right\} \end{array}$$

The deductive closure $\overline{C}(E)$ looks as follows:

$$\{E\} \subseteq \llbracket E \rrbracket$$

$$\llbracket x_1 \rrbracket \subseteq \{E\}$$

$$x \subseteq \llbracket x_1 \rrbracket$$

$$x \subseteq \llbracket x_2 \rrbracket$$

$$\{\mathsf{Int}\} \subseteq \llbracket \mathsf{succ} x_2 \rrbracket$$

$$\llbracket x_2 \rrbracket \subseteq \{\mathsf{Int}\}$$

$$x \subseteq \{E\}$$

$$x \subseteq \{\mathsf{Int}\}$$

Intuitively, this deductive closure is obtained by observing that no constraint matches the condition of any of the two conditional constraints, and by using the transitivity rule twice.

As also stated in Section 1, T(E) looks as follows:

$$\begin{aligned} x \to \llbracket x_1(\mathsf{succ} \ x_2) \rrbracket &\leq \llbracket E \rrbracket \\ \llbracket x_1 \rrbracket &\leq \llbracket \mathsf{succ} \ x_2 \rrbracket \to \llbracket x_1(\mathsf{succ} \ x_2) \rrbracket \\ x &\leq \llbracket x_1 \rrbracket \\ x &\leq \llbracket x_2 \rrbracket \\ \mathsf{Int} &\leq \llbracket \mathsf{succ} \ x_2 \rrbracket \\ \llbracket x_2 \rrbracket &\leq \mathsf{Int} \end{aligned}$$

The deductive closure $\overline{T}(E)$ looks as follows:

$$\begin{aligned} x \to \llbracket x_1(\operatorname{succ} x_2) \rrbracket &\leq \llbracket E \rrbracket \\ \llbracket x_1 \rrbracket &\leq \llbracket \operatorname{succ} x_2 \rrbracket \to \llbracket x_1(\operatorname{succ} x_2) \rrbracket \\ x &\leq \llbracket x_1 \rrbracket \\ x &\leq \llbracket x_2 \rrbracket \\ \operatorname{Int} &\leq \llbracket \operatorname{succ} x_2 \rrbracket \\ \llbracket x_2 \rrbracket &\leq \operatorname{Int} \\ x &\leq \llbracket \operatorname{succ} x_2 \rrbracket \to \llbracket x_1(\operatorname{succ} x_2) \rrbracket \\ x &\leq \operatorname{Int} \end{aligned}$$

This deductive closure is obtained by using the transitivity rule twice.

It can be verified by inspection that Theorem 9 is true for E, that is, $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic. Moreover, $\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}, \text{ and } \overline{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}, \text{ where } V, V' \in X_E \cup Y_E.$

We are now ready to focus on the equivalence proof. We will go through that proof and illustrate each construction in the case of E.

The equivalence proof may be summarized as follows. The proof demonstrates several instances of how to transform a solution of one constraint system into a solution of an other constraint system. It may be helpful to think of a step as transforming the output of the previous step, as follows. The starting point is a solution φ of C(E). This φ is then transformed into a solution ψ of T(E). This ψ is also a solution of $\overline{T}(E)$, and it is then transformed into a solution $\overline{\varphi}$ of $\overline{C}(E)$. Having such a solution implies that certain constraints are not in $\overline{C}(E)$ (condition 5), and also that certain constraints are not in $\overline{T}(E)$ (condition 6). The function $\overline{\varphi}$ need not be a solution of C(E), but we can construct the least solution of C(E) from $\overline{C}(E)$. In one picture, the transformations go as follows:

 $\varphi \rightarrow \psi \rightarrow \overline{\varphi}$

We will now follow a particular φ as it tours this diagram.
As starting point, we choose the least solution φ of C(E) which was also stated in Section 1. It looks as follows:

$$\begin{split} \varphi(\llbracket E \rrbracket) &= \{E\} \\ \varphi(\llbracket \mathsf{succ} \ x_2 \rrbracket) &= \{\mathsf{Int}\} \\ \varphi(\llbracket x_1(\mathsf{succ} \ x_2) \rrbracket) &= \varphi(x) = \varphi(\llbracket x_1 \rrbracket) = \varphi(\llbracket x_2 \rrbracket) = \emptyset \end{split}$$

To get the solution ψ of T(E), we need to construct the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and compose it with φ . The automaton $\mathcal{A}(E,\varphi,S)$ can be illustrated as follows:

Notice that we have not pointed to the start state; it is a parameter of the specification. The illustration gives both the name and the label of each state. There are just two transitions, both from the state $\{E\}$ to the state \emptyset . Observe that

$$t_{\mathcal{A}(E,\varphi,S)} = \begin{cases} \bot \to \bot & \text{if } S = \{E\} \\ \mathsf{Int} & \text{if } S = \{\mathsf{Int}\} \\ \bot & \text{if } S = \emptyset \end{cases}$$

We can then obtain the mapping ψ :

$$\begin{split} \psi(\llbracket E \rrbracket) &= \bot \to \bot \\ \psi(\llbracket \operatorname{succ} x_2 \rrbracket) &= \operatorname{Int} \\ \psi(\llbracket x_1(\operatorname{succ} x_2) \rrbracket) &= \psi(x) = \psi(\llbracket x_1 \rrbracket) = \psi(\llbracket x_2 \rrbracket) = \bot \end{split}$$

It can be verified by inspection that ψ is a solution of T(E) and $\overline{T}(E)$.

To get the solution $\overline{\varphi}$ of $\overline{C}(E)$, we need to compute $(\psi(V))(\epsilon)$ for every $V \in X_E \cup Y_E$. For example,

$$(\psi(\llbracket E \rrbracket))(\epsilon) \; = \; (\bot \to \bot)(\epsilon) \; = \to \;$$

Plugging this into the definition of $\overline{\varphi}$ yields:

$$\begin{split} \overline{\varphi}(\llbracket E \rrbracket) &= \{E\}\\ \overline{\varphi}(\llbracket \mathsf{succ} \ x_2 \rrbracket) &= \{\mathsf{Int}\}\\ \overline{\varphi}(\llbracket x_1(\mathsf{succ} \ x_2) \rrbracket) &= \overline{\varphi}(x) = \overline{\varphi}(\llbracket x_1 \rrbracket) = \overline{\varphi}(\llbracket x_2 \rrbracket) = \emptyset \end{split}$$

So, $\varphi = \overline{\varphi}$, and it can be verified by inspection that $\overline{\varphi}$ is a solution of $\overline{C}(E)$.

Finally, to construct φ' where

$$\varphi' = \lambda V.\{ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E) \}$$

notice that the constraints in $\overline{C}(E)$ that have the form $\{k\} \subseteq V$ where $V \in X_E \cup Y_E$ are:

 $\{E\} \subseteq \llbracket E \rrbracket \\ \{\mathsf{Int}\} \subseteq \llbracket \mathsf{succ} \ x_2 \rrbracket$

So, $\varphi = \overline{\varphi} = \varphi'$, and hence φ' is the least solution of C(E).

It is only in special cases that $\varphi = \overline{\varphi}$. Next we consider a slightly more complicated example where this does not occur.

5.2 $(\lambda x.xx)(\lambda y.y)$

We give each of the two occurrences of x a label so that the λ -term reads $(\lambda x.x_1x_2)(\lambda y.y)$. For brevity, let $E = (\lambda x.x_1x_2)(\lambda y.y)$. Notice that $\mathsf{Abs}(E) = \{\lambda x.x_1x_2, \lambda y.y\}$. The constraint system C(E) looks as follows:

$$\begin{array}{lll} \lambda x.x_1x_2 & \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \\ \lambda y.y & \{\lambda y.y\} \subseteq \llbracket \lambda y.y \rrbracket \\ E & \llbracket \lambda x.x_1x_2 \rrbracket \subseteq \mathsf{Abs}(E) \\ x_1x_2 & \llbracket x_1 \rrbracket \subseteq \mathsf{Abs}(E) \\ x_1 & x \subseteq \llbracket x_1 \rrbracket \\ x_2 & x \subseteq \llbracket x_2 \rrbracket \\ y & y \subseteq \llbracket y \rrbracket \\ x_1x_2 \text{ and } \\ \lambda x.x_1x_2 & \{\lambda x.x_1x_2\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_2 \rrbracket \subseteq x \\ \{\lambda x.x_1x_2\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \subseteq \llbracket x_1x_2 \rrbracket \\ x_1x_2 \text{ and } \\ \{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_2 \rrbracket \subseteq y \\ \lambda y.y & \{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket y \rrbracket \subseteq \llbracket x_1x_2 \rrbracket \\ E \text{ and } \\ \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ E \text{ and } \\ \{\lambda y.y\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ \{\lambda y.y\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ \lambda y.y \rrbracket \subseteq \llbracket x \\ E \text{ and } \\ \{\lambda y.y\} \subseteq \llbracket x_1x_2 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \subseteq \llbracket x \\ \{\lambda y.y\} \subseteq \llbracket x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ E \text{ and } \\ \{\lambda y.y\} \subseteq \llbracket x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x \\ x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x_1x_2 \rrbracket \Rightarrow \llbracket x_1y.y \rrbracket \subseteq \llbracket x_1x_2 \rrbracket = \llbracket x \\ x_1x_2 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket = \llbracket x_1x_2 \rrbracket$$

To the left of the constraints, we have indicated from where they arise. The constraint system T(E) looks as follows:

From $\lambda x.x_1x_2$	$x \to \llbracket x_1 x_2 \rrbracket \le \llbracket \lambda x. x_1 x_2 \rrbracket$
From $\lambda y.y$	$y \to [\![y]\!] \le [\![\lambda y.y]\!]$
From E	$[\![\lambda x.x_1x_2]\!] \le [\![\lambda y.y]\!] \to [\![E]\!]$
From $x_1 x_2$	$\llbracket x_1 \rrbracket \le \llbracket x_2 \rrbracket \to \llbracket x_1 x_2 \rrbracket$
From x_1	$x \le \llbracket x_1 \rrbracket$
From x_2	$x \le \llbracket x_2 \rrbracket$
From y	$y \leq \llbracket y \rrbracket$

The deductive closures $\overline{C}(E)$ and $\overline{T}(E)$ look as follows.

$\overline{C}(E)$	$\overline{T}(E)$
$\{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket$	$x \to \llbracket x_1 x_2 \rrbracket \le \llbracket \lambda x. x_1 x_2 \rrbracket$
$\{\lambda x. x_1 x_2\} \subseteq Abs(E)$	$x \to \llbracket x_1 x_2 \rrbracket \le \llbracket \lambda y. y \rrbracket \to \llbracket E \rrbracket$
$\llbracket \lambda x. x_1 x_2 \rrbracket \subseteq Abs(E)$	$\llbracket \lambda x. x_1 x_2 \rrbracket \le \llbracket \lambda y. y \rrbracket \to \llbracket E \rrbracket$
$\{\lambda y.y\} \subseteq \llbracket \lambda y.y rbracket$	$y \to [\![y]\!] \le [\![\lambda y.y]\!]$
$\{\lambda y.y\} \subseteq x$	$y \to [\![y]\!] \le x$
$\{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket$	$y \to \llbracket y \rrbracket \leq \llbracket x_1 \rrbracket$
$\{\lambda y.y\} \subseteq Abs(E)$	$y \to \llbracket y \rrbracket \leq \llbracket x_2 \rrbracket \to \llbracket x_1 x_2 \rrbracket$
$\{\lambda y.y\} \subseteq [\![x_2]\!]$	$y \to [\![y]\!] \le [\![x_2]\!]$
$\{\lambda y.y\} \subseteq y$	$y \to [\![y]\!] \le y$
$\{\lambda y.y\} \subseteq \llbracket y \rrbracket$	$y \to [\![y]\!] \le [\![y]\!]$
$\{\lambda y.y\} \subseteq [\![x_1x_2]\!]$	$y \to [\![y]\!] \le [\![x_1 x_2]\!]$
$\{\lambda y.y\} \subseteq \llbracket E rbracket$	$y \to \ y\ \le \ E\ $
$[\![\lambda y.y]\!] \subseteq x$	$\left\ \lambda y.y\right\ \le x$
$ \ \lambda y.y\ \subseteq \ x_1\ $	$\ \lambda y.y\ \le \ x_1\ $
$[\lambda y.y] \subseteq Abs(E)$	$\ \lambda y.y\ \le \ x_2\ \to \ x_1x_2\ $
$ [\![\lambda y.y]\!] \subseteq [\![x_2]\!] $	$\ \lambda y.y\ \le \ x_2\ $
$\llbracket \lambda y.y \rrbracket \subseteq y$	$\ \lambda y.y\ \le y$
$\llbracket \lambda y. y \rrbracket \subseteq \llbracket y \rrbracket$	$\begin{bmatrix} \lambda y. y \end{bmatrix} \leq \begin{bmatrix} y \end{bmatrix}$
$\llbracket \lambda y.y \rrbracket \subseteq \llbracket x_1 x_2 \rrbracket$	$\begin{bmatrix} \lambda y. y \end{bmatrix} \leq \begin{bmatrix} x_1 x_2 \end{bmatrix}$ $\begin{bmatrix} \lambda y. y \end{bmatrix} \leq \begin{bmatrix} F \end{bmatrix}$
$\llbracket \land g . g \rrbracket \subseteq \llbracket L \rrbracket$	$\begin{bmatrix} X y \cdot y \end{bmatrix} \leq \begin{bmatrix} L \end{bmatrix}$ $x \leq \begin{bmatrix} x \end{bmatrix}$
$\begin{array}{c} x \leq \llbracket x_1 \rrbracket \\ x \subset \Deltabs(E) \end{array}$	$\begin{array}{c} x \leq \llbracket x_1 \rrbracket \\ x \leq \llbracket x_2 \rrbracket \longrightarrow \llbracket x_1 x_2 \rrbracket \end{array}$
$\begin{array}{c} x \subseteq ADS(L) \\ r \subset \llbracket r_0 \rrbracket \end{array}$	$\begin{array}{c} x \leq \left[x_2 \right] \forall \left[x_1 x_2 \right] \\ r < \left[r_2 \right] \end{array}$
$\begin{array}{c} x \leq \left[x_2 \right] \\ x \subset y \end{array}$	$\begin{array}{c} x \leq \left\lfloor x \right\rfloor_{2} \\ x \leq y \end{array}$
$\begin{array}{c} x \leq y \\ x \subset \llbracket y \rrbracket$	$x \leq y$ $x \leq [u]$
$x \subseteq \llbracket y \rrbracket$ $x \subset \llbracket x_1 x_2 \rrbracket$	$\begin{array}{c} x = \llbracket y \rrbracket \\ x < \llbracket x_1 x_2 \rrbracket \end{array}$
$x \subseteq \llbracket E \rrbracket$	$x \leq \llbracket E \rrbracket$
$\llbracket x_1 \rrbracket \subset Abs(E)$	$\begin{bmatrix} x_1 \\ x_1 \end{bmatrix} < \begin{bmatrix} x_2 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
$\llbracket x_2 \rrbracket \subseteq y$	$\begin{bmatrix} x_2 \end{bmatrix} \leq y$
$\llbracket x_2 \rrbracket \subseteq \llbracket y \rrbracket$	$\ x_2\ \leq \ y\ $
$\llbracket x_2 \rrbracket \subseteq \llbracket x_1 x_2 \rrbracket$	$\llbracket x_2 \rrbracket \leq \llbracket x_1 x_2 \rrbracket$
$\bar{\llbracket}x_2\bar{\rrbracket}\subseteq \bar{\llbracket}E \rrbracket$	$[\![x_2]\!] \leq [\![E]\!]$
$y \subseteq \llbracket y \rrbracket$	$y \leq \llbracket y rbracket$
$y \subseteq \llbracket x_1 x_2 \rrbracket$	$y \le \llbracket x_1 x_2 \rrbracket$
$y \subseteq \llbracket E \rrbracket$	$y \leq \llbracket E \rrbracket$
$\llbracket y \rrbracket \subseteq \llbracket x_1 x_2 \rrbracket$	$\llbracket y \rrbracket \leq \llbracket x_1 x_2 \rrbracket$
$\llbracket y \rrbracket \subseteq \llbracket E \rrbracket$	$\llbracket y \rrbracket \leq \llbracket E \rrbracket$
$[\![x_1x_2]\!] \subseteq [\![E]\!]$	$[x_1x_2] \le [E]$

It can be verified by inspection that Theorem 9 is true for E, that is, $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic. Moreover, $\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$, and $\overline{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}$, where $V, V' \in X_E \cup Y_E$.

As for the previous example, we will now go through the equivalence proof and illustrate each construction in the case of E.

As starting point, we choose the least solution φ of C(E). It looks as follows:

$$\varphi(V) = \begin{cases} \{\lambda x. x_1 x_2\} & \text{if } V = [\![\lambda x. x_1 x_2]\!] \\ \{\lambda y. y\} & \text{otherwise} \end{cases}$$

To get the solution ψ of T(E), we need to construct the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and compose it with φ . The automaton $\mathcal{A}(E,\varphi,S)$ can be illustrated as follows:



As before, notice that we have not pointed to the start state; it is a parameter of the specification. Notice also that we have abbreviated the names of some of the states. Observe that $t_{\mathcal{A}(E,\varphi,S)} = \mu \alpha.\alpha \rightarrow \alpha$, if S is a non-empty subset of $\{\lambda x.x_1x_2, \lambda y.y\}$. We can then obtain the mapping ψ . It is a constant function:

$$\psi(V) = \mu \alpha. \alpha \to \alpha$$

It can be verified by inspection that ψ is a solution of T(E) and $\overline{T}(E)$.

As an aside, note that although $\psi(V)$ is an infinite tree for all V, there are other solutions of T(E) and $\overline{T}(E)$ where all the involved types are finite. For example, consider the solution ψ' where

$$\psi'(\llbracket\lambda x.x_1x_2\rrbracket) = (\top \to \top) \to \top$$

$$\psi'(\llbracket\lambda y.y\rrbracket) = \psi'(x) = \psi'(\llbracket x_1\rrbracket) = \top \to \top$$

$$\psi'(\llbracket x_2\rrbracket) = \psi'(y) = \psi'(\llbracket y\rrbracket) = \psi'(\llbracket x_1x_2\rrbracket)$$

$$= \psi'(\llbracket E\rrbracket) = \top$$

To get the solution $\overline{\varphi}$ of $\overline{C}(E)$, observe that

 $(\psi(V))(\epsilon) = \rightarrow$.

Plugging this into the definition of $\overline{\varphi}$ yields that $\overline{\varphi}$ is a constant function:

 $\overline{\varphi}(V) = \mathsf{Abs}(E)$

Notice that $\varphi \neq \overline{\varphi}$. It can be verified by inspection that $\overline{\varphi}$ is a solution of $\overline{C}(E)$.

Finally, to construct φ' where

 $\varphi' = \lambda V.\{ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E) \}$

notice that the constraints in $\overline{C}(E)$ that have the form $\{k\} \subseteq V$ where $V \in X_E \cup Y_E$ are:

$$\{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket \lambda y.y \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket x_2 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket x_2 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket y \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket y \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket x_1x_2 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket x_1x_2 \rrbracket$$

$$\{\lambda y.y\} \subseteq \llbracket E \rrbracket$$

So, $\varphi = \varphi'$, and hence φ' is the least solution of C(E).

6 Extensions

Various extensions to the type system and flow analysis have equivalent typability and safety problems. We now show an example of such an extension: the conditional construct if0. For simplicity, we consider if0 rather than a more usual if, to avoid introducing booleans. Thus, the set of types T_{Σ} and the abstract domain Cl(E) for the safety analysis remain the same.

The syntax for the extension of our example language is:

 $E ::= \dots \mid ifo \ E_1 E_2 E_3$

The intension is that if E_1 evaluates to 0, then E_2 is evaluated; if E_1 evaluates to a non-zero integer, then E_3 is evaluated; and if E_1 evaluates to a non-integer, then an error occurs. We need one new type rule:

$$\frac{A \vdash E : \operatorname{Int} \quad A \vdash E_2 : t \quad A \vdash E_3 : t}{A \vdash \operatorname{if0} E_1 E_2 E_3 : t} \tag{7}$$

As before, we can rephrase the type inference problem in terms of solving a system of type constraints. We need three new type constraints:

• for every occurrence of a subterm of the form if $0 E_1 E_2 E_3$, we generate the three inequalities

$$\begin{split} \llbracket E_1 \rrbracket &\leq & \mathsf{Int} \\ \llbracket E_2 \rrbracket &\leq & \llbracket \mathsf{if0} \ E_1 E_2 E_3 \rrbracket \\ \llbracket E_3 \rrbracket &\leq & \llbracket \mathsf{if0} \ E_1 E_2 E_3 \rrbracket \ . \end{split}$$

It is straightforward to check that Theorem 5 remains true, that is, the solutions of the constraint system correspond to the possible type annotations.

We need three new safety constraints:

• for every occurrence of a subterm of the form if $0 E_1 E_2 E_3$, we generate the three constraints

$$\begin{split} \llbracket E_1 \rrbracket &\subseteq & \{\mathsf{Int}\} \\ \llbracket E_2 \rrbracket &\subseteq & \llbracket \mathsf{if0} \ E_1 E_2 E_3 \rrbracket \\ \llbracket E_3 \rrbracket &\subseteq & \llbracket \mathsf{if0} \ E_1 E_2 E_3 \rrbracket \ . \end{split}$$

It is straightforward to check that Theorem 9 and Theorem 12 remain true, that is, $\overline{C}(E)$ are $\overline{T}(E)$ are isomorphic, and the type system and the safety analysis accept the same programs. Moreover, the safety analysis analysis and type inference algorithms remain the same.

The addition of polymorphic let, products, sums and atomic subtypes with coercions should also be straightforward. Dynamic or soft typing systems are also candidates for formulating equivalent type and flow analysis systems.

A different challenge is to formulate a type system equivalent to the safety analysis for object-oriented languages presented in [14]. Yet another challenge is to find two equivalent binding-time analyses, one based on type systems and one based on flow analysis. Results in this direction were presented in [13].

7 Conclusion

We have described a type system and a flow analysis and proved that the corresponding typability and safety problems are equivalent. We also obtained a cubic time algorithm for typability. This problem has been open since the type system was first presented by Amadio and Cardelli in 1991 [1].

For a given program, the system of type constraints and the system of flow constraints are radically different. For the particular language studied in this paper, however, we demonstrated that the deductive closures of those systems are isomorphic. This property does not seem to be either a necessary or a sufficient condition for the equivalence result, but in itself it suggests a close relationship between the systems.

Tang and Jouvelot [19] has demonstrated that type analysis and flow analysis can be combined in a single framework. The type system part of their approach is that of simple types. A challenge is to extend their framework such that the type system part is that of this paper. Wand and Steckler [22] presented a framework for proving correctness of flow-based compiler optimizations. A challenge is to investigate if their framework can be simplified when the flow analysis is replaced by for example the framework of Tang and Jouvelot [19] or possibly an extended one.

An example of an area in which a relationship between a typing problem and a flow problem may be helpful is debugging and explaining type inferencing results for end users. A flow analysis point of view might provide a concrete illustration of why the type inferencer produced a particular type assignment or typing error [20].

The two systems considered in this paper use inequalities, that is, subtyping in the type system and set inclusion in the flow analysis. One might consider changing the inequalities to equalities and look for an equivalence result similar to the one on this paper. On the type system side, this would result in a simply-typed lambda-calculus with a type inference algorithm based on unification. On the flow analysis side, it would result in an analysis resembling the one of Bondorf and Jørgensen [4]. Current work addresses obtaining an equivalence between two such systems [6].

In conclusion, we find that a type system and a flow analysis can in some cases be equivalent ways of looking at the same problem.

Acknowledgements

We thank Mitchell Wand for encouragement and helpful discussions. We also thank Torben Amtoft and the anonymous referees for helpful comments on a draft of the paper. The results of this paper were obtained while the first author was at Northeastern University, Boston.

References

- Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. ACM Transactions on Programming Languages and Systems, 15(4):575–631, 1993. Also in Proc. POPL'91.
- [2] Torben Amtoft. Minimal thunkification. In Proc. WSA'93, pages 218–229, 1993.
- [3] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.
- [4] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. Journal of Functional Programming, 3(3):315–346, 1993.
- [5] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Proc. PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 145–154, 1993.
- [6] Nevin Heintze. Personal communication. 1994.

- [7] Nevin Heintze. Set-based analysis of ML programs. In Proc. ACM Conference on LISP and Functional Programming, pages 306–317, 1994.
- [8] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. Journal of Computer and System Sciences, 49(2):306–324, 1994. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.
- [9] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. Mathematical Structures in Computer Science, 1995. To appear. Also in Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.
- [10] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In Proc. Conference on Functional Programming Languages and Computer Architecture, pages 260–272, 1989.
- [11] Jens Palsberg. Closure analysis in constraint form. ACM Transactions on Programming Languages and Systems, 1995. To appear. Also in Proc. CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (LNCS 787), pages 276–290, Edinburgh, Scotland, April 1994.
- [12] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.
- [13] Jens Palsberg and Michael I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. In Proc. ICCL'94, Fifth IEEE International Conference on Computer Languages, pages 289–298, Toulouse, France, May 1994.
- [14] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Systems. John Wiley & Sons, 1994.
- [15] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. Information and Computation, 118(1):128–141, 1995.
- [16] Peter Sestoft. Analysis and Efficient Implementation of Functional Programs. PhD thesis, DIKU, University of Copenhagen, October 1991.
- [17] Olin Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, CMU, May 1991. CMU-CS-91-145.
- [18] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, Topics in Advanced Language Implementation, pages 47–87. MIT Press, 1991.
- [19] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In Proc. TACS'94, Theoretical Aspects of Computing Sofware, pages 224–243. Springer-Verlag (LNCS 789), 1994.

- [20] Mitchell Wand. Finding the source of type errors. In *Thirteenth Symposium on Principles* of Programming Languages, pages 38–43, 1986.
- [21] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Infor*mation and Computation, 93(1):1–15, 1991.
- [22] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In Proc. POPL'94, 21st Annual Symposium on Principles of Programming Languages, pages 434– 445, 1994.

Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions

Sumit Gulwani¹ and Ashish Tiwari²

¹ Microsoft Research, Redmond, WA 98052, sumitg@microsoft.com ² SRI International, Menlo Park, CA 94025, tiwari@csl.sri.com

Abstract. This paper presents results on the problem of checking equality assertions in programs whose expressions have been abstracted using combination of linear arithmetic and uninterpreted functions, and whose conditionals are treated as non-deterministic.

We first show that the problem of assertion checking for this combined abstraction is coNP-hard, even for loop-free programs. This result is quite surprising since assertion checking for the individual abstractions of linear arithmetic and uninterpreted functions can be performed efficiently in polynomial time.

Next, we give an assertion checking algorithm for this combined abstraction, thereby proving decidability of this problem despite the underlying lattice having infinite height. Our algorithm is based on an important connection between unification theory and program analysis. Specifically, we show that weakest preconditions can be strengthened by replacing equalities by their unifiers, without losing any precision, during backward analysis of programs.

1 Introduction

We use the term *equality assertion* or simply *assertion* to refer to an equality between two program expressions. By *assertion checking*, we mean checking whether a given assertion is an invariant at a given program point.

Reasoning about assertions in programs is an undecidable problem. Hence, assertion checking is typically performed over some (sound) abstraction of the program. This may give rise to false positives, i.e., some assertions that are true in the original program may not be true in the abstract version. There is an efficiency-precision trade-off in the choice of the abstraction. A more precise abstraction leads to fewer false positives but is also harder to reason about.

Linear arithmetic and uninterpreted functions³ are two most commonly used expression languages for creating program abstractions. There are several papers

³ An uninterpreted function F of arity n satisfies only one axiom: If $e_i = e'_i$ for $1 \le i \le n$, then $F(e_1, \ldots, e_n) = F(e'_1, \ldots, e'_n)$. Uninterpreted functions are commonly used to abstract programming language operators that are otherwise hard to reason about. They are also used to abstract procedure calls.



Fig. 1. This program illustrates the difference between precision of performing analysis over the abstractions of linear arithmetic (which can verify only the first assertion), uninterpreted functions (which can verify only the second assertion), and their combination (which can verify all assertions). F denotes some function without any side-effects and can be modeled as an uninterpreted function for purpose of proving the assertions.

that describe how to do assertion checking for each of these abstractions. (Section 6 on related work describes some of this work.) The combined expression language of linear arithmetic and uninterpreted functions yields a more precise abstraction than the ones obtained from either of these two expression languages. For example, consider the program shown in Figure 1. Note that all assertions at the end of the program are true. If this program is analyzed over the abstraction of linear arithmetic (using, for example, the abstract interpreter described in [14] or [6]), then only the first assertion can be validated. This is because discovering the relationship between b_1 and b_2 , and between c_1 and c_2 , involves reasoning about uninterpreted functions. Similarly, if this program is analyzed over the abstraction of uninterpreted functions (using, for example, the abstract interpreter described in [10]), then only the second assertion can be validated. However, an analysis over the combined abstraction of linear arithmetic and uninterpreted functions can verify all assertions.

Even though there has been a lot of work for reasoning about the abstractions of linear arithmetic and that of uninterpreted functions, the problem of assertion checking over the combined abstraction of linear arithmetic and uninterpreted functions has not been considered before. In this paper, we consider the problem of checking equality assertions in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions. We also abstract all program conditionals as non-deterministic because otherwise the problem is easily shown to be undecidable even for the individual abstractions of linear arithmetic [17] and uninterpreted functions [16]. (An analysis that performs an imprecise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions but takes conditional guards into account would also be useful in practice, and can be used, for example, for array bounds checking. The related work section mentions our recent work on combining abstract interpreters, which can be used to construct such an analysis.) The abstracted program model is formally described in Section 2.

In Section 3, we show that the problem of assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions is coNP-hard. This is true even for loop-free programs, in which case it is coNP-complete. This result is quite surprising because assertion checking in the individual abstractions of linear arithmetic and uninterpreted functions entails polynomial-time algorithms (even for programs with loops). Karr's algorithm [14, 17] can be used to perform assertion checking when program expressions have been abstracted using linear arithmetic operators. Gulwani and Necula's algorithm [9, 10] performs assertion checking in programs whose expressions have been abstracted using uninterpreted functions. Both these algorithms run in polynomial time. However, our coNP-hardness result shows that there is no way to combine these algorithms to do assertion checking for the combined abstraction in polynomial time (unless P=coNP). A similar combination problem has been studied extensively in the context of decision procedures. Nelson and Oppen have given a famous combination result for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. The theories of linear arithmetic and uninterpreted functions are disjoint, convex, and quantifier-free and have polynomial time decision procedures. Hence, the Nelson-Oppen combination methodology can be used to construct a polynomial-time decision procedure for the combination of these theories. In this paper, we show that, unfortunately, there is no polynomial-time combination scheme for assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions (unless P = coNP).

In Section 4, we give an assertion checking algorithm for the combined abstraction (of linear arithmetic and uninterpreted functions) thereby showing that this problem is decidable. This result is again surprising because the underlying abstract lattice has infinite height, which implies that a standard abstract interpretation [6] based algorithm cannot terminate in a finite number of steps. However, our algorithm leverages the fact that our goal is not to discover all equality invariants, but to check whether a given assertion is an invariant. A central component of our algorithm is a general result that allows replacement of equalities generated in weakest precondition computation by their unifiers (Lemma 2). For theories that admit a singleton or finite complete set of unifiers, respectively called unitary and finitary theories, this replacement can be effectively done. The significance of this connection between assertion checking and unification is discussed further in Section 5. We make the paper self-contained by presenting (in Section 4.1) a novel unification algorithm for the combined theory of linear arithmetic and uninterpreted functions, which is used in our assertion checking algorithm.



Fig. 2. Flowchart nodes in our abstracted program model.

2 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 2. In the assignment node, x refers to a program variable while e denotes some expression in the underlying abstraction. We refer to the language of such expressions as *expression language of the program*. The expression languages for the abstractions of linear arithmetic, uninterpreted functions and their combination are as follows:

- Linear arithmetic:

$$e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e$$

Here y denotes some variable while c denotes some arithmetic constant. - Uninterpreted functions:

$$e ::= y \mid F^n(e_1, \dots, e_n)$$

Here F^n denotes some uninterpreted function of arity n. We allow n to be zero (for representing nullary uninterpreted functions).

- Combination of linear arithmetic and uninterpreted functions:

 $e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e \mid F^n(e_1, \dots, e_n)$

A non-deterministic assignment x :=? denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely.

Non-deterministic conditionals, represented by *, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of guarded conditionals, which our abstraction cannot handle precisely. We abstract away the guards in conditionals because otherwise the problem of assertion checking (when the expression language of the program involves combination of linear arithmetic and uninterpreted functions) can be easily shown undecidable from either of the following two results. Müller-Olm and Seidl have shown that the problem of assertion checking in programs that use guarded conditionals and linear arithmetic expressions is undecidable [17]. Müller-Olm, Rüthing, and Seidl have also proved a similar undecidability result when the expression language involves uninterpreted functions [16].

A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to multiple join nodes each with two incoming edges.

3 coNP-hardness of Assertion Checking

In this section, we show that the problem of assertion checking when the expression language of the program involves combination of linear arithmetic and uninterpreted functions (and the flowchart representation of the program consists of nodes shown in Figure 2) is coNP-hard.

The key observation in proving this result is that a disjunctive assertion of the form $g = a \lor g = b$ can be encoded as the non-disjunctive assertion F(a) + F(b) = F(g) + F(a + b - g). The procedure Check(g,m) generalizes this encoding for the disjunctive assertion $g = 0 \lor ... \lor g = m - 1$ (which has m - 1disjuncts), as stated in Lemma 1. Once such a disjunction can be encoded, we can reduce the unsatisfiability problem to the problem of assertion checking as follows.

Consider the program shown in Figure 3. We will show that the assert statement in the program is true iff the input boolean formula ψ is unsatisfiable. Note that, for a fixed ψ , the procedures IsUnSatisfiable and Check can be reduced to one procedure whose flowchart representation consists of only the nodes shown in Figure 2. (These procedures use procedure calls and loops with guarded conditionals only for expository purposes.) This can be done by unrolling the loops and inlining procedure Check inside procedure IsUnSatisfiable. The size of the resulting procedure is polynomial in the size of the input boolean formula ψ .

The procedure IsUnSatisfiable contains k non-deterministic conditionals, which together choose a truth value assignment for the k boolean variables in the input boolean formula ψ , and accordingly set its clauses to true (1) or false (0). The boolean formula ψ is unsatisfiable iff at least one of its clauses remains unsatisfied in every truth value assignment to its variables, or equivalently, $g \in$ $\{0, \ldots, m-1\}$ in all executions of the procedure IsUnSatisfiable. The procedure Check(g, m) performs the desired check as stated in the following lemma.

Lemma 1. The assert statement in Check(g, m) is true iff $g \in \{0, ..., m-1\}$.

Proof. The following properties hold for all $0 \le i \le m - 1$.

- E1. If $0 \le j \le i$, then $h_{i,j} = h_{i,0}$.
- E2. If $g \in \{0, ..., m-1\}$, then $h_i = h_{i,g}$.
- E3. If $g \notin \{0, ..., m-1\}$, then h_i cannot be expressed as a linear combination of $h_{i,0}, ..., h_{i,m-1}$.

 $\texttt{IsUnSatisfiable}(\psi)$ % Suppose formula ψ has k variables x_1, \ldots, x_k % and m clauses numbered 1 to m. % Let variable x_i occur in positive form in clauses # $A_i[0], \ldots, A_i[c_i]$ % and in negative form in clauses # $B_i[0], \ldots, B_i[d_i]$. for i = 1 to $m \ \mathrm{do}$ $e_i := 0$; % e_i represents whether clause i is satisfiable or not. for i = 1 to $k\ \mathrm{do}$ if (*) then % set x_i to true for j = 0 to c_i do $e_{A_i[j]} := 1;$ else % set x_i to false for j = 0 to d_i do $e_{B_i[j]} := 1;$ $g:=e_1+e_2+\ldots+e_m$; % Count how many clauses have been satisfied. Check(q,m);Check(g,m)% This procedure checks whether $g \in \{0, \dots, m-1\}$.

```
 \begin{split} h_0 &:= F(g); \\ \text{for } j = 0 \text{ to } m-1 \text{ do} \\ h_{0,j} &:= F(j); \\ \text{for } i = 1 \text{ to } m-1 \text{ do} \\ s_{i-1} &:= h_{i-1,0} + h_{i-1,i}; \\ h_i &:= F(h_{i-1}) + F(s_{i-1} - h_{i-1}); \\ \text{for } j = 0 \text{ to } m-1 \text{ do} \\ h_{i,j} &:= F(h_{i-1,j}) + F(s_{i-1} - h_{i-1,j}); \\ \text{Assert}(h_{m-1} = h_{m-1,0}); \end{split}
```

Fig. 3. A program that illustrates the coNP-hardness of assertion checking when the expression language uses combination of linear arithmetic and uninterpreted functions.

The above properties can be proved easily by induction on *i*. If $g \in \{0, ..., m-1\}$, then the assert statement is true because:

$$h_{m-1} = h_{m-1,g}$$
 (follows from property E2)
= $h_{m-1,0}$ (follows from property E1)

If $g \notin \{0, ..., m-1\}$, then it follows from property E3 that the assert statement is falsified. \blacksquare

Lemma 1 implies that the assert statement in procedure $\texttt{IsUnSatisfiable}(\psi)$ is true iff the input boolean formula ψ is unsatisfiable. Hence, the following theorem holds.

Theorem 1. Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is coNP-hard.

Since IsUnSatisfiable can be represented as a loop-free program, Theorem 1 holds even for loop-free programs.

4 Algorithm for Assertion Checking

In this section, we give an assertion checking algorithm for our abstracted program model when the expression language of the program involves combination of linear arithmetic and uninterpreted functions. We prove that this algorithm terminates, which establishes the decidability of assertion checking for the combined abstraction. It remains an open problem to establish an upper complexity bound for this algorithm.

For purpose of describing and proving correctness of our algorithm, we first establish some results on unification in the combined theory of linear arithmetic and uninterpreted functions in the next sub-section.

4.1 Unification in the Combined Theory

A substitution σ is a mapping that maps variables to expressions such that for every variable x, the expression $\sigma(x)$ contains variables only from the set $\{y \mid \sigma(y) = y\}$. A substitution mapping σ can be (homomorphically) lifted to expressions such that for every expression e, we define $\sigma(e)$ to be the expression obtained from e by replacing every variable x by its mapping $\sigma(x)$. Often, we denote the application of a substitution σ to an expression e using postfix notation as $e\sigma$. We sometimes treat a substitution mapping σ as the following formula, which is a conjunction of non-trivial equalities between variables and their mappings:

$$\bigwedge_{x:x \neq x\sigma} x = x\sigma$$

A substitution σ is a *unifier* for an equality $e_1 = e_2$ (in theory T) if $e_1\sigma = e_2\sigma$ (in theory T). A substitution σ is a unifier for a set of equalities E if σ is a unifier for each equality in E. A substitution σ_1 is *more-general* than a substitution σ_2 if there exists a substitution σ such that $x\sigma_2 = (x\sigma_1)\sigma$ for all variables x. ⁴ A set C of unifiers for E is *complete* when for any unifier σ for E, there exists a unifier $\sigma' \in C$ that is more-general than σ . Theories can be classified based on whether all equalities in that theory have a complete set of unifiers whose cardinality is at most 1 (unitary theory), or finite (finitary theory), or whether some equality does not have any finite complete set of unifiers (infinitary theory).

In the remaining part of this section, we show that the combined theory of linear arithmetic and uninterpreted functions is finitary. For this purpose, we describe an algorithm that computes a complete set of unifiers for an equality in the combined theory. We describe this algorithm using a set of inference rules (listed in table 1) in the style of [4].

Table 1 describes some inference rules that operate on states. A state (E, σ) is a pair consisting of a set E of equalities (between expressions involving combination of linear arithmetic and uninterpreted functions) and a substitution σ . The Unif0 rule removes trivial equalities from E. The Unif1 rule can be applied after

⁴ The more-general relation is reflexive, i.e., a substitution is more-general than itself.

Unif0:
$$\frac{(E \cup \{e = e\}, \sigma)}{(E, \sigma)}$$

Unif1

if1:
$$\frac{(E \cup \{x = e\}, \sigma)}{(E\sigma', \sigma\sigma')}$$

if x does not occur in e. Here $\sigma' = \{x \mapsto e\}$ and $E\sigma'$ denotes $\{e_1\sigma' = e_2\sigma' \mid (e_1 = e_2) \in E\}$.

Unif2:
$$\frac{(E \cup \{F(e_1, \dots, e_n) = F(e'_1, \dots, e'_n) + e\}, \sigma)}{(E \cup \{e_1 = e'_1, \dots, e_n = e'_n, e = 0\}, \sigma)}$$

Table 1. Inference rules for unification in the combination theory.

selecting some equality from E that can be rewritten in the form x = e such that variable x does not occur in expression e. The Unif2 rule is applied after selecting some equality that can be rewritten in the form $F(e_1, \ldots, e_n) = F(e'_1, \ldots, e'_n) + e$ for some uninterpreted function F and expressions e_i, e'_i and e.

The notation $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}$ denotes the substitution mapping that maps variable x_i to e_i (for $1 \leq i \leq k$) and all other variables to themselves. We use the notation $(E, \sigma) \vdash (E', \sigma')$ to denote that the state (E', σ') is obtained from (E, σ) by applying some inference rule. Similarly, $(E, \sigma) \vdash^* (E', \sigma')$ denotes that the state (E', σ') can be obtained from the state (E, σ) by applying some sequence of inference rules.

To generate a complete set of unifiers C for an equality $e_1 = e_2$, we start with the state ($\{e_1 = e_2\}, I$), where I is the identity mapping, and apply the inference rules repeatedly until no more inference rules can be applied. For all derivations that end with some state of the form (\emptyset, σ) , we put σ in C. Theorem 2 stated below implies that the set C thus obtained is indeed a set of unifiers for the equality $e_1 = e_2$. Theorem 3 implies that this set C of unifiers is complete. The proofs of these theorems are by induction on the length of the derivation and are given in the full version of this paper [13].

Theorem 2 (Soundness). If $(E, I) \vdash^* (\emptyset, \sigma)$, then σ is a unifier for E.

Theorem 3 (Completeness). Suppose σ is a unifier for E. Then there is a derivation $(E, I) \vdash^* (\emptyset, \sigma_0)$ such that σ_0 is a more-general unifier for E than σ .

The following theorem implies that the set C is finite.

Theorem 4 (Finite Complete Set of Unifiers). Every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite number of steps. Consequently, E has a finite complete set of unifiers.

The proof of Theorem 4 is given in the full version of this paper [13]. The key proof idea is to show that every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite number of steps and then use Konig's lemma to bound the total number of derivations.

We next illustrate the application of the inference system.

Example 1. Consider the following derivation of a unifier for the equality Fx + Fy = Fa + Fb.

$(\{Fx + Fy = Fa + Fb\}, I)$	
$(\{Fx = Fb, y = a\}, I)$	Unif2
$(\{x = b, y = a\}, I)$	Unif2
$(x = b, \{y \mapsto a\})$	Unif1
$(\emptyset, \{x \mapsto b, y \mapsto a\})$	Unif1

Thus $\{x \mapsto b, y \mapsto a\}$ is a unifier for Fx + Fy = Fa + Fb. Note that the alternate choice for the first Unif2 application yields another unifier $\{x \mapsto a, y \mapsto b\}$ for the given equality. No other unifier can be generated by applying the inference rules. Hence, these two unifiers constitute a complete set of unifiers for the given equality.

Example 2. As another example, consider generating a complete set of unifiers for the equality x + Fx + Fy = a + Fa + F(a + 1). Since each variable occurs below an uninterpreted symbol, only the Unif2 rule is applicable. There are four choices, either x = a, or x = a + 1, or y = a, or y = a + 1. We show a derivation for the second choice below.

$$\begin{array}{ll} (\{x+Fx+Fy=a+Fa+F(a+1)\},I) \\ (\{x+Fy=a+Fa,x=a+1\},I) & {\rm Unif2} \\ (\{a+Fa-Fy=a+1\},\{x\mapsto a+Fa-Fy\}) & {\rm Unif1} \\ (\{a=a+1,a=y\},\{x\mapsto a+Fa-Fy\}) & {\rm Unif2} \\ (\{0=1\},\{x\mapsto a+Fa-Fy,y\mapsto a\}) & {\rm Unif1} \end{array}$$

The above derivation is now stuck with no inference rule being applicable. Note that only the first choice x = a and the fourth choice y = a + 1 successfully generate a unifier, which in both cases is $\{x \mapsto a, y \mapsto a + 1\}$. This unifier yields a singleton complete set of unifiers for the given equality.

4.2 Algorithm

Our algorithm for assertion checking over the combined abstraction is based on weakest precondition computation. It represents invariants at each program point by a formula that is a disjunction of substitution mappings. We show that any program invariant in our abstracted program model can be represented using such formulas (Lemma 2).

Suppose the goal is to check whether an assertion $e_1 = e_2$ is an invariant at program point π . The algorithm performs a backward analysis of the program computing a formula ψ (which is a disjunction of substitution mappings) at each program point such that ψ must hold for the assertion $e_1 = e_2$ to be true at program point π . This formula is computed at each program point from the formulas at the successor program points in an iterative manner. The algorithm uses the transfer functions described below to compute these formulas across the flowchart nodes shown in Figure 2. The algorithm declares $e_1 = e_2$ to be an invariant at π if the formula computed at the beginning of the program after fixed-point computation is a tautology in the combined theory of linear arithmetic and uninterpreted functions.

In the following transfer functions, we use the notation Unif(E), where E is some conjunction of equalities E, to denote the formula that is a disjunction of all unifiers in some complete set of unifiers for E. (If E is unsatisfiable, then E does not have any unifier and Unif(E) is simply *false*.) The formula Unif(E)can be computed by using the algorithm described in Section 4.1.

Initialization: The formula at all program points except π is initialized to be the trivial formula *true*. The formula at program point π is initialized to be Unif $(e_1 = e_2)$.

Assignment Node: See Figure 2 (a).

The formula ψ' before an assignment node x := e is obtained from the formula ψ after the assignment node by substituting x by e in ψ , and invoking Unif on each resulting disjunct.

$$\psi' = \bigvee_i \operatorname{Unif}(\psi^i[e/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Assignment Node: See Figure 2 (b).

The formula ψ' before a non-deterministic assignment node x :=? is obtained from the formula ψ after the non-deterministic assignment node by substituting program variable x by some fresh constant (i.e., a fresh nullary uninterpreted function symbol) α , and invoking Unif on each resulting disjunct.

$$\psi' = \bigvee_i \operatorname{Unif}(\psi^i[\alpha/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Conditional Node: See Figure 2 (c).

The formula ψ before a non-deterministic conditional node is obtained by taking the conjunction of the formulas ψ_1 and ψ_2 on the two branches of the conditional, and invoking Unif on each resulting disjunct.

$$\psi = \bigvee_{i,j} \operatorname{Unif}(\psi_1^i \wedge \psi_2^j), \text{ where } \psi_1 = \bigvee_i \psi_1^i \text{ and } \psi_2 = \bigvee_j \psi_2^j$$

Join Node: See Figure 2 (d).

The formulas ψ_1 and ψ_2 on the two predecessors of a join node are same as the formula ψ after the join node.

$$\psi_1 = \psi$$
 and $\psi_2 = \psi$

Fixed-point Computation: In presence of loops in procedures, the algorithm goes around each loop until the formulas computed at each program point in two successive iterations of a loop are equivalent, or if any formula becomes *false*.

Correctness We now prove that the above algorithm is correct, i.e., an assertion $e_1 = e_2$ holds at program point π iff the algorithm claims so. For this purpose, we first state a useful lemma (Lemma 2) that states an interesting connection between program analysis and unification theory. This lemma is true is general: it is independent of the logical theory and also holds for programs with guarded conditionals. The proof of this lemma is given in the full version of this paper [13].

Lemma 2. An equality $e_1 = e_2$ holds at a program point π iff $\text{Unif}(e_1 = e_2)$ holds at π . In fact, a formula ϕ containing $e_1 = e_2$ holds at a program point π iff $\phi[\text{Unif}(e_1 = e_2)/(e_1 = e_2)]$ holds at π .

Lemma 2 implies that the formula computed by our algorithm before the flowchart is the (real) weakest precondition of the formula after those nodes. Also, note that the algorithm starts with a formula which is an invariant at π iff the given assertion is an invariant at π (follows from Lemma 2). The correctness of the algorithm now follows from the fact that the algorithm starts with the correct assertion at π and iteratively computes the correct weakest precondition at each program point in a backward analysis.

Termination We now prove that the above algorithm terminates in a finite number of steps. It suffices to show that the weakest precondition computation across a loop terminates in a finite number of iterations. This follows from the following lemma.

Lemma 3. Let C be a chain ψ_1, ψ_2, \ldots of formulas that are disjunctions of substitutions. Let $\psi_i = \bigvee_{\ell=1}^{m_i} \psi_i^{\ell}$ for some integer m_i and substitutions ψ_i^{ℓ} . Suppose (a) $\psi_{i+1} = \bigvee_{\ell=1}^{m_i} \bigvee_{j=1}^{n_i} \text{Unif}(\psi_i^{\ell} \wedge \alpha_i^j)$, for some substitutions α_i^j . (b) $\psi_i \neq \psi_{i+1}$. Then, C is finite.

The proof of Lemma 3 is by establishing a well founded ordering on $\psi'_i s$, and is given in the full version of this paper [13]. Lemma 3 implies termination of our assertion checking algorithm. (Note that the weakest preconditions $\psi_1, \psi_2, ...$ generated by our algorithm at any given program point inside a loop in successive iterations satisfy condition (a), and hence $\psi_{i+1} \Rightarrow \psi_i$ for all *i*. Lemma 3 implies that there exists *j* such that $\psi_j \Rightarrow \psi_{j+1}$ and hence $\psi_j \equiv \psi_{j+1}$, at which point the fixed-point computation across that loop terminates.) Hence, the following theorem holds.

Theorem 5. Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is decidable.

The decidability of assertion checking for the combined abstraction is rather surprising given that the abstract lattice over sets of equalities between expressions in the combined theory has an infinite height. This suggests that an abstract interpretation based forward analysis algorithm that operates over this lattice may not terminate across loops (unless widening techniques are employed, which may lead to imprecise analysis). For example, consider the following program.

InfiniteHeightExample() x := 0;while (*) do { x := x + 1 }; Assert($x = 0 \lor \cdots \lor x = m$);

The disjunctive assertion at the end of the program can be encoded using an equality assertion. The procedure Check(x,m) (on page 6) does exactly this. Clearly, the assertion at the end of the program is not true. To invalidate this assertion, the abstract interpreter will have to go around the loop m times. Hence, it will not terminate across loops (because if it did terminate in say tsteps, then it will not be able to invalidate the assertion $x = 0 \lor \cdots \lor x = t$). Our algorithm terminates because it performs a backward analysis (which is good enough for assertion checking) instead of performing a forward analysis (which is required for discovering all valid equalities).

5 Assertion Checking and Unification

The results in this paper point out an interesting connection between assertion checking in programs over a given abstraction and the unification problem for the theory defining that abstraction. Lemma 2 implies that we can replace an assertion by a formula representing a complete set of unifiers for that assertion. This result is quite general and holds for programs with even guarded conditionals and any expression language. This allows for strengthening of weakest preconditions computed using standard transfer functions, by applying Unif() to the result *without* losing any precision. This observation is the basis for the close connection between assertion checking and unification.

The theories of linear arithmetic and uninterpreted functions are unitary. However, equalities in the combined theory of linear arithmetic and uninterpreted functions may not have a complete set of unifiers with a cardinality of at most 1. This disparity appears to be responsible for the coNP-hardness of assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions (as opposed to the fact that the abstractions of linear arithmetic and uninterpreted functions have polynomial-time assertion checking algorithms [14, 10]). The presence of multiple unifiers in a minimal complete set allows for encoding of disjunctions in the combined abstraction. For example, the assertion F(x) + F(3-x) = F(1) + F(2) has two unifiers x = 1 and x = 2 in its minimal complete set of unifiers. This assertion will be true at any program point iff x = 1 or x = 2 on all paths leading to this assertion.

The decidability of assertion checking for the combined abstraction (of linear arithmetic and uninterpreted functions) can be attributed to fact that the combined theory is finitary. Observe that the weakest precondition computation of an assertion, as described in Section 4.2, terminates across a loop because there are only finitely many ways that the assertion can be true.

6 Related Work

We are not aware of any work related to assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions. However, there has been a lot of work on assertion checking and invariant generation over individual abstractions of linear arithmetic and uninterpreted functions.

Program analysis over abstraction of linear arithmetic. Karr described an algorithm to reason about programs using the abstraction of linear equalities. This algorithm performs a forward analysis of the program and computes a set of linear equalities at each program point [14, 17] in an iterative manner. Gulwani and Necula gave a randomized algorithm that performs an equally precise reasoning but more efficiently [8]. Cousot gave a more precise algorithm that reasons about programs using the abstraction of linear inequalities wherein the facts computed at each program point are linear inequality relationships between program variables [7]. Müller-Olm and Seidl have described a modular linear arithmetic analysis to reason about finite-bit machine arithmetic [19]. There has also been some work on extending some of these analyses to an interprocedural setting [18, 11].

Program analysis over abstraction of uninterpreted functions. Kildall's algorithm [15] performs abstract interpretation over the lattice of sets of Herbrand equivalences (i.e., equivalences between expressions involving uninterpreted functions) but it runs in exponential time. Alpern, Wegman, and Zadeck (AWZ) gave a polynomial-time algorithm that reasons about programs treating all operators as uninterpreted functions [1]. The AWZ algorithm is less precise than Kildall's algorithm, but is quite popularly used for global value numbering in compilers. Rüthing, Knoop and Steffen's (RKS) polynomial-time algorithm also reasons about programs using the abstraction of uninterpreted functions. The RKS algorithm is more precise than the AWZ algorithm but remains less precise than Kildall's algorithm. Recently, Gulwani and Necula gave a polynomial-time algorithm that is as precise as Kildall's algorithm with respect to assertion checking in programs using the abstraction of uninterpreted functions [9, 10].

Combination of Abstract Interpreters. We have recently described a general methodology to combine abstract interpreters for two abstractions to construct an abstract interpreter for the combination of those abstractions [12]. This methodology can be used to construct an efficient polynomial-time algorithm that performs analysis over the combined abstraction of linear arithmetic and uninterpreted functions and also takes conditional guards into account. However, this algorithm does not perform the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions. Note that the algorithm that we have described in this paper performs the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions, but it does not take conditional guards into account.

Unification for combination of theories. The unification problem for the combined theory of linear arithmetic and uninterpreted functions is a simple variant of the unification problem for abelian groups with additional uninterpreted functions. This latter problem is usually referred to as the general unification problem for abelian groups [3]. The first algorithm for generating unifiers for the general unification problem for abelian groups was obtained as a corollary of the general result for combining unification algorithms [21] and was later refined [2]. The generic combination unification algorithm involves solving the so-called "unification with constants" and "constant elimination" problems [21], or "unification with linear constant restriction" [2] problem for the individual theories. In this paper, we have presented a different unification algorithm for the combined theory of linear arithmetic and uninterpreted functions. Our presentation of this unification algorithm is using inference rules, which are simple to understand and implement.

Decision procedures for combination of theories. Nelson and Oppen gave a general methodology for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. Shostak gave an efficient variant of this algorithm for the specific case of solvable theories. Clark, Dill and Levitt have described a decision procedure, based on Shostak's method, for combination of linear arithmetic and uninterpreted functions in presence of boolean connectives [5]. It must be mentioned that the problem of assertion checking in programs over a certain abstraction (and in particular for combination of two abstractions) is harder than developing a decision procedure for that abstraction. This is because even though a decision procedure can be used to verify an assertion along a particular program path, a program can potentially have an infinite number of paths. However, if a program is annotated with appropriate invariants at all join points, then a decision procedure can be easily used to verify those invariants as well as assertions across straight-line program fragments.

7 Conclusion

In this paper, we show that assertion checking in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions is coNP-hard (even for loop-free programs). We also give an algorithm for assertion checking for this abstraction, thereby proving decidability of this problem. These results are obtained by closely analyzing the expressiveness of a theory and its effect on the assertion checking problem. First, the ability to encode disjunctions is identified to be an important factor in making assertion checking hard. Second, the classification of a theory as unitary, finitary, or infinitary—based on whether it admits a singleton, finite, or infinite complete set of unifiers has bearing on the hardness and tractability of the assertion checking problem. We show that assertions can be replaced by their unifiers for purpose of checking if they are invariant. We believe that these observations will be significant when other similar or more general abstractions are considered for program analysis.

References

- B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In 15th Annual ACM Symposium on POPL, pages 1–11, 1988.
- F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In 11th International Conference on Automated Deduction, volume 607 of LNAI, pages 50–65, 1992.
- F. Baader and W. Snyder. Unification theory. In Handbook of Automated Reasoning, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
- L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. J. of Automated Reasoning, 31(2):129–168, 2003.
- C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, 1996.
- P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4th Annual ACM Symposium on POPL, pages 234–252, 1977.
- P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In 5th ACM Symposium on POPL, pages 84–96, 1978.
- S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In 30th Annual ACM Symposium on POPL, Jan. 2003.
- S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In 31st Annual ACM Symposium on POPL, Jan. 2004.
- 10. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227, 2004.
- 11. S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In 32nd Annual ACM Symposium on POPL, Jan. 2005.
- 12. S. Gulwani and A. Tiwari. Combining abstract interpreters. *Submitted for publication*, Nov. 2005.
- S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. Technical Report MSR-TR-2006-01, Microsoft Research, Jan. 2006.
- M. Karr. Affine relationships among variables of a program. In Acta Informatica, pages 133–151. Springer, 1976.
- G. A. Kildall. A unified approach to global program optimization. In 1st ACM Symposium on POPL, pages 194–206, Oct. 1973.
- M. Müller-Olm, O. Rüthing, and H. Seidl. Checking herbrand equalities and beyond. In VMCAI, volume 3385 of LNCS, pages 79–96. Springer, Jan. 2005.
- 17. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *31st International Colloquium on Automata, Languages and Programming*, pages 1016–1028, 2004.
- M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In 31st ACM Symposium on POPL, pages 330–341, Jan. 2004.
- M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In European Symposium on Programming, pages 46–60, 2005.
- G. Nelson and D. Oppen. Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems, 1(2):245–257, Oct. 1979.
- M. Schmidt-Schauss. Unification in a combination of arbitrary disjoint equational theories. J. Symbolic Computation, 8(1-2):51–99, 1989.

Undecidability of Context-Sensitive Data-Dependence Analysis

THOMAS REPS University of Wisconsin

A number of program-analysis problems can be tackled by transforming them into certain kinds of graph-reachability problems in labeled directed graphs. The edge labels can be used to filter out paths that are not of interest: A path P from vertex s to vertex t only counts as a "valid connection" between s and t if the word spelled out by P is in a certain language. Often the languages used for such filtering purposes are languages of matching parentheses:

- In some cases, the matched-parenthesis condition is used to filter out paths with mismatched calls and returns. This leads to so-called "context-*sensitive*" program analyses, such as context-*sensitive* interprocedural slicing and context-*sensitive* interprocedural dataflow analysis.
- In other cases, the matched-parenthesis condition is used to capture a graph-theoretic analog of McCarthy's rules: $(cons(x,y)) = x^{2}$ and $(cons(x,y)) = y^{2}$. That is, in the code fragment

c = cons(a,b); d = car(c);

the fact that there is a "structure-transmitted data dependence" from a to d, but not from b to d, is captured in a graph by using (i) a vertex for each variable, (ii) an edge from vertex *i* to vertex *j* when *i* is used on the right-hand side of an assignment to *j*, (iii) parentheses that match as the labels on the edges that run from a to c and c to d, and (iv) parentheses that do not match as the labels on the edges that run from b to c and c to d.

However, structure-transmitted data-dependence analysis is context-*insensitive*, because there are no constraints that filter out paths with mismatched calls and returns. Thus, a natural question is whether these two kinds of uses of parentheses can be combined to create a context-*sensitive* analysis for structure-transmitted data dependences. This paper answers the question in the negative: In general, the problem of context-*sensitive*, structure-transmitted data-dependence analysis is undecidable.

The results of this paper imply that, in general, both context-*sensitive* set-based analysis and ∞ -CFA (when data constructors and selectors are taken into account) are also undecidable.

CR Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*compilers; optimization*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*computability theory*; F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages—*decision problems*; G.2.2 [Discrete Mathematics]: Graph Theory—*path and circuit problems*;

General Terms: Languages, Theory

Additional Key Words and Phrases: Context-sensitive program analysis, dependence analysis, graph-reachability problem, structure-transmitted data dependence, set-based analysis, set constraints, control-flow analysis, ∞-CFA, linear matched-parenthesis language

This work was supported in part by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, by a grant from IBM, and by a Vilas Associate Award from the University of Wisconsin.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

Author's address: Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706. E-mail: reps@cs.wisc.edu.

1. INTRODUCTION

A number of program-analysis problems can be tackled by transforming them into certain kinds of graphreachability problems in labeled directed graphs [20,9,5,4,6,15,19,28,30,16,29,25,32]. It is useful to consider not just ordinary reachability (*e.g.*, transitive closure), but a generalization in which the edge labels are used, in effect, to filter out paths that are not of interest [4,15,28,30,16,29,25,32]: A path *P* from vertex *s* to vertex *t* only counts as a "valid connection" between *s* and *t* if the word spelled out by *P* (*i.e.*, the concatenation, in order, of the labels on the edges of *P*) is in a certain language.

Definition 1.1. Let *L* be a language over alphabet Σ_L , and let *G* be a graph whose edges are labeled with members of Σ_L . Each path in *G* defines a word over Σ_L , namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in *G* is an *L*-path if its word is a member of *L*. An instance of the (single-source/single-target) *L*-path problem asks whether there exists an *L*-path in *G* from a given source vertex *s* to a given target vertex *t*.

Let \mathcal{L} be a family of languages, and $L \in \mathcal{L}$ be a language over alphabet Σ_L . An instance of the \mathcal{L} -reachability problem is an L-path problem instance $\langle L, \Sigma_L, G, s, t \rangle$. \Box

Example. When the family of languages \mathcal{L} in Defn. 1.1 is the context-free languages, we have the *CFL*-*reachability* problem [38]. Consider the graph and the context-free grammar shown below. Note that L(matched) is the context-free language that consists of strings of matched parentheses and square brackets, with zero or more *e*'s interspersed.¹



In this graph, there is exactly one L(matched)-path from s to t: The path goes exactly once around the cycle, and generates the word "[(e[])eee[e]]". \Box

A number of program-analysis problems can be viewed as instances of the CFL-reachability problem [32]. In program-analysis problems, the languages used for such filtering purposes are often languages of matching parentheses. In some cases, the matched-parenthesis condition is used to filter out paths with mismatched calls and returns in order to implement so-called "context-*sensitive*" program analyses.

Example 1.2. The use of CFL-reachability in context-*sensitive* program analysis, as opposed to ordinary graph reachability, is illustrated by the following example:²

¹In this paper, the word "parentheses" is used in both the generic sense—to mean any kind of matching delimiter (*e.g.*, round parentheses, square brackets, curly braces, angle brackets, *etc.*)—as well as in the specific sense of round parentheses. It should always be clear from the context which of these two meanings is intended.

²In this example, we use C syntax. In later examples, we use C augmented with the operator cons, which denotes a pairing constructor; the operators car and cdr, which select the first and second components of a pair, respectively; and the operator atom, which constructs an atomic object (different from NULL) from a given string. This notation is used to simplify the way storage-allocation operations are expressed in our examples. The use of these operators does *not* imply that our results apply only to the analysis of LISP programs.



The diagram on the right shows the program's data-dependence graph. (Strictly speaking, neither the two large ovoid shapes nor the rectangular boxes labeled Call, Enter, and Exit are part of the data-dependence graph. The two ovoids indicate which elements belong to which procedure; the rectangular boxes provide some context about the control points in the program to which the various vertices are associated.) Each directed edge in the graph represents a data dependence (also known as a flow dependence [21,22]): An edge from vertex v_1 to vertex v_2 indicates that the value produced at v_1 may be used at vertex v_2 . For instance, the edge

0 **→** x=0

in the dependence graph for procedure f indicates that the value of x after the execution of the statement x = 0 could be (and, in this case, must be) 0.

In the above program, procedure g has no parameters. However, our data-dependence graphs reflect a somewhat nonstandard treatment of global variables: A global variable such as x is treated as if it were a "hidden" value-result parameter whose value (and subsequent return value) is passed from one scope to another via the special scope-transfer variables x_{in} and x_{out} . For example, the interprocedural data-dependence edge

$$x_{in=x} \xrightarrow{\{_1 \ x=x_{in}\}} x=x_{in}$$

represents the passing of x from f's scope to g's scope at the first call on g. Note that data-dependence edges for dependences transmitted from the caller to the callee (*i.e.*, from f to g) are labeled by the symbols " $\{_1$ " and " $\{_2$ ", whereas data-dependence edges for dependences transmitted from the callee back to the caller are labeled by the symbols " $\}_1$ " and " $\}_2$ ". In particular, the data-dependence edges that represent how x and y are passed from f's scope to g's scope at the first call on g are labeled with $\{_1$; the data-dependence edges that represent how x and y are passed from f's scope to g's scope to g's scope at the second call on g are labeled with $\{_2$. Likewise, the data-dependence edges that represent how x and y are passed back from g's scope to f's scope after the two calls finish are labeled with $\}_1$ and $\}_2$, respectively.

Our data-dependence graphs also have vertices that correspond to various annotations in the program; annotations, denoted here as comments, indicate that we are interested in a given variable at a given point in the program. In the example above, the comments at p1 and p2 give rise to the vertices p1:y and p2:y.

A context-*insensitive* analysis that tracks dependences between constants and variables in the program will report that y depends on 0 and 1 at both p1 and p2. The reason is that a context-*insensitive* analysis *ignores* the fact that the only paths that can possibly be feasible execution paths are those in which returns are matched with corresponding calls. For instance, the existence of the (mismatched) path

$$0 \longrightarrow x=0 \longrightarrow x_in=x \xrightarrow{\{_1 \ x=x_in \longrightarrow y=x} y=x$$

$$\longrightarrow y_out=y \xrightarrow{\}_2} y=y_out \longrightarrow p2:y \qquad (1.3)$$

in the graph shown above serves as "evidence" that p2:y depends on 0 (*i.e.*, that the value of y at p2 could be 0).

In contrast, a context-*sensitive* analysis only reports possible transmissions of values along paths in which returns are matched with corresponding calls. For this example, it reports that p1:y depends on 0 but not on 1 and that p2:y depends on 1 but not on 0 (*i.e.*, y could have only the value 0 at p1 and 1 at p2). The context-*sensitive* analysis can be expressed as a CFL-reachability problem with respect to a language of matched indexed curly braces. It would say that path (1.3) is not a valid connection between 0 and p2:y, because the label $\begin{cases} 1 & \text{on the interprocedural data-dependence edge } x_in=x \xrightarrow{l_1} x=x_in \\ \text{(which represents the transfer of x's value from f to g as g is entered at the first call site) does not match the label <math>\begin{cases} 2 & \text{out}=y \\ y=y_out \\ y=$

$$1 \longrightarrow x=1 \longrightarrow x_{in=x} \xrightarrow{l_2} x=x_{in} \longrightarrow y=x$$
$$\longrightarrow y_{out=y} \xrightarrow{l_2} y=y_{out} \longrightarrow p2:y \qquad (1.4)$$

does count as a valid connection between 1 and p2:y, and this path serves as evidence that the value of y at p2 could be 1. \Box

Problems in which CFL-reachability has been used to devise context-*sensitive* program analyses include interprocedural slicing [15,28] and interprocedural dataflow analysis [30,16].³

CFL-reachability—and, in particular, a matched-parenthesis constraint—has also been used to capture a graph-theoretic analog [29,25] of McCarthy's rules [24] (*i.e.*, "car(cons(x,y))=x" and "cdr(cons(x,y))=y"), as illustrated in the following example.

Example 1.5. The following program illustrates the use of CFL-reachability in (context-*insensitive*) structure-transmitted data-dependence analysis:

³There is an unfortunate clash in terminology that the reader should be aware of. The term "context-sensitive analysis" is standard in the programming-languages community, where it means a static-analysis method in which the analysis of a called procedure is "sensitive" to the context in which it is called: A context-sensitive analysis captures the fact that different call sites that call the same procedure may have different effects on a program's possible execution states. Context-sensitive analysis should not be confused with the "context-sensitive languages" of formal-language theory. Unfortunately, the principle that context-*free*-language reachability is useful in formalizing approaches to context-*sensitive* analysis was fully articulated [32] only after the term "context-sensitive analysis" had been adopted by the programming-languages community [7,37].



In this graph, the labels on the data-dependence edges serve a different purpose than the labels used in Example 1.2: Here an edge labeled "(" corresponds to a data construction in which the value is placed in the first position of a cons; an edge labeled "[" corresponds to a data construction in which the value is placed in the second position of a cons; an edge labeled ")" corresponds to a selection via car; an edge labeled "]" corresponds to a selection via car; an edge labeled "]" corresponds to a selection via car.

The matched-parenthesis path

NULL
$$\xrightarrow{(} x=cons(NULL, NULL) \xrightarrow{(} x=cons(x, NULL))$$

 $\xrightarrow{)} y=car(x) \xrightarrow{)} y=car(y) \xrightarrow{} p3:y$

from NULL to p3:y serves as evidence that the value of y at p3 could be NULL. The path

NULL
$$\xrightarrow{(}$$
 x=cons(NULL,NULL) $\xrightarrow{(}$ x=cons(x,NULL) $\xrightarrow{)}$ y=car(x) \longrightarrow p2:y

does not serve as evidence that the value of y at p2 could be NULL, because the first occurrence of "(" has no matching ")". In fact, there is *no* matched-parenthesis path from NULL to p2:y, and a (context-*insensi-tive*) structure-transmitted data-dependence analysis would conclude that the value of y at p2 cannot be NULL. \Box

The structure-transmitted data-dependence analyses given in [29] and [25] are context-*insensitive*, because there are no constraints that filter out paths with mismatched calls and returns. Thus, a natural question is whether the two kinds of uses of matching delimiters illustrated in Examples 1.2 and 1.5 can be combined to create a context-*sensitive* analysis for structure-transmitted data dependences. The following interprocedural variation on Example 1.5 illustrates what we would hope to gain from a context-*sensitive*, structure-transmitted data-dependence analysis:

Example 1.6. Consider the following example program:

```
List *x, *y;
void g() {
    y = car(x);
}
void f() {
    x = cons(NULL,NULL);
    g();
p1: /* Could y be NULL here? */
    x = cons(x,NULL);
    g();
p2: /* Could y be NULL here? */
    x = y;
    g();
p3: /* Could y be NULL here? */
}
```

1

The relevant portions of this program's data-dependence graph are shown below:



For this example, a context-*sensitive* analysis should report that variable y can have the value NULL at p1 and p3, but not at p2. The following path from NULL to program point p3 serves as evidence that the value of y at p3 could be NULL:

In contrast, a context-*insensitive* analysis would also use the following path from NULL to program point p2 as evidence that the value of y at p2 could be NULL:

The problem with this path is that there is a mismatch between the labels on the edge $x_{in=x} \xrightarrow{\{3\}} x=x_{in}$ and the subsequent edge $y_{out=y} \xrightarrow{\}_2} y=y_{out}$. Consequently, this path would be excluded from consideration by a context-*sensitive* analysis. \Box

The other fact to note about Example 1.6 is that in path (1.7), although "(" symbols match with ")" symbols, and " $\{i$ " symbols match with " $\}_i$ " symbols, the two patterns of matched symbols are *interleaved*.⁴ This observation serves to motivate the study of interleaved matched-parenthesis languages carried out in Sections 2 and 3.

This paper shows that it is impossible to create an algorithm that captures all, and only, interleaved matched-parenthesis paths of the kind illustrated in Example 1.6: In general, the problem of context-*sensi-tive*, structure-transmitted data-dependence analysis is undecidable. In other words, you can capture either (i) the matching of calls and returns, or (ii) "car(cons(x,y)) =x cancellation", but not both simultaneously, in any amount of time. (Of course, there may be useful algorithms that compute approximate, but safe, solutions to this problem, *cf.* [13].)

In terms of the programming-language features needed for this result to apply, higher-order functions are not required: The main result of this paper implies that context-*sensitive*, structure-transmitted data-dependence analysis is undecidable for *first-order* languages (both functional and imperative). This result applies to such languages as C, C++, Java, ML, and Scheme, as well as to many others.

It should be noted that questions of the kind posed in Example 1.6 (*i.e.*, "Does a given variable have a given value at a particular point in a program?") often turn out to be undecidable in their most general form, and often there are several independent reasons why the problem is undecidable (*e.g.*, it is undecidable whether a given statement is ever executed; it is undecidable whether a given path is ever executed; it is undecidable whether a given path is ever executed; *etc.*). This paper shows that context-*sensitive*, structure-transmitted data-dependence analysis is undecidable even if a conservative approximation is made that, for many other program-analysis problems, overcomes other sources of undecidability. (In particular, we assume that all paths in a procedure's control-flow graph are executable.)

The remainder of the paper is organized into four sections: The undecidability result is shown by a reduction from a variant of Post's Correspondence Problem (PCP); Section 2 defines PCP and discusses a variant of it that is particularly suited to our needs. Example 1.6 motivates our interest in languages with interleaved patterns of matching delimiters; Section 3 shows that a certain set of *L*-path problem instances where *L* is a language of strings formed by interleaving two languages of matching parentheses—is unde-

⁴In this paper, the term "interleaved" is used in a somewhat restricted sense, compared to the standard usage in formal-language theory (*cf.* [14, pp. 282]). The exact nature in which patterns of matched symbols are allowed to be woven together is defined precisely in Sections 3 and 4.

cidable. Section 4 relates the result from Section 3 to the undecidability of context-*sensitive*, structuretransmitted data-dependence analysis. Section 5 discusses what our results imply about other programanalysis problems.

2. A VARIANT OF POST'S CORRESPONDENCE PROBLEM

Our undecidability result is shown by a reduction from a variant of Post's Correspondence Problem:

Definition 2.1. An instance of Post's Correspondence Problem, or PCP, consists of two lists of strings, X and Y, where X and Y each consist of k strings in $\{0, 1\}^+$:

$$X = x_1, x_2, \cdots, x_k$$

$$Y = y_1, y_2, \cdots, y_k$$

The instance of PCP has a solution if there exists a nonempty sequence of integers $i_1, i_2, \dots, i_j, \dots, i_m$ such that (i) for all $1 \le j \le m$, we have $1 \le i_j \le k$, and (ii) $x_{i_1} x_{i_2} \cdots x_{i_j} \cdots x_{i_m} = y_{i_1} y_{i_2} \cdots y_{i_j} \cdots y_{i_m}$. \Box

Example 2.2. Consider the following instance of PCP, where *k* is 3:

X = 0101, 101, 111Y = 01, 011, 0111101

This instance of PCP has the solution 1, 2, 3, 1 because

$$x_1 x_2 x_3 x_1 = 0101 \ 101 \ 111 \ 0101 = 01 \ 011 \ 0111101 \ 01 = y_1 y_2 y_3 y_1.$$

PCP is known to be undecidable; for proofs, see Hopcroft and Ullman [14, pp. 193-198], Lewis and Papadimitriou [23, pp. 289-293], or Harrison [8, pp. 249-256].

For our purposes, it is more convenient to work with the following variant of PCP:

Definition 2.3. (Parenthesis-PCP) Given an instance of PCP,

 $X = x_1, x_2, \cdots, x_k$ $Y = y_1, y_2, \cdots, y_k$

we define the corresponding instance of *parenthesis-PCP*, or P-PCP, as

$$X = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k$$

$$\bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_k^R$$

where, for $1 \le i \le k$,

- \bar{x}_i is the string in { (, [}⁺ equal to x_i with 0 replaced by "(" and 1 replaced by "[".
- \bar{y}_i is the string in { },] }⁺ equal to y_i with 0 replaced by ")" and 1 replaced by "]".
- The superscript "*R*" on a string denotes the reversed string.

A solution to an instance of P-PCP is defined with the aid of the following linear context-free grammar:⁵

$$\begin{array}{rrrr} balanced & \longrightarrow & (\ balanced \) \\ & \mid & [\ balanced \] \\ & \mid & (\ \# \) \\ & \mid & [\ \# \] \end{array}$$

An instance of P-PCP has a solution if there exists a nonempty sequence of integers $i_1, i_2, \dots, i_j, \dots, i_m$ such that (i) for all $1 \le j \le m$, we have $1 \le i_j \le k$, and (ii) $\bar{x}_{i_1} \bar{x}_{i_2} \cdots \bar{x}_{i_j} \cdots \bar{x}_{i_m} \# \bar{y}_{i_m}^R \cdots \bar{y}_{i_2}^R \bar{y}_{i_1}^R \in L(balanced)$. \Box

Example 2.4. The instance of P-PCP that corresponds to Example 2.2 is

⁵A *linear* context-free grammar is one in which at most one nonterminal appears on the right-hand side of each production.

 $\bar{X} = ([([, [([, [[[]$ $<math>\bar{Y}^{R} =]),]]),]])])$

This instance of P-PCP has the solution 1, 2, 3, 1 because

Clearly an instance of PCP has a solution if and only if the corresponding instance of P-PCP has a solution.

For a given instance of P-PCP,

$$\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_k^R$$

every solution (if one exists) corresponds to a string in the language generated by the following linear context-free grammar:

While not every string in $L(S_0)$ corresponds to a solution, all strings in $L(S_0)$ that are also in L(balanced) correspond to a solution. That is, the given instance of P-PCP has a solution exactly when the language $L(S_0) \cap L(balanced)$ is non-empty. This observation implies the following theorem (*cf.* [23, pp. 293-294]):

THEOREM 2.5. It is undecidable for arbitrary linear context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2)$ is empty.

The fact that the existence of a solution to a given instance of P-PCP can be characterized by the nonemptiness of the intersection of two linear context-free grammars underlies our result on the undecidability of context-sensitive data-dependence analysis. However, for the purpose of investigating the latter problem, it is useful to develop a slightly more elaborate way of characterizing the solutions to an instance of P-PCP:

Definition 2.6. If an instance of P-PCP

$$\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_k^R$$

has a solution i_1, i_2, \dots, i_m , we say that the following string exhibits the solution in tagged form:⁶

$$\left\{_{i_{1}} \bar{x}_{i_{1}} \left\{_{i_{2}} \bar{x}_{i_{2}} \cdots \left\{_{i_{m}} \bar{x}_{i_{m}} \# \bar{y}_{i_{m}}^{R}\right\}_{i_{m}} \cdots \bar{y}_{i_{2}}^{R}\right\}_{i_{2}} \bar{y}_{i_{1}}^{R}\right\}_{i_{1}}.$$
(2.7)

In general, suppose that $i_1, i_2, \dots, i_j, \dots, i_p$, is some nonempty sequence of integers such that, for all $1 \le j \le p$, we have $1 \le i_j \le k$. Regardless of whether $i_1, i_2, \dots, i_j, \dots, i_p$ is actually a solution to the instance of P-PCP, we say that a string of the form

$$\left\{_{i_{1}} \bar{x}_{i_{1}} \left\{_{i_{2}} \bar{x}_{i_{2}} \cdots \left\{_{i_{j}} \bar{x}_{i_{j}} \cdots \left\{_{i_{p}} \bar{x}_{i_{p}} \# \bar{y}_{i_{p}}^{R}\right\}_{i_{p}} \cdots \bar{y}_{i_{j}}^{R}\right\}_{i_{j}} \cdots \bar{y}_{i_{2}}^{R}\right\}_{i_{2}} \bar{y}_{i_{1}}^{R}\right\}_{i_{1}}$$

exhibits a candidate solution in tagged form. \Box

For instance, because the sequence 2, 1, 2, 3, 1 is not a solution to Example 2.4, the following string exhibits a candidate solution in tagged form (but *not* a solution in tagged form):

$$\left\{ {}_{2} \left[\left(\left[\left\{ {}_{1} \left[\left[\left\{ {}_{2} \left[\left[\left\{ {}_{3} \left[\left[\left\{ {}_{1} \left[\left[\left[\left[\# \right] \right) \right\}_{1} \right] \right] \right] \right] \right\}_{2} \right] \right] \right\}_{1} \right] \right] \right\}_{2} \right] \right\}_{2} \right] \right\}_{2} \right]$$

$$(2.8)$$

In contrast, the following string exhibits a candidate solution to Example 2.4 in tagged form that is also a solution in tagged form:

⁶For technical reasons having to do with the details of the constructions given in Sections 3 and 4, we will also work with some slight variants of (2.7). For instance, in Section 3 we use a version of (2.7) in which each occurrence of "{" is immediately preceded by two occurrences of the symbol e, and each occurrence of "}" is immediately followed by two occurrences of e.

$$\left\{_{1}\left(\left[\left\{_{2}\left[\left[\left\{_{3}\left[\left[\left\{_{1}\left[\left[\left([\#]\right)\right\}_{1}\right]\right]\right]\right]\right\}_{3}\right]\right]\right\}_{2}\right]\right)\right\}_{1}\right]\right\}_{1}\right\}$$
(2.9)

3. AN UNDECIDABLE FAMILY OF L-PATH PROBLEM INSTANCES

In this section, we show how to formulate P-PCP in graph-theoretic terms. In particular, we construct an undecidable family of L-path problem instances, where each problem instance corresponds to an instance of P-PCP. Throughout the remainder of the paper, we assume that we have been given a fixed, but arbitrary, instance of P-PCP consisting of the k pairs of strings

$$\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k$$
$$\bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_k^R$$

Our interest in P-PCP is motivated by the fact that a string that exhibits a P-PCP solution in tagged form has two interleaved patterns of matched delimiters:

- (i) The string $\bar{x}_{i_1} \bar{x}_{i_2} \cdots \bar{x}_{i_j} \cdots \bar{x}_{i_m} \# \bar{y}_{i_m}^R \cdots \bar{y}_{i_j}^R \cdots \bar{y}_{i_2}^R \bar{y}_{i_1}^R$ is in *L*(*balanced*).
- (ii) The string $\{_{i_1} \{_{i_2} \cdots \{_{i_j} \cdots \{_{i_m} \# \}_{i_m} \cdots \}_{i_j} \cdots \}_{i_2} \}_{i_1}$ is in *L*(*balanced'*), the language of balanced strings made up of $\{_i \text{ and } \}_i$, for $1 \le i \le k$, defined by the following linear context-free grammar:

$$\begin{array}{rcl} balanced' & \longrightarrow & \left\{ {}_{i} & balanced' & \right\}_{i} & \text{ for } 1 \leq i \leq k \\ & & \mid & \left\{ {}_{i} & \# & \right\}_{i} & & \text{ for } 1 \leq i \leq k \end{array}$$

It is important to note that, in general, in a string that exhibits a P-PCP solution in tagged form, *the two* balancing processes can be out of sync. For instance, in the following prefix of string (2.9)

 $\left\{ {}_{1}([([\left\{ {}_{2}[([\left\{ {}_{3}[[[\left\{ {}_{1}([([\#]) \right\}]_{1} \right]$

S

the last symbol, namely " $\}_1$ ", does not match with the seventh-from-last symbol, namely "[". However, we can capture the structure of P-PCP solutions in tagged form via the intersection of two linear context-free languages:

(i) The language $L(S_1)$ consists of strings of L(balanced) with an arbitrary number of symbols of the form " $\{i$ " interspersed among the open-parenthesis and open-bracket symbols, and an arbitrary number of symbols of the form " $\}_i$ " interspersed among the close-parenthesis and close-bracket symbols:

(ii) The language $L(S_2)$ consists of all possible candidate solutions, in tagged form, to the given instance of P-PCP:

$$\begin{array}{cccc} S_2 & \longrightarrow & \left\{ \begin{smallmatrix} i & \bar{x}_i & S_2 & \bar{y}_i^R \\ i & \bar{x}_i & \# & \bar{y}_i^R \end{smallmatrix} \right\}_i & & \text{for } 1 \le i \le k \end{array}$$

Languages $L(S_1)$ and $L(S_2)$ capture the two interleaved patterns of matched delimiters noted above: $L(S_2)$ consists of all possible candidate solutions, in tagged form, to the given instance of P-PCP; $L(S_1)$ consists of strings such that when all " $\{i$ " and " $\}_i$ " symbols are excluded, we are left with a string in L(balanced). Furthermore, the language $L(S_1) \cap L(S_2)$ consists of exactly the solutions, in tagged form, to the given instance of P-PCP. For instance, for Example 2.4, strings (2.8) and (2.9) are both in $L(S_2)$, but only string (2.9) is in $L(S_1)$; that is,

 $\Big\{_1(\big[\big(\big[\Big\{_2\big[\big(\big[\Big\{_3\big[\big[\big\{_1(\big[([\#])\big\}_1\big]\big)\big]\big]\big]\Big)\Big\}_3\big]\big]\Big)\Big\}_2\big]\big)\Big\}_1\in L(S_1)\cap L(S_2).$

Similar to what we observed for the language $L(S_0) \cap L(balanced)$, the given instance of P-PCP has a solution exactly when the language $L(S_1) \cap L(S_2)$ is non-empty.

We now show how to construct a graph (with two distinguished vertices, *s* and *t*) such that there is an $L(S_1) \cap L(S_2)$ -path from *s* to *t* if and only if the given instance of P-PCP has a solution. Fig. 1 shows a schematic diagram that illustrates the construction. For an instance of P-PCP $\bar{X} = \bar{x}_1, \bar{x}_2, \dots, \bar{x}_j, \dots, \bar{x}_k, \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \dots, \bar{y}_j^R, \dots, \bar{y}_k^R$, the graph contains *k* regions of the form



Call the left part of such a region an *x*-string segment, and the right part a \bar{y}^R -string segment. Note that the $j^{th} \bar{x}$ -string segment begins with the sequence " $e\{j^n, w\}$ whereas the $j^{th} \bar{y}^R$ -string segment ends with the



Figure 1. A schematic diagram of the graph that would be constructed for an instance of P-PCP $\bar{X} = \bar{x}_1, \bar{x}_2, \dots, \bar{x}_j, \dots, \bar{x}_k; \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \dots, \bar{y}_j^R, \dots, \bar{y}_k^R.$

sequence " $\}_{j}e$ ". The dotted edges labeled *e* around the outsides of Fig. 1 serve to connect each \bar{x} -string segment to all of the other \bar{x} -string segments, and each \bar{y}^{R} -string segment to all of the other \bar{y}^{R} -string segments. Any number of \bar{x} -string segments can be concatenated together to form a path in any order; however, each such segment is labeled in the word of the path by the appropriate "{" symbol. Similarly, any number of \bar{y}^{R} -string segments can be concatenated together to form a path in any order; however, each such segment is labeled in the word of the path by the appropriate "{" symbol.

The fact that certain edges in Fig. 1 are dotted has no special significance; they are displayed in this way to highlight the fact that these edges correspond to interprocedural data-dependence edges in dependence graphs. This correspondence will be made clear in Section 4 (cf. Fig. 4).

By following one of the edges that is labeled with "#", a path can pass from an \bar{x} -string segment to a \bar{y}^{R} -string segment. However, once a #-edge is taken, the path can only be extended with \bar{y}^{R} -string segments. Consequently, all paths from s to t are of the form:

s \xrightarrow{e} some number of \bar{x} -string segments $\xrightarrow{\#}$ some number of \bar{y}^{R} -string segments \xrightarrow{e} t

Here we see the reason for the remark made in footnote 6: In the word of a path from s to t, each occurrence of "{" is immediately preceded by two occurrences of the symbol e, and each occurrence of "}" is immediately followed by two occurrences of e. Technically, the definition of a (candidate) P-PCP solution in tagged form should be changed accordingly, and also each occurrence of "{" in grammars S_1 and S_2 should be replaced with "e e {"," and each occurrence of "}" should be replaced with "e e {".

We now observe that

- If there is an L(S₁) ∩ L(S₂)-path P from s to t, a solution to the instance of P-PCP can be read off from the word of P by reading it as a P-PCP solution in tagged form.
- If the instance of P-PCP has the solution $i_1, i_2, \dots, i_j, \dots, i_m$, then we can find an $L(S_1) \cap L(S_2)$ -path *P* from *s* to *t* by
 - (i) following the e edge from s,
 - (ii) choosing \bar{x} -string segments in the order $\bar{x}_{i_1}, \bar{x}_{i_2}, \dots, \bar{x}_{i_j}, \dots, \bar{x}_{i_m}$, thereby generating a subpath whose word is $e \{_{i_1} \bar{x}_{i_1} e e \{_{i_2} \bar{x}_{i_2} \dots e e \{_{i_n} \bar{x}_{i_n} \dots e e \{_{i_m} \bar{x}_{i_m}, \dots e e \} \}$
 - (iii) following the #-edge from the $i_m^{th} \bar{x}$ -string segment to the $i_m^{th} \bar{y}^R$ -string segment,
 - (iv) choosing \bar{y}^R -string segments in the order $\bar{y}_{i_m}^R, \dots, \bar{y}_{i_j}^R, \dots, \bar{y}_{i_2}^R, \bar{y}_{i_1}^R$, thereby generating a subpath whose word is $\bar{y}_{i_m}^R$ $\Big|_{i_m} e e \cdots \bar{y}_{i_j}^R \Big|_{i_j} e e \cdots \bar{y}_{i_2}^R \Big|_{i_2} e e \bar{y}_{i_1}^R \Big|_{i_1} e$,
 - (v) following the e edge to t.

The word of this path exhibits the solution to the instance of P-PCP in tagged form (modulo the extra occurrences of *e*). As we observed earlier, $L(S_1) \cap L(S_2)$ consists of exactly the solutions, in tagged form, to the given instance of P-PCP. Hence, the path constructed above is an $L(S_1) \cap L(S_2)$ -path from *s* to *t*.

We shall call a graph constructed in the manner described above a *P-PCP graph*. For the particular case of the instance of P-PCP introduced in Example 2.4, the corresponding P-PCP graph is shown in Fig. 2.

The above observations prove the following lemma:

LEMMA 3.1. Given an instance of P-PCP (with corresponding grammars S_1 and S_2 , and P-PCP graph G), there is an $L(S_1) \cap L(S_2)$ -path from s to t in G if and only if the instance of P-PCP has a solution.

UNDECIDABILITY OF CONTEXT-SENSITIVE, STRUCTURE-TRANSMITTED DATA-DEPEN-DENCE ANALYSIS

In this section, we show how a slight modification of the construction presented in the previous section implies that it would be impossible to create a precise algorithm for context-*sensitive*, structure-transmitted data-dependence analysis. In particular, we construct a family of programs whose data-dependence graphs encode the P-PCP graphs.


Figure 2. The P-PCP graph that would be constructed for the instance of P-PCP given in Example 2.4. This graph contains an $L(S_1) \cap L(S_2)$ -path from *s* to *t* whose word is

 $e e \left\{ {}_{1}([(e e \left\{ {}_{2}[(e e \left\{ {}_{3}[[e e \left\{ {}_{1}([([\#]) \right\}_{1} e e])]]] \right\}_{3} e e]]) \right\}_{2} e e]) \right\}_{1} e e,$ which indicates that the instance of P-PCP given in Example 2.4 has the solution 1, 2, 3, 1.

For instance, Fig. 3 shows a C program fragment whose data-dependence graph (see Fig. 4) corresponds to the instance of P-PCP given in Example 2.4. A context-*sensitive*, structure-transmitted data-dependence analysis should report that variable x may have the value atom("A") at program point t (which corresponds to the fact that this instance of P-PCP has the solution 1, 2, 3, 1). In Fig. 3, the symbols " $\{j$ " and " $\}_j$ " correspond to data dependences associated with the call from procedure f to procedure fj and the return from fj to f, respectively; the symbol "(" corresponds to a data construction in which the value is placed in the first position of a cons; the symbol "[" corresponds to a selection via car; the symbol "[" corresponds to a selection via car; the symbol "]" corresponds to a selection via cdr. Data dependences associated with calls to procedure f are labeled by symbols of the form " \langle_i "; data dependences associated with corresponding returns from f are labeled by " \rangle_i "; the symbol "#" corresponds to a data dependence that occurs when a (recursive) call on f is finally

bypassed.7

When inspecting Fig. 3, the reader should keep in mind that the left-to-right encoding of a string—where a string consists of either all open parentheses or all closed parentheses—corresponds to working one's way out from the inner occurrence of x in the expression that encodes the string.

The idea illustrated in Figs. 3 and 4 carries over to all instances of P-PCP: In general, the pair of strings \bar{x}_i , \bar{y}_i^R is encoded by a procedure of the form

```
List *x;
void f1() {
    x = cons(NULL, cons(cons(NULL, cons(x, NULL)), NULL));
                                                            /* Encodes ([([
                                                                                */
    if (. . .) {
        f();
    }
    x = car(cdr(x));
                                                            /* Encodes ])
                                                                                */
}
void f2() {
    x = cons(NULL,cons(cons(NULL,x),NULL));
                                                            /* Encodes [([
                                                                                * /
    if (. . .) {
        f();
    }
    x = car(cdr(cdr(x)));
                                                            /* Encodes ]])
                                                                                */
}
void f3() {
    x = cons(NULL, cons(NULL, cons(NULL, x)));
                                                            /* Encodes [[[
                                                                                */
    if (. . .) {
        f();
    }
    x = car(cdr(cdr(cdr(cdr(cdr(x))))));
                                                            /* Encodes ])]]]) */
}
void f() {
    if (. . .) fl();
    else if (. . .) f2();
    else f3();
}
void main() {
 s: x = atom("A");
                         /* A special value used nowhere else in the program */
    f();
 t: /* Could x be atom("A") here? */
}
```

Figure 3. The C program scheme that would be constructed for the instance of P-PCP given in Example 2.4. The relevant part of this program's data-dependence graph is shown in Fig. 4.

⁷The role of the labels " \langle " and " \rangle " is similar to that of "{" and "}", respectively, in that both kinds of parenthesis pairs encode procedure call/return. However, \langle and \rangle have been introduced as separate symbols to emphasize the fact that the calls to procedure f play a different role in the construction than the calls to the fj procedures.

Procedure f has the form

void f() {
 if (. . .) f1();
 else if (. . .) f2();
 .
 .
 else if (. . .) fk-1();
 else fk();
}

Procedure main is the same as in Fig. 3.

The undecidability of context-*sensitive*, structure-transmitted data-dependence analysis follows from two properties: (i) the data-dependence graph for a program of the form given above is very much like the P-PCP graph for the given instance of P-PCP (see Fig. 1), and (ii) variable x can have the value a tom("A") at program point t if and only if there is a path from s to t whose word is in a language very similar to $L(S_1) \cap L(S_2)$ (except that $\langle s \text{ and } \rangle$'s must also be balanced).

For instance, the portion of the data-dependence graph shown in Fig. 4 (for the program given in Fig. 3) is identical to the P-PCP graph shown in Fig. 2, except that certain dotted edges, corresponding to calls to f and returns from f, now have labels of the form $\langle_i \text{ or } \rangle_i$.⁸ Therefore, the path language of interest for identifying context-*sensitive*, structure-transmitted data dependences must now incorporate the labels $\langle_i \text{ and } \rangle_i$ (for $0 \le i \le k$). Formally, this is accomplished by considering $L(S_1') \cap L(S_2')$ -paths, where the grammars S_1' and S_2' are defined as follows:

$$S_{1}' \rightarrow e \langle_{0} S_{1}'' \rangle_{0} e$$

$$S_{1}'' \rightarrow (S_{1}'')$$

$$\mid [S_{1}'']$$

$$\mid (\#)$$

$$\mid [\#]$$

$$\mid e \{_{i} S_{1}'' \quad \text{for } 1 \leq i \leq k$$

$$\mid S_{1}'' \}_{i} e \quad \text{for } 1 \leq i \leq k$$

$$\mid S_{1}'' \rangle_{i} \quad \text{for } 1 \leq i \leq k$$

$$\mid S_{1}'' \rangle_{i} \quad \text{for } 1 \leq i \leq k$$

$$S_{2}' \rightarrow e \langle_{0} S_{2}'' \rangle_{0} e$$

$$S_{2}'' \rightarrow e \{_{i} \bar{x}_{i} \langle_{i} S_{2}'' \rangle_{i} \bar{y}_{i}^{R} \}_{i} e \quad \text{for } 1 \leq i \leq k$$

$$\mid e \{_{i} \bar{x}_{i} \# \bar{y}_{i}^{R} \}_{i} e \quad \text{for } 1 \leq i \leq k$$

We also change the notion of a P-PCP solution in tagged form to one in which each occurrence of \bar{x}_i (except for the innermost one) is immediately followed by an occurrence of ζ_i , and each occurrence of \bar{y}_i^R (except for the innermost one) is immediately preceded by an occurrence of λ_i .

⁸The only elements omitted from the data-dependence graph shown in Fig. 4 concern dependences on NULL. These are irrelevant to the question of how atom("A") flows through the program—and to our embedding of P-PCP graphs in data-dependence graphs.



Figure 4. Relevant parts of the data-dependence graph for the program shown in Fig. 3. This corresponds to the P-PCP graph shown in Fig. 2, except that certain dotted edges now have labels of the form $\langle_i \text{ or } \rangle_i$.

By the same argument used to prove Lemma 3.1, there is an $L(S_1') \cap L(S_2')$ -path from s to t in the datadependence graph if and only if the given instance of P-PCP has a solution. Therefore, a context-*sensitive*, structure-transmitted data-dependence analysis would determine that x could have the value atom("A")at program point t if and only if the given instance of P-PCP has a solution. Consequently, context-*sensitive*, structure-transmitted data-dependence analysis is undecidable (*i.e.*, an algorithm for context-*sensitive*, structure-transmitted data-dependence analysis cannot exist).

5. CONCLUSIONS AND IMPLICATIONS FOR OTHER PROGRAM-ANALYSIS FRAMEWORKS

Earlier work by the author and his colleagues has demonstrated the usefulness of formulating programanalysis problems in terms of graph-reachability questions [32]. This approach has been used to obtain a number of positive results about program-analysis problems (specifically, polynomial-time algorithms for solving a variety of different problems [15,28,30,16,29,34,25]). The present paper demonstrates that this viewpoint is also valuable from the standpoint of obtaining negative results about program-analysis problems (see also [31]).

The undecidability result in this paper concerns a situation in which there are two interleaved patterns of matching "events". Viewed more broadly, the notions of "interleaved matched-parenthesis paths" and "P-PCP solutions in tagged form" are two concepts that can provide insight into whether other program-analysis problems are undecidable. For instance, Ramalingam showed recently that synchronization-sensitive, context-sensitive interprocedural analysis of multi-tasking concurrent programs is undecidable [27]. His

result was inspired by the one given in the present paper, using the insight that synchronization-sensitive, context-sensitive interprocedural analysis also involves two interleaved patterns of matching events.

Set Constraints and Set-Based Analysis

Following earlier work by Reynolds [33] and Jones and Muchnick [18], a number of people in recent years have explored the use of set constraints for analyzing programs. Set constraints are typically used to collect a superset of the set of values that the program's variables may hold during execution. Typically, a set variable is created for each program variable at each program point; set constraints are generated that approximate the program's behavior; program analysis then becomes a problem of finding the least solution of the set-constraint problem. Set constraints have been used both for program analysis [33,18,1,11,12], and for type inference [2,3].

Melski and Reps have obtained a number of results on the relationship between certain classes of set constraints and CFL-reachability [25]. Their results establish relationships in both directions: They showed that CFL-reachability problems and a subclass of what have been called *definite set constraints* [10] are equivalent. That is, given a CFL-reachability problem, it is possible to construct a set-constraint problem whose answer gives the solution to the CFL-reachability problem; likewise, given a set-constraint problem, it is possible to construct a CFL-reachability problem whose answer gives the solution to the CFL-reachability problem whose answer gives the solution to the set-constraint problem. It is also shown in [25] that CFL-reachability is equivalent to a class of set constraints that was designed to be useful for (context-*insensitive*) analysis of programs written in a higher-order language—so-called "set-based analysis" [11]. The results of Sections 3 and 4 imply that if you start with a version of context-*insensitive* set-based analysis that is at least as precise as the context-*insensitive* structure-transmitted data-dependence analysis illustrated in Example 1.5, then it is impossible to create an algorithm for the context-*sensitive* version of your set-based analysis, even for a first-order language.

Control-Flow Analysis

In [35], Sharir and Pnueli defined two methods for carrying out interprocedural dataflow analysis so as to ensure that the propagation of dataflow information respects the fact that when a procedure finishes it returns to the site of the most recent call. In one of their methods, the so-called "call-strings approach", each piece of dataflow information is tagged with a call string that records the history of uncompleted procedure calls along which that data has propagated. The call string on a piece of information is updated whenever a propagation step associated with a call statement or return statement is performed. The information that would be obtained, in principle, if call strings were allowed to grow arbitrarily long is called the call-strings- ∞ solution.

Sharir and Pnueli show that, for distributive dataflow-analysis problems over a finite semilattice, it is possible to restrict the length of call strings to some fixed length (where the bound on the length required is quadratic in the size of the lattice and linear in the number of call sites in the program) and yet still obtain a result that is equivalent in precision to the call-strings- ∞ solution. By suitable means, approximate (but safe) solutions can also be obtained using shorter call strings; limiting call strings to length *k* defines the call-strings-*k* solution.

In considering algorithms for interprocedural dataflow analysis, one should be careful not to confuse two separate issues:

- (i) Whether an algorithm computes a solution equal in precision to the call-strings- ∞ solution.
- (ii) Whether an algorithm computes its solution by *actually tracking* entities labeled by call strings (*e.g.*, of some length k).

A type-(ii) algorithm typically has worst-case running time that is exponential in k. However, for suitably restricted classes of interprocedural dataflow-analysis problems, there are algorithms with property (i), yet

their worst-case running times are *polynomial* in the size of the program;⁹ these algorithms use dynamic programming, rather than utilizing entities labeled with explicit call strings [30,34]. For the same class of problems, a type-(ii) algorithm will, in general, have exponential running time.

These results provide an interesting contrast with those that have been obtained on a program-analysis problem of interest to the functional-programming community: the problem of "k-CFA", or "control-flow analysis" (for higher-order programming languages) using call strings of length k [36,17,13,26]. The goal of control-flow analysis is to track data and control flow in the presence of first-class (anonymous) functions, data constructors, and selectors. Many of the algorithms that have been given for k-CFA are type-(ii) algorithms (in the sense mentioned above), in that they actually track entities labeled by call strings of length $\leq k$. In general, the running time of these algorithms is exponential in k.

Similar to the concept of the call-strings- ∞ solution to an interprocedural dataflow-analysis problem, the ∞ -CFA solution is what would be obtained, in principle, if call strings were allowed to grow arbitrarily long. The results of Sections 3 and 4 imply that, in general, when data constructors and selectors are to be taken into account, ∞ -CFA is undecidable. That is, in the presence of data constructors and selectors, the ∞ -CFA solution cannot be computed.

ACKNOWLEDGEMENTS

I am grateful for discussions that I had about the problem with F. Nielson, G. Ramalingam, S. Horwitz, and D. Melski.

REFERENCES

- Aiken, A. and Murphy, B.R., "Implementing regular tree expressions," pp. 427-447 in *Func. Prog. and Comp. Arch., Fifth ACM Conf.*, (Cambridge, MA, Aug. 26-30, 1991), *Lec. Notes in Comp. Sci.*, Vol. 523, ed. J. Hughes, Springer-Verlag, New York, NY (1991).
- 2. Aiken, A. and Murphy, B.R., "Static type inference in a dynamically typed language," pp. 279-290 in *Conf. Rec. of the Eighteenth ACM Symp. on Princ. of Prog. Lang.*, (Orlando, FL, Jan. 1991), ACM, New York, NY (1991).
- Aiken, A. and Wimmers, E.L., "Type inclusion constraints and type inference," pp. 31-41 in Sixth Conf. on Func. Prog. and Comp. Arch., (Copenhagen, Denmark), (June 1993).
- Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proc. of the ACM SIGPLAN* 88 Conf. on Prog. Lang. Design and Implementation, (Atlanta, GA, June 22-24, 1988), SIGPLAN Not. 23(7) pp. 47-56 (July 1988).
- Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* 23(7) pp. 57-66 (July 1988).
- 6. Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conf. Rec. of the Sixteenth ACM Symp. on Princ. of Prog. Lang.*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
- Emami, M., Ghiya, R., and Hendren, L.J., "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *Proc. of the ACM SIGPLAN 94 Conf. on Prog. Lang. Design and Implementation*, (Orlando, FL, June 22-24, 1994), *SIG-PLAN Not.* 29(6) pp. 242-256 (June 1994).
- 8. Harrison, M.A., Introduction to Formal Language Theory, Addison-Wesley, Reading, MA (1978).
- 9. Hecht, M.S., Flow Analysis of Computer Programs, North-Holland, New York, NY (1977).

To achieve a polynomial time bound, the tricks are:

- To restrict attention to a certain class of semilattices. However, this class can include semilattices whose size is exponential in the size of the program (*e.g.*, the powerset of the program points, the powerset of the program's variables, *etc.*).
- To tabulate the Sharir-Pnueli φ functions pointwise (e.g., on singletons, rather than entire sets).

⁹In their Theorem 7-3.4, Sharir and Pnueli establish that the greatest fixed point of a certain set of equations equals the meet-over-all-valid-paths (MOVP) solution. In their Theorem 7-4.6, they establish that the MOVP solution equals the call-strings- ∞ solution. As for algorithms, Sharir and Pnueli give a worklist algorithm for finding the greatest fixed point of the set of equations. However, Section 7-3 of [35] presents a fairly general framework: in particular, the only assumption about the semilattice is that it is finite. Because of the way their dynamic-programming algorithm tabulates information, when the size of the semilattice is exponential in the size of the program.

This is essentially what is done in [30] for the class of finite-distributive-subset problems, and in [34] for the larger class of distributive environment problems.

- Heintze, N. and Jaffar, J., "A decision procedure for a class of set constraints," Tech. Rep. CMU-CS-91-110, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (1991).
- Heintze, N., "Set-based analysis of ML programs," Tech. Rep. CMU-CS-93-193, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (July 1993).
- 12. Heintze, N. and Jaffar, J., "Set constraints and set-based analysis," in 2nd Workshop on Principles and Practice of Constraint Programming, (May 1994).
- Heintze, N. and McAllester, D., "Linear-time subtransitive control flow analysis," Proc. of the ACM SIGPLAN 97 Conf. on Prog. Lang. Design and Implementation, (Las Vegas, Nevada, June 15-18, 1997), SIGPLAN Not. 32(5) pp. 261-272 (May 1997).
- 14. Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA (1979).
- Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," ACM Trans. Program. Lang. Syst. 12(1) pp. 26-60 (Jan. 1990).
- Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," SIGSOFT 95: Proc. of the Third ACM SIGSOFT Symp. on the Found. of Softw. Eng., (Wash., DC, Oct. 10-13, 1995), ACM SIGSOFT Softw. Eng. Notes 20(4) pp. 104-115 (1995).
- Jagannathan, S. and Weeks, S., "A unified treatment of flow analysis in higher-order languages," pp. 393-406 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.*, (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
- Jones, N.D. and Muchnick, S.S., "Flow analysis and optimization of Lisp-like structures," pp. 102-131 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
- Khedker, U.P. and Dhamdhere, D.M., "A generalized theory of bit vector data flow analysis," ACM Trans. Program. Lang. Syst. 16(5) pp. 1472-1511 (Sept. 1994).
- 20. Kou, L.T., "On live-dead analysis for global data flow problems," J. ACM 24(3) pp. 473-483 (July 1977).
- 21. Kuck, D.J., The Structure of Computers and Computations, Vol. 1, John Wiley & Sons, New York, NY (1978).
- Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conf. Rec. of the Eighth ACM Symp. on Princ. of Prog. Lang.*, (Williamsburg, VA, Jan. 26-28, 1981), ACM, New York, NY (1981).
- 23. Lewis, H.R. and Papadimitriou, C.H., Elements of the Theory of Computation, Prentice-Hall, Englewood Cliffs, NJ (1981).
- 24. McCarthy, J., "A basis for a mathematical theory of computation," pp. 33-70 in *Computer Programming and Formal Systems*, ed. Braffort and Hershberg, North-Holland, Amsterdam (1963).
- 25. Melski, D. and Reps, T., "Interconvertibility of a class of set constraints and context-free language reachability," *Theor. Comp. Sci.* 248(1-2)(Nov. 2000). (To appear.)
- Nielson, F. and Nielson, H.R., "Infinitary control flow analysis: A collecting semantics for closure analysis," pp. 332-345 in *Conf. Rec. of the Twenty-Fourth ACM Symp. on Princ. of Prog. Lang.*, (Paris, France, Jan. 15-17, 1997), ACM, New York, NY (1997).
- 27. Ramalingam, G., "Context-sensitive synchronization-sensitive analysis is undecidable," Res. Rep. RC 21493, IBM T.J. Watson Res. Cent., Yorktown Heights, NY (May 1999).
- Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," SIGSOFT 94: Proc. of the Second ACM SIGSOFT Symp. on the Found. of Softw. Eng., (New Orleans, LA, Dec. 7-9, 1994), ACM SIGSOFT Softw. Eng. Notes 19(5) pp. 11-20 (Dec. 1994).
- 29. Reps, T., "Shape analysis as a generalized path problem," pp. 1-11 in *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip. (PEPM 95)*, (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
- 30. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.*, (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
- 31. Reps, T., "On the sequential nature of interprocedural program-analysis problems," Acta Inf. 33 pp. 739-757 (1996).
- 32. Reps, T., "Program analysis via graph reachability," *Information and Software Technology* **40**(11-12) pp. 701-726 Elsevier Science, (Nov./Dec. 1998).
- Reynolds, J.C., "Automatic computation of data set definitions," pp. 456-461 in *Information Processing 68: Proc. of the IFIP Congress* 68, North-Holland, New York, NY (1968).
- 34. Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comp. Sci.* **167** pp. 131-170 (1996).
- Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: The*ory and Applications, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
- Shivers, O., "Control flow analysis in Scheme," Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation, (Atlanta, GA, June 22-24, 1988), SIGPLAN Not. 23(7) pp. 164-174 (July 1988).
- 37. Wilson, R.P. and Lam, M.S., "Efficient context-sensitive pointer analysis for C programs," *Proc. of the ACM SIGPLAN 95 Conf.* on Prog. Lang. Design and Implementation, (La Jolla, CA, June 18-21, 1995), SIGPLAN Not. **30**(6) pp. 1-12 (June 1995).
- Yannakakis, M., "Graph-theoretic methods in database theory," pp. 230-242 in Proc. of the Ninth ACM Symp. on Princ. of Database Syst., (1990).

The Undecidability of Aliasing

G. RAMALINGAM

IBM T. J. Watson Research Center

Alias analysis is a prerequisite for performing most of the common program analyses such as reaching-definitions analysis or live-variables analysis. Landi [1992] recently established that it is impossible to compute statically precise alias information—either may-alias or must-alias—in languages with if statements, loops, dynamic storage, and recursive data structures: more precisely, he showed that the may-alias relation is not recursive, while the must-alias relation is not even recursively enumerable. This article presents simpler proofs of the same results.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.4.1 [**Mathematical Logic**]: Computability Theory; F.4.3 [**Formal Languages**]: Decision Problems

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, pointer analysis

1. INTRODUCTION

Compilers and various other programming tools make good use of static program analysis. To solve most program analysis problems, such as the problem of determining live variables, one requires alias information, or information about whether two L-valued expressions may/must have the same value at some program point. Informally, two names or L-valued expressions are said to alias each other at a particular point during program execution if both refer to the same location. In the may-alias problem, one is interested in identifying aliases that can occur during some execution of the program, while in the must-alias problem, one is interested in identifying aliases that occur in all executions of the program. Obviously, such information is relevant to most dataflow analysis problems.

Program analysis is commonly performed under the conservative assumption that all paths in the program are executable, since the problem of deciding if an arbitrary path in a program is executable is undecidable. This simplifying assumption makes it possible to solve a number of program analysis problems. Unfortunately, even this assumption is not sufficient to make the may-alias or must-alias problem decidable.

© 1994 ACM 0164-0925/94/0900-1467 \$03.50

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994, Pages 1467-1471.

Author's address: IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1468 • G. Ramalingam

Names a and b are said to may-alias each other at a program point if there exists a path P from the program entry to that program point, such that a and b refer to the same location after execution along path P. Names a and b are said to must-alias each other at a program point if, for all paths P from the program beginning to the program point, a and b both refer to the same location after execution along path P. Landi [1992] recently established that even the simpler intraprocedural versions of the may-alias and must-alias problems are undecidable for languages that permit recursive data structures to be built. Here, we present a simpler proof of the same result. We establish the undecidability of this problem by reducing the Post's Correspondence Problem, or PCP, to the may-alias problem. Section 2 of the article presents proofs of the undecidability results, while Section 3 discusses related work.

2. THE UNDECIDABILITY OF ALIASING

The decision version of the may-alias problem is the following: given a program point in a program and two names, decide if the may-alias relation holds between the given names at the given program point. More generally, we are interested in generating the set of all may-alias facts that hold true. This set can be made finite by restricting attention to the names and L-valued expressions that occur in the program.

Definition 2.1. The Post's Correspondence Problem, or PCP is the following: Given arbitrary lists A and B of r strings each in $\{0, 1\}^+$, say

$$A = w_1, w_2, \dots, w_r$$
$$B = z_1, z_2, \dots, z_r$$

does there exist a nonempty sequence of integers i_1, i_2, \ldots, i_k such that

$$w_{i_1}w_{i_2}\cdots w_{i_k}=z_{i_1}z_{i_2}\cdots z_{i_k}$$

THEOREM 2.2. The PCP is undecidable [Hopcroft and Ullman 1979].

THEOREM 2.3. The intraprocedural may-alias problem is undecidable for languages with if statements, loops, dynamic storage, and recursive data structures.

PROOF. We relate PCP to the may-alias problem as follows. Consider a binary tree with root *root*. A binary string consisting of 0s and 1s can be interpreted as representing a path from the root of the binary tree with 0 representing a left branch and 1 a right branch. Define branch(0) to be *left* and branch(1) to be *right*. For any binary string $b_0b_1 \cdots b_n$, define $path(b_0b_1 \cdots b_n)$ to be $branch(b_0) \rightarrow branch(b_1) \rightarrow \cdots \rightarrow branch(b_n)$. Let α and β be two binary strings. Then, $\alpha = \beta$ iff $root \rightarrow path(\alpha)$ and $root \rightarrow path(\beta)$ refer to the same node in the binary tree. Essentially, this is the idea behind our reduction of PCP to the may-alias problem.

Given an instance of PCP, we construct the program in Figure 1. The program is written in C, but it can be written in any language with if statements, loops, dynamic storage, and recursive data structures. The may-

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

alias relation holds between $*(q \rightarrow left)$ and node at line 39 iff the given instance of PCP has a solution, as explained below. Ignore, for the moment, lines 7 through 19, and assume that root points to a binary tree at line 20. There are r different branches inside the loop of lines 22 through 35—number these branches 1 through r. Any path P in the program from line 20 to line 36 that iterates through the loop (lines 22 through 35) t times corresponds to a sequence σ of t integers, where the *j*th element in the sequence denotes the branch taken during the *j*th time through the loop. Furthermore, *p and *q will alias each other at the end of path P iff the sequence σ is a solution to the given PCP instance, provided that root pointed to a "sufficiently large" binary tree at line 20.

Instead of actually constructing a binary tree, we use the code in lines 9 through 19 to "generate" all possible paths through a binary tree. In doing this, the pointer fields of newly allocated tree nodes are not initialized to a null pointer as might be done usually. Instead, these fields are initialized to point to a special node called undefined whose left and right fields are initialized to point to itself. This ensures that every possible path from the program beginning to line 36 can be executed without raising any errors such as dereferencing a null pointer. Consequently, at line 36, either pointer p has a "proper value" and points to some node allocated in line 12 or 15, or pointer p points to the node *undefined*. The same claim holds true for pointer q. Consequently, the given instance of PCP has a solution iff there exists some execution path to line 36, at the end of which $p = q \neq \&$ undefined. Checking for this condition can be converted into checking for a may-alias fact using line 38. Obviously, the may-alias relation holds between $*(q \rightarrow left)$ and node at line 39 iff the given instance of PCP has a solution.

The may-alias relation, however, is recursively enumerable because we can enumerate all paths in the program, and for any given path, we can determine the aliases that hold after execution along that path.

THEOREM 2.4. The intraprocedural must-alias problem is undecidable for languages with if statements, loops, dynamic storage, and recursive data structures. The intraprocedural must-alias relation is not even recursively enumerable.

PROOF. The undecidability of the must-alias problem follows immediately from the undecidability of the may-alias problem. Consider Figure 1. Line 40 shows how must-alias information can be used to compute may-alias information. The must-alias relation holds between *node* and *(node.left) in line 41 iff the may-alias relation does *not* hold between *node* and $*(q \rightarrow left)$ in line 39. The complement of a recursively enumerable but nonrecursive set is not recursively enumerable. It follows that the must-alias relation is not even recursively enumerable. \Box

3. RELATED WORK

Kam and Ullman [1977] established that the problem of computing the meet-over-all-paths solution to a monotonic dataflow analysis framework, or monotone MOP problem, is undecidable by reducing a modified version of

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

1470 • G. Ramalingam

```
[1]
        main() {
[2]
                int i;
[3]
                struct tree node {
[4]
                         int value:
                         struct tree_node *left, *right;
[5]
[6]
                 } *root, *p, *q, *t, node, undefined;
                undefined.left = &undefined; undefined.right = &undefined;
[7]
[8]
                root = malloc(sizeof(struct tree_node)); root->left = &undefined; root->right = &undefined;
                t = root:
[9]
[10]
                while ( · · · ) {
                         if ( · · · ) {
[11]
[12]
                                 t->right = malloc(sizeof(struct tree_node));
[13]
                                 t = t -> right;
                         | else {
[14]
                                 t->left = malloc(sizeof(struct tree_node));
[15]
[16]
                                 t = t -> left:
[17]
[18]
                         t->left = &undefined; t->right = &undefined;
[19]
                 }
[20]
                p = root;
                q = root;
[21]
                do {
[22]
[23]
                         i = • • • :
                         if (i == 1) {
[24]
[25]
                                 p = p -> path(w_1); q = q -> path(z_1);
[26]
                         | else if (i == 2) |
[27]
                                 p = p \rightarrow path(w_2); q = q \rightarrow path(z_2);
[28]
                         } else if (i == 3) {
[29]
[30]
                         else if (i == r-1)
                                 p = p \rightarrow path(w_{r-1}); q = q \rightarrow path(z_{r-1});
[31]
[32]
                         } else {
[33]
                                  p = p \rightarrow path(w_r); q = q \rightarrow path(z_r);
[34]
[35]
                 } while ( · · · )
[36]
                 /* The given PCP instance has an affirmative answer iff \exists some execution path to this point after
[37]
                  * which p = q \neq \& undefined. */
[38]
                 p->left = &node; undefined.left = &undefined;
1391
                 /* The given PCP instance has an affirmative answer iff *(q->left) may-alias node at this point. */
[40]
                 node.left = &node; q->left->left = &undefined;
[41]
                 /* node must-alias *(node.left) here iff not *(q->left) may-alias node in line 39. */
[42]
        1
```



PCP to the monotone MOP problem. The proof presented in this article is similar to the proof of Kam and Ullman. However, as Kam and Ullman observe, their result says only that no algorithm that solves *any* monotonic dataflow analysis problem exists. However, they do not rule out the existence of algorithms for a *specific* monotonic dataflow analysis problem, such as the may-alias problem. In other words, the meet-over-all-paths problem for arbitrary monotonic dataflow analysis frameworks is more general than the may-alias problem. Consequently, the undecidability of the latter problem is a stronger result than the undecidability of the former problem.

Larus [1988; 1989] showed that alias analysis was NP-hard in languages with recursive data structures. Landi [1992] presented the first proof that the

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

may-alias problem is not recursive and that the must-alias problem is not even recursively enumerable. He established these results by reducing the halting problem to these problems, and this article presents simpler proofs for the same results.

In the absence of recursively defined data structures, various versions of the aliasing problems become decidable, but remain difficult. Myers [1981] showed that many interprocedural static-analysis problems are NP-complete. Refer to Landi's thesis [1991] for a comprehensive classification of the complexity of various restricted versions of the aliasing problems. Not surprisingly, the problem of computing conservative approximations to the mayalias and must-alias relations in the presence of pointers has attracted and continues to attract much attention. Pfeiffer's thesis [1991] presents a comprehensive overview of this area. Refer to Landi and Ryder [1992] and Choi et al. [1993] for more recent work in this area.

ACKNOWLEDGMENTS

The author thanks the anonymous referees, Charles Fischer, and William Landi for their comments, which helped improve this article and made the proofs more precise.

REFERENCES

- CHOI, J.-D., BURKE, M. G. AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S. Carolina). ACM, New York, 232-245.
- HOPCROFT, J. E. AND ULLMAN, J. D. 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, Mass.
- KAM, J. B. AND ULLMAN, J. D. 1977. Monotone data flow analysis frameworks. In Acta Informatica 7, 305–317.
- LANDI, W. 1992. Undecidability of static analysis. 1992. Lett. Program. Lang. Syst. 1, 4 (Dec.).
- LANDI, W. 1991. Interprocedural aliasing in the presence of pointers. Ph.D. thesis, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.
- LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for pointer-induced aliasing. SIGPLAN Not. 27, 7 (July), 235-248.
- LARUS, J. R. 1989. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. thesis, Univ. of California, Berkeley, Calif. (May).
- LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. SIGPLAN Not. 23, 7 (July), 21–34.
- MYERS, E. 1981. A precise inter-procedural data flow algorithm. In Conference Record of the 8th ACM Symposium on Principles of Programming Languages (Williamsburg, Va., Jan. 26-28). ACM, New York.
- PFEIFFER, P. 1991. Dependence-based representations for programs with reference variables. Ph.D. dissertation and Tech. Rep. TR-1037, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wis. (Aug.).

Received June 1993; revised October 1993 and March 1994; accepted May 1994

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

A Generalized Theory of Bit Vector Data Flow Analysis

UDAY P. KHEDKER and DHANANJAY M. DHAMDHERE Indian Institute of Technology

The classical theory of data flow analysis, which has its roots in unidirectional flows, is inadequate to characterize bidirectional data flow problems. We present a generalized theory of bit vector data flow analysis which explains the known results in unidirectional and bidirectional data flows and provides a deeper insight into the process of data flow analysis. Based on the theory, we develop a worklist-based generic algorithm which is uniformly applicable to unidirectional and bidirectional data flow problems. It is simple, versatile, and easy to adapt for a specific problem. We show that the theory and the algorithm are applicable to all bounded monotone data flow problems which possess the property of the separability of solution.

The theory yields valuable information about the complexity of data flow analysis. We show that the complexity of worklist-based iterative analysis is the same for unidirectional and bidirectional problems. We also define a measure of the complexity of round-robin iterative analysis. This measure, called *width*, is uniformly applicable to unidirectional and bidirectional problems and provides a tighter bound for unidirectional problems than the traditional measure of *depth*. Other applications include explanation of isolated results in efficient solution techniques and motivation of new techniques for bidirectional flows. In particular, we discuss edge splitting and edge placement and develop a feasibility criterion for decomposition of a bidirectional flow into a sequence of unidirectional flows.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—compilers; optimization; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—complexity of proof procedures

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Bidirectional data flows, data flow analysis, data flow frameworks

1. INTRODUCTION

Data flow analysis is the process of collecting information about the uses and definitions of data items in a program. This information is put to a variety of uses, viz., program design, debugging, optimization, maintenance, and docu-

© 1994 ACM 0164-0925/94/0900-1472 \$03.50

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994, Pages 1472-1511.

This work was done when the first author was a research student at the Indian Institute of Technology.

Authors' addresses: U. P. Khedkar, Department of Computer Science, University of Poona, Pune 411 007, India; email: uday@pucsd.ernet.in; D. M. Dhamdhere, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay 400 076, India; email: dmd@cse.iitb.ernet.in.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

mentation. Compilers typically use data flow analysis to collect information for the purpose of code optimization.¹

Data flows used in code optimization involve *unidirectional* dependencies mostly i.e., the data flow information available at a node in the program flow graph is influenced either by its predecessors or by its successors. Such data flows can be readily classified into *forward* and *backward* data flows [Aho et al. 1986]. In *bidirectional* problems, the information available at a node depends on its predecessors as well as its successors. The Morel and Renvoise Algorithm for partial redundancy elimination [Morel and Renvoise 1979] (also called MRA) is a representative bidirectional problem. The advantage of bidirectional problems is that they unify several optimizations reducing both the size and the running time of an optimizer. For example, MRA unifies the traditional optimizations of code movement, common-subexpression elimination, and loop optimization. The Composite Hoisting and Strength Reduction Algorithm [Joshi and Dhamdhere 1982a; 1982b] unifies code movement, strength reduction, and loop optimization.

Though bidirectional data flow problems have been known for over a decade, it has not been possible to explain the intricacies of bidirectional flows using the traditional theory of data flow analysis. Although a fixed-point solution for a bidirectional problem exists, the flow of information and the safety of an assignment cannot be characterized formally. Because of this theoretical lacuna, efficient solutions to bidirectional problems have not been found though some isolated and ad hoc results have been obtained [Dhamdhere 1988a; Dhamdhere and Patil 1993; Dhamdhere et al. 1992].

In this article we present a theory which handles undirectional as well as bidirectional data flow problems uniformly. Apart from explaining the known results in unidirectional and bidirectional flows, it provides deeper insights into the process of data flow analysis. Though the exposition of the theory is based on the *bit vector problems*, the theory is applicable to all bounded monotone data flow problems which possess the property of separability of solution. Several proofs have been omitted from the article for brevity; they can be found in Khedker and Dhamdhere [1992].

Section 2 introduces MRA which is used as a representative example throughout the article. Section 3 reviews the classical theory of data flow analysis. Section 4 defines bit vector problems formally, generalizes the traditional concepts, and provides generic data flow equations which facilitate uniform specification of data flow problems. A worklist-based generic algorithm is developed in Section 5. Arising out of a generalized theory, it is uniformly applicable to unidirectional and bidirectional data flow problems. This section also analyses the performance of the generic algorithm and shows that the complexity of the worklist-based iterative analysis is the same for unidirectional and bidirectional problems.

¹Data flow analysis can be either *interprocedural* or *intraprocedural*. We restrict ourselves to the latter in this article except that the interprocedural information at the entry/exit of a procedure is considered for analyzing data flows within the procedure.

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

1474 • U. P. Khedker and D. M. Dhamdhere

Section 6 discusses several applications of the generalized theory in the complexity of data flow analysis. A new measure called the *width* (w) of a graph for a data flow framework is defined which is shown to bound the number of iterations of round-robin analysis for unidirectional and bidirectional problems. We show that the width provides a tighter bound for unidirectional problems than the traditional measure of *depth*. This section also explains several known results in the solution of bidirectional data flows, viz., edge splitting and edge placement, and develops a feasibility criterion for decomposition of bidirectional flows into a sequence of unidirectional flows. Section 7 discusses the significance and applicability of the results presented in this article.

2. BIDIRECTIONAL DATA FLOWS: AN EXAMPLE

This section introduces the Morel and Renvoise Algorithm (MRA) [Morel and Renvoise 1979] for partial redundancy elimination which is used as a representative bidirectional problem throughout the article.

MRA unifies the traditional optimizations of code movement, common-subexpression elimination, and loop optimization. The importance of the MRA framework lies in the fact that unification of many classical optimizations reduces the size as well as the running time of an optimizer; a 35% reduction in the size and a 30% to 70% reduction in the execution cost has been reported in the literature [Morel and Renvoise 1979]. It has been implemented in at least two important production compilers (MIPS and PL.8) and has inspired several other unifications [Chow 1988; Dhamdhere 1988a; Joshi and Dhamdhere 1982a; 1982b].

The data flow properties and the data flow equations for MRA are given in Figure 1. Note that PPIN, is the bit vector for node i which represents the property PPIN_i for all expressions, whereas PPIN^l_i is the bit representing the expression e_l .

Local property ANTLOC^l represents *local anticipability*, i.e., the existence of an upward exposed expression e_i in node *i*, while TRANSP^{*i*} reflects transparency, i.e., the absence of definition(s) of any operand(s) of e_l in the node. The global property of *anticipability* (ANTIN^l_{*i*}/ANTOUT^l_{*i*}) indicates whether expression e_i is very busy at the entry/exit of node *i*—a necessary and sufficient condition for the safety of placing an evaluation of e_i at the entry/exit of the node [Kennedy 1972]. Equations (1) and (2) do not use ANTIN^{*i*}/ANTOUT^{*i*} properties explicitly; they are implied by PPIN^{*i*}/PPOUT^{*i*} properties. The data flow property of *availability* $(AVIN_i^l/AVOUT_i^l)$ is computed using the classical forward data flow problem [Aho et al. 1986]. The partial redundancy of an expression is represented by the partial availability of the expression (PAVIN^l_i) at the entry of node i. PPIN^l_i indicates the feasibility of placing an evaluation of e_i at the entry of i while PPOUT¹_i indicates the feasibility of placing it at the exit. Computations of an expression e_i are inserted at the exit of node *i* if INSERT^{*l*}_{*i*} = **T**. REDUND^{*l*}_{*i*} indicates that the upward exposed occurrence of e_i in node *i* is redundant and may be deleted.

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

LOCAL DATA FLOW PROPERTIES :

ANTLOC ¹	Node <i>i</i> contains a computation of e_i , not preceded by a d	lefinition
$\operatorname{COMP}_{i}^{l}$	of any of its operands. Node <i>i</i> contains a computation of e_i , not followed by a de of any of its operands	efinition
TRANSP_{i}^{l}	Node <i>i</i> does not contain a definition of any operand of e_i	
GLOBAL DATA FLO	W PROPERTIES :	
AVIN [!] /AVOUT [!] PAVIN [!] /PAVOUT [!] ANTIN [!] /ANTOUT [!] PPIN [!] /PPOUT [!] INSERT [!] REDUND [!]	e_i is available at the entry/exit of node <i>i</i> . e_i is partially available at the entry/exit of node <i>i</i> . e_i is anticipated at the entry/exit of node <i>i</i> . Computation of e_i may be placed at the entry/exit of node Computation of e_i should be inserted at the exit of node First computation of e_i existing in node <i>i</i> is redundant.	de i. i.
DATA FLOW EQUAT	IONS :	
PPIN	$i = PAVIN_i \cdot (ANTLOC_i + TRANSP_i \cdot PPOUT_i)$	(1)
	$\prod_{j \in pred(i)} (AVOUT_j + PPOUT_j)$	
PPOUT	$i_{i} = \prod_{k \in succ(i)} (PPIN_{k})$	(2)
INSERT REDUND	$i = PPOUT_i \cdot \neg AVOUT_i \cdot (\neg PPIN_i + \neg TRANSP_i)$ $i = PPIN_i \cdot ANTLOC_i$	
	Fig. 1. The Morel-Renvoise algorithm.	

The PPIN_i equation is slightly different from the original equation in MRA; the term $\text{ANTIN}_i \cdot (\text{PAVIN}_i + \neg \text{ANTLOC}_i \cdot \text{TRANSP}_i)$ in the original MRA equations is replaced by the term PAVIN_i to prohibit redundant hoisting when the expression is not partially available. The PAVIN_i term represents the *profitability* of hoisting in that there exists at least one possible execution path along which the expression is computed more than once. The other two terms in the PPIN_i equation represent the *feasibility* of hoisting.

Bidirectional dependencies of MRA arise as follows. *Redundancy* of an expression is based on the notion of availability of the expression which gives rise to forward data flow dependencies (reflected by the Π term in the PPIN_i equation). The *safety* of code movement is based on the notion of anticipability of the expression which introduces backward dependencies in the data flow problem (reflected by the Π term in the PPOUT_i equation).

Example 2.1. Consider the program flow graph in Figure 2. The partial redundancy elimination performed by MRA subsumes the following three optimizations:

-Loop-Invariant Movement: The computations of a a * b in node 4 and node 5 are hoisted out of the loops and are inserted in node 2 (REDUND₄^l, REDUND₅^l, and INSERT₂^l are **T**).

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.



Node	Transp	Antloc	Pavin	Avout	Ppin	Ppout	Insert	Redund
1	Т	F	F	F	F	F	F	F
2	Т	F	F	F	F	Т	Ť	F
3	Т	F	T	F	Т	T	F	F
4	T	Т	Т	Т	Т	F	F	Т
5	T	Т	Т	Ť	Т	T	F	T
6	T	Т	Т	T	Т	F	F	Т
7	F	F	Т	F	F	Т	Т	F
8	Т	F	F	F	F	F	F	F
9	F	F	F	F	F	F	F	F
10	Т	Т	F	T	F	F	F	F
11	T	Т	Т	Т	F	Т	F	F
12	Т	Т	Т	T	T	F	F	Т

Fig. 2. Program flow graph and properties for Example 2.1.

- -Code Hoisting: The partially redundant computation of a * b in node 12 is hoisted to node 7. As a result of suppressing this partial redundancy, the path 1-8-11-12 would have only *one* computation of a * b; the unoptimized program has two.
- -Common-Subexpression Elimination: The totally redundant computation of a * b in node 6 is deleted as an instance of common-subexpression elimination.

Note that the partially redundant computation a * b in node 11 is not suppressed since hoisting it to node 8 would be unsafe—the path 1-8-9 had no computation of a * b in the original program.

Example 2.2. Bidirectional data flows have been used also in register assignment and strength reduction optimizations. Figure 3 presents the data flow equations of two such algorithms. The SPPIN/SPPOUT problem of LSIA

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

1476

• BASIC LOAD STORE INSERTION ALGORITHM (LSIA) [Dhamdhere 1988b]

$$SPPIN_{i} = \prod_{j \in pred(i)} (SPPOUT_{j})$$

$$SPPOUT_{i} = DPANTOUT_{i} \cdot (DCOMP_{i} + DTRANSP_{i} \cdot SPPIN_{i}) \cdot \prod_{k \in succ(i)} (DANTIN_{k} + SPPIN_{k})$$

• COMPOSITE HOISTING AND STRENGTH REDUCTION ALGORITHM (CHSA) [Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b]

$$NOCOMIN_{i} = CONSTA_{i} \cdot NOCOMOUT_{i} + \sum_{j \in pred(i)} CONSTB_{i} \cdot NOCOMOUT_{j}$$
$$NOCOMOUT_{i} = CONSTC_{i} + CONSTD_{i} \cdot NOCOMIN_{i} + \sum_{k \in succ(i)} CONSTE_{i} \cdot NOCOMIN_{k}$$

Fig. 3. Data flow equations of some other bidirectional problems.

performs sinking of STORE instructions using partial redundancy elimination techniques [Dhamdhere 1988b]. The NOCOMIN/NOCOMOUT problem of CHSA is used to inhibit the placement of an update computation following a high-strength computation [Joshi and Dhamdhere 1982a; 1982b].

3. NOTIONS FROM CLASSICAL DATA FLOW ANALYSIS

This section presents an overview of the classical theory of data flow analysis and compares various solution methods and their complexities. Our description is based mostly on Graham and Wegman [1976], Hecht [1977], and Marlowe and Ryder [1990]. A more detailed treatment can be found in Aho et al. [1986], Graham and Wegman [1976], Hecht [1977], Kam and Ullman [1977], Kildall [1973], Marlowe and Ryder [1990], and Rosen [1980]. The concluding part of this section motivates the need for a more general setting.

3.1 Preliminaries

A data flow framework is defined as a triple $\mathbf{D} = \langle \mathscr{L}, \sqcap, \mathscr{F} \rangle$ (Figure 4). Elements in \mathscr{L} represent the information associated with the entry/exit of a basic block. \sqcap is the set union or intersection operation which determines the way the global information is combined when it reaches a basic block. A function $f_i \in \mathscr{F}$ represents the effect on the information as it flows through basic block $i.^2$

²Alternatively, the functions can be associated with in-edges (out-edges) of node i for forward (backward) flow problems.

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994.

Stack Size Analysis for Interrupt-driven Programs¹

Krishnendu Chatterjee^a Di Ma^c Rupak Majumdar^e Tian Zhao^d Thomas A. Henzinger^{a,b} Jens Palsberg^{e,*}

^aDepartment of Electrical Engineering and Computer Sciences University of California, Berkeley, CA 94720, USA {c_krish,tah}@eecs.berkeley.edu

^bSchool of Computer and Communication Sciences École Polytechnique Fédérale de Lausanne, Switzerland

^cDepartment of Computer Science Purdue University, West Lafayette, IN 47907, USA madi@cs.purdue.edu

^dDepartment of Computer Science University of Wisconsin, Milwaukee, WI 53211, USA tzhao@cs.uwm.edu

^eDepartment of Computer Science University of California, Los Angeles, CA 90095, USA {rupak,palsberg}@cs.ucla.edu

Abstract

We study the problem of determining stack boundedness and the exact maximum stack size for three classes of interrupt-driven programs. Interrupt-driven programs are used in many real-time applications that require responsive interrupt handling. In order to ensure responsiveness, programmers often enable interrupt processing in the body of lower-priority interrupt handlers. In such programs a programming error can allow interrupt handlers to be interrupted in a cyclic fashion to lead to an unbounded stack, causing the system to crash. For a restricted class of interrupt-driven programs, we show that there is a polynomial-time procedure to check stack boundedness, while determining the exact maximum stack size is PSPACE-complete. For a larger class of programs, the two problems are both PSPACE-complete, and for the largest class of programs we consider, the two problems are high, our algorithms are exponential only in the number of handlers, and polynomial in the size of the program.

Key words: Program analysis, stack bounds, interrupt programs.

Preprint submitted to Elsevier Science

1 Introduction

Most embedded software runs on resource-constrained processors, often for economic reasons. Once the processor, RAM, etc. have been chosen for an embedded system, the programmer has to fit everything into the available space. For example, on a Z86 processor, the stack exists in the 256 bytes of register space, and it is crucial that the program does not overflow the stack, corrupting other data. Estimating the stack size used by a program is therefore of paramount interest to the correct operation of these systems. A tight upper bound is necessary to check if the program fits into the available memory, and to prevent precious system resources (e.g., registers) from being allocated unnecessarily.

Stack size analysis is particularly challenging for interrupt-driven software. Interrupt-driven software is often used in embedded real-time applications that require fast response to external events. Such programs usually have a fixed number of external interrupt sources, and for each interrupt source, a handler that services the interrupt. When an external interrupt occurs, control is transferred automatically to the corresponding handler if interrupt processing is enabled. To maintain fast response, interrupts should be enabled most of the time, in particular, higher-priority interrupts are enabled in lower-priority handlers. Interrupt handling uses stack space: when a handler is called, a return address is placed on the stack, and if the handler itself gets interrupted, then another return address is placed on the stack, and so on. A programming error occurs when the interrupt handlers can interrupt each other indefinitely, leading to an unbounded stack. Moreover, since stack boundedness violations may occur only for particular interrupt sequences, these errors are difficult to replicate and debug, and standard testing is often inadequate. Therefore, algorithms that statically check for stack boundedness and automatically provide precise bounds on the maximum stack size will be important development tools for interrupt-driven systems.

In this paper, we provide algorithms for the following two problems (defined formally in Section 2.3) for a large class of interrupt-driven programs:

• Stack boundedness problem. Given an interrupt-driven program, the stack boundedness problem asks if the stack size is bounded by a finite constant. More precisely, given a program p, the stack boundedness problem returns "yes" if there exists a finite integer K such that on all executions of

A preliminary version of this paper appeared in the Proceedings of the Static Analysis Symposium (SAS 2003), Lecture Notes in Computer Science 2694, Springer-Verlag, pages 109–126, 2003.

^{*} Corresponding Author

the program p, the stack size never grows beyond K, and "no" if no such K exists.

• Exact maximum stack size problem. Given an interrupt-driven program, the exact maximum stack size problem asks for the maximum possible stack size. More precisely, given a program p, the exact maximum stack size problem returns an integer K such that for all executions of the program p, the stack size never grows beyond K, and such that there is a possible schedule of interrupts and an execution of the program p such that the stack size becomes K; the problem returns ∞ if there is an execution where the stack can grow unbounded.

We model interrupt-driven programs in the untyped *interrupt calculus* of Palsberg and Ma [4]. The interrupt calculus contains essential constructs for programming interrupt-driven systems. For example, we have found that the calculus can express the core aspects of seven commercial micro-controllers from Greenhill Manufacturing Ltd. A program in the calculus consists of a main part and some interrupt handlers. In the spirit of such processors as the Intel MCS-51 family (8051, etc.), Motorola Dragonball (68000 family), and Zilog Z86, the interrupt calculus supports an interrupt mask register (imr). An imr value consists of a master bit and one bit for each interrupt source. For example, the Motorola Dragonball processor can handle 22 interrupt sources. An interrupt handler is enabled, if *both* the master bit and the bit for that interrupt handler is set. When an interrupt handler is called, a return address is stored on the stack, and the master bit is automatically turned off. At the time of return, the master bit is turned back on (however, the handler can turn the master bit on at any point). A program execution has access to:

- the interrupt mask register, which can be updated during computation,
- a stack for storing return addresses, and
- a memory of integer variables; output is done via memory-mapped I/O.

Each element on the stack is a return address. When we measure the size of the stack, we simply count the number of elements on the stack. Our analysis is approximate: when doing the analysis, we ignore the memory of integer variables and the program statements that manipulate this memory. In particular, we assume that both branches of a conditional depending on the memory state can be taken. Of course, all the problems analyzed in this paper become undecidable if integer variables are considered in the analysis, since we can then easily encode two-counter machines.

We consider three versions of Palsberg and Ma's interrupt calculus, here presented in increasing order of generality:

• Monotonic programs. These are interrupt calculus programs that satisfy the following monotonicity restriction: when a handler is called with an imr

Calculus	Problem	Complexity	Reference
Monotonic	Stack boundedness	NLOGSPACE-complete	Theorem 7
	Exact maximum stack size	PSPACE-complete	Theorems 13,25
Monotonic	Stack boundedness	PSPACE-complete	Theorems 22,25
(enriched)	Exact maximum stack size	PSPACE-complete	Theorems 13,25
Enriched	Stack boundedness	PSPACE-hard, EXPTIME	Theorems 22,30
	Exact maximum stack size	PSPACE-hard, EXPTIME	Theorems 13,30

Table 1

Complexity results

value imr_b , then it returns with an imr value imr_r such that $imr_r \leq imr_b$, where \leq is the logical bitwise implication ordering. In other words, every interrupt that is enabled upon return of a handler must have been enabled when the handler was called (but could have possibly been disabled during the execution of the handler).

- Monotonic enriched programs. This calculus enriches Palsberg and Ma's calculus with conditionals on the interrupt mask register. The monotonicity restriction from above is retained.
- Enriched programs. These are programs in the enriched calculus, without the monotonicity restriction.

We summarize our results in Table 1. We have determined the complexity of stack boundedness and exact maximum stack size both for monotonic programs and for monotonic programs enriched with tests. For general programs enriched with tests, we have a PSPACE lower bound and an EXPTIME upper bound for both problems; tightening this gap remains an open problem. While the complexities are high, our algorithms are *polynomial* (linear or cubic) in the size of the program, and exponential only in the number of interrupts. In other words, our algorithms are polynomial if the number of interrupts is fixed. Since most real systems have a fixed small number of interrupts (for example Motorola Dragonball processor handles 22 interrupt sources), and the size of programs is the limiting factor, we believe the algorithms should be tractable in practice. Experiments are needed to settle this.

We reduce the stack boundedness and exact stack size problems to state space exploration problems over certain graphs constructed from the interruptdriven program. We then use the structure of the graph to provide algorithms for the two problems. Our first insight is that for monotonic programs, the maximum stack bounds are attained without any intermediate handler return. The polynomial-time algorithm for monotonic programs is reduced to searching for cycles in a graph; the polynomial-space algorithm for determining the exact maximum stack size of monotonic enriched programs is based on finding the longest path in a (possibly exponential) acyclic graph. Finally, we can reduce the stack boundedness problem and exact maximum stack size problem for enriched programs to finding context-free cycles and context-free longest paths in graphs. Our EXPTIME algorithm for enriched programs is based on a novel technique to find the longest context-free path in a DAG. Our lower bounds are obtained by reductions from reachability in a DAG (which is NLOGSPACE-complete), satisfiability of quantified boolean formulas (which is PSPACE-complete), and reachability for polynomial-space Turing Machines (which is PSPACE-complete). We also provide algorithms that determine, given an interrupt-driven program, whether it is monotonic. In the nonenriched case, monotonicity can be checked in polynomial time (NLOGSPACE); in the enriched case, in co-NP. In Section 2, we recall the interrupt calculus of Palsberg and Ma [4]. In Section 3, we consider monotonic programs, in Section 4, we consider monotonic enriched programs, and in Section 5, we consider enriched programs without the monotonicity restriction.

1.1 Related Work

Brylow, Damgaard, and Palsberg [1] do stack size analysis of a suite of microcontroller programs by running a context-free reachability algorithm for model checking. They use, essentially, the same abstraction that our EXPTIME algorithm uses for enriched programs. Our paper gives more algorithmic details and clarifies that the complexity is exponential in the number of handlers.

Palsberg and Ma [4] present a type system and a type checking algorithm for the interrupt calculus that guarantees stack boundedness and certifies that the stack size is within a given bound. Each type contains information about the stack size and serves as documentation of the program. However, this requires extensive annotations from the programmer (especially since the types can be exponential in the number of handlers), and the required type information is absent in legacy programs. Our work can be seen as related to *type inference* for the interrupt calculus. In particular, we check stack properties of programs without annotations. From our algorithms, we should be able to infer the types of [4]. It remains to be seen whether our algorithms can be as successful on legacy programs as the algorithm of Brylow, Damgaard, and Palsberg [1].

Regehr, Reid, and Webb [8] integrated a stack size analysis in the style of [1] with aggressive abstract interpretation of ALU operations and conditional branches, and they showed how global function inlining can significantly decrease the stack space requirements.

Hughes, Pareto, and Sabry [3,7] use *sized types* to reason about liveness, termination, and space boundedness of reactive systems. However, they require types with explicit space information, and do not address interrupt handling. Wan, Taha, and Hudak [11] present event-driven Functional Reactive Programming (FRP), which is designed such that the time and space behavior of a program are necessarily bounded. However, the event-driven FRP programs are written in continuation-style, and therefore do not need a stack. Hence stack boundedness is not among the resource issues considered by Wan et al.

Context free reachability has been used in interprocedural program analysis [9]. Recently, Reps, Schwoon, and Jha [10] consider context-free reachability on weighted pushdown graphs to check security properties.

Hillebrand, Kanellakis, Mairson, Vardi [2] studied a boundedness problem that is related to ours, namely whether the depth of recursion of a given Datalog program is *independent* of the input. They showed that several variations of the problem are undecidable. Our boundedness problem focuses on whether the "depth of recursion" (that is, stack size) is finite under *all* possible inputs. The two boundedness problems are different. The problem studied by Hillebrand et al. is to decide whether the depth of recursion is always the same. Our problem is about whether the "depth of recursion" is bounded across all inputs. Note that our problem will allow the "depth of recursion" to vary with the input, as long as there is a common bound that works for all inputs. A further difference is that the *input* to a Datalog program is a finite database, while the "input" to an interrupt-calculus program is an infinite stream of interrupts.

2 The Interrupt Calculus

2.1 Syntax

We recall the (abstract) syntax of the interrupt calculus of [4]. We use x to range over a set of program variables, we use *imr* to range over bit strings, and we use c to range over integer constants.

(program) $p ::= (m, \overline{h})$ (main) $m ::= \text{loop } s \mid s ; m$ (handler) $h ::= \text{iret} \mid s ; h$ (statement) $s ::= x = e \mid \text{imr} = \text{imr} \land imr \mid \text{imr} = \text{imr} \lor imr \mid$ $\text{if0}(x) s_1 \text{ else } s_2 \mid s_1 ; s_2 \mid \text{skip}$ (expression) $e ::= c \mid x \mid x + c \mid x_1 + x_2$

The pair p = (m, h) is an *interrupt program* with main program m and interrupt handlers \bar{h} . The over-bar notation \bar{h} denotes a sequence $h_1 \dots h_n$ of

handlers. We use the notation $\overline{h}(i) = h_i$. We use a to range over m and h.

2.2 Semantics

We use R to denote a *store*, that is, a partial function mapping program variables to integers. We use σ to denote a *stack* generated by the grammar $\sigma ::= \mathsf{nil} \mid a :: \sigma$. We define the size of a stack as $|\mathsf{nil}| = 0$ and $|a :: \sigma| = 1 + |\sigma|$.

We represent the imr as a bit sequence $imr = b_0b_1 \dots b_n$, where $b_i \in \{0, 1\}$. The 0th bit b_0 is the master bit, and for i > 0, the *i*th bit b_i is the bit for interrupts from source *i*, which are handled by handler *i*. Notice that the master bit is the most significant bit, the bit for handler 1 is the second-most significant bit, and so on. This layout is different from some processors, but it simplifies the notation used later. For example, the imr value 101b means that the master bit is set, the bit for handler 1 is not set, and the bit for handler 2 is set. We use the notation imr(i) for bit b_i . The predicate *enabled* is defined as

$$enabled(imr, i) = (imr(0) = 1) \land (imr(i) = 1), \qquad i \in 1..n.$$

We use 0 to denote the imr value where all bits are 0. We use t_i to denote the imr value where all bits are 0's except that the *i*th bit is set to 1. We will use \land to denote bitwise logical conjunction, \lor to denote bitwise logical disjunction, \leq to denote bitwise logical implication, and \neg to denote bitwise logical negation. Notice that $enabled(t_0 \lor t_i, j)$ is true if i = j, and false otherwise. The imr values, ordered by \leq , form a lattice with bottom element 0.

A program state is a tuple $\langle \bar{h}, R, imr, \sigma, a \rangle$ consisting of interrupt handlers \bar{h} , a store R, an interrupt mask register imr, a stack σ of return addresses, and a program counter a. We refer to a as the current statement; it models the instruction pointer of a CPU. The interrupt handlers \bar{h} do not change during computation; they are part of the state to ensure that all names are defined locally in the semantics below. We use P to range over program states. If $P = \langle \bar{h}, R, imr, \sigma, a \rangle$, then we use the notation $P.stk = \sigma$. For $p = (m, \bar{h})$, the initial program state for executing p is $P_p = \langle \bar{h}, \lambda x.0, 0, \mathsf{nil}, m \rangle$, where the function $\lambda x.0$ is defined on the variables that are used in the program p.

A small-step operational semantics for the language is given by the reflexive,

transitive closure of the relation \rightarrow on program states:

$$\langle \bar{h}, R, imr, \sigma, a \rangle \to \langle \bar{h}, R, imr \land \neg t_0, a :: \sigma, \bar{h}(i) \rangle$$
 (1)

if enabled(imr, i)

$$\langle \bar{h}, R, imr, a :: \sigma', iret \rangle \rightarrow \langle \bar{h}, R, imr \lor t_0, \sigma', a \rangle$$
 (2)

$$\langle \bar{h}, R, imr, \sigma, \mathsf{loop} \ s \rangle \to \langle \bar{h}, R, imr, \sigma, s; \mathsf{loop} \ s \rangle$$
 (3)

$$\langle \bar{h}, R, imr, \sigma, x = e; a \rangle \rightarrow \langle \bar{h}, R\{x \mapsto eval_R(e)\}, imr, \sigma, a \rangle$$
 (4)

$$, R, imr, \sigma, \mathsf{imr} = \mathsf{imr} \land imr'; a \rangle \rightarrow \langle \bar{h}, R, imr \land imr', \sigma, a \rangle$$

$$(5)$$

$$\bar{h}, R, imr, \sigma, imr = imr \lor imr'; a
angle \to \langle \bar{h}, R, imr \lor imr', \sigma, a
angle$$
(6)

$$h, R, imr, \sigma, (\text{if0}(x) \ s_1 \text{ else } s_2); a \rangle \rightarrow \langle h, R, imr, \sigma, s_1; a \rangle \text{ if } R(x) = 0$$
 (7)

$$\bar{h}, R, imr, \sigma, (if0 (x) s_1 else s_2); a \rightarrow \langle \bar{h}, R, imr, \sigma, s_2; a \rangle$$
 if $R(x) \neq 0$ (8)

$$\langle \bar{h}, R, imr, \sigma, \mathsf{skip}; a \rangle \to \langle \bar{h}, R, imr, \sigma, a \rangle$$
 (9)

where the function $eval_R(e)$ is defined as:

$$eval_R(c) = c \qquad eval_R(x+c) = R(x) + c$$
$$eval_R(x) = R(x) \qquad eval_R(x_1+x_2) = R(x_1) + R(x_2).$$

Rule (1) models that if an interrupt is enabled, then it may occur. The rule says that if enabled(imr, i), then it is a possible transition to push the current statement on the stack, make $\bar{h}(i)$ the current statement, and turn off the master bit in the imr. Notice that we make no assumptions about the interrupt arrivals; any enabled interrupt can occur at any time, and conversely, no interrupt has to occur. Rule (2) models interrupt return. The rule says that to return from an interrupt, remove the top element of the stack, make the removed top element the current statement, and turn on the master bit. Rule (3) is an unfolding rule for loops. It implies that interrupt calculus programs do not terminate, a feature common in reactive systems. Rules (4)–(9) are standard rules for statements. Let \rightarrow^* denote the reflexive transitive closure of \rightarrow .

A program execution is a sequence $P_p \to P_1 \to P_2 \to \cdots \to P_k$ of program states. Consider a program execution γ of the form $P_p \to^* P_i \to P_{i+1} \to^* P_j \to P_{j+1}$ with $P_i = \langle \bar{h}, R, imr_b, \sigma, a \rangle$ and $P_j = \langle \bar{h}, R', imr', \sigma', a' \rangle$. The handler $\bar{h}(i)$ is called in γ with imr_b from state P_i and returns with imr_r from state P_j if

$$P_i \to P_{i+1} = \langle \bar{h}, R, imr_b \land \neg t_0, a :: \sigma, \bar{h}(i) \rangle \quad \text{and } enabled(imr_b, i),$$
$$P_j \to P_{j+1} = \langle \bar{h}, R', imr_r, \sigma, a \rangle \quad \text{and} \quad \sigma' = a :: \sigma,$$

```
handler 2 {
imr = imr or 111b
                                 imr = imr and 110b
loop { imr = imr or 111b }
                                 imr = imr or
                                                010b
handler 1 {
                                 imr = imr or
                                                100b
   imr = imr and 101b
                                 imr = imr and 101b
   imr = imr or
                100b
                                 iret
   iret
                              }
}
```



and $P_k.stk \neq \sigma$ for all $i < k \leq j$. We say that there is no handler call in γ between P_i and P_j if for all $i \leq k < j$, the transition $P_k \to P_{k+1}$ is not a transition of the form (1). Similarly, given an execution $P_p \to^* P_i \to^* P_j$, there is no handler return between P_i and P_j if for all $i \leq k < j$, the transition $P_k \to P_{k+1}$ is not a transition of the form (2).

2.3 Stack Size Analysis

We consider the following problems of stack size analysis.

- Stack boundedness problem Given an interrupt program p, the stack boundedness problem returns "yes" if there exists a finite integer K such that for all program states P', if $P_p \to^* P'$, then $|P'.stk| \leq K$; and returns "no" if there is no such K.
- Exact maximum stack size problem For a program state P we define maxStackSize(P) as the least $K \ge 0$ such that for all P', if $P \to P'$, then $|P'.stk| \le K$; and "infinite" in case no such K exists. The exact maximum stack size problem is given an interrupt program p and returns $maxStackSize(P_p)$.

Figure 1 shows an example of a program in the real interrupt calculus syntax, where " \wedge " and " \vee " are represented by "and" and "or" respectively. The bit sequences such as 111b are imr constants. Notice that each of the two handlers can be called from different program points with different imr values. The bodies of the two handlers manipulate the imr, and both are at some point during the execution open to the possibility of being interrupted by the other handler. However, the maximum stack size is 3. This stack size happens if handler 1 is first called with 111b, then handler 2 with 101b, and then handler 1 again with 110b, at which time there are three return addresses on the stack.

We shall analyze interrupt programs under the usual program analysis assumption that all paths in the program are executable. More precisely, our analysis assumes that each data assignment statement x = e in the program has been replaced by skip, and each conditional if $0(x) s_1$ else s_2 has been replaced by if 0 (*) s_1 else s_2 , where * denotes nondeterministic choice. While this is an overapproximation of the actual set of executable paths, we avoid trivial undecidability results for deciding if a program path is actually executable. In the following, we assume that the relation \rightarrow is defined on this abstract program.

3 Monotonic Interrupt Programs

We first define monotonic interrupt programs and then analyze the stack boundedness and exact maximum stack size problems for such programs. A handler h_i of program p is monotonic if for every execution γ of p, if h_i is called in γ with an imr value imr_b and returns with an imr value imr_r , then $imr_r \leq imr_b$. The program p is monotonic if all handlers $h_1 \dots h_n$ of p are monotonic. The handler h_i of p is monotonic in isolation if for every execution γ of p, if h_i is called in γ with an imr value imr_b from a state P_i and returns with an imr value imr_r from a state P_j such that there is no handler call between P_i and P_j , then $imr_r \leq imr_b$.

We first show that a program $p = (m, \bar{h})$ is monotonic iff every handler $h_i \in \bar{h}$ is monotonic in isolation. Moreover, a handler h_i is monotonic in isolation iff, whenever h_i is called with imr value $t_0 \vee t_i$ from state P_i and returns with imr_r from state P_j , with no handler calls between P_i and P_j , then $imr_r \leq t_0 \vee t_i$. These observations can be used to efficiently check if an interrupt program is monotonic: for each handler, we check that the return value imr_r of the imr when called with $t_0 \vee t_i$ satisfies $imr_r \leq t_0 \vee t_i$.

Lemma 1 A program $p = (m, \bar{h})$ is monotonic iff every handler $h_i \in \bar{h}$ is monotonic in isolation.

Proof. If there is a handler h which violates monotonicity in isolation then h is not monotonic and hence the program p is not monotonic. For the converse, suppose that all handlers are monotonic in isolation, but the program p is not monotonic. Consider an execution sequence γ which violates the monotonicy condition. In γ , we can choose a handler h which is called with an imr value imr_b and returns with an imr value imr_r such that

$$imr_b \geq imr_r$$
 (10)

but any handler h' which was called from within h with an imr value $imr_{b_{h'}}$ returned with an imr value $imr_{r_{h'}}$ satisfying $imr_{r_{h'}} \leq imr_{b_{h'}}$. From γ we now construct a simpler execution sequence γ' which also violates the monotonicity condition. We construct γ' by omitting from γ all calls from within h. In γ' there are no calls between the call to h and the return of h. Each of the omitted

calls are monotonic, so in γ' h will return with an imr value imr'_r such that

$$imr_r \le imr'_r \tag{11}$$

Since in this sequence no handler is called from h and h is monotonic in isolation it follows that:

$$imr'_r \le imr_b$$
 (12)

From (10), (11), and (12), we have a contradiction. Hence p is monotonic.

Lemma 2 Given a program $p = (m, \bar{h})$, a handler $h_i \in \bar{h}$ is monotonic in isolation iff when h_i is called with imr value $t_0 \vee t_i$ from program state P_i , and returns with imr value imr_r from program state P_j , with no handler calls between P_i and P_j , then $imr_r \leq t_0 \vee t_i$.

Proof. If the right-hand side of the "iff" is not satisfied, then h_i is not monotonic in isolation. Conversely, suppose the right-hand side of the "iff" is satisfied but h_i is not monotonic in isolation. Suppose further that h_i is called with the imr value imr_b and it follows some sequence of execution in the handler h_i to return imr'_r , with $imr_b \geq imr'_r$. Hence there is a bit j such that the j-th bit is on in imr'_r but the j-th bit is off in imr_b . Since the conditionals do not depend on imr, the same sequence of execution can be followed when the handler is called with $t_0 \vee t_i$. In this case, the return value imr_r will have the j-th bit on, and hence $t_0 \vee t_i \geq imr_r$. This is a contradiction.

Proposition 3 It can be checked in linear time (NLOGSPACE) if an interrupt program is monotonic.

Proof. It follows from Lemma 1 that checking monotonicity of a program p can be achieved by checking monotonicity of the handlers in isolation. It follows from Lemma 2 that checking monotonicity in isolation for a handler h_i can be achieved by checking if h_i is monotonic when called with t_i . Thus checking monotonicity is just checking the return value of the imr when called with t_i . This can be achieved in polynomial time by a standard bitvector dataflow analysis. Since the conditionals do not test the value of imr, we can join the dataflow information (i.e., bits of the imr) at merge points. It is clear that finding bit by bit the return value when called with t_i can be achieved in NLOGSPACE.

3.1 Stack Boundedness

We now analyze the complexity of stack boundedness of monotonic programs. Our main insight is that the maximum stack size is achieved without any intermediate handler returns. First observe that if handler h is enabled when the imr is imr_1 , then it is enabled for all imr $imr_2 \ge imr_1$. We argue the case where the maximum stack size is finite, the same argument can be formalized in case the maximum stack size is infinite. Fix an execution sequence that achieves the maximum stack size. Let h be the last handler that returned in this sequence (if there is no such h then we are done). Let the sequence of statements executed be $s_0, s_1, \ldots, s_{i-1}, s_i, \ldots, s_j, s_{j+1}, \ldots$ where s_i was the starting statement of h and s_j the iret statement of h. Suppose h was called with imr_b and returned with imr_r such that $imr_r \leq imr_b$. Consider the execution sequence of statements $s_0, s_1, \ldots, s_{i-1}, s_{j+1}, \ldots$ with the execution of handler hbeing omitted. In the first execution sequence the imr value while executing statement s_{j+1} is imr_r and in the second sequence of statements and same sequence of calls to handlers with h omitted gives the same stack size. Following a similar argument, we can show that all handlers that return intermediately can be omitted without changing the maximum stack size attained.

Lemma 4 For a monotonic program p, let P_{\max} be a program state such that $P_p \rightarrow^* P_{\max}$ and for any state P', if $P_p \rightarrow^* P'$ then $|P_{\max}.stk| \ge |P'.stk|$. Then there is a program state P'' such that $P_p \rightarrow^* P''$, $|P''.stk| = |P_{\max}.stk|$, and there is no handler return between P_p and P''.

We now give a polynomial-time algorithm for the stack boundedness problem for monotonic programs. The algorithm reduces the stack boundedness question to the presence of cycles in the *enabled graph* of a program. Let $h_1 \ldots h_n$ be the *n* handlers of the program. Given the code of the handlers, we build the enabled graph $G = \langle V, E \rangle$ as follows.

- There is a node for each handler, i.e., $V = \{h_1, h_2, \dots, h_n\}$.
- Let the instructions of h_i be $C_i = i_1, i_2, \ldots i_m$. There is an edge between (h_i, h_j) if any of the following conditions holds.
- (1) There is l, k such that $l \leq k$, the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \lor imr$ with $t_0 \leq imr$, the instruction at i_k is $\mathsf{imr} = \mathsf{imr} \lor imr$ with $t_j \leq imr$ and for all statements i_m between i_l and i_k , if i_m is $\mathsf{imr} = \mathsf{imr} \land imr$ then $t_0 \leq imr$.
- (2) There is l, k such that $l \leq k$, the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \lor imr$ with $t_j \leq imr$, the instruction at i_k is $\mathsf{imr} = \mathsf{imr} \lor imr$ with $t_0 \leq imr$ and for all statements i_m between i_l and i_k , if i_m is $\mathsf{imr} = \mathsf{imr} \land imr$ then $t_j \leq imr$.
- (3) We have i = j and there is l such that the instruction at i_l is $\mathsf{imr} = \mathsf{imr} \lor imr$ with $t_0 \le imr$ and for all statements i_m between i_1 and i_l , if i_m is $\mathsf{imr} = \mathsf{imr} \land imr$ then $t_i \le imr$. This gives a self-loop (h_i, h_i) .

Since we do not model the program variables, we can analyze the code of h_i and detect all outgoing edges (h_i, h_j) in time linear in the length of h_i . We only need to check that there is an \vee statement with an imr constant with *j*th bit 1 and then the master bit is turned on with no intermediate disabling of the *j*th bit or vice versa. Hence the enabled graph for program *p* can be constructed in time $n^2 \times |p|$ (where |p| denotes the length of *p*). **Lemma 5** Let G_p be the enabled graph for a monotonic interrupt program p. If G_p has a cycle, then the stack is unbounded, that is, for all positive integers K, there is a program state P' such that $P_p \rightarrow^* P'$ and |P'.stk| > K.

Proof. Consider a cycle $C = \langle h_{i_1}, h_{i_2}, \ldots, h_{i_k}, h_{i_1} \rangle$ such that for any two consecutive nodes in the cycle there is an edge between them in G_p . Consider the following execution sequence. When h_{i_1} is executed, it turns h_{i_2} and the master bit on. Then, an interrupt of type h_{i_2} occurs. When h_{i_2} is executed, it turns on h_{i_3} and the master bit. Then, an interrupt of type h_{i_3} occurs, and so on. Hence h_{i_1} can be called with h_{i_1} on stack and the sequence of calls can be repeated. If there is a self-loop at the node h_i , then h_i can occur infinitely many times. This is because handler h_i can turn the master bit on without disabling itself, so an infinite sequence of interrupts of type h_i will make the stack grow unbounded.

Since cycles in the enabled graph can be found in NLOGSPACE, the stack boundedness problem for monotonic programs is in NLOGSPACE. Note that the enabled graph of a program can be generated on the fly in logarithmic space. Hardness for NLOGSPACE follows from the hardness of DAG reachability.

Lemma 6 Stack Boundedness for monotonic interrupt programs is NLOGSPACE-hard.

Proof. We reduce reachability in a DAG to the Stack Boundedness checking problem. Given a DAG G = (V, E) where $V = \{1, 2, ..., n\}$ we write a program p with n handlers $h_1, h_2, ..., h_n$ as follows:

- The code of handler h_i disables all handlers and then enables all its successors in the DAG and the master bit.
- the handler h_n disables all the other handlers and enables itself and the master bit and then disables itself.

Hence the enabled graph of the program will be a DAG with only the node n with a self-loop. So the stack size is bounded iff n is not reachable. Hence stack boundedness checking is NLOGSPACE-hard.

Theorem 7 Stack boundedness for monotonic interrupt programs can be checked in time linear in the size of the program and quadratic in the number of handlers. The complexity of stack boundedness for monotonic interrupt programs is NLOGSPACE-complete.

In case the stack is bounded, we can get a simple upper bound on the stack size as follows. Let G_p be the enabled graph for a monotonic interrupt program p. If G_p is a DAG, and the node h_i of G_p has order k in topological sorting order, then we can prove by induction that the corresponding handler h_i of p can occur at most $2^{(k-1)}$ times in the stack.

Lemma 8 Let G_p be the enabled graph for a monotonic interrupt driven program p. If G_p is a DAG, and the node h_i of G_p has order k in topological sorting order, then the corresponding handler h_i of p can occur at most $2^{(k-1)}$ times in the stack.

Proof. We prove this by induction. Let h_i be the node with order 1. It has no predecessors in the enabled graph. No node in the enabled graph has a self-loop, since our assumption is that the enabled graph G_p is a DAG. Hence h_i must turn its bit off before turning the master bit on. Hence when h_i occurs in the stack its bit is turned off. As no other handler turns it on when the master bit is on (since otherwise there would have been an edge to h_i) it cannot occur more than once in the stack. This proves the base case.

Consider a node h with order k. By hypothesis, all nodes with order j where $j \leq k-1$ can occur at most $2^{(j-1)}$ times in the stack. Now when the node h occurs in the stack its bit is turned off. So before it occurs again in the stack, one of the predecessors of h must occur in the stack. Hence the number of times h can occur in the stack is given by

 $1 + \sum$ Number of times its predecessors can occur $\leq 1 + \sum_{i=1}^{k-1} 2^{(i-1)}$ $= 2^{(k-1)}.$

1			

We get the following bound as an immediate corollary of Lemma 8. Let $p = (m, \bar{h})$ be a monotonic interrupt driven program with n handlers, and with enabled graph G_p . If G_p is a DAG, then for any program state P' such that $P_p \to^* P'$, we have $|P'.stk| \leq 2^n - 1$. This is because the maximum length of the stack is given by the sum of the number of times a individual handler can be in the stack. By Lemma 8 we know a node with order j can occur at most 2^{j-1} times. Hence the maximum length of the stack is given by

$$\sum_{i=1}^{n} 2^{(i-1)} = 2^n - 1$$

In fact, this bound is tight: there is a program with n handlers that achieves a maximum stack size of $2^n - 1$. We show that starting with an imr value of all 1's one can achieve the maximum stack length of $2^n - 1$ while keeping the stack bounded. We give an inductive strategy to achieve this. With one handler which does not turn itself on we can have a stack length 1 starting with imr value 11. By induction hypothesis, using n - 1 handlers starting with

Algorithm 1 Function MaxStackLengthBound

Input: An interrupt program p
Output: If the stack size is unbounded then ∞, else an upper bound on the maximum stack size
1. Build the Enabled Graph G from the Program p
2. If G has a cycle then the Maximum Stack Length is ∞.
3. If G is a DAG then topologically sort and order nodes of G
4.1 For i ← 1to |V[G]|
4.2 For a node h with order i
N[h]= 1 + ∑ N[h_j] where h_j is a predecessor of h

5. Upper Bound on Maximum Length of $\text{Stack}=\sum N[h]$ for all handlers h

imr value all 1's we can achieve a stack length of $2^{n-1} - 1$. Now we add the *n*th handler and modify the previous n - 1 handlers such that they do not change the bit for the *n*th handler. The *n*-th handler turns on every bit except itself, and then turns on the master bit. The following sequence achieves a stack size of $2^n - 1$. First, the first n - 1 handlers achieve a stack size of $2^{n-1} - 1$ using the inductive strategy. After this, the *n*th handler is called. It enables the n - 1 handlers but disables itself. Hence the sequence of stack of $2^{n-1} - 1$ can be repeated twice and the *n* the handler can occur once in the stack in between. The total length of stack is thus $1 + (2^{n-1} - 1) + (2^{n-1} - 1) = 2^n - 1$. Since none of the other handlers can turn the *n*th handler on, the stack size is in fact bounded.

We now give a polynomial time procedure to give an upper bound on the stack size if it is bounded. If the stack can possibly grow unbounded we report infinite. If the stack is bounded we compute an upper bound N[h] on the number of times a handler h can occur in the stack. The algorithm MaxStackLength-Bound is shown in Algorithm 1.

Lemma 9 Function MaxStackLengthBound correctly checks the stack boundedness of interrupt driven programs, that is, if MaxStackLengthBound returns ∞ then there is some execution of the program that causes the stack to be unbounded. It also gives an upper bound on the number of times a handler can occur in the stack, that is, if MaxStackBound(p) is N, then the maximum stack size on any execution sequence of p is bounded above by N.

Proof. It follows from Lemma 5 that if the enabled graph has a cycle then the stack can grow unbounded. This is achieved by Step 2 of MaxStackLength-Bound. It follows from Lemma 8 that the maximum number of times a handler can occur in the stack is one plus the sum of the number of times its predecessors can occur. This is achieved in Step 4 of MaxStackLengthBound. ■

Lemma 10 Function MaxStackLengthBound runs in time polynomial in size

of the interrupt driven program.

Proof. The enabled graph can be built in time $h^2 \times PC$ where h is the number of handlers and PC is the number of program statements. Steps 2,3 and 4 can be achieved in time linear in the size of the enabled graph, and hence in time linear in the size of the program.

While MaxStackLengthBound is simple, one can construct simple examples to show that it may exponentially overestimate the upper bound on the stack size. We show next that this is no accident: we now prove that the exact maximum stack size problem problem is PSPACE-hard. There is a matching upper bound: the exact maximum stack size problem can be solved in PSPACE. We defer this algorithm to Section 4, where we solve the problem for a more general class of programs.

3.2 Maximum Stack Size

We now prove that the exact maximum stack size problem is PSPACE-hard. We start with a little warm-up: first we show that the problem is both NP-hard and co-NP hard. We show this by showing that the problem is DP-hard, where DP is the class of all languages L such that $L = L_1 \cap L_2$ for some language L_1 in NP and some language L_2 in co-NP [6] (note that DP is not the class NP \cap co-NP [5,6]). We reduce the problem of EXACT-MAX Independent Set of a Graph and its complement to the problem of finding the exact maximum size of the stack of programs in interrupt calculus. The EXACT-MAX IND problem is the following:

EXACT-MAX IND = { $\langle G, k \rangle$ the size of the maximum independent set is k}

EXACT-MAX IND is DP-complete [5]. Given an undirected graph $G = \langle V, E \rangle$ where $V = \{1, 2, ..., n\}$, we construct an interrupt-driven program as follows. We create a handler h_i for every node *i*. Let $N_i = \{j : (i, j) \in E\}$ be the neighbors of node *i* in *G*. The code of h_i disables itself and all the handlers of N_i and then turns the master bit on. The main program enables all handlers and then enters an empty loop. Consider the maximum stack size and the handlers in it. First observe that once a handler is disabled, it is never reenabled. Hence, no handler can occur twice in the stack, as every handler disables itself and no other handler turns it on. Let h_i and h_j be two handlers in the stack such that h_i occurs before h_j . Then $(i, j) \notin E$, since if $(i, j) \in E$ then h_i would have turned h_j off, and thus h_j could not have occurred in the stack (since h_j is never re-enabled). Hence if we take all the handlers that occur in the stack the corresponding nodes in the graph form an independent set. Consider an independent set *I* in *G*. All the handlers corresponding to the nodes in *I* can occur in the stack as none of these handlers is disabled by any other. We have thus proved given a stack with handlers we can construct an independent set of size equal to the size of the stack. Conversely, given an independent set we can construct a stack size equal to the size of the independent set. Hence the EXACT-MAX IND problem can be reduced to the problem of finding the exact maximum size of the stack. Hence the problem of finding exact stack size in DP-hard. It follows that it is NP-hard and co-NP hard.

We now give the proof of PSPACE-hardness, which is considerably more technical. We define a subclass of monotonic interrupt calculus which we call simple interrupt calculus and show the exact maximum stack size problem is already PSPACE-hard for this class. It follows that exact maximum stack size is PSPACE-hard for monotonic interrupt-driven programs.

For imr', imr'' where imr'(0) = 0 and imr''(0) = 0, define $\mathcal{H}(imr'; imr'')$ to be the interrupt handler

$$\begin{split} & \operatorname{imr} = \operatorname{imr} \wedge \neg \operatorname{imr}'; \\ & \operatorname{imr} = \operatorname{imr} \lor (t_0 \lor \operatorname{imr}''); \\ & \operatorname{imr} = \operatorname{imr} \land \neg (t_0 \lor \operatorname{imr}''); \\ & \operatorname{iret.} \end{split}$$

A *simple interrupt calculus program* is an interrupt calculus program where the main program is of the form

```
\operatorname{imr} = \operatorname{imr} \lor (\operatorname{imr}_S \lor t_0);
loop skip
```

where $imr_S(0) = 0$ and every interrupt handler is of the form $\mathcal{H}(imr'; imr'')$. Intuitively, a handler of a simple interrupt calculus program first disables some handlers, then enables other handlers and enables interrupt handling. This opens the door to the handler being interrupted by other handlers. After that, it disables interrupt handling, and makes sure that the handlers that are enabled on exit are a subset of those that were enabled on entry to the handler.

For a handler h_i of the form $\mathcal{H}(imr'; imr'')$, we define function $f_i(imr) = imr \wedge (\neg imr') \vee imr''$. Given a simple interrupt calculus program p, we define a directed graph G(p) = (V, E) such that

- $V = \{ imr \mid imr(0) = 0 \},\$
- $E = \{ (imr, f_i(imr), i) \mid t_i \leq imr \}$ is a set of labeled edges from imr to $f_i(imr)$ with label $i \in \{1..n\}$.

The edge $(imr, f_i(imr), i)$ in G(p) represents the call to the interrupt handler

h(i) when imr value is *imr*. We define imr_S as the start node of G(p) and we define $\mathcal{M}(imr)$ as the longest path in G(p) from node *imr*. The notation $\mathcal{M}(imr)$ is ambiguous because it leaves the graph unspecified; however, in all cases below, the graph in question can be inferred from the context.

Lemma 11 For a simple interrupt calculus program p, we have that

$$maxStackSize(P_p) = |\mathcal{M}(imr_S)|.$$

Proof. By definition, the state of a simple interrupt program p = (m, h) is of the form $\langle \bar{h}, 0, imr, \sigma, a \rangle$ and stack size of p increases whenever an interrupt is handled and we have state transition of the form

$$\langle \bar{h}, 0, imr, \sigma, a \rangle \to \langle \bar{h}, 0, imr \land \neg t_0, a :: \sigma, \bar{h}(i) \rangle$$
 if $imr \ge t_i \lor t_0$.

Let a_i , $i \in \{1..4\}$ represent the four statements in the body of an interrupt handler such that any handler is of the form $a_1; a_2; a_3; a_4$. By definition of simple interrupt program., the master bit is enabled only between a_2 and a_3 , where calls to other handlers may occur. Also, after a call to a interrupt handler returns, the imr value is always less than or equal to the imr value before the call. Thus, during a call to handler h_i with initial imr value equal to imr, the only possible states where interrupts maybe be handled are of the form $\langle \bar{h}, 0, imr', \sigma, a_3; a_4 \rangle$, where $imr' \leq f_i(imr)$. Then, we only need to examine state transitions of the following form to compute $maxStackSize(P_p)$:

$$\langle \bar{h}, 0, imr, \sigma, a \rangle \rightarrow^* \langle \bar{h}, 0, imr', a :: \sigma, a_3; a_4 \rangle,$$

where $imr' \leq f_i(imr) \ \forall i$, such that $t_i \lor t_0 \leq imr$.

Let $P = \langle \bar{h}, 0, imr, \sigma, a \rangle$ and $P' = \langle \bar{h}, 0, imr', \sigma, a \rangle$. By an easy induction on execution sequences, we have that $maxStackSize(P) \leq maxStackSize(P')$ if $imr \leq imr'$. Therefore, it is sufficient to consider state transitions of the form

$$\langle \bar{h}, 0, imr, \sigma, a \rangle \rightarrow^* \langle \bar{h}, 0, f_i(imr), a :: \sigma, a_3; a_4 \rangle,$$

where $imr \ge t_i \lor t_0$. In the main loop, the possible states where interrupts may be handled are of the form

 $\langle \bar{h}, 0, imr_S \lor t_0, \mathsf{nil}, \mathsf{loop skip} \rangle$ and $\langle \bar{h}, 0, imr_S \lor t_0, \mathsf{nil}, \mathsf{skip}; \mathsf{loop skip} \rangle$.

Let a_0 be the statements of the form loop skip or skip; loop skip. To compute $maxStackSize(P_p)$, we only need to consider transitions of the form

$$\langle \bar{h}, 0, imr_S \lor t_0, \sigma, a_0 \rangle \to^* \langle \bar{h}, 0, f_i(imr_S) \lor t_0, a_0 :: \sigma, a_3; a_4 \rangle,$$
where $imr_S \geq t_i$, and

$$\langle \bar{h}, 0, imr, \sigma, a_3; a_4 \rangle \rightarrow^* \langle \bar{h}, 0, f_j(imr), a_3; a_4 :: \sigma, a_3'; a_4' \rangle,$$

where $imr \geq t_j \lor t_0$.

It is now clear that we can just use $imr \wedge \neg t_0$ to represent states that we are interested in with starting states represented by imr_S . The above two kinds of transitions can be uniquely represented by edges of the form $(imr_S, f_i(imr_S), i), (imr \wedge \neg t_0, f_j(imr \wedge \neg t_0), j)$ in graph G(p). Therefore, $maxStackSize(P_p)$ is equal to the length of the longest path in G(p) from the start node imr_S .

Lemma 12 For a simple interrupt calculus program p, and a subgraph of G(p), we have that if $imr \leq imr_1 \vee imr_2$, then $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$.

Proof. The lemma follows from the following claim. If $imr \leq imr_1 \vee imr_2$, and P is a path from node imr to imr', then we can find a path P_1 from imr_1 to imr'_1 and a path P_2 from node imr_2 to imr'_2 such that $|P| = |P_1| + |P_2|$ and $imr' \leq imr'_1 \vee imr'_2$.

Given this claim, the lemma following from the following reasoning. We can apply the above claim to the situation with $\mathcal{M}(imr)$ as the path P from imrto 0. Since $|\mathcal{M}(imr_1)| \geq |P_1|$ and $|\mathcal{M}(imr_2)| \geq |P_2|$, we have $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$.

We now prove the claim. We proceed by induction on the length of P. The base case of |P| = 0 is trivially true. Suppose the claim is true for |P| = k and that P' is P appended with an edge to imr''. We need to prove the case of P'. Since P ends at imr', there exists $t_i \leq imr'$ such that $imr'' = f_i(imr')$. By the induction hypothesis, $t_i \leq imr' \leq imr'_1 \lor imr'_2$. Thus, there exists $a \in \{1, 2\}$ such that $t_i \leq imr'_a$. Suppose that $t_i \leq imr'_1$ (the case of $t_i \leq imr'_1$ is similar and is omitted). We can let P'_1 be P_1 appended with an edge to imr''_1 where $imr''_1 = f_i(imr'_1)$. By definition of f_i , we have $f_i(imr') \leq f_i(imr'_1) \lor imr'_2$. Thus, we have $|P'| = |P| + 1 = |P_1| + 1 + |P_2| = |P'_1| + |P_2|$ and $imr'' \leq imr''_1 \lor imr'_2$.

We now show PSPACE-hardness for simple interrupt calculus. Our proof is based on a polynomial-time reduction from the *quantified boolean satisfiability* (QSAT) problem [5].

We first illustrate our reduction by a small example. Suppose we are given a QSAT instance $S = \exists x_2 \forall x_1 \phi$ with

$$\phi = (\mathsf{I}_{11} \lor \mathsf{I}_{12}) \land (\mathsf{I}_{21} \lor \mathsf{I}_{22}) = (\mathsf{x}_2 \lor \neg \mathsf{x}_1) \land (\mathsf{x}_2 \lor \mathsf{x}_1).$$



Fig. 2. Enable relation of interrupt handlers

We construct a simple interrupt program $p = (m, \bar{h})$ with an imr register, where $\bar{h} = \{h(x_i), h(\bar{x}_i), h(w_i), h(\bar{w}_i), h(l_{ij}) \mid i, j = 1, 2\}$ are 12 handlers. The imr contains 13 bits: a master bit, and each remaining bit 1-1 maps to each handler in \bar{h} . Let $\mathcal{D} = \{x_i, \bar{x}_i, w_i, \bar{w}_i, l_{ij} \mid i, j = 1, 2\}$. We use t_x , where $x \in \mathcal{D}$, to denote the imr value where all bits are 0's except the bit corresponding to handler h(x) is set to 1. The initial imr value imr_S is set to $imr_S = t_{x_2} \vee t_{\bar{x}_2}$.

We now construct \bar{h} . Let E(h(x)), $x \in \mathcal{D}$, be the set of handlers that h(x) enables. This *enable* relation between the handlers of our example is illustrated in Figure 2, where there is an edge from $h(x_i)$ to $h(x_j)$ iff $h(x_i)$ enables $h(x_j)$. Let D(h(x)), $x \in \mathcal{D}$, be the set of handlers that h(x) disables. Let $L = \{h(l_{ij}) \mid i, j = 1, 2\}$. The $D(h(x)), x \in \mathcal{D}$, are defined as follows:

$$D(h(x_2)) = D(h(\bar{x}_2)) = \{h(x_2), h(\bar{x}_2)\}$$
(13)

$$D(h(x_1)) = \{h(x_1)\} \ D(h(\bar{x}_1)) = \{h(\bar{x}_1)\}$$
(14)

$$D(h(w_2)) = D(h(\bar{w}_2)) = \{h(x_1), h(\bar{x}_1)\} \cup \{h(w_i), h(\bar{w}_i) \mid i = 1, 2\} \cup L(15)$$

$$D(h(w_1)) = D(h(\bar{w}_1)) = \{h(w_1), h(\bar{w}_1)\} \cup L$$

$$D(h(w_1)) = \{h(w_1), h(w_2)\} + \{h(w_2), h(w_2)\} + \{h(\bar{w}_2), h(w_2)\} + \{$$

$$D(h(l_{ij})) = \{h(l_{i1}), h(l_{i2})\} \cup \{h(w_k) \mid l_{ij} = \neg x_k\} \cup \{h(w_k) \mid l_{ij} = x_k\}.$$
 (17)

If $h(x) = \mathcal{H}(imr'; imr'')$, then $imr' = \bigvee_{h(y) \in E(h(x))} t_y$ and $imr'' = \bigvee_{h(z) \in D(h(x))} t_z$, where $x, y, z \in \mathcal{D}$. We claim that the QSAT instance S is satisfiable iff $|\mathcal{M}(imr_S)| = 10$, where $imr_S = t_{x_2} \vee t_{\bar{x}_2}$. We sketch the proof as follows.

Let $imr_L = \bigvee_{h(l) \in L} t_l$, where $l \in \mathcal{D}$. From (17) and Figure 2, it can be shown that $|\mathcal{M}(imr_L)| = 2$. From Figure 2, we have $E(h(x_1)) = \{h(w_1)\} \cup L$; and together with (16), and (17), it can be shown that

$$|\mathcal{M}(t_{x_1})| = 1 + |\mathcal{M}(t_{w_1} \vee imr_L)| \le 2 + |\mathcal{M}(imr_L)| = 4$$

and the equality holds iff $\exists j_1, j_2 \in 1, 2$, such that $|_{1j_1}, |_{2j_2} \neq \neg x_1$, because otherwise handler $h(w_1)$ would be surely disabled. Similarly, it can be shown that $|\mathcal{M}(t_{\bar{x}_1})| \leq 4$, and that

$$\left|\mathcal{M}(t_{x_1} \vee t_{\bar{x}_1})\right| \le \left|\mathcal{M}(t_{x_1})\right| + \left|\mathcal{M}(t_{\bar{x}_1})\right| \le 8,$$

where the equality holds iff $\exists j_1, j_2$, such that $|_{1j_1}, |_{2j_2} \neq \neg x_1$ and $\exists j'_1, j'_2$, such that

 $I_{1j'_1}, I_{2j'_2} \neq x_1$. From Figure 2, we have $E(h(x_2)) = \{h(w_2), h(x_1), h(\bar{x}_1)\}$. Thus,

$$|\mathcal{M}(t_{x_2})| = 1 + |\mathcal{M}(t_{w_2} \vee t_{x_1} \vee t_{\bar{x}_1})| \le 2 + |\mathcal{M}(t_{x_1} \vee t_{\bar{x}_1})| = 10,$$

and it can be shown from (15) and (17), that the equality holds iff $\exists j_1, j_2$ such that $I_{ij_1}, I_{ij_2} \neq \neg x_2, \neg x_1$ and $\exists j'_1, j'_2$ such that $I_{ij'_1}, I_{ij'_2} \neq \neg x_2, x_1$, which implies that both $x_2 = \text{true}, x_1 = \text{true}$ and $x_2 = \text{true}, x_1 = \text{false}$ are satisfiable truth assignments to ϕ . Similarly, it can be shown that $|\mathcal{M}(t_{\bar{x}_2})| = 10$ iff both $x_2 = \text{false}, x_1 = \text{false}$ are satisfiable truth assignments to ϕ .

From (13), we have $|\mathcal{M}(t_{x_2} \vee t_{\bar{x}_2})| = \max(|\mathcal{M}(t_{x_2})|, |\mathcal{M}(t_{\bar{x}_2})|)$. Therefore, $|\mathcal{M}(imr_S)| = 10$ iff there exists x_2 such that for all x_1 , ϕ is satisfiable, or equivalently iff **S** is satisfiable. For our example, **S** is satisfiable since $\exists x_2 = \mathsf{true}$ such that $\forall x_1, \phi$ is satisfiable. Correspondingly, $|\mathcal{M}(imr_S)| = |\mathcal{M}(x_2)| = 10$.

Theorem 13 The exact maximum stack size problem for monotonic interrupt programs is PSPACE-hard.

Proof. We will do a reduction from the QSAT problem. Suppose we are given an instance of QSAT problem

$$S = \exists x_n \forall x_{n-1} \dots \exists x_2 \forall x_1 \ \phi,$$

where ϕ is a 3SAT instance in conjunctive normal form of n variables x_n, \ldots, x_1 and L boolean clauses. Let ϕ_{ij} be the *j*th literal of the *i*th clause in ϕ and $\phi = \bigwedge_{i=1}^{L} \bigvee_{j=1}^{3} \phi_{ij}$. We construct a program $p = (m, \bar{h})$ and $\bar{h} = \{h(i) \mid i \in \{1 \ldots 3L + 4n\}\}$.

As before, we define a graph G(p) = (V, E) such that $V = \{imr \mid imr(0) = 0\}$ and $E = \{(imr, f_i(imr), i) \mid t_i \leq imr\}$, where $f_i(imr) = imr \land \neg imr' \lor imr''$ iff $h(i) = \mathcal{H}(imr'; imr'')$.

For clarity, we define three kinds of indices: $d_{ij} = 3(i-1) + j$, where $i \in \{1..L\}, j \in \{1..3\}; q_i^a = 3L + 4i - 3 + a$, and $w_i^a = 3L + 4i - 1 + a$, where $i \in \{1..n\}, a \in \{0, 1\}$.

Let

$$D = \{d_{i1}, d_{i2}, d_{i3} \mid \forall i \in \{1..L\}\}$$

$$D_{ij} = \{d_{i1}, d_{i2}, d_{i3}\} \cup \{w_k^a \mid (a = 1 \land \phi_{ij} = x_k) \lor (a = 0 \land \phi_{ij} = \neg x_k)\}$$

$$W_i = \{w_j^a \mid \forall j \in \{1..i\}, \forall a \in \{0, 1\}\}$$

$$Q_i = \{q_i^a \mid \forall j \in \{1..i\}, \forall a \in \{0, 1\}\}.$$

We will use the abbreviation

$$imr_0 = \bigvee_{i \in D} t_i, \ imr_k = t_{q_k^0} \lor t_{q_k^1} \ \forall k \in \{1..n\}.$$

Assume that n is even. For all $a \in \{0, 1\}$, let

$$\begin{split} f_{q_{2k-1}^{a}}(x) &= x \land \neg t_{q_{2k-1}^{a}} \lor (imr_{2k-2} \lor t_{w_{2k-1}^{a}}), & \forall k \in \{1..n/2\} \\ f_{q_{2k}^{a}}(x) &= x \land \neg imr_{2k} \lor (imr_{2k-1} \lor t_{w_{2k}^{a}}), & \forall k \in \{1..n/2\} \\ f_{w_{k}^{a}}(x) &= x \land \neg \bigvee_{i \in D \cup Q_{k-1} \cup W_{k}} t_{i}, & \forall k \in \{1..n\} \\ f_{d_{ij}}(x) &= x \land \neg \bigvee_{k \in D_{ij}} t_{k}, & \forall i \in \{1..L\}, j \in \{1, 2, 3\}. \end{split}$$

Given an imr value r, we define the graph $G_r(p)$ to be the subgraph of G(p)such that any edge labeled d_{ij} is removed for all i, j such that $\phi_{ij} = \neg x_k$ and $t_{w_k^0} \leq r$, or $\phi_{ij} = x_k$ and $t_{w_k^1} \leq r$. We use $\mathcal{M}_r(imr)$ to denote the longest path in $G_r(p)$ from *imr*. We organize the proof as a sequence of claims.

Claim 14 $\forall k \in \{1..\frac{n}{2}\}, |\mathcal{M}_r(imr_{2k})| = \max_{a \in \{0,1\}} |\mathcal{M}_r(t_{q_{2k}^a})|, and |\mathcal{M}_r(imr_0)| \leq L.$

Proof of Claim 14. By definition, we have that $\forall a \in \{0,1\}, f_{q_{2k}^a}(x) = x \land \neg imr_{2k} \lor (imr_{2k-1} \lor t_{w_{2k}^a})$, from which the claim follows.

By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, $\mathcal{M}(imr_0)$ can contain at most one edge with label d_{ij} , where $j \in \{1, 2, 3\}$. Thus, $|\mathcal{M}(imr_0)| \leq L$.

Claim 15 $|\mathcal{M}_r(imr_{2k-1})| = \sum_{b \in \{0,1\}} |\mathcal{M}_r(t_{q_{2k-1}})|.$

Proof of Claim 15. From Lemma 12, we have $|\mathcal{M}_r(imr_{2k-1})| \leq \sum_{b \in \{0,1\}} |\mathcal{M}_r(t_{q_{2k-1}^b})|.$

Let P be the path from imr_{2k-1} to $t_{q_{2k-1}^1}$ constructed from $\mathcal{M}_r(t_{q_{2k-1}^0})$ by replacing any node imr on $\mathcal{M}_r(t_{q_{2k-1}^0})$ with $imr \lor t_{q_{2k-1}^1}$. It is straightforward to show that if edge (imr, imr', i) is on $\mathcal{M}_r(t_{q_{2k-1}^0})$, then $(imr \lor t_{q_{2k-1}^1}, imr' \lor t_{q_{2k-1}^1}, i)$ is on P. If we concatenate P with $\mathcal{M}_r(t_{q_{2k-1}^1})$, then we have a path from imr_{2k-1} of length $|\mathcal{M}_r(t_{q_{2k-1}^0})| + |\mathcal{M}_r(t_{q_{2k-1}^1})|$.

Claim 16 $|\mathcal{M}(imr_n)| \le 2^{n/2}(6+L) - 6.$

Proof of Claim 16. It is sufficient to prove that $|\mathcal{M}(imr_{2k})| \leq 2^k(6+L) - 6$. For all $a \in \{0, 1\}$ we have:

$$\begin{aligned} |\mathcal{M}(t_{q_{2k}^a})| &= 1 + |\mathcal{M}(imr_{2k-1} \vee t_{w_{2k}^a})| \\ &\leq 2 + |\mathcal{M}(imr_{2k-1})| \\ &= 2 + \sum_{b \in \{0,1\}} |\mathcal{M}(t_{q_{2k-1}^b})| \\ &= 4 + \sum_{b \in \{0,1\}} |\mathcal{M}(imr_{2k-2} \vee t_{w_{2k-1}^b})| \\ &\leq 6 + 2|\mathcal{M}(imr_{2k-2})| \end{aligned}$$

$$|\mathcal{M}(imr_{2k})| = \max_{a \in \{0,1\}} (|\mathcal{M}(t_{q_{2k}^0})|, |\mathcal{M}(t_{q_{2k}^1})|)$$
$$\leq 6 + 2|\mathcal{M}(imr_{2k-2})|$$

From the last inequality and Claim 14, it is straightforward to show the claim by induction on k.

Claim 17 For any r and $a \in \{0, 1\}$, $|\mathcal{M}_r(t_{q_{2k}^a})| = 2^k (6+L) - 6$ iff $\forall b \in \{0, 1\}$, $|\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 2^{k-1} (6+L) - 4$, where $r' = r \vee t_{w_{2k}^a}$.

Proof of Claim 17. Suppose that $|\mathcal{M}_r(t_{q_{2k}^a})| = 2^k(6+L) - 6$. The path $\mathcal{M}_r(t_{q_{2k}^a})$ must contain the edge with label w_{2k}^a because otherwise

$$\begin{aligned} |\mathcal{M}_{r}(t_{q_{2k}^{a}})| &= 1 + |\mathcal{M}_{r}(imr_{2k-1} \vee t_{w_{2k}^{a}})| \\ &= 1 + |\mathcal{M}_{r}(imr_{2k-1})| \\ &\leq 1 + |\mathcal{M}(imr_{2k-1})| \leq 2^{k}(6+L) - 7. \end{aligned}$$

By definition of $f_{q_{2k}^a}$, for any node *imr* on the path $\mathcal{M}_r(imr_{2k-1} \vee t_{q_{2k}^a})$, we have $f_{w_{2k}^a}(imr) = 0$. Thus, the edge labeled w_{2k}^a can only be the last edge on $\mathcal{M}_r(t_{q_{2k}^a})$. By definition of $f_{d_{ij}}$, the longest path from $imr_{2k-1} \vee t_{w_{2k}^a}$ containing edge labeled w_{2k}^a does not contain any edge labeled d_{ij} for all i, j such that $\phi_{ij} = x_{2k}$ if a = 1, and $\phi_{ij} = \neg x_{2k}$ if a = 0. This path is the same path in $G_{r'}(p)$, where $r' = r \vee t_{w_{2k}^a}$. Therefore,

$$\begin{aligned} |\mathcal{M}_{r}(t_{q_{2k}^{a}})| &= 2^{k}(6+L) - 6 = |\mathcal{M}_{r'}(t_{q_{2k}^{a}})| \\ &= 1 + |\mathcal{M}_{r'}(imr_{2k-1} \lor t_{w_{2k}^{a}})| \\ &\leq 1 + |\mathcal{M}_{r'}(imr_{2k-1})| + |\mathcal{M}_{r'}(t_{w_{2k}^{a}})| \\ &= 2 + \sum_{b \in \{0,1\}} |\mathcal{M}_{r'}(t_{q_{2k-1}^{b}})|, \text{ and} \\ &\sum_{b \in \{0,1\}} |\mathcal{M}_{r'}(t_{q_{2k-1}^{b}})| \geq 2^{k}(6+L) - 8. \end{aligned}$$

Since

$$|\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 1 + |\mathcal{M}_{r'}(imr_{2k-2} \lor t_{w_{2k-1}^b})|$$

$$\leq 2 + |\mathcal{M}_{r'}(imr_{2k-2})|$$

$$\leq 2 + |\mathcal{M}(imr_{2k-2})| = 2^{k-1}(6+L) - 4,$$

we have $\forall b \in \{0,1\}, |\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 2^{k-1}(6+L) - 4.$

Conversely, assume that for all $b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 2^{k-1}(6+L) - 4$ where $r' = r \lor t_{w_{2k}^a}$. From Claim 2, we know that $|\mathcal{M}_{r'}(imr_{2k-1})| = \sum_{b \in \{0,1\}} |\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 2^k(6+L) - 8.$

Let P be a path from $imr_{2k-1} \vee t_{w_{2k}^a}$ to $t_{w_{2k}^a}$ constructed from $\mathcal{M}_{r'}(imr_{2k-1})$ by replacing any node imr on $\mathcal{M}_{r'}(imr_{2k-1})$ with $imr \vee t_{w_{2k}^a}$. It is straightforward to show that if edge (imr, imr', i) is on $\mathcal{M}_{r'}(imr_{2k-1})$, then $(imr \vee t_{w_{2k}^a}, imr' \vee t_{w_{2k}^a}, i)$ is on P as well.

If we concatenate P with $\mathcal{M}_{r'}(t_{w_{2k}^a})$, then we have a path from $imr_{2k-1} \vee t_{w_{2k}^a}$ in graph $G_{r'}(p)$ of length $2^k(6+L) - 7$. Thus, $|\mathcal{M}_r(t_{q_{2k}^a})| = |\mathcal{M}_{r'}(t_{q_{2k}^a})| = 2^k(6+L) - 6$.

Claim 18 For any r' and $b \in \{0, 1\}$, we have $|\mathcal{M}_{r'}(t_{q_{2k-1}^b})| = 2^{k-1}(6+L) - 4$ iff $|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6$, where $r'' = r' \vee t_{w_{2k-1}^b}$.

Proof of Claim 18. The proof is similar to the proof of Claim 17, we omit the details. \blacksquare

Claim 19 $|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6$ iff $\exists a_n \forall a_{n-1} \dots \exists a_2 \forall a_1 \in \{0,1\}$, such that for $r = \bigvee_{k \in \{1...n\}} t_{w_k^{a_k}}, |\mathcal{M}_r(imr_0)| = L.$

Proof of Claim 19. From Claim 14, we know that $|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6$ iff $\exists a \in \{0, 1\}$ such that $|\mathcal{M}_r(t_{q_{2k}^a})| = 2^k(6+L) - 6$. Together with Claim 17 and Claim 18, we have that for $k \in \{1..n/2\}$, $|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6$ iff there exists $a \in \{0, 1\}$ such that for all $b \in \{0, 1\}$, we have $|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6$, where $r'' = r \vee t_{w_{2k}^a} \vee t_{w_{2k-1}^b}$.

It is straightforward to prove by induction from k = n/2 to 1, that

$$|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6 \text{ iff } \exists a_n \forall a_{n-1} \dots \exists a_{2k} \forall a_{2k-1},$$

such that $|\mathcal{M}_r(imr_{2k-2})| = 2^{k-1}(6+L) - 6$, where $r = \bigvee_{i=2k-1}^n t_{w_i^{a_i}}$

The claim follows when k = 1.

Claim 20 *S* is satisfiable iff $\exists a_n \forall a_{n-1} \dots \exists a_2 \forall a_1 \in \{0,1\}$, we have $|\mathcal{M}_r(imr_0)| = L$, where $r = \bigvee_{k=1}^n t_{w_k^{a_k}}$.

Proof of Claim 20. It is sufficient to prove that ϕ is satisfiable iff $\exists a_n, \ldots, a_1 \in \{0, 1\}$, such that for $r = \bigvee_{k=1}^n t_{w_k^{a_k}}$, we have $|\mathcal{M}_r(imr_0)| = L$.

Suppose we have a_n, \ldots, a_1 such that $r = \bigvee_{k=1}^n t_{w_k^{a_k}}, |\mathcal{M}_r(imr_0)| = L$. We can construct a truth assignment T by defining $T(x_k) = true$ if $a_k = 0$ and $T(x_k) = false$ if $a_k = 1$. By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, there exists a j such that the edge labeled d_{ij} is on $\mathcal{M}_r(imr_0)$. By definition of \mathcal{M}_r , if an edge labeled d_{ij} is on $\mathcal{M}_r(imr_0)$ and $\phi_{ij} = x_k$, then $a_k = 0$, and $T(x_k) = true$; and if $\phi_{ij} = \neg x_k$, then $a_k = 1$ and $T(x_k) = false$. $T(\phi_{ij}) = true$ in both cases. Therefore, T satisfies ϕ .

Conversely, suppose T satisfies ϕ . We can construct $r = \bigvee_{k=1}^{n} t_{w_{k}^{a_{k}}}$ from T by defining $a_{k} = 0$ if $T(x_{k}) = true$ and $a_{k} = 1$ if $T(x_{k}) = false$. For each $i \in \{1..L\}$, there exists j such that $T(\phi_{ij}) = true$, which means that the edge labeled d_{ij} can be on the path $\mathcal{M}_{r}(imr_{0})$. Therefore, $|\mathcal{M}_{r}(imr_{0})| = L$.

We now proceed with the proof of the theorem. We conclude

S is satisfiable

$$\text{iff } \exists a_n \forall a_{n-1} \dots \exists a_2 \forall a_1: \text{ for } r = \bigvee_{k=1}^n t_{w_k^{a_k}}, |\mathcal{M}_r(imr_0)| = L \quad (\text{Claim } 20)$$

$$\text{iff } |\mathcal{M}(imr_S)| = 2^{n/2}(6+L) - 6, \text{ where } imr_S = imr_n \qquad (\text{Claim } 19)$$

$$\text{iff } maxStackSize(P_p) = 2^{n/2}(6+L) - 6 \qquad (\text{Lemma } 11),$$

so the exact maximum stack size problem is PSPACE-hard.

Notice that we can combine the last part of the proof of Theorem 13 with Claim 16 to get that S is not satisfiable iff $maxStackSize(P_p) < 2^{n/2}(6+L) - 6$.

4 Monotonic Enriched Interrupt Programs

We now introduce an enriched version of the interrupt calculus, where we allow conditionals on the interrupt mask register. The conditional can test if some bit of the imr is on, and then take the bitwise or of the imr with a constant bit sequence; or it can test if some bit of the imr is off, and then take the bitwise and of the imr with a constant. The syntax for *enriched interrupt programs* is given by the syntax from Section 2 together with the following clauses:

> (statement) $s ::= \cdots$ | if(bit *i* on) imr = imr \lor *imr* | if(bit *i* off) imr = imr \land *imr*

The small-step operational semantics is given below:

Unlike the conditional statement if $0(x) s_1$ else s_2 on data that has been overapproximated, our analysis will be path sensitive in the imr-conditional.

Proposition 21 Monotonicity of enriched interrupt programs can be checked in time exponential in the number of handlers (in co-NP).

Proof. It follows from Lemma 1 that a program is monotonic iff each handler is monotonic in isolation. To check nonmonotonicity, we guess a handler and an imr value that shows it is nonmonotonic, and check in polynomial time that the handler is not monotonic for that imr. \blacksquare

For monotonic enriched interrupt programs, both the stack boundedness problem and the exact maximum stack size problem are PSPACE-complete. To show this, we first show that the stack boundedness problem is PSPACEhard by a generic reduction from polynomial-space Turing machines. We fix a PSPACE-complete Turing machine M. Given input x, we construct in polynomial time a program p such that M accepts x iff p has an unbounded stack. We have two handlers for each tape cell (one representing zero, and the other representing one), and a handler for each triple (i, q, b) of head position i, control state q, and bit b. The handlers encode the working of the Turing machine in a standard way. The main program sets the bits corresponding to the initial state of the Turing machine, with x written on the tape. Finally, we have an extra handler that enables itself (and so can cause an unbounded stack) which is set only when the machine reaches an accepting state. We provide the formal proof below.

Theorem 22 The stack boundedness problem for monotonic enriched interrupt programs is PSPACE-hard.

Proof. Fix a PSPACE-complete Turing Machine M which on any input x uses at most r(|x|) space to decide whether M accepts x, where r is a polynomial. Given any input x, the TM M always halts and answers *accept* or *reject*. It is PSPACE-complete to decide given M and x whether M(x) accepts or rejects

[5]. Let the states of M be $Q = \{q_1, q_2, ..., q_t\}$ with q_t as the accepting state. Given such a machine M and an input x we reduce the problem of whether q_t is reachable to the stack boundedness analysis of a interrupt driven program such that the stack size is infinite iff q_t is reachable. We construct, from M and x a monotone interrupt program p(M, x) such that M accepts x iff the stack of p(M, x) is unbounded.

Now we describe the *imr* and handlers. The total number of bits in the imr is $1 + 2 \times r(|x|) + 2 \times |Q| \times r(|x|)$. The first (0th) bit is the master bit. There are two bits for each position of the tape, so 2r(|x|) bits encode the tape of the TM M. Further, the imr has one bit for every tuple (i, q, σ) for each tape position $i \in \{1, \ldots, r(|x|)\}$, TM state $q \in Q$, and symbol $\sigma \in \{0, 1\}$.

For $k \in \{1, \ldots, r(|x|)\}$, the k-th tape cell is stored in bits 2k - 1 and 2k. For all $1 \leq k \leq r(|x|)$, the (2k - 1)-th bit is 1 and 2k-th bit is 0 if the tape cell in the k-th position of M is 0. Similarly, or all $1 \leq k \leq r(|x|)$, the (2k - 1)-th bit is 0 and 2k-th bit is 1 if the tape cell in the k-th position of M is 1.

The bit for (i, q, σ) bit is turned on when the head of M is in the *i*-th cell, the control is in state q and σ is written in the *i*-th cell. Formally, the bit (2r(|x|) + 2kr(|x|) + 2i - 1) is 1 if the head of TM M is in position *i*, the TM is in state q_k , and the *i*th tape cell reads 0. The code for this handler implements the transition for the TM M corresponding to $(q_k, 0)$. Similarly, the (2r(|x|) + 2kr(|x|) + 2i)-th bit is 1 if the head of TM M is in position *i*, the TM is in state q_k , and the *i*th tape cell reads 1. The code for this handler implements the transition for the TM M corresponding to $(q_k, 1)$.

The first $2 \times r(|x|)$ handlers which encode the tape cells do not change the status of the imr, that is, the body of the handler H_i contains only the statement iret for $i = 1, \ldots, 2r(|x|)$.

We show how to encode the transition of TM M in the code for the handler. We first introduce some short-hand notation for readability.

- The operation write (σ, i) writes σ in the *i*-th cell tape of M. This is shorthand for the following code:
- (1) if $\sigma = 0$, the code is $\operatorname{imr} = \operatorname{imr} \land \neg t_{2 \times i}$; $\operatorname{imr} = \operatorname{imr} \lor t_{2 \times i-1}$ (recall that t_i denotes the imr with all bits 0's and only the *j*-th bit 1).
- (2) if $\sigma = 1$, the code is $\operatorname{imr} = \operatorname{imr} \land \neg t_{2 \times i-1}$; $\operatorname{imr} = \operatorname{imr} \lor t_{2 \times i}$.
- The operation set(*i*-th bit on) sets the *i*-th bit of the imr on. This is shorthand for imr = imr ∨ t_i.
- The operation set(*i*-th bit off) sets the *i*-th bit of the imr off. This is shorthand for $imr = imr \land \neg t_i$.

We now give the encoding of the transition relation of the TM.

Fig. 3. Code for Turing Machine transition

Main {

}

```
    imr = imr ∨ c
where c is an imr constant which correctly encodes
the starting configuration of M on x.
    loop skip
```

Fig. 4. Code for the main program

- (1) Consider Handler H_j where j = 2r(|x|) + 2kr(|x|) + 2i 1 and the transition for $(q_k, 0)$ is $(q_{k'}, \sigma, R)$. Let l = 2r(|x|) + 2k'r(|x|) + 2(i + 1). The code for handler H_j is shown in Figure 3. Note that the two consecutive imr_{or} statements in line 3.1 and 3.2 and 4.1 and 4.2 can be folded into a single imr_{or} statement, we separate them for readability. We can encode the other transition types similarly.
- (2) The code for a handler H_j corresponding to an accepting state sets the master bit on, and returns.
- (3) The main program initializes the imr with the initial configuration, and enters an empty loop. The code for the main program is shown in Figure 4.

Lemma 23 If M accepts x then the stack of p(M, x) grows unbounded.

Proof. We show that there is a sequence of interrupt occurrences such that a handler corresponding to the accepting state is called. Whenever the *l*-th or the (l-1)-th bit in turned on and the master bit is turned on in lines 3.1,3.2 or 4.1,4.2 of Figure 3, an interrupt of type *l* or (l-1) occurs. Hence following this sequence a handler corresponding to the accepting state is called and then the stack can grow unbounded as the handler sets the master bit on without disabling itself. ■

Lemma 24 If M halts without accepting x then the stack of p(M, x) is bounded.

Proof. If any handler which encodes the transitions of the M(x) returns it sets all the bits of *imr* to 0 (Statement 6 in Figure 3). Hence all the following checks in the statements 3 and 4 will fail and the master bit will not be set any further. Hence the stack would go empty. So if the stack is unbounded then no handler which encodes the configuration of the machine M returns. If the stack is unbounded and the accepting state is not reached then there is a handler h which encodes the transition of the machine M and it occurs infinitely many times in the stack. This means one of the configurations of M(x) can be repeated. This means there is a cycle in the configuration graph of M(x) and hence it cannot halt. But this is a contradiction, since our TM always halts. This proves that if the accepting state is not reached then the stack is bounded.

From Lemmas 23, 24 the theorem follows.

We now give a PSPACE algorithm to check the exact maximum stack size. Since we restrict our programs to be monotonic it follows from Lemma 4 that the maximum length of the stack can be achieved with no handler returning in between. Given a program p with m statements and n handlers, we label the statements as $pc_1, \ldots pc_m$. Let PC denote the set of all statements, i.e., $PC = \{pc_1, \ldots pc_m\}$. Consider the graph G_p where there is a node v for every statement with all possible imr values (i.e., $v = \langle pc, imr \rangle$ for some value among PC and some imr value). Let $v = \langle pc, imr \rangle$ and $v' = \langle pc', imr' \rangle$ be two nodes in the graph. There is an edge between v, v' in G if any of the following two conditions hold:

- on executing the statement at *pc* with imr value *imr* the control goes to *pc'* and the value of imr is *imr'*. The weight of this edge is 0.
- pc' is a starting address of a handler h_i and enabled(imr, i) and $imr' = imr \land \neg t_0$. The weight of this edge is 1.

We also have a special node in the graph called *target* and add edges to *target* of weight 0 from all those nodes which correspond to a $pc \in PC$ which is a *iret* statement. This graph is exponential in the size of the input as there are $O(|PC| \times 2^n)$ nodes in the graph. The starting node of the graph is the node with pc_1 and imr = 0. If there is a node in the graph which is the starting address of a handler h and which is reachable from the start node and also self-reachable then the stack length would be infinite. This is because the sequence of calls from the starting statement to the handler h is first executed and then the cycle of handler calls is repeated infinitely many times. As the handler h is in stack when it is called again the stack would grow infinite. Since there is a sequence of interrupts which achieves the maximum stack length without

any handler returning in between (follows from Lemma 4) if there is no cycle in G_p we need to find the longest path in the DAG G_p .

Theorem 25 The exact maximum stack size for monotonic enriched interrupt programs can be found in time linear in the size of the program and exponential in the number of handlers. The complexity of exact maximum stack size for monotonic enriched interrupt programs is PSPACE.

In polynomial space one can generate in lexicographic order all the nodes that have a *pc* value of the starting statement of a handler. If such a node is reachable from the start node, and also self-reachable, then the stack size is infinite. Since the graph is exponential, this can be checked in PSPACE. If no node has such a cycle, we find the longest path from the start node to the target. Again, since longest path in a DAG is in NLOGSPACE, this can be achieved in PSPACE. It follows that both the stack boundedness and exact maximum stack size problems for monotonic enriched interrupt programs are PSPACE-complete.

5 Nonmonotonic Enriched Interrupt Programs

In this section we consider interrupt programs with tests, but do not restrict handlers to be monotonic. We give an EXPTIME algorithm to check stack boundedness and find the exact maximum stack size for this class of programs. The algorithm involves computing longest context-free paths in context-free DAGs, a technique that may be of independent interest.

5.1 Longest Paths in Acyclic Context-free Graphs

We define a context-free graph as in [9]. Let Σ be a finite alphabet. A contextfree graph is a tuple $G = (V, E, \Sigma)$ where V is a set of nodes and $E \subseteq (V \times V \times (\Sigma \cup \{\tau\}))$ is a set of labeled edges (and τ is a special symbol not in Σ).

We shall particularly consider the context-free language of matched parentheses. Let $\Sigma = \{(^1, (^2, \ldots, (^k,)^1,)^2, \ldots,)^k\}$ be the alphabet of opening and closing parentheses. Let \mathcal{L} be the language generated by the context-free grammar

$$M \to M(^{i}S \mid S \text{ for } 1 \le i \le k$$
$$S \to \epsilon \mid (^{i}S)^{i}S \text{ for } 1 \le i \le k$$

from the starting symbol M. Thus \mathcal{L} defines words of matched parentheses with possibly some opening parentheses mismatched. From this point, we restrict our discussion to this Σ and the language \mathcal{L} .

Algorithm 2 Function LongestContextFreePath

Input: A context-free DAG G, a vertex v_1 of G **Output:** For each vertex v of G, return the length of the longest context-free path from v to v_1 , and 0 if there is no context-free path from v to v_1 1. For each vertex $v_j \in V$: $val[v_j] = 0$ 2. Construct the transitive closure matrix T such that T[i, j] = 1 iff there is a context-free path from i to j3. For j = 1 to n: 3.1 For each immediate successor v_i of v_j such that the edge e_{v_j,v_i} from v_j to v_i satisfies $wt(e_{v_j,v_i}) \ge 0$: $val[v_j] = \max\{val[v_j], val[v_i] + wt(e_{v_j,v_i})\}$ 3.2 For each vertex $v_i \in V$: 3.2.1 if(T[i, j]) $(v_j$ is context-free reachable from v_i) $val[v_i] = \max\{val[v_i], val[v_j]\}$

We associate with each edge of G a weight function $wt : E \to \{0, +1, -1\}$ defined as follows:

- wt(e) = 0 if e is of the form (v, v', τ) ,
- wt(e) = -1 if e is of the form $(v, v',)^i)$ for some i,
- wt(e) = 1 if e is of the form (v, v', (i) for some i.

A context-free path π in a context-free graph G is a sequence of vertices $v_1, v_2, \ldots v_k$ such that for all $i = 1 \ldots k - 1$, there is an edge between v_i and v_{i+1} , i.e., there is a letter $\sigma \in \Sigma \cup \{\tau\}$ such that $(v_i, v_{i+1}, \sigma) \in E$ and the projection of the labels along the edges of the path to Σ is a word in \mathcal{L} . Given a context-free path π with edges $e_1, e_2, \ldots e_k$ the cost of the path $Cost(\pi)$ is defined as $\sum_i wt(e_i)$. Note that $Cost(\pi) \geq 0$ for any context-free path π . A context-free graph G is a context-free DAG iff there is no cycle C of G such that $\sum_{e \in C} wt(e) > 0$. Given a context-free DAG $G = (V, E, \Sigma)$ we define an ordering order : $V \to \mathbb{N}$ of the vertices satisfying the following condition: if there is a path π in G from vertex v_i to v_j and $Cost(\pi) > 0$ then $order(v_j) < order(v_i)$. This ordering is well defined for context-free DAGs. Let G be a context-free DAG G, and let $V = \{v_1, v_2, \ldots v_n\}$ be the ordering of the vertex set consistent with order (i.e., $order(v_i) = i$). We give a polynomial-time procedure to find the longest context-free path from any node v_i to v_1 in G.

The correctness proof of our algorithm uses a function Num from paths to \mathbb{N} . Given a path π we define $Num(\pi)$ as $\max\{order(v) \mid v \text{ occurs in } \pi\}$. Given a node v let $L_v = \{L_1, L_2, \ldots, L_k\}$ be the set of longest paths from v to v_1 . Then we define $Num_{v_1}(v) = \min\{Num(L_i) \mid L_i \in L_v\}$. The correctness of the algorithm follows from the following set of observations. **Lemma 26** If there is a longest path L from a node v to v_1 such that L starts with an opening parenthesis (ⁱ that is not matched along the path L then $order(v) = Num_{v_1}(v)$.

Proof. Consider any node v' in the path L. Since the first opening parenthesis is never matched, the sub-path L(v, v') of L from v to v' satisfies Cost(L(v, v')) > 0. Hence it follows that for all nodes v' in L, we have order(v') < order(v). Thus $Num_{v_1}(v) = order(v)$.

Lemma 27 A node v in the DAG G satisfies the following conditions.

- If $Num_{v_1}(v) = order(v) = j$ then within the execution of Statement 3.1 of the *j*-th iteration of Loop 3 of function LongestContextFreePath, val[v] is equal to the cost of a longest path from v to v_1 .
- If order(v) < Num_{v1}(v) = j then by the j-th iteration of Loop 3 of function LongestContextFreePath val[v] is equal to the cost of a longest path from v to v₁.

Proof. We prove by induction on $Num_{v_1}(v)$. The base case holds when $Num_{v_1}(v) = 1$, since $v = v_1$. We now prove the inductive case. If the value of the longest path is 0 then it was fixed initially and it cannot decrease. Otherwise, there is a positive cost longest path from v to v_1 . We consider the two cases when $order(v) = Num_{v_1}(v)$ and when $order(v) < Num_{v_1}(v)$.

Case $order(v) = Num_{v_1}(v) = j$. Let $L(v, v_1)$ be a longest path from v to v_1 such that $Num(L(v, v_1)) = order(v)$. We consider the two possible cases.

- (1) The longest path $L(v, v_1)$ is such that it starts with a opening parenthesis which is never matched. Let v'' be the successor of v in $L(v, v_1)$. Hence $order(v'') < order(v) = Num_{v_1}(v) = j$. Also the sub-path $L(v'', v_1)$ of $L(v, v_1)$ is a longest path from v'' to v_1 (since otherwise we could have a greater cost path from v to v_1 by following the path from v to v'' and then the path from v'' to v_1). Hence $Num_{v_1}(v'') < Num_{v_1}(v) = j$. By the induction hypothesis, before the *j*-th iteration val[v''] is equal to the cost of the longest path from v'' to v_1 . Hence during the *j*-th iteration of Loop 3, when the loop of statement 3.1 is executed and v'' is chosen as v's successor then val[v] is set to the cost of the longest path from v to v_1 .
- (2) The longest path $L(v, v_1)$ goes through a node v' such that the cost of the subpath of L(v, v') of $L(v, v_1)$ satisfies Cost(L(v, v')) = 0and there is a opening parenthesis from v' which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from v' to v_1 as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 26 that $Num_{v_1}(v') = order(v') = k < j$. By the induction hypothesis, by the end of Statement 3.1 of k-th iteration

val[v'] is equal to the longest path from v' to v_1 . As v can context-free reach v' we have during the execution of statement 3.2 of the k-th iteration val[v] is equal to the longest path from v to v_1 .

Case $order(v) < Num_{v_1}(v) = j$. Let $L(v, v_1)$ be a longest path from v to v_1 . The longest path $L(v, v_1)$ goes through a node v' such that the cost of the subpath of L(v, v') of $L(v, v_1)$ satisfies Cost(L(v, v')) = 0 and there is an opening parenthesis from v' which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from v' to v_1 as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 26 that $Num_{v_1}(v') = order(v') = k$. By hypothesis by the end of Statement 3.1 of k-th iteration val[v'] is equal to the cost of the longest path from v' to v_1 . As v can context-free reach v' we have during the execution of statement 3.2 of the k-th iteration val[v] is equal to the longest path from v to v_1 . As $Num_{v_1}(v) \ge order(v')$ (since v' occurs in the path) it follows by the end of j-th iteration val[v] is equal to the cost of the longest path from v to v_1 .

Notice also that every time val[v] is updated (to c, say), it is easy to construct a witness path that shows that the cost of the longest path is at least c. This concludes the proof.

From the above two lemmas, we get the following.

Corollary 28 At the end of function LongestContextFreePath (G, v_1) , for each vertex v, the value of val[v] is equal to the longest context-free path to v_1 , and equal to zero if there is no context-free path to v_1 .

We now consider the time complexity of the function LongestContextFreePath. In the Function LongestContextFreePath the statement 3.2.1 gets executed at most n^2 times since the loop on line 3 gets executed *n* times at most and the nested loop on line 3.2 also gets executed *n* times at most. The context-free transitive closure can be constructed in $O(n^3)$ time [12]. Hence the complexity of our algorithm is polynomial and it runs in time $O(n^2 + n^3) = O(n^3)$.

Theorem 29 The longest context-free path of a context-free DAG can be found in time cubic in the size of the graph.

To complete our description of the algorithm, we must check if a given contextfree graph is a context-free DAG, and generate the topological ordering *order* for a context-free DAG. We give a polynomial-time procedure to check whether a given context-free graph is a DAG. Let $G = (V, E, \Sigma)$ be a given context-free graph, and let $V = \{1, 2, \ldots n\}$. For every node $k \in V$ the graph G can be unrolled as a DAG for depth |V|, and it can be checked if there is a path π from k to k such that $Cost(\pi) > 0$. Given the graph G and a node k we create a context-free DAG $G_k = (V_k, E_k, \Sigma)$ as follows:

1.
$$V_k = \{k_0\} \cup \{(i, j) \mid 1 \le i \le n - 2, 1 \le j \le n\} \cup \{k_{n-1}\}$$

2. $E_k = \{\langle k_0, (1, j), * \rangle \mid \langle k, j, * \rangle \in E\} \cup \{\langle (i, j), (i + 1, j'), * \rangle \mid \langle j, j', * \rangle \in E\}$
 $\cup \{\langle (n - 2, j), k_{n-1}, * \rangle \mid \langle j, k, * \rangle \in E\}$
 $\cup \{\langle k_0, (1, k), \tau \rangle\} \cup \{\langle (i, k), (i + 1, k), \tau \rangle\}$

where * can represent a opening parenthesis, closing parenthesis or can be τ . Notice that the edges in the last line ensure that if there is a cycle of positive cost from k to itself with length t < n then it is possible to go from k_0 to (n - t - 1, k) and then to reach k_{n-1} by a path of positive cost.

We can find the longest context-free path from k_0 to k_n in G_n (by the function LongestContextFreePath). If the length is positive, then there is a positive cycle in G from k to k. If for all nodes the length of the longest path in G_n is 0, then G is a context-free DAG and the longest context-free path can be computed in G. Given a context-free DAG G we can define order(v) in polynomial time. If a vertex v can reach v' and v' can reach v put them in the same group of vertices. Both the path from v to v' and v' to v must be cost 0 since there is no cycle of positive cost. Hence the ordering of vertices within a group can be arbitrary. We can topologically order the graph induced by the groups and then assign an order to the vertices where vertices in the same group are ordered arbitrarily.

5.2 Stack Size Analysis

We present an algorithm to check for stack boundedness and exact maximum stack size. The idea is to perform context-free longest path analysis on the state space of the program. Given a program p with m statements and nhandlers, we label the statements as $pc_1, pc_2, \ldots pc_m$. Let $PC = \{pc_1, \ldots pc_m\}$ as before. We construct a context-free graph $G_p = \langle V, E, \Sigma \rangle$, called the *state* graph of p, where $\Sigma = \{(^1, (^2, \ldots, (^m,)^1,)^2, \ldots)^m\}$ as follows:

- $V = PC \times IMR$, where IMR is the set of all 2^n possible imr values.
- $E \subseteq (V \times V \times (\Sigma \cup \{\tau\}) \text{ consists of the following edges.}$
- (1) Handler call: $(v, v', (i) \in E \text{ iff } v = (pc_i, imr_1) \text{ and } v' = (pc_j, imr_2) \text{ and } pc_j$ is the starting address of some handler h_j such that $enabled(imr_1, j)$ and $imr_2 = imr_1 \wedge \neg t_0$.
- (2) Handler return: $(v', v,)_i) \in E$ iff $v = (pc_i, imr_1)$ and $v' = (pc_j, imr_2)$ and pc_j is the iret statement of some handler and $imr_1 = imr_2 \lor t_0$.
- (3) Statement execution: $(v, v', \tau) \in E$ iff $v = (pc_i, imr_1)$ and $v' = (pc_j, imr_2)$ and executing the statement at pc_i with imr value imr_1 the control goes to pc_j and the imr value is imr_2 .

The vertex $(pc_1, 0)$ is the starting vertex of G_p . Let G'_p be the induced subgraph of G_p containing only nodes that are context-free reachable from the start

Algorithm 3 Function StackSizeGeneral

Input: Enriched interrupt program p
Output: maxStackSize(P_p)
1. Build the state graph G_p = ⟨V, E, Σ⟩ from the program p
2. Let V' = {v' | there is a context-free path from the starting vertex to v'}
3. Let G'_p be the subgraph of G_p induced by the vertex set V'
4. If G'_p is not a context-free DAG then return "infinite"
5. Else create G''_p = (V'', E'', Σ) as follows :
5.1 V'' = V' ∪ {target} and E'' = E' ∪ {(v, target, τ) | v ∈ V'}
6. Return the value of the longest context-free path from the starting vertex to target

node. If G'_p is not a context-free DAG then we report that stack is unbounded. Otherwise, we create a new DAG G''_p by adding a new vertex *target* and adding edges to *target* from all nodes of G'_p of weight 0. Then, we find the value of a longest context-free path from the start vertex to *target* in the DAG G''_p .

From the construction of the state graph, it follows that there is a contextfree path from a vertex v = (pc, imr) to v' = (pc', imr') in the state graph G_p if there exists stores R, R' and stacks σ, σ' such that $\langle \bar{h}, R, imr, \sigma, pc \rangle \rightarrow^* \langle \bar{h}, R', imr', \sigma'pc' \rangle$. Moreover, if G'_p is the reachable state graph then there exists K such that for all P' such that $P_p \rightarrow^* P'$ we have $|P'.stk| \leq K$ iff G'_p is a context-free DAG. To see this, first notice that if G'_p is not a context-free DAG then there is a cycle of positive cost. Traversing this cycle infinitely many times makes the stack grow unbounded. On the other hand, if the stack is unbounded then there is a program address that is visited infinitely many times with the same imr value and the stack grows between the successive visits. Hence there is a cycle of positive cost in G'_p . These observations, together with Theorem 29 show that function StackSizeGeneral correctly computes the exact maximum stack size of an interrupt program p.

Theorem 30 The exact maximum stack size of nonmonotonic enriched interrupt programs can be found in time cubic in the size of the program and exponential in the number of handlers.

Proof. The number of vertices in G_p is $m \times 2^n$, for m program statements and n interrupt handlers. It follows from Theorem 29 and the earlier discussion that the steps 1, 2, 3, 4, 5, and 6 of StackSizeGeneral can be computed in time polynomial in G_p . Since G_p is linear in the size of the input program p, and exponential in the number of handlers, we have a procedure for determining the exact maximum stack size of nonmonotonic enriched interrupt programs that runs in $O(m^3 8^n)$. This gives an EXPTIME procedure for the exact maximum stack size problem. ■

While our syntax ensures that all statements that modify the imr are monotonic, this is not a fundamental limitation for the above algorithm. Indeed, we can extend the syntax of the enriched calculus to allow any imr operations, and the above algorithm still solves the exact maximum stack size problem, with no change in complexity.

We leave open whether the exact maximum stack size problem for nonmonotonic interrupts programs, in the nonenriched and enriched cases, is EXPTIME-hard or PSPACE-complete (PSPACE-hardness follows from Theorem 22). One can note that the time to execute the algorithms grows exponentially with the *number* of interrupt handlers, which is typically small, and cubically with the *size* of the interrupt handler programs.

Acknowledgments

Palsberg, Ma, and Zhao were supported by the NSF ITR award 0112628. Henzinger, Chatterjee, and Majumdar were supported by the AFOSR grant F49620-00-1-0327, the DARPA grants F33615-C-98-3614 and F33615-00-C-1693, the MARCO grant 98-DT-660, and the NSF grants CCR-0208875 and CCR-0085949.

References

- D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *ICSE: International Conference on Software Engineering*, pp. 47–56. ACM/IEEE, 2001.
- [2] G. G. Hillebrand, P. C. Kanellakis, H. G. Mairson, and M. Y. Vardi. Undecidable boundedness problems for datalog programs. Journal of Logic Programming 25(2):163–190, 1995.
- [3] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL: Principles of Programming Languages*, pp. 410–423. ACM, 1996.
- [4] J. Palsberg and D. Ma. A typed interrupt calculus. In *FTRTFT: Formal Techniques in Real-Time and Fault-tolerant Systems*, LNCS 2469, pp. 291–310. Springer, 2002.
- [5] C.H. Papadimitriou. Computational Complexity. Addision-Wesley, 1994.
- [6] C.H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). Journal of Computer and System Sciences 28:244-259, 1984.

- [7] L. Pareto. Types for Crash Prevention. PhD thesis, Chalmers University of Technology, 2000.
- [8] J. Regehr, A. Reid, and K. Webb, Eliminating stack overflow by abstract interpretation, In EMSOFT'03: Third International Workshop on Embedded Software, 2003. To appear.
- T.W. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pp. 49–61. ACM, 1995.
- [10] T.W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In SAS 03: Static Analysis Symposium, LNCS 2694, pp. 189–213. Springer, 2003.
- [11] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In PADL: Practical Aspects of Declarative Languages, LNCS 2257, pp. 155–172. Springer, 2002.
- [12] M. Yannakakis. Graph-theoretic methods in database theory. In PODS: Principles of Database Systems, pp. 203–242. ACM, 1990.

A Proof of the Schroder-Bernstein Theorem

Jens Palsberg

July 26, 2008

The following proof is a slightly modified version of C. A. Gunter and D. S. Scott's proof in their article *Semantic Domains* in Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pages 633–674, 1990.

Theorem 1 (Schroder-Bernstein) Let S and T be sets. If $f : S \to T$ and $g : T \to S$ are injections, then there is a bijection $h : S \to T$.

Proof. Let us for a moment assume that we can find a set $Y \subseteq T$ that satisfies the equation:

$$T \setminus Y = f^*(S \setminus g^*(Y)) \tag{1}$$

Define $h: S \to T$ by

$$h(x) = \begin{cases} y & \text{if } x \in g^*(Y), \text{ that is, } x = g(y) \text{ for some } y \in Y \\ f(x) & \text{if } x \in (S \setminus g^*(Y)) \end{cases}$$

To see that h is well defined for $x \in g^*(Y)$, notice that we have a unique choice of y because g is an injection.

To see that h is an injection, suppose $x_1, x_2 \in S$ and $h(x_1) = h(x_2)$. We have four cases. First, if $x_1, x_2 \in g^*(Y)$, then $x_1 = g(h(x_1)) = g(h(x_2)) = x_2$. Second, if $x_1, x_2 \in (S \setminus g^*(Y))$, then $x_1 = x_2$ because f is an injection. Third, if $x_1 \in g^*(Y)$ and $x_2 \in (S \setminus g^*(Y))$, then we have $h(x_1) \in Y$ and $h(x_2) \in f^*(S \setminus g^*(Y)) = (T \setminus Y)$, where the last step is equation (1). Now $h(x_1) \in Y$ and $h(x_2) \in (T \setminus Y)$ contradicts $h(x_1) = h(x_2)$. Fourth, if $x_1 \in (S \setminus g^*(Y))$ and $x_2 \in g^*(Y)$, then in a manner that is similar to the third case, we can reach a contradiction of $h(x_1) = h(x_2)$.

To see that h is a surjection, suppose $y \in T$. We have two cases. If $y \in Y$, then h(g(y)) = y. If $y \in (T \setminus Y)$, then we have from equation (1) that $y \in (T \setminus Y) = f^*(S \setminus g^*(Y))$, so y = f(x) = h(x) for some $x \in (S \setminus g^*(Y))$.

We must finally prove that we can find a set $Y \subseteq T$ that satisfies the equation (1). The function $Y \mapsto (T \setminus f^*(S)) \cup f^*(g^*(Y))$ from subsets of T to subsets of T is easily seen to be continuous with respect to the inclusion ordering. Hence, by the Fixed Point Theorem, there is a subset $Y = (T \setminus f^*(S)) \cup f^*(g^*(Y))$. We have

$$T \setminus Y = T \setminus [(T \setminus f^*(S)) \cup f^*(g^*(Y))]$$

= $f^*(S) \cap (T \setminus f^*(g^*(Y)))$
= $f^*(S \setminus g^*(Y))$

So our assumption is valid and we conclude that h is a bijection.

Closure Analysis in Constraint Form

JENS PALSBERG

Aarhus University

Flow analyses of untyped higher-order functional programs have in the past decade been presented by Ayers, Bondorf, Consel, Jones, Heintze, Sestoft, Shivers, Steckler, Wand, and others. The analyses are usually defined as abstract interpretations and are used for rather different tasks such as type recovery, globalization, and binding-time analysis. The analyses all contain a global *closure analysis* that computes information about higher-order control-flow. Sestoft proved in 1989 and 1991 that closure analysis is correct with respect to call-by-name and call-by-value semantics, but it remained open if correctness holds for arbitrary beta-reduction.

This article answers the question; both closure analysis and others are correct with respect to arbitrary beta-reduction. We also prove a subject-reduction result: closure information is still valid after beta-reduction. The core of our proof technique is to define closure analysis using a constraint system. The constraint system is equivalent to the closure analysis of Bondorf, which in turn is based on Sestoft's.

Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—semantics; D.3.2 [Programming Languages]: Language Classifications—applicative languages; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—logics of programs

General Terms: Languages, Theory

Additional Key Words and Phrases: Constraints, correctness proof, flow analysis

1. INTRODUCTION

1.1 Background

The optimization of higher-order functional languages requires powerful program analyses. The traditional framework for such analyses is *abstract interpretation*, and for *typed* languages, suitable abstract domains can often be defined by induction on the structure of types. For example, function spaces can be abstracted into function spaces. For *untyped* languages such as the λ -calculus, or dynamically typed languages such as Scheme, abstract domains cannot be defined by abstracting function spaces into function spaces. Other domains can be used, but it may then be difficult to relate the abstract interpretation to the *denotational* semantics. In this article we consider a style of program analysis where the result is an abstraction of the *operational* semantics.

In the past decade, program analyses of untyped languages has been presented

Author's address: Computer Science Department, Aarhus University, Ny Munkegade, DK–8000 Aarhus C, Denmark; email: palsberg@daimi.aau.dk.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM Transactions on Programming Languages and Systems, 17(1):47–62, January 1995. Also in Proc. CAAP'94, pages 276–290. ©

by Ayers [1992], Bondorf [1991], Consel [1990], Jones [1981], Heintze [1992], Sestoft [1989; 1991], Shivers [1991a; 1991b], Wand and Steckler [1994], and others. Although the analyses are used for rather different tasks such as type recovery, globalization, and binding-time analysis, they are all based on essentially the same idea:

Key idea. In the absence of types, define the abstract domains in terms of *program points*.

For example, consider the following λ -term:

 $(\lambda x.\lambda y.y(xI)(xK))\Delta$ where $I = \lambda a.a, K = \lambda b.\lambda c.b$, and $\Delta = \lambda d.dd$.

Giannini and Rocca [1988] proved that this strongly normalizing term has no higherorder polymorphic type. Still, a program analysis might answer basic questions such as:

(1) For every application point, which abstractions can be applied?

(2) For every abstraction, to which arguments can it be applied?

Each answer to such questions should be a subset of the program points in this particular λ -term. Thus, let us label all abstractions and applications. Also variables will be labeled: if a variable is bound, then it is labeled with the label of the λ that binds it, and if it is free, then with an arbitrary label. By introducing an explicit application symbol, we get the following abstract syntax for the above λ -term.

 $\begin{array}{l} (\lambda^1 x.\lambda^2 y.y^2 \ \mathbf{@}_7 \ (x^1 \ \mathbf{@}_8 \ I) \ \mathbf{@}_9 \ (x^1 \ \mathbf{@}_{10} \ K)) \ \mathbf{@}_{11} \ \Delta \\ \text{where} \ I = \lambda^3 a.a^3, \ K = \lambda^4 b.\lambda^5 c.b^4, \ \text{and} \ \Delta = \lambda^6 d.d^6 \ \mathbf{@}_{12} \ d^6. \end{array}$

An analysis might be able to find out that no matter how reduction proceeds:

- —"I can only be applied to I," that is, an abstraction with label 3 can only be applied to abstractions with label 3;
- "At the application point dd (in Δ) both I and K can be applied," that is, at an application point labeled 12 there can only be applied abstractions with labels 3 and 4; and
- —"the abstraction $\lambda c.b$ will never be applied," that is, at no application point can an abstraction with label 5 be applied.

The quoted sentences give the intuitive understanding of the precise statements that follow. In this particular example, the labels are rather unnecessary because no name clashes happen during any reduction and because I, K, and $\lambda c.b$ are in normal form. In the presence of name clashes or reduction under a λ , however, it is crucial to use sets of program points as the abstract values.

The above questions have turned out to be of paramount importance in many analyses of untyped functional programs. Following Sestoft and Bondorf, we will call any analysis that can answer them conservatively a *closure analysis*. On top of a closure analysis, one can build for example type recovery analyses, globalization analyses, and binding-time analyses. The closure analysis answers questions about higher-order control flow, and the extension answers the questions one is really interested in, for example, about type recovery. The role of closure analysis is thus as follows:

3

"Higher-order analysis = first-order analysis + closure analysis."

Closure analysis is useful for higher-order languages in general, for example, objectoriented languages (see Palsberg and Schwartzbach [1991; 1994b]). It is also useful for typed functional languages because type information is usually not specific enough to tell which functions among the type-correct ones are called at each application point.

Closure analysis and its extensions can be defined as abstract interpretations. They differ radically from traditional abstract interpretations, however, in that the abstract domain is defined in terms of the program to be analyzed. This means that such analyses are *global*: before the abstract domain can be defined, the complete program is required. Moreover, the program cannot take higher-order input because that would add program points. Also the minimal function graph approach to program analysis uses abstract domains defined in terms of the input program. In contrast, traditional abstract interpretations can analyze pieces of a program in isolation. We will refer to all analyses based on closure analysis as *flow analyses*.

Examples of large-scale implementations of such analyses can be found in the Similix system of Bondorf [Bondorf 1993; Bondorf and Danvy 1991], the Schism system of Consel [1990], and the system of Agesen et al. [1993] for analyzing Self programs [Ungar and Smith 1987]. The last of these implementations demonstrates that closure analysis can handle dynamic and multiple inheritance.

Closure analysis and its extensions have been formulated using constraints by others, for example, Heintze [1992; 1994], and Wand and Steckler [1994]. Their constraint systems are in spirit close to ours, although they are technically somewhat different. A key difference between Heintze's definition [Heintze 1994] and ours is that he attempts to avoid analyzing code that will not be executed under call-by-value. This goal is shared by an analysis of Palsberg and Schwartzbach [1992a]. The idea of defining program analyses using constraints over set variables is called *set-based analysis* by Heintze.

Sestoft [1989; 1991] proved that closure analysis is correct with respect to callby-name and call-by-value semantics, but it remained open if correctness holds for arbitrary beta-reduction.

1.2 Our Results

We prove that closure analysis is correct with respect to arbitrary beta-reduction. We also prove a subject-reduction result: closure information is still valid after beta-reduction. The correctness result implies that closure analysis is correct with respect to any reduction strategy.

- —We present a novel *specification* of closure analysis that allows arbitrary betareduction to take place and which subsumes all previous specifications.
- -We present a closure analysis that uses a *constraint system*. The constraint system characterizes the result of the analysis without specifying how it is computed. An example of such a constraint system is given in Section 1.3.
- —We prove that the constraint-based analysis is equivalent to the closure analysis of Bondorf [1991], which in turn is based on Sestoft's [Sestoft 1989]. We also

prove that these analyses are equivalent to a novel simplification of Bondorf's definition.

The proofs of correctness and subject-reduction then proceed by considering only the constraint-based definition of closure analysis.

In contrast to the closure analyses by abstract interpretation, the one using a constraint system does *not* depend on labels being distinct. This makes it possible to analyze a λ -term, beta-reduce it, and then analyze the result *without* relabeling first. The abstract interpretations might be modified to have this property also, but it would be somewhat messy. This indicates that a direct proof of correctness of such a modified abstract interpretation would be more complicated than the proof presented in this article.

Our technique for proving correctness generalizes without problems to analyses based on closure analysis. The following two results are not proved in this article:

- —The *safety analysis* of Palsberg and Schwartzbach [1992a; 1992b] is correct with respect to arbitrary beta-reduction. This follows from the subject-reduction property: terms stay safe after beta-reduction.
- —The *binding-time analysis* of Palsberg and Schwartzbach [1994a] that was proved correct by Palsberg [1993], can be proved correct more elegantly with our new technique.

The constraint-based definition of closure analysis is straightforward to extend to practical languages. For a medium-sized example see Palsberg and Schwartzbach [1994b] where the analysis is defined for an object-oriented language.

1.3 Example

The constraint system that expresses closure analysis of a λ -term is a set of Horn clauses. If the λ -term contains n abstractions and m applications, then the constraint system contains $n + (2 \times m \times n)$ constraints. Thus, the size of a constraint system is in the worst-case quadratic in the size of the λ -term. Space constraints disallow us to show a full-blown example involving name clashes and reduction under a λ , so consider instead the λ -term $(\lambda x.xx)(\lambda y.y)$ which has the abstract syntax $(\lambda^1 x.x^1 \ \mathfrak{O}_3 \ x^1) \ \mathfrak{O}_4 \ (\lambda^2 y.y^2)$. The constraint system that expresses closure analysis of this λ -term looks as follows.

From λ^1	$\{1\} \subseteq \llbracket \lambda^1 \rrbracket$	
From λ^2	$\{2\} \subseteq [\![\lambda^2]\!]$	
From \mathbf{Q}_3 and λ^1	$\begin{cases} \{1\} \subseteq \llbracket \nu^1 \rrbracket \Rightarrow \llbracket \nu^1 \rrbracket \Leftrightarrow \\ \{1\} \subset \llbracket \nu^1 \rrbracket \Rightarrow \llbracket \mathfrak{Q}_2 \rrbracket \end{cases}$	$ \begin{bmatrix} [\nu^1] \\ \llbracket \mathbf{Q}_2 \end{bmatrix} $
From Q_3 and λ^2	$\begin{cases} \{2\} \subseteq \llbracket \nu^1 \rrbracket \Rightarrow \llbracket \nu^1 \rrbracket \Leftrightarrow \llbracket \nu^2 \rrbracket \Leftrightarrow \llbracket \nu^1 \rrbracket \Leftrightarrow \llbracket \nu^2 \rrbracket $	$\begin{bmatrix} 1 & -3 \end{bmatrix}$ $\begin{bmatrix} 1 & -3 \end{bmatrix}$ $\begin{bmatrix} 1 & -3 \end{bmatrix}$
From \mathbf{Q}_4 and λ^1	$\begin{cases} \{2\} \subseteq \llbracket \nu \rrbracket \Rightarrow \llbracket \nu \rrbracket \\ \{1\} \subseteq \llbracket \lambda^1 \rrbracket \Rightarrow \llbracket \lambda^2 \rrbracket \\ \{1\} \subset \llbracket \lambda^1 \rrbracket \Rightarrow \llbracket \alpha \rrbracket \end{cases}$	$ = \llbracket \boldsymbol{\psi}_3 \rrbracket \\ \subseteq \llbracket \boldsymbol{\nu}^1 \rrbracket \\ = \llbracket \boldsymbol{\alpha}_1 \rrbracket $
From \mathbf{Q}_4 and λ^2	$\begin{cases} \{1\} \subseteq \llbracket \lambda \rrbracket \implies \llbracket \mathfrak{G} \rrbracket \end{cases} \llbracket \mathfrak{G} \rrbracket \\ \{2\} \subseteq \llbracket \lambda^1 \rrbracket \implies \llbracket \lambda^2 \rrbracket \\ \{2\} \subseteq \llbracket \lambda^1 \rrbracket \implies \llbracket \nu^2 \rrbracket \end{aligned}$	$\stackrel{=}{\subseteq} \llbracket v^2 \rrbracket$ $\stackrel{\subseteq}{\subseteq} \llbracket \mathfrak{Q}_4 \rrbracket$

Symbols of the forms $[\![\nu^l]\!]$, $[\![\lambda^l]\!]$, and $[\![\mathfrak{Q}_i]\!]$ are metavariables. They relate to variables with label l, abstractions with label l, and applications with label i, respectively. Notice that we do *not* assume, for example, that there is just one abstraction

with label l. The reason is that we want to do closure analysis of *all* terms, also those arising after beta-reduction which may copy terms and hence labels.

To the left of the constraints, we have indicated from where they arise. The first two constraints express that an abstraction may evaluate to an abstraction with the same label. The rest of the constraints come in pairs. For each application point Q_i and each abstraction with label l there are two constraints of the form:

 $\{l\} \subseteq$ "metavar. for operator of \mathbb{Q}_i " \Rightarrow "metavar. for operand of \mathbb{Q}_i " $\subseteq \llbracket \nu^l \rrbracket$ $\{l\} \subseteq$ "metavar. for operator of \mathbb{Q}_i " \Rightarrow "metavar. for body of abst." $\subseteq \llbracket \mathbb{Q}_i \rrbracket$

Such constraints can be read as:

- The first constraint. If the operator of \mathfrak{Q}_i evaluates to an abstraction with label l, then the bound variable of that abstraction may be substituted with everything to which the operand of \mathfrak{Q}_i can evaluate.
- —*The second constraint.* If the operator of \mathfrak{Q}_i evaluates to an abstraction with label l, then everything to which the body of the abstraction evaluates is also a possible result of evaluating the whole application \mathfrak{Q}_i .

In a solution of the constraint system, metavariables are assigned closure information. The minimal solution of the above constraint system is a mapping Lwhere:

$$\begin{split} & L\llbracket\lambda^1\rrbracket = \{1\} \\ & L\llbracket\lambda^2\rrbracket = L\llbracket\nu^1\rrbracket = L\llbracket\nu^2\rrbracket = L\llbracket \mathbf{0}_3\rrbracket = L\llbracket \mathbf{0}_4\rrbracket = \{2\} \end{split}$$

For example, the whole λ -term will, if normalizing, evaluate to an abstraction with label 2 $(L[[@_4]] = \{2\})$; at the application point $@_3$ there can only be applied abstractions with label 2 $(L[[\nu^1]] = \{2\})$; the application point $@_3$ is the only point where abstractions with label 2 can be applied $(L[[\lambda^1]] = \{1\})$; and such abstractions can only be applied to λ -terms that either do not normalize or evaluate to an abstraction with label 2 $(L[[\nu^2]] = \{2\})$.

One of our theorems says that the computed closure information is correct. One might also try to do closure analysis of the above λ -term using Bondorf's abstract interpretation; another of our theorems says that we will get the same result.

Now contract the only redex in the above λ -term. The result is a λ -term with abstract syntax $(\lambda^2 y. y^2) \, \mathfrak{a}_3 \, (\lambda^2 y. y^2)$. One third of our theorems says that the mapping L above gives correct closure information also for this λ -term.

In the following section we define three closure analyses: Bondorf's, a simpler abstract interpretation, and one in constraint form. In Section 3 we prove that they are equivalent, and in Section 4 we prove that they are correct.

2. CLOSURE ANALYSIS

Recall the λ -calculus [Barendregt 1981].

Definition 2.1. The language Λ of λ -terms has an abstract syntax which is defined by the grammar:

$$E ::= x^{l} \quad (\text{variable}) \\ \mid \lambda^{l} x.E \quad (\text{abstraction}) \\ \mid E_{1} @_{i} E_{2} \quad (\text{application}) \end{cases}$$

The labels on variables, abstraction symbols, and application symbols have no semantic impact; they mark program points. The label on a bound variable is the same as that on the λ that binds it. Labels are drawn from the infinite set Label. The symbols l, l', i range over labels. The labels and the application symbols are not part of the concrete syntax of Λ . We identify terms that are α -congruent. The α -conversion changes only bound variables, not labels. We assume the Variable Convention of Barendregt [1981]: when a λ -term occurs in this article, all bound variables are chosen to be different from the free variables. This can be achieved by renaming bound variables. An occurrence of $(\lambda^l x.E) @_i E'$ is called a redex. The semantics is as usual given by the rewriting-rule scheme:

$$(\lambda^l x.E) \ \mathfrak{O}_i \ E' \to E[E'/x^l] \qquad (\text{beta-reduction}) \ .$$

Here, $E[E'/x^l]$ denotes the term E with E' substituted for the free occurrences of x^l . Notice that by the Variable Convention, no renaming of bound variables is necessary when doing substitution. In particular, when we write $(\lambda^l y.E)[E'/x^{l'}]$, we have that $y^l \not\equiv x^{l'}$ and that y^l is not among the free variables of E'. Thus, $(\lambda^l y.E)[E'/x^{l'}] = \lambda^l y.(E[E'/x^{l'}])$. We write $E_S \to^* E_T$ to denote that E_T has been obtained from E_S by 0 or more beta-reductions. A term without redexes is in normal form.

The abstract domain for closure analysis of a λ -term E is called $\mathsf{CMap}(E)$ and is defined as follows.

Definition 2.2. A metavariable is of one of the forms $\llbracket \nu^{l} \rrbracket$, $\llbracket \lambda^{l} \rrbracket$, and $\llbracket \mathfrak{Q}_{i} \rrbracket$. The set of all metavariables is denoted Metavar. A λ -term is assigned a metavariable by the function var, which maps x^{l} to $\llbracket \nu^{l} \rrbracket$, $\lambda^{l} x.E$ to $\llbracket \lambda^{l} \rrbracket$, and $E_{1} \mathfrak{Q}_{i} E_{2}$ to $\llbracket \mathfrak{Q}_{i} \rrbracket$.

For a λ -term E, $\mathsf{Lab}(E)$ is the set of labels on abstractions (but not applications) occurring in E. Notice that $\mathsf{Lab}(E)$ is finite. The set $\mathsf{CSet}(E)$ is the powerset of $\mathsf{Lab}(E)$; $\mathsf{CSet}(E)$ with the inclusion ordering is a complete lattice. The set $\mathsf{CMap}(E)$ consists of the total functions from Metavar to $\mathsf{CSet}(E)$. The set $\mathsf{CEnv}(E)$ contains each function in $\mathsf{CMap}(E)$ when restricted to metavariables of the form $\llbracket \nu^{l} \rrbracket$. Both $\mathsf{CMap}(E)$ and $\mathsf{CEnv}(E)$ with pointwise ordering, written \sqsubseteq , are complete lattices where the least upper bound is written \sqcup . The function $\langle V \mapsto S \rangle$ maps the metavariable V to the set S and maps all other metavariables to the empty set. Finally, we define $upd V S L = \langle V \mapsto S \rangle \sqcup L$.

2.1 The Specification of Closure Analysis

We can then state precisely what a closure analysis is. An intuitive argument follows the formal definition.

Definition 2.1.1. For a λ -term E and for every $L \in \mathsf{CMap}(E)$, we define a binary relation T_L on λ -terms, as follows. $T_L(E_X, E_Y)$ holds if and only if the following four conditions hold:

- -If E_Y equals $\lambda^l x.E$, then $\{l\} \subseteq L(\mathsf{var}(E_X))$.
- —If E_Y contains $\lambda^{l'} y.(\lambda^l x.E)$, then E_X contains $\lambda^{l'} z.E'$ such that $\{l\} \subseteq L(\mathsf{var}(E'))$.
- —If E_Y contains $(\lambda^l x.E) \, \mathbb{Q}_i \, E_2$, then E_X contains $E_1 \, \mathbb{Q}_i \, E'_2$ such that $\{l\} \subseteq L(\mathsf{var}(E_1))$.

-If E_Y contains $E_1 \ \mathfrak{Q}_i \ (\lambda^l x. E)$, then E_X contains $E'_1 \ \mathfrak{Q}_i \ E_2$ such that $\{l\} \subseteq L(\operatorname{var}(E_2))$.

A closure analysis of E produces $L \in \mathsf{CMap}(E)$ such that if $E \to^* E'$, then $T_L(E, E')$.

Intuitively, if $E_X \to^* E_Y$, then we can get conservative information about the abstractions in E_Y by doing closure analysis of E_X . For example, the first condition in Definition 2.1.1 can be illustrated as follows.



In this case, E_Y is an abstraction with label l. Thus, E_X can evaluate to an abstraction with label l. The first condition says that in this case the mapping L must satisfy $\{l\} \subseteq L(var(E_X))$. In other words, the analysis must be aware that such an abstraction is a possible result of evaluating E_X .

The three other conditions in Definition 2.1.1 cover the cases where abstractions are proper subterms of E_Y . The second condition covers the case where an abstraction in E_Y is the body of yet another abstraction. The third and fourth conditions cover the cases where an abstraction is the operator and the operand of an application, respectively. Here, we will illustrate just the first of these three conditions; the others are similar.



In this case, E_Y contains an abstraction with label $l(\lambda^l x.E)$. This abstraction is in turn the body of an abstraction with label $l'(\lambda^l y.\lambda^l x.E)$. The second condition

in Definition 2.1.1 says that in this case there must be an abstraction in E_X with label $l'(\lambda^{l'}z.E')$, the bound variable may be different) such that the mapping Lsatisfies $\{l\} \subseteq L(var(E'))$. In other words, the analysis must be aware that some abstraction $\lambda^{l'}z.E'$ in E_X can evolve into an abstraction with a body being an abstraction with label l.

Notice the possibility that more than one abstraction in E_X has label l'. Thus, if we want closure information for "the body of the abstraction with label l'" we must compute the *union* of information for the bodies of *all* abstractions in E_X with label l'. A similar comment applies to the third and fourth condition in Definition 2.1.1. Such use of closure information is not of concern in this article, however.

2.2 Bondorf's Definition

We now recall the closure analysis of Bondorf [1991], with a few minor changes in the notation compared to his presentation. The analysis assumes that all labels are distinct. Bondorf's definition was originally given for a subset of Scheme; we have restricted it to the λ -calculus. Note that Bondorf's definition is based on Sestoft's [Sestoft 1989].

We have simplified Bondorf's definition as follows. Bondorf's original definition assigns *distinct* metavariables to different occurrences of a variable; in contrast we assign the *same* metavariable to each occurrence of a variable. The simplified definition is equivalent to Bondorf's original definition; see below.

We will use the notation that if $\lambda^{l} x.E$ is a subterm of the term to be analyzed, then the partial function *body* maps the label l to E.

Definition 2.2.1. We define

$$\begin{split} B: & (E:\Lambda) \to \mathsf{CMap}(E) \times \mathsf{CEnv}(E) \\ B(E) &= fix(\lambda(\mu,\rho).b(E)\mu\rho) \\ b: & (E:\Lambda) \to \mathsf{CMap}(E) \to \mathsf{CEnv}(E) \to \mathsf{CMap}(E) \times \mathsf{CEnv}(E) \\ & b(x^l)\mu\rho = (upd\,\llbracket \nu^l \rrbracket \, \rho \llbracket \nu^l \rrbracket \, \mu, \rho) \\ & b(\lambda^l x.E)\mu\rho = \operatorname{let}\,(\mu',\rho') \,\operatorname{be}\, b(E)\mu\rho \\ & \text{ in } (upd\,\llbracket \lambda^l \rrbracket \, \{l\} \, \mu',\rho') \\ & b(E_1 \ \mathfrak{e}_i \ E_2)\mu\rho = \operatorname{let}\,(\mu',\rho') \,\operatorname{be}\, (b(E_1)\mu\rho) \sqcup (b(E_2)\mu\rho) \,\operatorname{in} \\ & \operatorname{let}\, c \,\operatorname{be}\, \mu'(\operatorname{var}(E_1)) \,\operatorname{in} \\ & \operatorname{let}\, \mu'' \,\operatorname{be}\, upd\,\llbracket \mathfrak{e}_i \rrbracket \, (\sqcup_{l \in c}\, \mu'(\operatorname{var}(body(l)))) \, \mu' \,\operatorname{in} \\ & \operatorname{let}\, \rho'' \,\operatorname{be}\, \rho' \sqcup (\sqcup_{l \in c}\, (upd\,\llbracket \nu^l \rrbracket \, \mu'(\operatorname{var}(E_2))) \, \rho') \\ & \operatorname{in}\,(\mu'',\rho'') \, . \end{split}$$

We can now do closure analysis of E by computing fst(B(E)).

If we modify the above definition such that different occurrences of a variable are assigned distinct metavariables, then we obtain Bondorf's original definition. That definition will assign the *same* set to all metavariables for the occurrences of a given variable, and moreover, the computed closure information will be the same as that computed by the stated analysis (we leave the details to the reader).

2.3 A Simpler Abstract Interpretation

Bondorf's definition can be simplified considerably. To see why, consider the second component of $\mathsf{CMap}(E) \times \mathsf{CEnv}(E)$. This component is updated only in

9

 $b(E_1 \ \mathfrak{Q}_i \ E_2)\mu\rho$ and read only in $b(x^l)\mu\rho$. The key observation is that both these operations can be done on the first component instead. Thus, we can omit the use of $\mathsf{CEnv}(E)$. By rewriting Bondorf's definition according to this observation, we arrive at the following definition. As with Bondorf's definition, we assume that all labels are distinct.

Definition 2.3.1. We define

We can now do closure analysis of E by computing fix(m(E)).

A key question is: is the simpler abstract interpretation equivalent to Bondorf's? We might attempt to prove this using fixed-point induction, but we find it much easier to do using a particular constraint system as a "stepping stone."

2.4 A Constraint System

For a λ -term E, the constraint system is a finite set of Horn clauses over inclusions of the form $P \subseteq P'$, where P and P' are either metavariables or elements of $\mathsf{CSet}(E)$. A solution of such a system is an element of $\mathsf{CMap}(E)$ that satisfies all Horn clauses.

The constraint system is defined in terms of the λ -term to be analyzed. We need *not* assume that all labels are distinct.

The set $R(E_1 \ \mathbf{Q}_i \ E_2, \lambda^l x. E)$ consists of the two elements

$$\begin{aligned} \{l\} &\subseteq \mathsf{var}(E_1) \Rightarrow \mathsf{var}(E_2) \subseteq \llbracket \nu^l \rrbracket \\ \{l\} &\subseteq \mathsf{var}(E_1) \Rightarrow \mathsf{var}(E) \subseteq \llbracket \mathfrak{G}_i \rrbracket \end{aligned}$$

For a λ -term E, the constraint system C(E) is the union of the following sets of constraints.

- —For every $\lambda^l x.E'$ in E, the singleton constraint set consisting of $\{l\} \subseteq [\![\lambda^l]\!]$.
- —For every $E_1 \ \mathbb{Q}_i \ E_2$ in E and for every $\lambda^l x \cdot E'$ in E, the set $R(E_1 \ \mathbb{Q}_i \ E_2, \lambda^l x \cdot E')$.

Each C(E) has a least solution, namely, the intersection of all solutions.

We can now do closure analysis of E by computing a solution of C(E). The canonical choice of solution is of course the least one.

The closure analysis of Bondorf and Jørgensen [1993] can be understood as adding two constraints to each $R(E_1 \ \mathfrak{Q}_i \ E_2, \lambda^l x.E')$ such that in effect the inclusions $\operatorname{var}(E_2) \subseteq \llbracket \nu^l \rrbracket$ and $\operatorname{var}(E) \subseteq \llbracket \mathfrak{Q}_i \rrbracket$ are changed to equalities. Thus, their closure analysis computes more approximate information than ours. In return, their analysis can be computed in almost-linear time, using an other formulation of the problem [Bondorf and Jørgensen 1993], whereas the fastest known algorithm for computing the least solution of C(E) uses transitive closure (see Palsberg and Schwartzbach [1992a; 1994b]).

3. EQUIVALENCE

We now prove that the three closure analyses defined in Section 2 are equivalent (when applied to λ -terms where all labels are distinct). We will use the standard

terminology that μ is a *prefixed point* of m(E) if $m(E)\mu \sqsubseteq \mu$.

LEMMA 3.1. If μ is a prefixed point of m(E), then so is it of m(E') for every subterm E' of E.

PROOF. By induction on the structure of E.

LEMMA 3.2. C(E) has least solution fix(m(E)).

PROOF. We prove a stronger property: the solutions of C(E) are exactly the prefixed points of m(E). There are two inclusions to be considered.

First, we prove that every solution of C(E) is a prefixed point of m(E). We proceed by induction on the structure of E. In the base case, consider x^l . Clearly, every μ is a prefixed point of $m(x^l)$. In the induction step, consider first $\lambda^l x.E$. Suppose μ is a solution of $C(\lambda^l x.E)$. Then μ is also a solution of C(E), so by the induction hypothesis, μ is a prefixed point of m(E). Hence, we get $m(\lambda^l x.E)\mu =$ $(m(E)\mu) \sqcup \langle [\lambda^l] \mapsto \{l\} \rangle \sqsubseteq \mu \sqcup \langle [\lambda^l] \mapsto \{l\} \rangle = \mu$, by using the definition of m, that μ is a prefixed point of m(E), and that since $C(\lambda^l x.E)$ has solution μ , $\{l\} \subseteq \mu([\lambda^l])$.

Consider then $E_1 \, \mathbb{Q}_i \, E_2$. Suppose μ is a solution of $C(E_1 \, \mathbb{Q}_i \, E_2)$. Then μ is also a solution of $C(E_1)$ and $C(E_2)$, so by the induction hypothesis, μ is a prefixed point of $m(E_1)$ and $m(E_2)$. Hence, we get $m(E_1 \, \mathbb{Q}_i \, E_2)\mu = \mu$, by using the definition of m, that μ is a prefixed point of $m(E_1)$ and $m(E_2)$, and that $C(E_1 \, \mathbb{Q}_i \, E_2)$ has solution μ .

Second, we prove that every prefixed point of m(E) is a solution of C(E). We proceed by induction on the structure of E. In the base case, consider x^l . Clearly, every μ is a solution of $C(x^l)$. In the induction step, consider first $\lambda^l x.E'$. Suppose μ is a prefixed point of $m(\lambda^l x.E')$. Then, by Lemma 3.1, μ is also a prefixed point of m(E'). By the induction hypothesis, μ is a solution of C(E'). Thus, we need to prove that μ satisfies $\{l\} \subseteq [\![\lambda^l]\!]$ and for every $E_1 \ \mathbb{Q}_i \ E_2$ in $E', R(E_1 \ \mathbb{Q}_i \ E_2, \lambda^l x.E')$. For the first of these, use that μ is a prefixed point of $m(\lambda^l x.E')$ to get $\mu \supseteq m(\lambda^l x.E')\mu = (m(E')\mu) \sqcup \langle [\![\lambda^l]\!] \mapsto \{l\} \rangle \supseteq \langle [\![\lambda^l]\!] \mapsto \{l\} \rangle$, from which the result follows. For the second one, consider $E_1 \ \mathbb{Q}_i \ E_2$ in E'. By Lemma 3.1, μ is also a prefixed point of $m(E_1 \ \mathbb{Q}_i \ E_2)$. Using the assumption that we get $\mu \supseteq m(E_1 \ \mathbb{Q}_i \ E_2)\mu \supseteq \bigsqcup_{l \in \mu(\operatorname{Var}(E_1))}(\langle [\![\nu^l]\!] \mapsto \mu(\operatorname{var}(E_2)) \rangle \sqcup \langle [\![\mathbb{Q}_i]\!] \mapsto$ $\mu(\operatorname{var}(body(l))) \rangle$, from which the result follows.

Consider then $E_1 \ \mathbb{Q}_i \ E_2$. Suppose μ is a prefixed point of $m(E_1 \ \mathbb{Q}_i \ E_2)$. Then, by Lemma 3.1, μ is also a prefixed point of both $m(E_1)$ and $m(E_2)$. By the induction hypothesis, μ is a solution of both $C(E_1)$ and $C(E_2)$. Thus, we need to prove that for every $\lambda^l x.E'$ in $E_1 \ \mathbb{Q}_i \ E_2$, μ satisfies $R(E_1 \ \mathbb{Q}_i \ E_2, \lambda^l x.E')$. From μ being a prefixed point of $m(E_1 \ \mathbb{Q}_i \ E_2)$, we get $\mu \supseteq m(E_1 \ \mathbb{Q}_i \ E_2)\mu \supseteq \bigsqcup_{l \in \mu(\mathsf{var}(E_1))}(\langle \llbracket \nu^l \rrbracket \mapsto$ $\mu(\mathsf{var}(E_2)) \rangle \sqcup \langle \llbracket \mathbb{Q}_i \rrbracket \mapsto \mu(\mathsf{var}(body(l))) \rangle$, from which the result follows. \Box

LEMMA 3.3. C(E) has least solution fst(B(E)).

PROOF. Similar to the proof of Lemma 3.2. \Box

THEOREM 3.4. The three closure analyses defined in Section 2 are equivalent.

PROOF. Combine Lemmas 3.2 and 3.3. \Box

4. CORRECTNESS

We now prove that the three closure analyses defined in Section 2 are correct. The key is to define an entailment relation $A \rightsquigarrow A'$ (Definition 4.1) meaning that all constraints in the constraint system A' can be logically derived from those in A. A central result (Theorem 4.10) is that if $E_X \to E_Y$, then $C(E_X) \rightsquigarrow C(E_Y)$. This theorem is proved without at all considering solutions of the involved constraint systems.

Definition 4.1. If A is a constraint system, and H is a Horn clause, then the judgment $A \vdash H$ ("A entails H") holds if it is derivable using the following five rules:

$$\overline{A \vdash H} \quad \text{if } H \in A \tag{Discharge}$$

 $\overline{A \vdash P \subseteq P} \tag{Reflexivity}$

$$\frac{A \vdash P \subseteq P' \qquad A \vdash P' \subseteq P''}{A \vdash P \subseteq P''}$$
(Transitivity)

$$\frac{A \vdash X \qquad A \vdash X \Rightarrow Y}{A \vdash Y}$$
(Modus Ponens)

$$\frac{A \vdash P \subseteq P'' \Rightarrow Q' \subseteq Q'' \qquad A \vdash P' \subseteq P'' \qquad A \vdash Q \subseteq Q'}{A \vdash P \subseteq P' \Rightarrow Q \subseteq Q''}$$
(Weakening)

If A, A' are constraint systems, then $A \rightsquigarrow A'$ if and only if $\forall H \in A' : A \vdash H$.

LEMMA 4.2. \rightsquigarrow is reflexive, transitive, and solution-preserving. If $A \supseteq A'$, then $A \rightsquigarrow A'$.

PROOF. The last property is immediate using Discharge. Reflexivity of \rightsquigarrow is a consequence of the last property. For transitivity of \rightsquigarrow , suppose $A \rightsquigarrow A'$ and $A' \rightsquigarrow A''$. The statement "if $A' \vdash H$ then $A \vdash H$ " can be proved by induction on the structure of the proof of $A' \vdash H$. To prove $A \rightsquigarrow A''$, suppose then that $H \in A''$. From $A' \rightsquigarrow A''$ we get $A' \vdash H$, and from the above statement we finally get $A \vdash H$. To prove that \rightsquigarrow is solution-preserving, suppose $A \rightsquigarrow A'$ and that A has solution L. We need to prove that for every $H \in A'$, H has solution L. This can be proved by induction on the structure of the proof of $A \vdash H$.

The following lemmas are structured such that Modus Ponens is only used in the proof of Lemma 4.3, and Weakening is only used in the proof of Lemma 4.6.

To aid intuition we can informally read $A \vdash \mathsf{var}(E) \subseteq \mathsf{var}(E')$ as "under the assumption A, the λ -term E has smaller flow information than the λ -term E'."

The next lemma states that two specific constraints can be derived from the constraint system for a redex. Informally, the first constraint says that the argument has smaller flow information than the bound variable, and the second constraint says that the body of the abstraction has smaller flow information than the whole redex.

LEMMA 4.3. If $A \rightsquigarrow C((\lambda^l x.E) \ \mathfrak{Q}_i \ E_2)$, then $A \vdash \mathsf{var}(E_2) \subseteq \llbracket \nu^l \rrbracket$ and $A \vdash \mathsf{var}(E) \subseteq \llbracket \mathfrak{Q}_i \rrbracket$.

PROOF. We have $A \vdash \{l\} \subseteq [\![\lambda^l]\!]$ and $A \rightsquigarrow R((\lambda^l x.E) @_i E_2, \lambda^l x.E)$. The result then follows from $\operatorname{var}(\lambda^l x.E) = [\![\lambda^l]\!]$ and Modus Ponens. \Box

The next lemma is a substitution lemma. Informally, it states that a λ -term gets smaller flow information if a subterm gets substituted by one with smaller flow information.

LEMMA 4.4. If $A \vdash \operatorname{var}(U) \subseteq \llbracket \nu^{l} \rrbracket$, then $A \vdash \operatorname{var}(E[U/x^{l}]) \subseteq \operatorname{var}(E)$.

PROOF. By induction on the structure of E, using Reflexivity repeatedly.

Informally, the next lemma states that beta-reduction creates λ -terms with smaller flow information.

LEMMA 4.5. If
$$A \sim C(E_X)$$
 and $E_X \to E_Y$, then $A \vdash \mathsf{var}(E_Y) \subseteq \mathsf{var}(E_X)$.

PROOF. We proceed by induction on the structure of E_X . In the base case, consider x^l . The conclusion is immediate since x^l is in normal form.

In the induction step, consider first $\lambda^l x.E$. Suppose $E \to E'$. Notice that $\operatorname{var}(\lambda^l x.E) = \operatorname{var}(\lambda^l x.E') = [\![\lambda^l]\!]$. Using Reflexivity we get $A \vdash [\![\lambda^l]\!] \subseteq [\![\lambda^l]\!]$.

Consider finally $E_1 \ \mathfrak{Q}_i E_2$. There are three cases. Suppose $E_1 \to E'_1$. Notice that $\operatorname{var}(E_1 \ \mathfrak{Q}_i E_2) = \operatorname{var}(E'_1 \ \mathfrak{Q}_i E_2) = \llbracket \mathfrak{Q}_i \rrbracket$. Using Reflexivity we get $A \vdash \llbracket \mathfrak{Q}_i \rrbracket \subseteq \llbracket \mathfrak{Q}_i \rrbracket$.

Suppose then that $E_2 \to E'_2$. Notice that $\operatorname{var}(E_1 \ \mathbb{Q}_i \ E_2) = \operatorname{var}(E'_1 \ \mathbb{Q}_i \ E_2) = \llbracket \mathbb{Q}_i \rrbracket$. Using Reflexivity we get $A \vdash \llbracket \mathbb{Q}_i \rrbracket \subseteq \llbracket \mathbb{Q}_i \rrbracket$.

Suppose then that $E_1 = \lambda^l x.E$ and that $E_1 @_i E_2 \to E[E_2/x^l]$. From Lemma 4.3 we get $A \vdash \mathsf{var}(E_2) \subseteq \llbracket \nu^l \rrbracket$ and $A \vdash \mathsf{var}(E) \subseteq \llbracket @_i \rrbracket$. From the former of these and Lemma 4.4 we get $A \vdash \mathsf{var}(E[E_2/x^l]) \subseteq \mathsf{var}(E)$. Using Transitivity we can finally conclude that $A \vdash \mathsf{var}(E[E_2/x^l]) \subseteq \llbracket @_i \rrbracket$. \Box

Informally, the next lemma states that entailment is robust under beta-reduction and substitution.

LEMMA 4.6. Suppose $A \rightsquigarrow R(E_1 \ \mathfrak{G}_i \ E_2, \lambda^l x. E_3) \cup C(E_1) \cup C(E_2) \cup C(E_3)$. If $E_j = E'_j \text{ or } E_j \rightarrow E'_j \text{ or } E'_j = E_j[U_j/x_j^{l_j}]$ where $A \vdash \operatorname{var}(U_j) \subseteq \operatorname{var}(x_j^{l_j})$ for $j \in 1..3$, then $A \rightsquigarrow R(E'_1 \ \mathfrak{G}_i \ E'_2, \lambda^l x. E'_3)$.

PROOF. For $j \in 1..3$, we get $A \vdash \mathsf{var}(E'_j) \subseteq \mathsf{var}(E_j)$ from either Reflexivity, Lemma 4.5, or Lemma 4.4. The result then follows using Weakening. \Box

The following definition is needed for stating and proving Lemma 4.9.

Definition 4.7. The set W(E, E') is the union of the following sets of constraints. $-C(E) \cup C(E').$

—For every $E_1 \ \mathfrak{Q}_i \ E_2$ in E and for every $\lambda^l x \cdot E_3$ in E', the set $R(E_1 \ \mathfrak{Q}_i \ E_2, \lambda^l x \cdot E_3)$.

—For every $E_1 \ \mathfrak{Q}_i \ E_2$ in E' and for every $\lambda^l x \cdot E_3$ in E, the set $R(E_1 \ \mathfrak{Q}_i \ E_2, \lambda^l x \cdot E_3)$.

LEMMA 4.8. $W(E_1, E_2) \subseteq C(E_1 \ \mathfrak{G}_i \ E_2)$. Moreover, if E'_1 is a subterm of E_1 , then $W(E'_1, E_2) \subseteq W(E_1, E_2)$.

PROOF. Immediate. \square

13

The next lemma is a substitution lemma. Like Lemma 4.6, it states that entailment is robust under substitution.

LEMMA 4.9. If $A \rightsquigarrow W(E, U)$ and $A \vdash \mathsf{var}(U) \subseteq \llbracket \nu^l \rrbracket$, then $A \rightsquigarrow C(E[U/x^l])$.

PROOF. Let ρ denote the substitution $[U/x^l]$. We proceed by induction on the structure of E. In the base case, consider $E = y^{l'}$. If $x^l \equiv y^{l'}$, then $E\rho = U$ so the result follows from $A \rightsquigarrow W(E, U)$ and Lemma 4.2. If $x^l \not\equiv y^{l'}$, then $E\rho = E$ so again the result follows from $A \rightsquigarrow W(E, U)$ and Lemma 4.2.

In the induction step, consider first $E = \lambda^{l'} y.E'$. If $x^{l} \equiv y^{l'}$, then $E\rho = E$ so also in this case the result follows from $A \rightsquigarrow W(E, U)$ and Lemma 4.2. If $x^{l} \not\equiv y^{l'}$, $E\rho = \lambda^{l'} y.(E'\rho)$. By the induction hypothesis, $A \rightsquigarrow C(E'\rho)$. Thus, we need to show $A \vdash \{l'\} \subseteq [\lambda^{l'}]$ and for every $E_1 @_i E_2$ in $E'\rho$, $A \rightsquigarrow R(E_1 @_i E_2, \lambda^{l'} y.(E'\rho))$. The first follows from $A \rightsquigarrow C(\lambda^{l'} y.E')$. For the second, consider any $E_1 @_i E_2$ in $E'\rho$. Notice that either $E_1 @_i E_2$ is a subterm of E', or $E_1 @_i E_2 = (E'_1 @_i E'_2)\rho =$ $(E'_1\rho) @_i (E'_2\rho)$ where $E'_1 @_i E'_2$ is a subterm of E', or $E_1 @_i E_2$ is a subterm of U. In each case the result follows from $A \rightsquigarrow W(E, U)$ and Lemma 4.6.

Consider finally $E = E_1 \ \mathfrak{a}_i \ E_2$. Notice that $(E_1 \ \mathfrak{a}_i \ E_2)\rho = (E_1\rho) \ \mathfrak{a}_i \ (E_2\rho)$. By the induction hypothesis, $A \rightsquigarrow C(E_1\rho) \cup C(E_2\rho)$. Thus, we need to show that for every $\lambda^{l'}y.E'$ in $(E_1 \ \mathfrak{a}_i \ E_2)\rho$, $A \rightsquigarrow R((E_1 \ \mathfrak{a}_i \ E_2)\rho, \lambda^{l'}y.E')$. Consider any $\lambda^{l'}y.E'$ in $(E_1 \ \mathfrak{a}_i \ E_2)\rho$. Notice that either $\lambda^{l'}y.E'$ is a subterm of $E_1 \ \mathfrak{a}_i \ E_2$, or $\lambda^{l'}y.E' = \lambda^{l'}y.(E'\rho)$ where $\lambda^{l'}y.E'$ is a subterm of $E_1 \ \mathfrak{a}_i \ E_2$, or $\lambda^{l'}y.E'$ is a subterm of U. In each case the result follows from $A \rightsquigarrow W(E, U)$ and Lemma 4.6. \Box

We can now prove that if we beta-reduce E_X to E_Y , then the constraint system for E_X entails the constraint system for E_Y .

THEOREM 4.10. If $E_X \to E_Y$, then $C(E_X) \rightsquigarrow C(E_Y)$.

PROOF. We proceed by induction on the structure of E_X . In the base case of x^l , the conclusion is immediate since x^l is in normal form.

In the induction step, consider first $\lambda^l x.E$. Suppose $E \to E'$. By the induction hypothesis, $C(E) \to C(E')$, so also $C(\lambda^l x.E) \to C(E')$. Thus, we need to show $C(\lambda^l x.E) \vdash \{l\} \subseteq [\![\lambda^l]\!]$ and for every $E_1 \ @_i \ E_2$ in $\lambda^l x.E'$, $C(\lambda^l x.E) \to R(E_1 \ @_i \ E_2, \lambda^l x.E')$. The first follows using Discharge. For the second, there are four cases. Notice that by Discharge we have $C(\lambda^l x.E) \to R(E'_1 \ @_i \ E'_2, \lambda^l x.E)$ for every $E'_1 \ @_i \ E'_2$ in $\lambda^l x.E$. In the first case, suppose $E_1 \ @_i \ E_2$ is also a subterm of $\lambda^l x.E$. The result then follows from Lemma 4.6. In the second case, consider a subterm $E'_1 \ @_i \ E'_2$ of $\lambda^l x.E$ such that $E'_1 \to E_1$. Again, the result follows from Lemma 4.6. In the fourth case, consider a subterm $E'_1 \ @_i \ E'_2$ of $\lambda^l x.E$ such that $E_1 \ @_i \ E_2 = (E'_1 \ @_i \ E'_2)[E_S/y^{l'}]$. The substitution arises because of the contraction of a redex. From Lemma 4.6.

Consider finally $E_1 \, \mathbb{Q}_i \, E_2$. For every $\lambda^l x \cdot E$ in $E_1 \, \mathbb{Q}_i \, E_2$, we have $C(E_1 \, \mathbb{Q}_i \, E_2) \rightsquigarrow R(E_1 \, \mathbb{Q}_i \, E_2, \lambda^l x \cdot E)$. There are three cases.

Suppose that $E_1 \to E'_1$. By the induction hypothesis, $C(E_1) \rightsquigarrow C(E'_1)$, so also $C(E_1 \ \mathfrak{Q}_i \ E_2) \rightsquigarrow C(E'_1)$. Thus we need to show that for every $\lambda^l x.E'$ in $E'_1 \ \mathfrak{Q}_i \ E_2$, $C(E_1 \ \mathfrak{Q}_i \ E_2) \rightsquigarrow R(E'_1 \ \mathfrak{Q}_i \ E_2, \lambda^l x.E')$. There are three cases. In the first case, suppose $\lambda^l x.E'$ is a subterm of $E_1 \ \mathfrak{Q}_i \ E_2$. The result then follows from Lemma 4.6.

In the second case, consider a subterm $\lambda^l x.E$ of $E_1 @_i E_2$ such that $E \to E'$. Again, the result follows from Lemma 4.6. In the third case, consider a subterm $\lambda^l x.E$ of $E_1 @_i E_2$ such that $\lambda^l x.E' = \lambda^l x.(E[E_S/y^{l'}])$. The substitution arises because of the contraction of a redex. From Lemma 4.3 we see $C(E_1 @_i E_2) \vdash \operatorname{var}(E_S) \subseteq [\![\nu^{l'}]\!]$. The result then follows from Lemma 4.6.

Suppose then that $E_2 \to E'_2$. The proof in this case is similar to the case of $E_1 \to E'_1$ so we omit the details.

Suppose then that $E_1 = \lambda^l x \cdot E$ and that $E_1 @_i E_2 \to E[E_2/x^l]$. From Lemma 4.3 we see $C(E_1 @_i E_2) \vdash \mathsf{var}(E_2) \subseteq \llbracket \nu^l \rrbracket$. From Lemma 4.8 we see that $W(E, E_2) \subseteq C(E_1 @_i E_2)$. The result then follows from Lemma 4.9. \Box

THEOREM 4.11. The three closure analyses defined in Section 2 are correct.

PROOF. From Theorem 3.4 we see that the three analyses are equivalent when applied to λ -terms where all labels are distinct. Thus, it is sufficient to prove that the one defined using a constraint system is correct. The proof has two steps.

In Step 1, use Lemmas 4.3, 4.4, and 4.5 to prove that if $A \rightsquigarrow C(E_X)$ and $E_X \rightarrow E_Y$, then both of the following properties hold:

—If E_Y contains $\lambda^l y.E$, then E_X contains $\lambda^l z.E'$ such that $A \vdash \mathsf{var}(E) \subseteq \mathsf{var}(E')$.

-If E_Y contains $E_1 @_i E_2$, then E_X contains $E'_1 @_i E'_2$ such that $A \vdash var(E_1) \subseteq var(E'_1)$ and $A \vdash var(E_2) \subseteq var(E'_2)$.

In Step 2, suppose $C(E_X)$ has solution L, and suppose $E_X \to^* E_Y$. We will prove $T_L(E_X, E_Y)$ by induction on the length of $E_X \to^* E_Y$.

In the base case, $T_L(E_X, E_X)$ is immediate. In the induction step, suppose $E_X \to E_Z \to^n E_Y$. By Theorem 4.10, $C(E_X) \rightsquigarrow C(E_Z)$. By Lemma 4.2, $C(E_Z)$ has solution L. By the induction hypothesis, $T_L(E_Z, E_Y)$. To prove $T_L(E_X, E_Y)$, there are four cases to be considered.

First suppose $E_Y = \lambda^l x.E$. From $T_L(E_Z, E_Y)$ we get $\{l\} \subseteq L(\mathsf{var}(E_Z))$. From Lemma 4.5 we get $C(E_X) \vdash \mathsf{var}(E_Z) \subseteq \mathsf{var}(E_X)$. Finally, the result follows by using that $C(E_X)$ has solution L.

Then suppose E_Y contains $\lambda^{l'} y.(\lambda^l x.E)$. From $T_L(E_Z, E_Y)$ we get that E_Z contains $\lambda^{l'} z.E'$ such that $\{l\} \subseteq L(\operatorname{var}(E'))$. From Step 1 of this proof, we get that E_X contains $\lambda^{l'} w.E''$ such that $C(E_X) \vdash \operatorname{var}(E') \subseteq \operatorname{var}(E'')$. Finally, the result follows by using that $C(E_X)$ has solution L.

In the last two cases, suppose E_Y contains either $(\lambda^l x.E) \, \mathfrak{a}_i \, E_2$ or $E_1 \, \mathfrak{a}_i \, (\lambda^l x.E)$, respectively. Both cases are similar to the second one, so we omit the details. \Box

Finally, we prove our subject-reduction result.

THEOREM 4.12. If C(E) has solution L and $E \to E'$, then C(E') has solution L.

PROOF. Immediate from Theorem 4.10 and Lemma 4.2. \Box

ACKNOWLEDGMENTS

The author thanks Torben Amtoft, Nils Klarlund, and the anonymous referees for helpful comments on a draft of the article. REFERENCES

- AGESEN, O., PALSBERG, J., AND SCHWARTZBACH, M. I. 1993. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of ECOOP'93, 7th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, New York, 247–267.
- AYERS, A. 1992. Efficient closure analysis with reachability. In Proceedings of WSA'92, Analyse Statique. IRISA, Rennes, France, 126–134.
- BARENDREGT, H. P. 1981. The Lambda Calculus: Its Syntax and Semantics. North-Holland, Amsterdam.
- BONDORF, A. 1993. *Similix 5.0 Manual.* DIKU, University of Copenhagen, Denmark. Included in Similix 5.0 distribution.
- BONDORF, A. 1991. Automatic autoprojection of higher order recursive equations. Sci. Comput. Program. 17, 1–3 (Dec.), 3–34.
- BONDORF, A. AND DANVY, O. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. Sci. Comput. Program. 16, 151–195.
- BONDORF, A. AND JØRGENSEN, J. 1993. Efficient analyses for realistic off-line partial evaluation. J. Functional Program. 3, 3, 315–346.
- CONSEL, C. 1990. Binding time analysis for higher order untyped functional languages. In Proceedings of the ACM Conference on Lisp and Functional Programming. ACM, New York, 264–272.
- GIANNINI, P. AND ROCCA, S. R. D. 1988. Characterization of typings in polymorphic type discipline. In Proceedings of LICS'88, 3rd Annual Symposium on Logic in Computer Science. IEEE, New York, 61–70.
- HEINTZE, N. 1994. Set-based analysis of ML programs. In Proceedings of the ACM Conference on LISP and Functional Programming. ACM, New York, 306–317.
- HEINTZE, N. 1992. Set based program analysis. Ph. D. thesis, CMU–CS–92–201, Carnegie Mellon University, Pittsburgh, Pa.
- JONES, N. D. 1981. Flow analysis of lambda expressions. In Proceedings of the 8th Colloquium on Automata, Languages, and Programming. Lecture Notes in Computer Science, vol. 115. Springer-Verlag, New York, 114–128.
- PALSBERG, J. 1993. Correctness of binding-time analysis. J. Functional Program. 3, 3, 347-363.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1994a. Binding-time analysis: Abstract interpretation versus type inference. In Proceedings of ICCL'94, 5th IEEE International Conference on Computer Languages. IEEE, New York, 289–298.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1994b. *Object-Oriented Type Systems*. John Wiley and Sons, New York.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1992a. Safety analysis versus type inference. *Inf. Comput.* To be published.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1992b. Safety analysis versus type inference for partial types. Inf. Process. Lett. 43, 175–180.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In Proceedings of OOPSLA'91, ACM SIGPLAN 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications. ACM, New York, 146–161.
- SESTOFT, P. 1991. Analysis and efficient implementation of functional programs. Ph. D. thesis, DIKU, University of Copenhagen.
- SESTOFT, P. 1989. Replacing function parameters by global variables. M.S. thesis, DIKU, University of Copenhagen.
- SHIVERS, O. 1991a. Control-flow analysis of higher-order languages. Ph. D. thesis, CMU–CS– 91–145, Carnegie Mellon University, Pittsburgh, Pa.
- SHIVERS, O. 1991b. Data-flow analysis and type recovery in Scheme. In *Topics in Advanced Language Implementation*, P. Lee, Ed. MIT Press, Cambridge, Mass., 47–87.
- UNGAR, D. AND SMITH, R. B. 1987. SELF: The power of simplicity. In Proceedings of OOP-SLA'87, Object-Oriented Programming Systems, Languages and Applications. ACM, New

York, 227–241. Also published in Lisp and Symbolic Computation 4(3), Kluwer Acadamic Publishers, June 1991.

WAND, M. AND STECKLER, P. 1994. Selective and lightweight closure conversion. In Proceedings of POPL'94, 21st Annual Symposium on Principles of Programming Languages. ACM, New York, 434–445.

Received May 1994; revised October 1994; accepted October 1994
Access Rights Analysis for Java

Larry Koved IBM T.J. Watson Research Center P.O. Box 704 Yorktown Heights, New York 10598 koved@us.ibm.com Marco Pistoia IBM T.J. Watson Research Center P.O. Box 704 Yorktown Heights, New York 10598 pistoia@us.ibm.com Aaron Kershenbaum IBM T.J. Watson Research Center P.O. Box 704 Yorktown Heights, New York 10598 aaronk@us.ibm.com

ABSTRACT

Java[™] 2 has a security architecture that protects systems from unauthorized access by mobile or statically configured code. The problem is in manually determining the set of security access rights required to execute a library or application. The commonly used strategy is to execute the code, note authorization failures, allocate additional access rights, and test again. This process iterates until the code successfully runs for the test cases in hand. Test cases usually do not cover all paths through the code, so failures can occur in deployed systems. Conversely, a broad set of access rights is allocated to the code to prevent authorization failures from occurring. However, this often leads to a violation of the "Principle of Least Privilege."

This paper presents a technique for computing the access rights requirements by using a context sensitive, flow sensitive, interprocedural data flow analysis. By using this analysis, we compute at each program point the set of access rights required by the code. We model features such as multi-threading, implicitly defined security policies, the semantics of the Permission.implies method and generation of a security policy description. We implemented the algorithms and present the results of our analysis on a set of programs. While the analysis techniques described in this paper are in the context of Java code, the basic techniques are applicable to access rights analysis issues in non-Java-based systems.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Computer-aided software engineering (CASE).

D.2.4 [Software/Program Verification]: Format methods, Validation.

D.2.5 [**Testing and Debugging**]: Code inspections and walk-throughs, Diagnostics, Testing tools (e.g., data generators, coverage testing), Tracing.

General Terms

Security, Languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

Copyright 2002 ACM 1-58113-471-1/02/0011...\$5.00.

Keywords

Security, call graph, invocation graph, data flow analysis, Java security, access rights.

1. Introduction

JavaTM 2 has a security architecture intended to protect client and server systems from dynamically installed (e.g., mobile code) or statically configured malicious code [13] [14] [15] [23]. Applet code is downloadable from the Internet into a Web browser [23] [19], and uploadable via RMI [22] into a server application. The Java 2 security system contains an authentication subsystem and an authorization subsystem. This paper focuses on the authorization subsystem, automating the determination of access rights needed to execute the code.

Prior to deploying application or library code in Java, a critical question arises: "What Java access rights are required to allow the code to execute?" In practice this problem is solved empirically. The developer reads documentation for libraries used (including the Java run-time libraries) and deduces the required access rights. Unfortunately, this documentation is often missing, misleading, or out of date. In the absence of reliable documentation, the developer executes the new code and observes authorization failures. The developer then grants additional access rights and retests. The developer repeats this process, possibly many times, until there are no authorization failures. However, required access rights requirements can remain undiscovered due to an insufficient number of test cases, rendering the code unstable.

An analogous situation arises in systems where mobile code is dynamically installed and the system administrator (e.g., the Web browser user) must determine the set of access rights to provide. The system administrator usually relies on the code developers'/distributors' recommendations, with the risk that too many access rights are granted and security holes are created. Alternatively, the system administrator runs the code with a smaller set of access rights, examines failures, and incrementally adds access rights as is necessary. This process is tedious and error-prone. As before, insufficient testing results in improper authorizations, creating security exposures or application instability.

This paper describes a technique based on context sensitive data and control flow analyses to automatically determine access rights required by Java programs or libraries. We use a modified interprocedural invocation graph, called an access rights invocation graph (ARIG), to compute the access rights.

In Java, access rights are modeled using the Permission class hierarchy. The root of the class hierarchy is the abstract class java.security.Permission. By default, all Permissions are "approval" of access rather than "denial." A Permission object is an instance of a subclass of java.security.Permission. For instance,

perm = new FilePermission("/tmp/abc", "read");

creates a Permission object to read the file /tmp/abc. In our analysis, we compute the set of Permission objects to associate with each program point by constructing an ARIG to propagate the access rights. An ARIG consists of nodes corresponding to AccessController's checkPermission and doPrivileged methods, which are the boundary nodes, as well as all nodes in the invocation graph in all paths between the boundary nodes and from the boundary nodes to the root nodes.

To summarize, the main contributions of this paper are:

- We present a context sensitive, flow sensitive analysis for computing the access rights requirements of a program.
- We model features such as multithreading, implicitly defined security policies, semantics of the Permission.implies method, and the generation of a security policy description.
- We use a modified invocation graph, an ARIG, to propagate access rights.
- We implemented the algorithms and present test results from the use of our tool.

•Our analysis technique is scalable enough to produce usable results on problems with an analysis scope of over 20,000 classes.

2. Prior Work

Both static and dynamic analysis techniques are employed in modeling security and authorization. Much of this work has been applied to eliminate or minimize redundant authorization tests or define alternatives to the current approach to defining authorization points within code.

Pottier, Skalka and Smith [25] extend and formalize Wallach's security passing style [31] via type theory using a λ -calculus, called λ_{sec} . Pottier, et al, were unable to model all of Java's authorization characteristics, including multithreaded code and "open world" analysis. Nor does it consider computing the authorization object, which often includes identifying the String parameters to the Permission objects' constructor. All of these strongly affect the completeness of an authorization analysis.

Jensen, Métayer and Thorn [17] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests. They assume all of the code is available for analysis, and that a call graph can be constructed for the code, though they do not do so themselves. The results are also limited by an assumption that security properties can be expressed solely in terms of the control flow and call graph of the program. For Java, essential authorization information is based on values (usually string constants) propagated to authorization tests.

Bartoletti, Degano and Ferrari [5] are interested in optimizing performance of run-time authorization testing, by eliminating redundant tests and relocating others as is needed. The reported results apply operational semantics to model the run-time stack. Similarly, Banerjee and Naumann [4] apply denotational semantics to show the equivalence of "eager" and "lazy" semantics for stack inspection, provide a static analysis of safety (the absence of security errors), and identify transformations that can remove unnecessary authorization tests. Significant limitations to this approach are that the analyses are limited to a single thread, and require whole program analysis. Also, the Permission.implies and Permissions.implies methods, including AllPermission, are not modeled. Modeling these classes and methods are important when simplifying the access rights policy descriptions so that the results are usable.

In the aforementioned works, assumptions are made that (1) call graph algorithms are available to translate the theoretical approach into a practical implementation, and (2) there is an authorization object, p, and a single authorization point, the checkPermission method. For Java 2 this is not correct. Almost all of the code in the Java runtime calls one of the SecurityManager authorization methods, though many of these methods subsequently call AccessController.checkPermission. Many of the well-known call graph and data flow algorithms [16] are too conservative to correctly identify authorization requirements. In this paper we describe an invocation graph and data flow analysis that minimizes the conservativeness to get more accurate authorization information.

Felten, Wallach, Dean and Balfanz have studied a number of security problems related to mobile code [32] [11] [31] [9] [30] [10] [8]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals (signers and/or origin of the code) currently active in a Thread stack at run-time, as is found in Java. In particular, an authorization optimization technique, called security passing style, encodes the security state of an application while the application is executing [31]. Each method is modified so that it passes a security token as part of each method invocation. The token represents an encoding of the active principals (security state) at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, security passing style explores subgraphs of the comparable invocation graph, and discovers the security states and authorizations associated with those states. Wallach assumes that all authorization tests are temporally invariant, so that once an authorization test succeeds or fails in a particular security state, it will always succeed or fail in that state. In Java 2 this invariance does not exist. A Permission implies invocation can revoke a class' access rights, or may use state unrelated to the runtime stack when determining the result of the test. In practice, especially in web server environments, access rights are revoked while the JVM is running. Our approach is not to optimize the authorization performance, but to discover authorization requirements by analyzing all possible paths

through the program, even those that may not be discovered by a limited number of test cases.

Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [12] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. We examine the authorization issue from the perspective of examining an existing system containing authorization test points. Through static analysis, we discover how the security policy needs to be modified / updated to enable the code to execute.

3. Invocation Graph Characteristics

We use a path-insensitive, flow-sensitive, context-sensitive invocation graph. A path-insensitive invocation graph analyzes all paths through all basic blocks in each method. Because Java 2 authorization is based on associating rights with classes, a pathinsensitive invocation graph construction algorithm is sufficient. The invocation graph is *flow-sensitive* for intraprocedural analysis because it considers the order of execution of the instructions within each basic block, accounting for local variable kills and casting of object references. The invocation graph (interprocedural analysis) is *context-sensitive* because it uniquely distinguishes each node by its *calling context*: the target method, receiver and parameters values.

For the purposes of describing the Permission culling algorithm, the invocation graph has the following characteristics:

- Each node in the graph represents a context sensitive method invocation.
- Each node in the graph is uniquely identified by its calling context, so no two nodes in the graph have the same calling context.
- Each node in the graph contains the following state:
 - The target method. 0
 - For instance methods, an allocation site (or type) for the 0 method's receiver.
 - All parameters to the method, represented as a vector of sets of possible allocation sites (or possible types).
 - 0 A set of possible return value allocation sites (or types) from this method.
- The edges in the graph are directed, where each edge points from a call site to a target method.
- The graph is rooted and may be cyclic.
- Our representation of the graph allows bi-directional traversal, even though the edges in the graph are unidirectional. Therefore, from any node within the graph, we can find all of its predecessor nodes.

In addition to the invocation graph construction, we use a data flow analysis with a precision to the level of allocation sites (CFA(1) [27]) and include the propagation of string constants. In a limited number of cases, data flow on Permission objects is computed using a CFA(2)-like algorithm to reduce the conservativeness of the analysis (see Section 4.). We are

particularly interested in the string constants since they are used as parameters to Permission object constructors. The string constant values passed to the constructor fully qualify the access rights requirement.

4. Authorization Model – Access Rights **Invocation Graph (ARIG)**

Each Java application class is loaded into the Java Virtual Machine and is associated with a set of rights, or privileges, granted to the code. Statically determining this set of rights is nontrivial because it involves identifying, as accurately as possible, the precise set of methods callable from any point in a program execution. If any method is omitted, the analysis is incomplete. Conversely, when the analysis is overly conservative, the large number of false positives violates the Principle of Least Privilege [26], rendering the analysis ineffective for practical use.

We model the Java 2 authorization algorithm using a graph and set theoretic approach as follows. Let $\mathbf{G} = (\mathbf{N}, \mathbf{E})$ be the invocation graph representing a program *P*. The nodes are described by $\mathbf{N} = \{n(M, R, \mathbf{P}) \mid M \text{ is the target method, with}$ receiver R and k parameters $\mathbf{P} = \langle P_i \rangle_{i=1,\dots,k}$, where parameter i can have possible types P_i . Each node represents the intraprocedural analysis of a method, including the virtual call sites within the method, and subsumes the control flow graph for that method. The edges are described by:

$$\mathbf{E} = \left\{ e(n_p(M_p, R_p, \mathbf{P}_p), n_s(M_s, R_s, \mathbf{P}_s)) | M_s \text{ is invoked within } M_p \right\}.$$

Given a node $n \in \mathbf{N}$, we define $\Gamma^+(n) := \{n' \mid \exists e(n, n') \in \mathbf{E}\}$ and $\Gamma^{-}(n) := \{n' \mid \exists e(n', n) \in \mathbf{E}\}, \text{ known as the outward and inward}$ adjacencies of n, respectively.

We can extend these definitions to sets of nodes. Thus, given $N \subseteq \mathbf{N}$, we define $\Gamma^+(N) := \bigcup_{n \in N} \Gamma^+(n)$ and $\Gamma^-(N) := \bigcup_{n \in N} \Gamma^-(n)$.

- We also define: $N_{\text{root}} := \{ n \in \mathbf{N} \mid \Gamma^{-}(n) = \Phi \}$, the set of root nodes in the invocation graph
- $N_{dp} := \{n(M, R, \mathbf{P}) | M \text{ is AccessController.doPrivileged} \}$
- $N_{cp} := \{n(M, R, \mathbf{P}) | M \text{ is AccessController.checkPermission} \}$ •
- $N_{\text{start}} := N_{\text{root}} \cup \Gamma^{-} (N_{\text{dp}})$
- $N_{\text{stop}} := N_{\text{cp}}$

For any node n, $\mathbf{RP}(n)$ indicates the set of required Permissions for *n*. Similarly, given a set of nodes $N \subseteq \mathbf{N}$, $\mathbf{RP}(N)$ denotes the required Permissions for the nodes in N. This implies that $\mathbf{RP}(N) = \bigcup \mathbf{RP}(n)$. For each node $n \in \mathbf{N}$, the algorithm determines $\mathbf{RP}(n)$ by starting from $\mathbf{RP}(N_{\text{stop}})$, and tracing paths

back from nodes in N_{stop} to nodes in N_{start} .

For each node *n* in N_{stop} , $\mathbf{RP}(n)$ is defined to be the set containing the single element *p*, where *p* is the Permission being checked at *n*. Thus, if we define $\mathbf{CP}(n)$ to be the Permission

checked at *n* itself, we have
$$\mathbf{CP}(n) = \begin{cases} \{p\}, \forall n \in N_{cp} \\ \boldsymbol{\Phi}, \forall n \notin N_{cp} \end{cases}$$

For any node $n \in \mathbf{N}$, if there is a path $\pi(n,s)$ from *n* to another node *s*, *n* requires all the Permissions that *s* does. Therefore, it must be $\mathbf{RP}(n) \supseteq \mathbf{RP}(s)$. We thus compute $\mathbf{RP}(n)$ recursively

from
$$\mathbf{RP}(n) = \mathbf{CP}(n) \cup \bigcup_{s \in \mathbf{N} \mid \exists \pi(n,s)} \mathbf{RP}(s).$$

Since, in general, **G** contains cycles, the algorithm is a fixed-point computation, starting with estimates for $\mathbf{RP}(s)$ for every s in N_{stop}

and working backwards, using the Γ^- function, along paths towards nodes in N_{start} . This process associates a set of required Permissions $\mathbf{RP}(n)$ with each node *n* in N_{start} . More precisely, it computes $\mathbf{RP}(n)$ for all $n \in \mathbf{N}$.

Finally, $\mathbf{RP}(C)$, the set of Permissions required for a class *C*, can then be computed as $\mathbf{RP}(C) = \bigcup_{n \in N(C)} \mathbf{RP}(n)$, where N(C) is the

sets of nodes *n* whose methods are declared in class *C*. Indeed, in this manner, we can compute $\mathbf{RP}(N)$ for any set of nodes $N \subseteq \mathbf{N}$.

Note that Permissions propagated upwards via a doPriviledged node do not propagate beyond the predecessor of the doPrivileged node. Thus, the above definition (or the algorithm based on it) must be refined to replace $\Gamma^{-}(n)$ by $\Gamma^{-}(n,p)$, where $[\phi, \text{ when } p \text{ was propagated upwards to } n$

 $\Gamma^{-}(n, p) = \begin{cases} \psi, \text{ when } p \text{ was propagated upwards to } n \\ \text{via a doPrivileged node} \\ \Gamma^{-}(n), \text{ otherwise} \end{cases}$

Lastly, many security authorization tests in Java 2 are made through calls to methods in the SecurityManager class. In the reference implementation of this class, most of the SecurityManager authorization tests are performed by calling AccessController.checkPermission with an appropriate Permission object. Details about the classes, objects and algorithms employed by Java 2 authorization are treated in-depth in Gong's and Pistoia's Java 2 security books [15] [23].

It can be seen that the data flow analysis described above does indeed converge to a fixed point by observing that the transfer function relating the value of $\mathbf{RP}(n)$ at the output of any invocation graph node, n, to its value at the input to that node is in fact the identity function and the value of $\mathbf{RP}(n)$ at the input to a

node *n* is formed from the values at the outputs of nodes in $\Gamma^-(n)$ by means of a set union operation. Thus, $\mathbf{RP}(n)$ is monotone, specifically it is a non-decreasing function as our computation proceeds. The values of the $\mathbf{RP}(n)$ at each invocation site form a

lattice [18] and, since the set of types within our analysis scope is finite, we are guaranteed that the computation converges to a unique fixed point in finite time, regardless of the order in which we visit the nodes in the invocation graph.

It should be noted also that uniquely identifying a node in the ARIG by its calling context does not introduce any additional conservativeness to the analysis. In fact, any two invocations of the same method with the same calling context would generate the same invocation subgraph. Therefore, they would require the same set of Permissions.

To clarify all of the concepts introduced in this section, consider the following simple example. A class C implements a method, methodC, which takes as argument the name of a file, and returns a FileInputStream for that file, as shown next:

```
FileInputStream methodC(String fileName) {
    return new FileInputStream(fileName);
}
```

Creating a FileInputStream involves a security check.

methodC is called from within methodA in class A and methodB in class B, the only difference being that while methodA calls methodC with a specific parameter, "file1.txt", methodB calls methodC with two possible parameters depending on the value of a boolean expression, as shown in the two following snippets of code:

```
FileInputStream methodA() {
   String fileName = "file1.txt";
   C c = new C();
   return c.methodC(fileName);
}
FileInputStream methodB() {
   String fileName;
   if (Math.random() > 0.5)
      fileName = "file.txt";
   else
      fileName = "file2.txt";
   C c = new C();
   return c.methodC(fileName);
}
```

Evaluating the boolean expression in methodB cannot be done during the static analysis of the code. Therefore, the conservative and most secure approach requires that the execution of both the branches be considered. Therefore, the call to methodC must be represented taking into account both the parameters "file1.txt" and "file2.txt". The following figure shows a simplified version of the invocation graph representing the program:



Figure 1. Sample Program ARIG Graphical Representation

As the ARIG in Figure 1, shows, classes B and C require both Permissions p_1 and p_2 , while class A require only p_1 .

5. The Permission Culling Algorithm

We describe an algorithm to statically identify the set of rights required by each analyzed class. The algorithm identifies paths in an invocation graph [2] [3] [6] [7] [28] [29] leading to AccessController.checkPermission nodes. By using a data flow analysis [21] the algorithm determines the set of possible Permission objects that could be passed as the argument to this method.

5.1 The Basic Permission Culling Algorithm

The basic algorithm uses the ARIG previously described to identify the set of Permissions representing the access rights required for each analyzed call site. The algorithm then aggregates these Permissions by the calling methods and their declaring classes. The algorithm identifies all nodes in each path bounded by any N_{start} node and an N_{stop} node and associates a set of Permissions with each of the nodes in the path. In a running system, each checkPermission method call has a single Permission object passed as an argument. In our analysis, which is path insensitive, this argument is a set of possible Permissions.

Each method in the path, and the method's declaring class, is marked as requiring the set of Permissions. In addition, the String parameters from Permission constructors are obtained through the data flow analysis. The parameter values provide necessary qualification of the authorization requirement. A typical authorization described call is by constructor FilePermission("/tmp/file1", "write"). Pseudo code for this algorithm is in Appendix 1. The following figure graphically represents the Basic Permission Culling algorithm:



Figure 2. The Basic Permission Culling Algorithm

5.2 Reducing the Conservativeness of the Access Rights Analysis

The basic algorithm as described leads to an overly conservative result as is shown in the following figure. The subgraph represents part of the standard Java 2 SecurityManager control and data flow:



Figure 3. Conservativeness Introduced by the SecurityManager

The problem is that the FilePermission allocation site corresponds to two different FilePermission constructor calls as a result of the two different paths leading to the checkRead method. The Basic Permission Culling Algorithm propagates the Permission set $\{p,q\}$ even when only p or q, but not both, is required. This violates the Principle of Least Privilege. We selectively reduce the conservativeness by using the node of the Permission's allocation site (a CFA(2)-like approach [27]) to differentiate the Permission allocations in the SecurityManager. In practice, this approach appears to be sufficient. The following figure shows the result:





Figure 5. Modeling Threads

Figure 4. Conservativeness Reduction through Selective CFA(2)

We define $\Gamma^{-}(n, P)$, the inward adjacency of a node *n* with respect to a set *P* of Permissions, as the set of predecessor nodes of *n* with respect to the allocation sites of the Permissions in *P*. The algorithm proceeds as before, with $\Gamma^{-}(n, P)$ in place of $\Gamma^{-}(n)$. In practice, this refinement is sufficient for those Permissions allocated in one of the SecurityManager check methods that exhibit the behavior similar to the checkRead method described above.

5.3 Threads

The construction of a new Thread object does not cause it to start execution. Nominally, a call to the Thread.start method results in the new Thread beginning execution at the Thread.run method. The new Thread can be started by a different Thread other than that which created it, or by a method in a class other than that which created the Thread. The run method becomes the root (starting) method for the Thread.

According to the Java 2 authorization model, a new Thread requires that all predecessor nodes of the newly created Thread's constructor node also require Permission set *P*. Also, $\Gamma^{-}(n)$, where *n* is a Thread.run node, does not require *P*. We extend the Γ^{-} function as follows:

$$\widetilde{\Gamma}^{-}(n, P) = \begin{cases} n_{\text{Th}}, \text{ if } n \text{ is a Thread.run node} \\ \Gamma^{-}(n, P), \text{ otherwise} \end{cases}$$

where n_{Th} is the Thread constructor node for *n*'s receiver.

The pseudo code is in Appendix 2. Graphically, we are rewriting the ARIG to contain a predecessor edge from a Thread.run call site to the Thread constructor as is shown in the following figure:

5.4 doPrivileged with an AccessControlContext

In addition to the AccessController.doPrivileged method described above, another form of the doPrivileged method takes an AccessControlContext instance as an argument. In addition to the previously described behavior, authorization tests include all of the predecessor nodes of the node where the AccessControlContext object was allocated. This is modeled similarly to how we model new Threads, by augmenting $\tilde{\Gamma}^-$ to include an edge from the doPrivileged node to the node where the AccessControlContext was allocated. Specifically, we extend $\tilde{\Gamma}^-$ as follows:

$$\hat{\Gamma}^{-}(n, P) = \begin{cases} N_{ACC}(n), \text{ if } n \text{ is a doPrivileged node with an} \\ AccessControlContext} \\ \widetilde{\Gamma}^{-}(n, P), \text{ otherwise} \end{cases}$$

where $N_{ACC}(n)$ is the set of nodes where AccessControlContext.<init> is invoked to create any of *n*'s AccessControlContext parameters.

6. Computational Experience

The practical usefulness of an ARIG depends on the conservativeness of the underlying invocation graph and data flow analysis. In this section we discuss the trade-offs of using relatively less conservative invocation graphs, and we explain through practical results why we selected a context sensitive, selectively CFA(2) invocation graph. We discuss also the limitations that naturally arise in the identification of string constants representing the parameters to Permission constructors. Finally, we show some experimental results.

6.1 Analysis Conservativeness

Previous work using static analysis for Java authorization analysis has not discussed the implications of conservativeness of invocation graph and data flow analysis techniques. The conservativeness of the invocation graph and data flow analysis greatly affects the Permission Culling Algorithm results. An overly conservative control and data flow analysis is likely to determine access rights requirements for classes that do not need them, thus violating the Principle of Least Privilege. Class Hierarchy Analysis (CHA)-style graph construction algorithms [7] result in java.security.Permission and all of its subclasses being required for all classes needing authorization. A similar result holds for Rapid Type Analysis (RTA)-style algorithms [2], except that the access rights requirements for any class needing any Permission will include all of those Permissions that have allocation sites within the call graph. A CFA(0) call graph [27] would lead to overly conservative results because it does not consider the calling contexts. Without the calling contexts, it not possible to differentiate the AccessController.checkPermission calls. Similarly for the calls to the SecurityManager, where CFA(2) is used to selectively differentiate the required Permissions, as shown in Section 5.2. In addition, doPrivileged nodes would be collapsed, causing all nodes prior to doPrivileged to require the same set of Permissions, even though the required Permissions should usually be different. Again, this is overly conservative, resulting in a violation of the Principle of Least Privilege.

Experience has shown that context-sensitive invocation graphs yield less conservative results. Propagation-based call graph construction algorithms have been studied extensively, and differ primarily in the number of sets used to approximate run-time values of expressions [16]. The *Cartesian Product Algorithm* (CPA) [1] uses an approach based on parametric polymorphism. Given a method invocation to analyze, CPA computes the Cartesian product of the types of the actual arguments to the method. The invocation graph we use is similar to Agesen's, except we use what he refers to as *megamorphic sets* to represent parameters. Also, we consider data polymorphism, while Agesen does not. The invocation graph construction algorithm we use is similar the one described by Plevyak and Chien [24].

To minimize conservativeness, we use a graph construction algorithm that is context sensitive, flow sensitive, and path insensitive. By *flow sensitive* we mean that the analysis considers the order of execution of instructions both intra- and interprocedurally thus improving the accuracy of the resulting graph. Part of our graph construction is flow sensitive for local variables, including support for local *field kills* – local fields that are overridden by subsequent assignments to the same field – and type casting. However, our handling of instance and class (static) fields is flow-insensitive because we use the weak assumption that all instance and class fields are subject to modification at any time due to multi-threading. To compensate for class and instance field flow-insensitivity, the data flow analysis tracks field values by allocation site. The practical problem that arises is that an allocation site does not directly map to a node in the invocation graph, thus making the analysis somewhat more conservative than we would otherwise like. Specifically, there is a one-to-many mapping of allocation sites to nodes in the graph. However, we have observed, by closely examining output from our tool and corresponding source code, that using allocation sites is sufficient to compensate for imprecision resulting from being interprocedurally flow insensitive.

Path insensitive intraprocedural analysis considers all paths through a method. This is conservative because input values or the values of constants defined within the program are not considered. For example, in the statement:

if (false) expl else exp2

both exp1 and exp2 are evaluated even though, in practice, exp1 never gets executed. While conservatively correct, this may result in additional graph nodes being generated for paths not occurring in practice. The net effect is that all required Permissions are included, though some additional Permissions may be included that are not strictly necessary in all executions of a program. A future version of the tool could consider adding a level of path sensitivity to minimize this conservative characteristic.

The graph construction algorithm that we use is selectively CFA(2). In particular, we use the node of the Permission's allocation site to differentiate Permission allocations in the SecurityManager. The CFA(1) portion of our algorithm defines the calling context to be the allocation sites of the possible receivers and parameters. Unfortunately, this approach alone does not distinguish between two different Permissions allocated at the same bytecode offset in the SecurityManager (see Section 5.2). As a result, it would be impossible to distinguish between two different Permissions of the same type, where the Permission object is allocated at the same location in the SecurityManager. As we explained in Section 5.2, the selective use of a CFA(2)-like approach eliminates this additional level of conservativeness without significantly impacting CPU and memory usage.

6.2 Limitations on String Constant Identification

As a practical matter, parameters to Permission constructors are string constants. In some cases, the parameter value may be available only at run-time. For example, it may be required that the name of a file to be opened be specified as input. To improve the analysis, it is possible to provide some metadata to reflect the run-time values. In other cases, the parameter value could be the result of a computation (e.g., string concatenations). A slightly more sophisticated data flow analysis is required in such circumstances.

Application	Classes		Methods	Instruction	Analysis time (sec.)			Nodes	Edges	Heap size
	Scope	Analyzed	Analyzed	bytes	Call graph	ARIG	Total			(MB)
javadoc	6,720	635	2,874	190,394	162	8	170	21,042	60,935	128
GetProperty	5,446	378	1,545	91,471	32	6	38	9,326	15,743	65
CountMain	5,451	383	1,552	91,809	32	6	38	9,339	15,763	65
java.lang.*	5,445	472	2,319	126,105	51	6	57	13,578	23,250	93
ECPerf Corp	21,291	1,912	2,854	426,685	94	7	101	18,330	36,479	120

Table 1. Comparison of Analyses

6.3 Experimental Results

Context and flow sensitive static analysis has a reputation for requiring significant processing power and memory. We have performed authorization analysis on a number of sample programs (see Appendix 3), parts of rt.jar, selected middleware, the Java API documentation generator (javadoc), and part of the ECPerf benchmark program. The results reported in Table 1. are from running our analysis on a system with an AMD Athalon 933 MHz processor, Windows 2000 SP2 with 768 MB RAM and using the Java Development Kit (JDK) V1.3.1. JDK V1.3 functionality was made part of the analysis scope by including the JDK V1.3 rt.jar.

Performance is improved by ignoring methods that do not lead to calls to AccessController.checkPermission, but whose data flow analysis requires a substantial amount of time. By forcing the underlying invocation graph to ignore the class constructor for sun.io.CharacterEncoding as well as all methods in classes Object, String, StringBuffer, and NullPointerException in package java.lang, and classes TimeZone and SimpleTimeZone in package java.util, we improved execution time significantly without affecting the resulting Permission sets identified. In particular, this optimization is a consequence of the fact that the analysis could be performed incrementally. Once it has been established that the invocation of a particular library will never lead to a security check, the data flow analysis for that library can be avoided.

The simplest example that we analyzed is an application called GetProperty, whose main method was the only root method and contained the following two lines of code:

System.setSecurityManager(new SecurityManager());
System.out.println(System.getProperty("user.home"));

The authorization requirements produced were precisely those that we expected based on examination of the source code:

```
java.lang.RuntimePermission "createSecurityManager"
java.lang.RuntimePermission "setSecurityManager"
java.util.PropertyPermission "user.home", "read"
```

The next example, CountMain, is more interesting because it makes use of privileged code. The method main was the only root method. The analysis computed access rights requirements that exactly reflected the presence of privileged code in the application. The source code for CountMain, as well as the computed access rights requirements, is reported in Appendix 3.

We also ran an analysis on the packages java.lang, java.lang.ref, and java.lang.reflect in rt.jar (part of the run-time classes for Java); all the public and protected methods of classes in these packages were considered as root nodes, including non-abstract public and protected methods in abstract classes, because they represent all the possible entry points that a program running on top of a library could invoke. The entire rt.jar V1.3 was included in the analysis scope. 2,811 root nodes were generated from 1,018 root methods, of which 336 were static methods, and the remaining 682 were instance methods. The number 2,811 comes from the fact that each static method gets counted once, because it has no receiver, and each instance method is weighted by the number of receivers. This results in an average of 3.629 receivers per root instance method.

The tool has successfully been run on large Java programs, the largest of which contain over 20,000 classes. Table 1. shows the statistics of running the analysis on the Corp part of the J2EE benchmark called ECPerf. To analyze this benchmark, the Java runtime plus all of the Enterprise JavaBeans (EJB) runtime classes are needed, resulting in an analysis scope of over 21,000 classes.

Other analyses were performed on large products (over 20,000 classes in the analysis scope) based on the Java security model of the JDK V1.1 platform. The goal was to identify its access rights requirements to allow it to successfully run with the Java 2 security model enabled.

7. Generation of a Security Policy Description

In Java 2, every concrete Permission class is required to implement the implies method. Given two Permission objects, p and q, when p.implies(q) returns true it means that any code that is granted p is also automatically granted q. Since we identify the String parameters to the Permission objects'

constructors, we instantiate the Permission objects during the analysis, and use their implies methods to filter out those Permissions that are already implied by other Permissions. This minimizes the list of required access rights. When the parameters to a Permission constructor are not determinable, we impose that the resulting Permission cannot imply any other Permissions, even though stronger Permissions, such as java.security.AllPermission, still imply it. This also allows us to generate a security policy file containing the access right requirements needed at run-time.

The access rights requirements are minimal modulo the conservativeness of the analysis and the possible inability to determine some string constants. The resulting policy file is useful for defining new security policy, update an existing policy, or validate whether a path through the program will result in an authorization failure.

8. Conclusions

For a given application or classes in a library, we are able to conservatively identify the set of Java 2 Permissions required for each class in the analysis scope. By using a context-sensitive invocation graph, we are able to accurately identify the classes in each path that contains a call to the Java 2 security authorization subsystem. Our level of precision is far greater than that required for Java 2 security because we are able to identify access rights requirements to the level of methods and call sites, rather then the coarser granularity of classes or libraries. A refinement of the Java 2 authorization algorithm could result in the minimization of authorization, bringing us closer to the application of the Principle of Least Privilege.

By using the analysis technique described in this paper, we can determine the access rights requirements of mobile code, such as applets, servlets, and code that exploits mobile code features of RMI. Prior to loading the mobile code, it is possible to prompt the administrator or end-user to authorize or deny the code access rights for restricted resources protected by the Java 2 authorization subsystem.

Automating the process of determining required access rights changes the relationship between the developer of the code and the administrator / end-user. Instead of relying solely on recommendations from the developer, or resorting to trial-anderror testing of the code to determine required access rights, our tool can analyze the code and make its own recommendations and/or validate recommendations made by the developer. This shifts the relationship from one that *requires* that the developer be trusted, to something that can be verified.

While the analysis described here is specific to Java, the basic techniques can be applied to resource access rights determination in other type safe languages. With stronger analysis techniques, it may even be possible to apply the same approach to languages that lack type explicit safety but could rely on other mechanisms such as typed assembly language [20].

9. Acknowledgment

The authors would like to thank Dr. Sreedhar Vugranam for his invaluable technical contributions to this paper.

10. References

- O. Agesen. The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism. In Proceedings of ECOOP '95, Aarhus, Denmark, August 1995. Springer-Verlag, 1995.
- [2] D.F. Bacon and P.F. Sweeney. Fast static analysis of C++ virtual function calls. In Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, Systems and Applications (OOPSLA'96), San Jose, CA. 1996, 324-341, ACM Press, New York. Also in ACM SIGPLAN Notices 31(10).
- [3] D.F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.
- [4] A. Banerjee and D. A. Naumann. A Simple Semantics and Static Analysis for Java Security. Stevens Institute of Technology, CS Report 2001-1, July 2001.
- [5] M. Bartoletti, P. Degano, and G. Ferrari. *Static Analysis for Stack Inspection*. Proceedings of ConCoord, Lipari, Italy, 6-8 July 2001, ENTCS 54, Elsevier Science B. V., 2001.
- [6] C. Chambers, D. Grove, G. DeFouw and J. Dean. Call graph construction in object-oriented languages. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), 108-124, Oct. 5-9, 1997, ACM Press, New York. Also in ACM SIGPLAN Notices 32(10).
- [7] J. Dean, D. Grove and C. Chambers. Optimization of objectoriented programs using static class hierarchy analysis. In Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95). Aarhus, Denmark, Aug. 1995. W. Olthoff, Ed., Springer-Verlag, 77-101.
- [8] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. Proceedings of the 1996 IEEE Syposium on Security and Privacy (Oakland, California), IEEE, May 1996.
- [9] D. Dean. The Security of Static Typing with Dynamic Linking. Proceedings of the Fourth ACM Conference on Computer and Communications Security. (Zürich, Switzerland), April 1997.
- [10] D. Dean, E. W. Felten, D.S. Wallach, and D. Balfanz. Java Security: Web Browsers and Beyond. Internet Beseiged: Counter Cyberspace Scofflaws, D.E. Denning and P.J. Denning, eds. ACM Press (NY, NY), October 1997.
- [11] R.D. Dean. Formal Aspects of Mobile Code Security. PhD thesis, Princeton University, Princeton, New Jersey, January 1999.

- [12] Ú. Erlingsson and F.B. Schneider. IRM Enforcement of Java Stack Inspection. Proceedings IEEE Symposium on Security and Privacy, pp. 246-255, Oakland, California, May 2000.
- [13] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. Proceedings of the USENIX Symposium on Internet Technologies and Systems, 103-112, Monterey, CA., December 1997.
- [14] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. Proceedings of the Internet Society Symposium on Network and Distributed System Security, 125-134, San Diego, CA., March 1998.
- [15] L. Gong. <u>Inside Java™ 2 Platform Security: Architecture,</u> <u>API Design, and Implementation</u>. Addison-Wesley, Reading, MA. 1999.
- [16] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. ACM TOPLAS, Vol. 23, No. 6, November 2001.
- [17] T. Jensen D. Le Métayer and T. Thorn. Verification of control flow based security properties. IRISA, Publication interne n° 1210, October 1998.
- [18] G. A. Kildall. A Unified Approach to Global Program Optimization. Proceedings of Principles of Programming Languages, pp. 194-206, 1973.
- [19] G. McGraw and E.W. Felten. <u>Securing Java™</u>. John Wiley & Sons, Inc., New York. 1999.
- [20] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to Typed Assembly Language. In ACM Transactions on Programming Languages and Systems, 21(3):528-569, May 1999.
- [21] S.S. Muchnick. <u>Advanced Compiler Design And</u> <u>Implementation</u>. Morgan Kaufmann Publishers, San Diego, CA, 1997.
- [22] R. Oberg. <u>Mastering RMI: Developing Enterprise</u> <u>Applications in Java and EJB</u>. John Wiley & Sons, Inc., New York. 2001.
- [23] M. Pistoia., D.F. Reller, D. Gupta., M. Nagnur., A.K. Ramani. <u>Java™ 2 Network Security</u>, Second Edition. Prentice Hall PTR, New Jersey, 1999.

- [24] J. Plevyak and A.A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. ACM OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications, pp. 324-340, Portland, Oregon, October 1994.
- [25] F. Pottier, C. Skalka and S. Smith. A Systematic Approach to Static Access Control. D. Sands (Ed.): ESOP 2001, LNCS 2028, pp.30-45, 2001. Springer-Verlag, Berlin Heidelberg 2001.
- [26] Saltzer J.H. and M.D.Schroeder. The Protection of Information in Computer Systems. Proceedings of the IEEE 63 9 (Sept.1975), 1278-1308.
- [27] O. Shivers. Control-flow Analysis in Scheme. ACM SIGPLAN Notices, 23(7):164-174, July 1988. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation.
- [28] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon and C. Godin. *Practical Virtual Method Call Resolution for Java*. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), 264-280, Oct. 15-19, 2000, ACM Press, New York. Also in ACM SIGPLAN Notices 35(10).
- [29] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), 264-280, Oct. 15-19, 2000, ACM Press, New York. Also in ACM SIGPLAN Notices 35(10).
- [30] D.S. Wallach, D. Balfanz, D. Dean, E.W. Felten. *Extensible Security Architectures for Java*. 16th Symposium on Operating Systems Principles (Saint-Malo, France), October 1997.
- [31] D.S. Wallach and E.W. Felten. Understanding Java Stack Inspection. Proceedings of the 1998 IEEE Symposium on Security and Privacy (Oakland, California), May 1998.
- [32] D.S. Wallach. A New Approach to Mobile Code Security. PhD thesis, Princeton University, Princeton, New Jersey, January 1999.

Appendix 1

The following pseudo-code embodies the Basic Permission Culling Algorithm.

```
// Identify all start and stop nodes in the graph.
// The start set includes all nodes in the call graph root set.
Set startSet = new Set(rootNodes);
Set stopSet = new Set();// Initially, the stop set is empty.
// Identify additional start nodes and the stop nodes
// by iterating over all nodes in the call graph.
Iterator nodesIter = graphNodes.iterator();
while (nodesIter.hasNext())
   Node node = nodesIter.next();
   if (isDoPrivileged(node)
      startSet.add(node);
      doPrivSet.add(node);
   elseif (isCheckPermission(node))
      stopSet.add(node);
// Find all nodes between stop to start nodes and get the required Permission.
// For each node, get its method and associated class.
// Associate the Permission with each class identified.
// For each checkPermission(perm), identify Permission
// perm and the classes needing perm.
Iterator stopIter = stopSet.iterator();
while (stopIter.hasNext())
   Node stopNode = stopIter.next();
   // Get the Permission from the checkPermission() node
   RequiredPermission perm = getPermission(stopNode);
   // Using a work list algorithm, find all nodes in all paths that
   // are bounded by the nodes in the start set and the stop node.
   // Note: The graph may be cyclic.
   Set pathNodes = getPathsNodes(stopNode, start);
   // For each such node, get the node's method and the class that
   // declared that method. Add the Permission as being required for the class.
   Iterator nodesIter = pathNodes.iterator();
   while (nodesIter.hasNext())
     Node node = nodesIter.next();
      nodePerm.add(node, perm);
      Method method = node.getMethod();
      Class declaringClass = method.getDeclaringClass();
      // Remember that this class needs this Permission
      requiredPerms.add(declaringClass, perm);
// Propagate the Permissions at the doPrivileged node to all of
// its predecessors as is required by Java 2
Iterator doPrivIter = doPrivSet.iterator();
while (doPrivIter.hasNext))
  Node node = doPrivIter.next();
   Set reqPerms = nodePerm.get(node);
   PropagatePermsToPredecessorNodesClass(node, reqPerms);
```

At the end of this algorithm, each class in the requiredPerms map is mapped to the set of the Permission objects that it requires. From the allocation sites, we identify the string constants used in the Permission constructors. These string constants are used to report the required access rights for each class.

Appendix 2

We build a lookup table that maps Thread allocation sites to the graph nodes where the respective inherited AccessControlContext constructor is called. This mapping allows us create the replacement predecessor edge for the Thread.run method.

The following pseudo-code embodies the basic algorithm.

Now, when we reach a Thread.run node in the invocation graph, we can find its new predecessors by looking up the Thread allocation site and use it as the replacement predecessor edge. From the algorithm above, the getNodes method is suitably modified to use allocCallSites to find replacement predecessor nodes when searching the call graph.

<u>Appendix 3</u>

To illustrate computational experience with Permission analysis, we have made use of an application called CountMain. It is an interesting test case because it contains privileged code. From its main method, CountMain creates and sets a new SecurityManager, and then instantiates a CountFileCaller1 object and a CountFileCaller2 object, as shown in the following code:

```
import java.io.FilePermission;
public class CountMain {
    public static void main(String[] args) {
        System.setSecurityManager(new SecurityManager());
        CountFileCaller1.main(args);
        CountFileCaller2.main(args);
    }
}
```

The purpose of both CountFileCaller1 and CountFileCaller2 is to read the file C:\AUTOEXEC.BAT from the local file system. The code for CountFileCaller1 is shown next:

```
public class CountFileCaller1 {
    public static void main(String[] args) {
        try {
            System.out.println("Instantiating CountFile1...");
            CountFile1 cf = new CountFile1();
        }
        catch(Exception e) {
            System.out.println("" + e.toString());
            e.printStackTrace();
        }
    }
}
```

The following is the code for CountFileCaller2:

```
public class CountFileCaller2 {
   public static void main(String[] args) {
      try {
        System.out.println("Instantiating CountFile2...");
        CountFile2 cf = new CountFile2();
        cf.countChars();
      }
      catch(Exception e) {
   }
}
```

```
System.out.println("" + e.toString());
e.printStackTrace();
}
}
}
```

To perform the file read operation, CountFileCaller1 uses a supporting class called CountFile1, whereas CountFileCaller2 makes use of CountFile2. The difference between these two supporting classes is that CountFile1 wraps the code that performs the file read operation into a privileged block, whereas CountFile2 does not. This is evident by looking at the source code for CountFile1:

```
import java.io.*;
import java.security.*;
class PrivExcAction implements PrivilegedExceptionAction {
   public Object run() throws FileNotFoundException {
      FileInputStream fis = new FileInputStream("C:\\AUTOEXEC.BAT");
      try {
         int count = 0;
         while (fis.read() != -1)
            count++;
         System.out.println("Hi! We counted " + count + " chars.");
      }
      catch (Exception e) {
         System.out.println("Exception " + e);
     return null;
   }
}
public class CountFile1 {
    public CountFile1() throws FileNotFoundException {
       try {
          AccessController.doPrivileged(
                        new PrivExcAction());
       catch (PrivilegedActionException e) {
          throw (FileNotFoundException) e.getException();
       }
    }
 }
```

CountFile2 attempts to gain file read access without using Privileged code, as shown next:

```
import java.io.*;
public class CountFile2 {
   int count=0;
   public void countChars() throws Exception {
      FileInputStream fis =
                              new FileInputStream("C:\\AUTOEXEC.BAT");
      try
         while (fis.read() != -1)
           count++;
         System.out.println("We counted " + count + " chars.");
      catch (Exception e) {
         System.out.println("No characters counted");
         System.out.println("Exception caught" + e.toString());
      }
   }
}
```

The following figure shows a graphical representation of the CountMain program structure:



The CountMain program will run as long as CountFileCaller2, CountFile1, CountFile2, and CountMain itself are all granted the Permission to read the file C:\AUTOEXEC.BAT. This requirement is waived for CountFileCaller1, which is temporarily given the Permission because CountFile1 invokes doPrivileged. In addition to that Permission, CountMain also needs the Permissions to create and set a new SecurityManager. The analysis reflected these Permission requirements exactly, as shown in the following table:

Classes	Permissions Determined by the Permission Culling Algorithm				
CountMain	java.io.FilePermission "C:\AUTOEXEC.BAT", "read"				
	java.lang.RuntimePermission "createSecurityManager"				
	java.lang.RuntimePermission "setSecurityManager"				
CountFileCaller1					
CountFile1	java.io.FilePermission "C:\AUTOEXEC.BAT", "read"				
PrivExcAction					
CountFileCaller2					
CountFile2					

Static Analyses for Eliminating Unnecessary Synchronization from Java Programs

Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers

Department of Computer Science and Engineering University of Washington Box 352350 Seattle, WA 98195 USA {jonal, chambers, egs, eggers}@cs.washington.edu

Abstract. This paper presents and evaluates a set of analyses designed to reduce synchronization overhead in Java programs. Monitor-based synchronization in Java often causes significant overhead, accounting for 5-10% of total execution time in our benchmark applications. To reduce this overhead, programmers often try to eliminate unnecessary lock operations by hand. Such manual optimizations are tedious, error-prone, and often result in poorly structured and less reusable programs. Our approach replaces manual optimizations with static analyses that automatically find and remove unnecessary synchronization from Java programs. These analyses optimize cases where a monitor is entered multiple times by a single thread, where one monitor is nested within another, and where a monitor is accessible by only one thread. A partial implementation of our analyses eliminates up to 70% of synchronization overhead and improves running time by up to 5% for several already hand-optimized benchmarks. Thus, our automated analyses have the potential to significantly improve the performance of Java applications while enabling programmers to design simpler and more reusable multithreaded code.

1. Introduction

Monitors [LR80] are appealing constructs for synchronization because they promote reusable code and present a simple model to the programmer. Many modern programming languages, such as Java [GJS96] and Modula-3, directly support monitors. While these constructs enable programmers to easily write multithreaded programs and reusable components, they can incur significant run time overhead. Reusable code modules may contain synchronization for the most general case of concurrent access, even though particular programs often use these modules in a context that is already protected from concurrency. For instance, a synchronized data structure may be accessed by only one thread at run time, or access to a synchronized data structure may be protected by another monitor in the program. In both cases, unnecessary synchronization increases execution overhead. As described in section 2, even singlethreaded Java programs typically spend 5-10% of their execution time on unnecessary synchronizations.

Synchronization overhead can be reduced by manually restructuring programs [SNR+97], but this typically involves trading off program performance against simplicity, maintainability, and reusability. To improve performance, synchronization annotations can be omitted where

they are not needed for correctness in the current version of the program, or synchronized methods can be modified to provide specialized, fast entry points for threads that already hold a monitor lock. Such specialized functions make the program more complex, and using them safely may require careful reasoning about object-oriented dispatch to ensure that the protecting lock is acquired on all paths to the function call. The assumption that a lock is held at a particular program point may be unintentionally violated by a change in some other part of the program, making program evolution and maintenance error-prone. Hand optimizations make code less reusable, because they make assumptions about synchronization that may not be valid when a component is reused in another setting. In general, complex manual optimizations make programs harder to understand, make program evolution more difficult, reduce the reusability of components, and create an opportunity for subtle concurrency bugs to arise.

In this paper, we present and evaluate static analyses that reduce synchronization overhead by automatically detecting and removing unnecessary synchronization. A synchronization operation is unnecessary if there can be no contention between threads for the synchronization operation. For example, if a monitor is only accessible by a single thread throughout the lifetime of the program, there can be no contention for the monitor, and thus all operations on that monitor can safely be eliminated. Similarly, if threads always acquire one monitor and hold it while acquiring another monitor, there can be no contention for the second monitor, and this unnecessary synchronization can safely be removed. Finally, when a monitor is acquired by the same thread multiple times in a nested fashion, the first monitor acquisition protects the others from contention and therefore all nested synchronization operations may be optimized away. In order to reason statically about synchronization, we assume the compiler has knowledge of the whole program at analysis time; future work may extend our techniques to handle Java's dynamic code loading and reflection features.

There are three main contributions of this paper. First, we describe several synchronization optimization opportunities and measure their frequency of occurrence in several Java programs. Second, we provide precise definitions for a family of analyses designed to detect unnecessary synchronization. Finally, we present a preliminary empirical evaluation of these analyses on a suite of benchmarks. Our partial implementation eliminates up to 70% of synchronization overhead and improves running time by up to 5% for typical Java benchmarks on a highly optimized platform.

The rest of the paper is structured as follows. The next section describes the Java synchronization model, and provides measurements of synchronization overhead for typical benchmarks. Section 3 identifies opportunities for optimizations. Section 4 provides a precise description for a set of analyses that detect and eliminate unnecessary synchronization operations. Section 5 summarizes the performance impact of these analyses on a set of benchmarks, section 6 discusses related work, and section 7 concludes.

2. Java Synchronization

Java provides a monitor construct to protect access to shared data structures in a multithreaded environment.

2.1 Semantics

The semantics of monitors in Java are derived from Mesa [GMS77]. Each object is implicitly associated with a monitor, and any method can be marked synchronized. When executing



Fig. 1. Overhead of Synchronization

a synchronized method, a thread acquires the monitor associated with the receiver object,¹ runs the method's code, and then releases the monitor. An explicit synchronization statement provides a way to manipulate monitors at program points other than method invocations. Java's monitors are reentrant, meaning that a single thread can acquire a monitor multiple times in a nested fashion. A reentrant monitor is only released when the thread exits the outermost method or statement that synchronizes on that monitor.

2.2 Cost

Synchronization represents a significant performance bottleneck for a set of Java benchmarks. To quantify the cost of synchronization operations, we compared singlethreaded Java programs to versions of the same programs where synchronization has been removed from both the application and the standard Java library. Since the correctness of multithreaded benchmarks depends on the presence of synchronization, we did not perform these measurements on multithreaded benchmarks. However, the unnecessary synchronization present in singlethreaded programs suggests that a significant amount of the synchronization in multithreaded programs is also unnecessary.

We used a binary rewriter [SGA+98] to eliminate all synchronization operations from the application binaries. This strategy allowed us to perform measurements on commercial Java virtual machines without having to instrument and recompile them at the source level.

We examine the benchmarks using two different Java implementations that are representative of different Java virtual machine implementations. The JDK 1.2.0 embodies a hybrid JIT compilation and interpretation scheme, and features an efficient implementation of lock operations. Consequently, it represents the state of the art in commercially available Java virtual machines. Vortex, an aggressively optimizing research compiler [DDG+96], produces natively compiled stand-alone executables and uses efficient synchronization primitives [BKM+98]. For these figures, we use the base Vortex system, which does not contain the analyses described in this paper.

Figure 1 shows the percentage of total execution time spent on synchronization in five singlethreaded benchmarks for each platform. Synchronization overhead averages 5-10% of

¹static synchronized methods acquire the monitor associated with the Class object for the enclosing class.

```
class Enclosing {
class Reentrant {
   synchronized foo() {
                                        Enclosed member;
                                        synchronized foo() {
       this.bar()
                                            member.bar();
   }
   synchronized bar()
                                        }
                                     }
       \{\ldots\}
}
                                     class Enclosed {
                                        synchronized bar()
                                            \{\ldots\}
                                     }
       Fig. 2. Reentrant Monitors
                                           Fig. 3. Enclosed Monitors
```

execution time, depending on the platform, and can be as high as 35%. The relative cost of synchronization varies between the platforms because of the varying amounts of optimization they perform in the compilation process, and their different synchronization implementations. For example, if Vortex is able to optimize the non-synchronization-related parts of a benchmark like *jlex* more effectively than the JDK 1.2.0, its synchronization overhead will be relatively more significant. In contrast, the benchmarks *javac* and *cassowary* may use synchronization in a way that is more expensive on the JDK platform than on Vortex. Despite the variations between platforms, synchronization overhead represents a significant portion of the execution time for these Java benchmarks, demonstrating that there is considerable room for performance improvement over current synchronization technology.

3. Optimization Opportunities

In this section, we describe three different opportunities for optimizing synchronization operations.

3.1 Reentrant Monitors

Reentrant monitors present the simplest form of unnecessary synchronization. As illustrated in Figure 2, a monitor is reentrant when one synchronized method calls another with the same receiver object. It is safe to remove synchronization from bar if all calls to bar reachable during program execution are within procedures that synchronize on the same receiver object. Our optimization generalizes this example to arbitrary call paths: synchronization on the receiver object O of method bar may be removed if along every reachable path in the call graph to bar there is a method or statement synchronized on the same object O.

If the receiver object's monitor has been entered along some, but not all, call paths to method bar, specialization can be used to create two versions of bar: an unsynchronized version for the call paths where the receiver is already synchronized, and a synchronized version for the other call paths. The synchronized version acquires the lock and then simply calls the unsynchronized version. For example, if bar is also called from the function main, where the receiver object is not synchronized, bar could be specialized so that main calls a synchronized version that acquires a monitor. Methods like foo that have already locked the receiver object can still call the more efficient, unsynchronized version of bar.



Fig. 4. Immutable Paths

3.2 Enclosed Monitors

An *enclosed monitor* is a monitor that is already protected from concurrent access by another monitor. The enclosing monitor is always entered first, and while it is held the enclosed monitor is acquired. Later, the enclosed monitor and then the enclosing monitor will be released. Because the enclosed monitor is only entered when the enclosing monitor is held, it is protected from concurrent access and is unnecessary. For example, in Figure 3 the monitor on the member object is enclosed by the monitor on the Enclosing object. Thus the synchronization on the bar function is unnecessary and may be removed.

In order to remove synchronization safely from a monitor M during static analysis, we must prove there is a unique, unchanging enclosing monitor that protects M, not one of several enclosing monitors. If there were several Enclosing objects in Figure 3, for example, different threads could access the Enclosed object concurrently by going through different Enclosing objects, and it would be unsafe to remove synchronization from bar. There are four ways we can ensure this is the case:

First, the enclosing monitor may store the enclosed monitor in an *unshared* field—a field that holds the only reference to the enclosed object. Since the unshared field holds the only reference to the enclosed object, the only way to enter the enclosed object's monitor is to go through the (unique) enclosing object. We can relax the "only reference" condition in the definition of an unshared field if we use the name of the field to identify the enclosing lock. As long each enclosed object is only stored in one instance (i.e., run-time occurrence) of that field, it is permissible for other fields and local variables to refer to the enclosed object, because the field name uniquely identifies the enclosing object.

Second, the enclosing monitor may be stored in an *immutable static field*, i.e. a global variable that does not change value. Because the enclosing monitor is identified by the static field, and only one object is ever stored in that static field, the field name uniquely identifies a monitor. The static field's monitor M encloses another monitor M' if all synchronization operations on M' execute from within monitor M.

Third, the enclosing monitor may be stored in an immutable field of the enclosed monitor. Since an immutable field cannot change, the same enclosing monitor is always entered before the enclosed monitor. This case occurs when a method first synchronizes on a field of the receiver object, then on the receiver object itself.



Fig. 5. Thread-Local Monitors

Fig. 6. Optimization Potential

Fourth, the cases above can be combined. For example, Figure 4 illustrates an example similar to cases in the JDK 1.2.0 I/O library when an stream object first synchronizes on an object in one of its fields, then calls a synchronized method on the object in another field. In the example, it is safe to remove the synchronization on StringWriter.write because the lock object of an enclosing stream is always locked before calling write. Since lock is an immutable field of PrintWriter and out is an unshared field of PrintWriter, we can use transitivity to determine that there is a unique enclosing object (lock) for each enclosed object (out). Using transitivity, we can combine a sequence of immutable and unshared fields into a *unique path* from the enclosed monitor to the enclosing monitor. A unique path identifies a unique enclosing object.

The general rule we have developed can be stated as follows:

A synchronization statement S may be removed if, for every other synchronization statement S that could synchronize on the same object as S, there exists an unique path of links such that:

- 1. The first link represents the object synchronized on by *S* and *S*'
- 2. Each subsequent link is either an unshared field of an object that encloses the link before or an immutable field that is enclosed by the link before
- 3. The last link represents an object that is synchronized on all call paths that reach *S* and is also synchronized on all call paths that reach *S*'

As in the case of reentrant monitors, synchronization statements on enclosed objects may be specialized if it is legal to remove synchronization on some instances of a class but not others. For example, the root node in a binary tree encloses all of the inner nodes, so specialization could create two kinds of nodes: one that is synchronized for creating the root of a binary tree, and one that is unsynchronized for creating the inner nodes of the tree.

3.3 Thread-Local Monitors

Figure 5 shows an example of a thread-local monitor. Instances of the Local class are only accessible by the thread that created them, because they are created on the stack and are not

accessible via any static field. Since static fields are the only base case for sharing data between threads in Java's memory model, it is safe to remove synchronization on methods of any class that is unreachable from static fields. In our model, Thread and its subclasses are stored in a global list, so that passing references from one thread to another during thread creation is handled correctly. Specialization can eliminate synchronization when some instances of a class are thread-local and other instances are not.

3.4 Optimization Potential

Figure 6 shows an estimate of the opportunities for optimization in our benchmark suite, demonstrating that different programs present different optimization opportunities. This data was collected from dynamic traces of the five Java programs running on the JDK 1.1.6. For each benchmark, it shows the percentages of dynamic monitor operations that were reentrant, enclosed (by a different monitor), and thread-local, representing an upper bound for how well our analyses could perform. The bars may add up to more than 100% because some synchronization operations may fall into several different categories. All the benchmarks do 100% of their synchronization on thread-local monitors because they are singlethreaded, and so no monitor is ever locked by more than one thread. Multithreaded benchmarks would have some synchronization that is not thread-local, but we believe that thread-local monitors would still represent a significant opportunity in these benchmarks.

The benchmarks differ significantly in the optimization opportunities they present. For example, 41% of the synchronization in *jlex* is reentrant but less than 1% is enclosed. In contrast, 97% of the synchronization in *javac* is enclosed and virtually none is reentrant. For these singlethreaded benchmarks, thread-local monitors present the greatest opportunity for optimization, with two programs gaining significant benefit from enclosing or reentrant monitors. This data demonstrates that each kind of optimization is important for some Java programs.

4. Analyses

We define a simplified analysis language and describe three analyses necessary to optimize the synchronization opportunities discussed above: lock analysis, unshared field analysis, and multithreaded object analysis. Lock analysis computes a description of the monitors held at each synchronization point so that reentrant locks and enclosed locks can be eliminated. Unshared field analysis identifies unshared fields so that lock analysis can safely identify enclosed locks. Finally, multithreaded object analysis identifies which objects may be accessible by more than one thread. This enables the elimination of all synchronization on objects that are not multithreaded. Our analyses can rely on Java's **final** annotation to detect immutable fields; an important area of future work is to detect immutable fields that are not explicitly annotated as **final**.

4.1 Analysis Language

We describe our analyses in terms of a simple expression-based core language, incorporating the essential synchronization-related aspects of Java. This allows us to focus on the details relevant to specifying the analyses while avoiding some of the complexity of a real language. It is straightforward to handle the missing features of Java—our prototype implementation

```
id, field, fn \in ID

label \in LABEL

key \in KEY

e, program \in E

E ::= new<sup>KEY</sup>

| ID

| let ID := E<sub>1</sub> in E<sub>2</sub>

| E.ID

| E<sub>1</sub>.ID := E<sub>2</sub>

| E<sub>1</sub> op E<sub>2</sub>

| synchronized<sup>LABEL</sup> (E<sub>1</sub>) { E<sub>2</sub> }

| if E<sub>1</sub> then E<sub>2</sub> else E<sub>3</sub>

| ID(E<sub>1</sub>,...,E<sub>n</sub>)
```

Fig. 7. Core Analysis Language

handles all of the Java language except reflection and dynamic code loading, which are omitted to enable static reasoning.

Figure 7 presents our analysis language. It is a simple, first-order language, incorporating object creation, field access and assignment, let-bound identifiers, synchronization expressions, and simple control flow. Each object creation point is labeled with a *class key* [GDD+97], which identifies the group of objects created at that point. In our implementation, there is a unique key for each **new** statement in the program; in other implementations a key could represent a class, or could represent another form of context sensitivity. We assume that all letbound identifiers are given unique names. Static field references are modeled as references to a field of the special object global, which is implicitly passed to every procedure. We assume all procedures are put into an implicit global table before evaluating the main expression. The *lookup* function returns the λ -expression associated with a particular procedure.

We model ordinary binary operators like + and ; (which evaluates and discards its first argument before returning the second) with the E_1 op E_2 syntax. Control flow operations include simple function calls and a functional **if** expression—facilities that can be combined to form other structures like loops and object-oriented dispatch. Finally, Java's synchronization construct is modeled by a **synchronized** statement, which locks the object referred to by E_1 and then evaluates E_2 before releasing the lock. Each **synchronized** statement in the program text is associated with a unique label \in LABEL that is used in our analyses.

4.2 Analysis Context

Our analyses are parameterized by other alias and class analyses, a feature of our approach that allows a tradeoff between analysis time and the precision of our analysis results. Our analyses also benefit from earlier copy propagation and must-alias analysis passes, which merge identifiers that point to the same object. We assume the following functions are defined from earlier analysis passes:

- *id_aliases*(e) the set of identifiers that may point to the same value as expression e
- field_aliases(field) the set of fields declarations whose instances may point to the same object as field. This information can be easily computed from a class analysis.

- is_immutable(field) true if field is immutable (i.e., write-once). This may be deduced from final annotations and constructor code.
- label_aliases(label) the set of labels of synchronization statements
 that may lock the same object as the synchronization statement
 associated with label

Some of our analyses deal with groups of objects, represented by class keys. We assume that an earlier class pass has found a conservative approximation to the set of objects that can be in each variable or field in the program. Our implementation uses the 1-1-CFA algorithm [S88][GDD+97], which considers each procedure in one level of calling context and analyzes objects from different creation points separately, and the 0-CFA algorithm, which lacks this context-sensitivity. We use the following functions to access this information:

field_keys(field, key) - the set of class keys to which field field may
 refer when accessed through a particular class key key
static_field_keys(field) - the set of class keys to which static field
 field may refer

4.3 Analyses

Our analyses compute the following functions:

is_multithreaded(key) - true if objects described by key may be accessible through static variables

We describe our first two analyses in syntax-directed form, where a semantic function maps an expression and a set of inherited attributes to a set of synthesized attributes. The third analysis uses only data from previous analyses and does not work directly over the program text.

Lock Analysis. Figure 8 defines the domains and helper functions that are used by our lock analysis flow functions. Our lock analysis, shown in Figure 9, describes locks in terms of paths and bipaths. A *path* names a particular object relative to an identifier, and consists of the identifier name and a series of field accesses. Thus, the path $id \rightarrow field_1 \rightarrow field_2$ represents the expression $id.field_1.field_2$. A *bipath* represents a bi-directional path. The forward links represent field dereferences, as in paths, while the backward links mean "is enclosed by"—that is, in a bipath of the form *bipath_{sub}* \leftarrow field, the expression denoted by *bipath_{sub}* is referenced by the field field of some unspecified object. In our descriptions, we use the notation $m[x \rightarrow y]$ to denote that we compute a new mapping $\in X \rightarrow Y$ that is identical to mapping *m* except that element $x \in X$ is mapped to $y \in Y$.

```
path \in PATH = ID + PATH \rightarrow ID
dir \in DIR = { \rightarrow, \leftarrow }
bipath \in BIPATH = ID + BIPATH \times DIR \times ID
lockset \in LOCKSET = 2^{BIPATH}
lockmap \in LOCKMAP = LABEL \rightarrow_{fin} LOCKSET
idmap \in \texttt{IDMAP} = \texttt{ID} \ \rightarrow_{fin} 2^{\texttt{PATH}}
is_immutable_path(path) : bool
     switch (path)
          case id : true
          case path' \rightarrow field : is_immutable(field) \land is_immutable_path(path')
is_prefix(bipath<sub>1</sub>, bipath<sub>2</sub>) : bool
     if (bipath_1 = bipath_2) then true
     else if (bipath<sub>2</sub> = id) then false
     else if (bipath<sub>2</sub> = bipath' dir field) then is_prefix(bipath<sub>1</sub>, bipath')
substitute(bipath<sub>1</sub>, path, bipath<sub>2</sub>) : BIPATH
     if (bipath<sub>1</sub> = path) then bipath<sub>2</sub>
     else if (bipath<sub>1</sub> = bipath' dir field)
          then substitute(bipath', path, bipath<sub>2</sub>) dir field
     else error
map_lock(bipath<sub>1</sub>, path, bipath<sub>2</sub>) : BIPATH \cup { not_defined }
     if (is_prefix(path, bipath<sub>1</sub>)) then substitute(bipath<sub>1</sub>, path, bipath<sub>2</sub>)
     else if (path = path' \rightarrow field \land is_unshared(field))
          then map\_lock(bipath<sub>1</sub>, path', bipath<sub>2</sub> \leftarrow field)
     else not_defined
```

```
map\_lockset(lockset, path_1, path_2) : LOCKSET 
{ map\_lock(bipath, path_1, path_2) | bipath \in lockset \} - \{ not\_defined \}
```

Fig. 8. Domains and Helper Functions for Lock Analysis

The lock analysis function L accepts four arguments in curried style. The first argument is an expression from the text of the program. The second argument, a lockset, is the set of bipaths representing locks held at this program point. The third argument, a lockmap, is the current mapping from synchronization labels to sets of bipaths representing locks held at each labeled synchronization statement. The final argument, an idmap, is a mapping from identifiers to paths that describe the different field expressions that the identifier aliases. The result of lock analysis is a lockmap that summarizes the locks held at every reachable synchronization label in the program. We analyze the expression representing the program in the context of an empty lockset (no lock encloses the entire program expression), an optimistic lockmap (no synchronization points have been analyzed yet), and an empty idmap (no identifiers are in scope).

Many of the analysis flow functions in Figure 9 are relatively straightforward; we discuss only the more subtle ones below. The rules for **let** and id expressions update the idmap for identifiers and return the pathset represented by an identifier. A field expression simply extends all paths in e's pathset with field.

 $L: \mathbf{E} \to \mathbf{LOCKSET} \to \mathbf{LOCKMAP} \to \mathbf{IDMAP} \to \mathbf{2}^{\mathbf{PATH}} \times \mathbf{LOCKMAP}$

get_locks(label):LOCKSET = **let** (pathset', lockmap') = L[[program]] $\emptyset \emptyset \emptyset$ **in** lockmap'(label) $L[new^{key}]$ lockset lockmap idmap = (Ø, lockmap) L[[id]] lockset lockmap idmap = ({ id } \cup idmap(id), lockmap) L[[let id := e₁ in e₂]] lockset lockmap idmap = **let** (pathset', lockmap') = *L*[[e₁]] lockset lockmap idmap **in** $L[[e_2]]$ lockset lockmap' idmap[id \rightarrow pathset'] *L*[[e.field]]lockset lockmap idmap = **let** (pathset', lockmap') = *L*[[e]] lockset lockmap idmap in ({ path \rightarrow field | path \in pathset' }, lockmap') *L*[[e₁.field := e₂]] lockset lockmap idmap = let (pathset', lockmap') = $L[[e_2]]$ lockset lockmap idmap in let (pathset", lockmap") = $L[[e_1]]$ lockset lockmap' idmap in (Ø, lockmap") $L[[e_1 \text{ op } e_2]]$ lockset lockmap idmap = **let** (pathset', lockmap') = *L*[[e₁]] lockset lockmap idmap in let (pathset", lockmap") = $L[[e_2]]$ lockset lockmap' idmap in (Ø, lockmap") $L[synchronized^{label} (e_1) \{ e_2 \}]] lockset lockmap idmap =$ **let** (pathset', lockmap') = *L*[[e₁]] lockset lockmap idmap **in** U let lockmap'' = lockmap'[label \rightarrow map_lockset(lockset, path, SYNCH)] in path∈ pathset' $L[e_2]$ (lockset \cup { path | path \in pathset' \land *is_immutable_path*(path) }) lockmap" idmap L[[if e_1 then e_2 else e_3]] lockset lockmap idmap = let (pathset', lockmap') = $L[[e_1]]$ lockset lockmap idmap in let (pathset", lockmap") = $L[[e_2]]$ lockset lockmap' idmap in **let** (pathset''', lockmap''') = *L*[[e₃]]lockset lockmap'' idmap **in** (pathset" \cap pathset"', lockmap'") L[[fn(e₁,...,e_n)]] lockset lockmap₀ idmap = let $[\lambda(formal_1, \ldots, formal_n)] \in [-lookup(fn)]$ in $\forall i \in 1..n$ let (pathset_i, lockmap_i) = $L[[e_i]]$ lockset lockmap_{i-1} idmap in map_lockset(lockset, path, formal;) in let lockset' = *i*∈1..*n* path∈ pathset let (pathset', lockmap') = $context_strategy(L[[e]]]lockset' lockmap_n \emptyset)$ in ({ *substitute*(path, formal, path') $| \text{path} \in \text{pathset'} \land \exists i \in 1... \text{ s.t. } is_prefix(\texttt{formal}_i, \texttt{path}) \land \texttt{path'} \in \texttt{pathset}_i \},$ lockmap')

Fig. 9. Lock Analysis Flow Functions

When a synchronization statement is encountered, the lockmap is updated with all of the bipaths in the lockset. Before being added to the lockset, however, these bipaths are converted to a normal form in terms of e_1 , the expression on which the statement synchronizes. This normal form allows us to compare the bipath descriptions of the locks held at different synchronization points in the program in the lock elimination optimization described below.

The normal form expresses a lock in terms of the special identifier **SYNCH** representing e_1 , the object being locked. The *map_lockset* function considers each bipath in the lockset in turn and uses *map_lock* to compute a new bipath in terms of a mapping from the pathset of e_1 to **SYNCH**. For each bipath *b* in the lockset, *map_lock* will substitute **SYNCH** into the lock expression bipath if the bipath is a prefix of *b*. For example, if the path corresponding to e_1 is id \rightarrow field₁ and the lockset is { id \rightarrow field₁ \rightarrow field₂ }, then *map_lock*(id \rightarrow field₁ \rightarrow field₂, id \rightarrow field₁, **SYNCH**) = **SYNCH** \rightarrow field₂, signifying that the field field₂ of the object referred to by e_1 is already locked at this point.

If the prefix rule does not apply and the last field field in the synchronization expression path is unshared, then map_lock will try to match against a shorter prefix with SYNCH \leftarrow field as the expression to be substituted. In Figure 5, the synchronization expression PrintWriter \rightarrow out is not a prefix of the currently locked object PrintWriter \rightarrow lock, so since out is an unshared field map_lock will attempt to substitute SYNCH \leftarrow out for PrintWriter instead. Thus the result we get is map_lock (PrintWriter \rightarrow lock, PrintWriter \rightarrow out, SYNCH) = SYNCH \leftarrow out \rightarrow lock. That is, at the current synchronization point the program holds a lock on the lock field of the object whose out field points to the object currently being synchronized. This is a correct description of the case in Figure 5.

Next, the expression inside the synchronization block is evaluated in the context of the current lockset combined with all paths in the synchronization expression's pathset that are unique. The *is_immutable_path* function, which checks that each field in a path is immutable, ensures that no lock description is added to the lockset unless it uniquely identifies the locked object in terms of the base identifier.

At function calls, we look up the definition of the called function and evaluate the actual parameters to produce a set of paths for each parameter and an updated lockmap. The *map_lockset* function is used to map actual paths to formal variables in each lock in the lockset. Information about locks that are not related to formal parameters (including the implicit formal parameter global mentioned subsection 4.2) cannot be used by the callee, since there would be no way to ensure that the locked object protects any synchronization statements there. The callee is analyzed in the context of the new lockset and lockmap, and the result is memoized to avoid needless recomputation.

Our analysis may be parameterized by a *context_strategy* function that allows a varying level of context sensitivity. The current implementation is context-insensitive—it simply computes the intersections of the incoming lockset with all other locksets and re-evaluates the callee in the context of the new lockset if the input information has changed since the last analysis. We avoid infinite recursion in our analysis by returning an empty pathset and the existing lockmap when a lock analysis flow function is called recursively with identical input analysis information; the analysis will automatically iterate until a sound fixpoint is reached. Since the lockset must decrease in size each time a function is reanalyzed, termination of our analysis is assured. Finally, the set of paths is returned from the function call by mapping back from formals to actuals.

fset, shared \in FSET = 2^{ID} $idstate \in IDSTATE = ID \rightarrow_{fin} FSET$ $U \colon \mathbf{E} \to \mathtt{IDSTATE} \to \mathtt{FSET} \to \mathtt{FSET} \times \mathtt{IDSTATE} \times \mathtt{FSET}$ *is_unshared*(field) = let (idstate, shared) = U[[program]]ØØ in (field ∉ shared) $U[[new^{key}]]$ idstate shared = (Ø, idstate, shared) *U*[[id]] idstate shared = (idstate(id), idstate, shared) U[[let id := e₁ in e₂]]idstate shared = let (fset', idstate', shared') = $U[[e_1]]$ idstate shared in $U[[e_2]]$ idstate'[id \rightarrow fset'] shared' *U*[[e.field]]idstate shared = let (fset', idstate', shared') = U[[e]] idstate shared in (field_aliases(field), idstate', shared') U[[e1.field := e2]]idstate shared = let (fset', idstate', shared') = $U[[e_1]]$ idstate shared in let (fset", idstate", shared") = $U[[e_2]]$ idstate' shared' in let fset'" = field ∪ fset" in (fset'", idstate"[id \rightarrow idstate"(id) \cup fset"' | id \in *id_aliases*(e₂)], **if** (fset" ∉ field) **then** shared" **else** shared" ∪ field) $U[[e_1 \text{ op } e_2]]$ idstate shared = **let** (fset', idstate', shared') = *U*[[e₁]] idstate shared **in** let (fset", idstate", shared") = $U[[e_2]]$ idstate' shared' in (fset' mergeop fset", idstate", shared") U[[synchronized^{label} (e₁) { e₂ }]]idstate shared = let (fset', idstate', shared') = $U[[e_1]]$ idstate shared in U[[e₂]]idstate' shared' U[[if e_1 then e_2 else e_3]] idstate shared = let (fset', idstate', shared') = $U[[e_1]]$ idstate shared in let (fset", idstate", shared") = $U[[e_2]]$ idstate' shared' in let (fset''', idstate''', shared''') = $U[[e_2]]$ idstate' shared' in (fset" \cup fset"', idstate" \cup idstate"', shared" \cup shared"') U[[fn(e₁,...,e_n)]]idstate₀ shared₀ = let $[[\lambda(formal_1, \ldots, formal_n) e]] = lookup(fn)$ in $\forall i \in 1..n$ let (fset_i, idstate_i, shared_i) = $U[[e_i]]$ idstate_{i-1} shared_{i-1} in let idstate' = { formal_i \rightarrow fset_i | i \in 1..n } in let (fset", idstate", shared") = $context_strategy(U[[e]]] idstate' shared_n)$ in (fset", $idstate_n[id \rightarrow idstate_n(id) \cup idstate''(formal_i) | i \in 1..n and id \in id_aliases(e_i)],$ shared")

Fig. 10. Unshared Field Analysis

Unshared Field Analysis. The unshared field analysis described in Figure 10 computes the set of fields that are *shared*, i.e. may refer to objects that are also stored in other instances (that is, run-time occurrences) of the same field. Unshared fields are in the complement of this shared field set. The result of this analysis is used in the *map_lock* function of the previous analysis to detect enclosing locks.

The information computed by unshared field analysis differs from the result of the *field_aliases* function in two essential ways. First, the *field_aliases* function cannot tell whether two instances of a given field declaration may point to the same object, which determines whether a given field is shared. Second, our unshared field analysis is flow-sensitive, enabling increased precision over non-flow-sensitive techniques.

The analysis works by keeping track of which fields each identifier and expression could alias. When a field is assigned a value that may have originated in another instance of the same field, the analysis marks the field shared. *U*, the analysis function for unshared field analysis, accepts as curried parameters a program expression, the set of currently shared fields, and a mapping from identifiers to the sets of fields whose contents they may point to. It then computes the set of fields the expression may alias and an updated set of shared fields. Our analysis is run on the program's top-level expression, using an initially empty identifier mapping (since no identifiers are initially in scope) and initially optimistically assuming that all fields are unshared. The rules for field references, field assignment, and function calls are the most interesting.

When a field field is dereferenced, the resulting expression may alias any field in *field_aliases*(field). At assignments to a field field, we must update the identifier mapping for any identifier that could alias the expression being assigned to field, since the values these identifiers point to could also be referenced by field due to the assignment. In fact, due to the actions of other threads, these identifiers could alias any field in *field_aliases*(field). For the purposes identifying unshared fields, however, we can optimistically assume that such aliasing does not occur when writing to a field. This enables our analysis to detect unshared fields even when the same object is written to two fields with different names. If this object is later copied from one field to another, the field written to will be correctly identified as shared because aliasing is accounted for when reading fields. If the expression being assigned may not alias the field being assigned, then the field being assigned may remain unshared; otherwise, it is added to the shared set. In expressions of the form e_1 op e_2 , the correct merge function for the expression's field set depends on the operator. For example, the merge function for the ; operand simply returns the field set of its second argument.

At a function call, we lookup the callee and evaluate all the argument expressions to get a set of fields for each of them as well as an updated identifier map and shared field set. The idstate for the callee consists of a mapping from its formal parameters to the field sets of each actual parameter expression. We then evaluate the callee in the context of the new idstate and the current shared set, and return the resulting field set and shared set. After evaluating the callee, it is also necessary to update the identifier state of the caller. Every id that may alias an actual expression could now reference any field that the formal parameter of the callee could reference after evaluating the callee. This update is necessary because some callee (possibly several levels down the call graph) may have assigned the parameter's value to a field.

Our *context_strategy* for this analysis is context sensitive, as we re-evaluate the callee for each different identifier mapping. In practice, context sensitivity enables results that are much more precise. For example, when a callee is called with a formal parameter aliased to field at one call site, we don't want all other call sites to see that the formal may alias field after the call and thus conservatively assume that the callee assigned that formal to field. Termination is assured because the results of each analysis are memoized, and the size of the field sets is bounded by the number of fields in the program. Recursive functions are handled

$$multi = \left(\bigcup_{f \in staticFields} static_field_keys(f)\right) \bigcup \left(\bigcup_{\substack{k \in multi\\ f \in fields(k)}} field_keys(f,k)\right)$$

Fig. 11. Multithreaded Object Analysis

by optimistically returning the empty set of fields at recursive calls, and the analyses subsequently iterate until a sound fixpoint is reached.

Multithreaded Object Analysis. We define *multi*, a set of class keys, as the smallest set satisfying the recursive equation shown in Figure 11. Then we define *is_multithreaded* as follows:

is_multithreaded(key) = key ∈ *multi*

Our implementation simply starts with class keys referenced by static fields, and for each class key it considers each field of that key and adds the keys that field may reference to the set *multi*. When the set reaches the least fixed point, the analysis is complete. The analysis must terminate because there is a finite number of class keys to consider.

4.4 Applying the Results

To apply the results of our analyses, we perform an optimization pass during code generation. At each statement of the form

synchronized^{label} (e_1) { e_2 }

we replace it with the statement e_1 ; e_2 if any of the following conditions holds:

- 1. SYNCH \in get_locks(label), or
- 2. $\forall label' \in label_aliases(label)$.
- $get_locks(label) \cap get_locks(label') \neq \emptyset$, or
- 3. $\forall \text{key} \in \text{label}_{keys}(\text{label})$. $\neg is_multithreaded(\text{key})$

The first condition represents a reentrant monitor—if the monitor associated with expression e_1 is already locked, then **SYNCH** \in *get_locks*(label). Here *get_locks*(label) is defined (in Figure 9) to be the result of lock analysis at the program point identified by label. We can safely replace the synchronized expression with a sequence that evaluates the lock expression (for potential side effects) and then evaluates and returns the expression protected within the synchronization statement. The second condition represents the generalization to enclosed locks: a synchronization statement *S* may be eliminated if, for every other synchronization statement *S*'that may lock the same object, some common lock is already held at both *S* and *S*'. The third condition removes synchronization statements that synchronize on an expression that refers only to non-multithreaded class keys.

Due to the complicated semantics of monitors in Java, our optimizations may not conform to the Java specification on some multiprocessor systems. According to the Java language specification, "locking any lock conceptually flushes all variables from a thread's working memory, and unlocking any lock forces the writing out to main memory of all variables that the thread has assigned." [GJS96] This implies, for example, that a legal Java program may pass data (in a timing-dependent manner) from one thread to another by having each thread synchronize on a thread-local object. This kind of "covert channel" communication could be broken by our optimizations. An implementation that synchronizes the caches of a multiprocessor even when other parts of a synchronization operation have been removed would comply with the Java specification, for example. Our optimizations are always safe, however, in a Java-like language with a somewhat looser synchronization guarantee which could be informally stated as follows: if thread T_1 writes to a variable V and then unlocks a lock and thread T_2 locks the same lock and reads variable V, then thread T_2 will read the value that T_1 wrote. We believe that most well written multithreaded programs in Java use this model of synchronization.

5. Results

A preliminary performance evaluation shows that a subset of our analysis is able to eliminate 30-70% of the synchronization overhead in several of our benchmarks. We have implemented prototype versions of reentrant lock analysis and multithreaded object analysis, and transformations that use the results of these analyses. Our implementation does not yet apply specialization to optimize different instances of an object or method separately. Although our results are preliminary, they demonstrate the promise of our approach. We plan to complete and evaluate a more robust and detailed implementation in the future, which will include unshared field analysis and enclosed lock analysis.

We demonstrate the performance benefit of our analyses on the five singlethreaded benchmarks presented earlier. While these benchmarks could be optimized trivially by removing all synchronization, they are real programs and may be partly representative of how synchronization is used in multithreaded programs as well. *Javac, javacup, jlex,* and *pizza* are all compiler tools; *cassowary* is a constraint solver. We hope to evaluate our techniques on multithreaded programs in the future.

Our prototype implementation is built on the Vortex compiler infrastructure [DDG+96] augmented with a simple, portable, non-preemptive, user-level threading package based on QuickThreads [K93]. We compiled all programs with a full suite of conventional optimizations, as well as interprocedural class analysis. For our small benchmarks, we used a 1-1-CFA call graph construction algorithm [GDD+97]; this did not scale well to *pizza, javac*, and *javacup*, so we used a simpler 0-CFA analysis for these programs, possibly missing some optimization opportunities due to more conservative alias information. Our lock implementation is already highly optimized, using an efficient lock implementation [BKM+98]. We compiled two versions—one with and one without our synchronization optimizations. Both versions included all other Vortex optimizations. All our runtime overhead measurements come from the average of five runs on a SPARC ULTRA 2 machine with 512 MB of memory. We ran the benchmarked program once before the data were collected to eliminate cold cache startup effects.

Table 1 shows statistics about how our analyses performed. The first two columns show the total number of classes in the program and the number identified as thread-local. Multithreaded object analysis identified a large fraction of classes as singlethreaded for the *jlex*, *javacup*, and *cassowary* benchmarks, but was less successful for *javac* or *pizza*. Since these benchmarks are singlethreaded, all their classes are thread-local. However, because our analysis assumes static field references make a class reachable by other threads, our analysis is only able to determine this for a subset of the classes in each program.

Benchmark	classes		total lock	lock ops removed			% overhead
			ops				removed
	total	thread-local		reentrant	thread-local	total	
jlex	56	34	27	2	8	9	67%
pizza	184	0	38	6	0	6	N/A
javacup	66	28	30	2	6	7	47%
cassowary	57	29	32	4	12	13	27%
javac	194	0	68	5	0	5	0%

Table 1. Synchronization Analysis Statistics

The next four columns of Table 1 show the total (static) number of synchronization operations, the number removed by reentrant lock analysis, the number of thread-local operations removed, and the total number of operations removed. The total is less than the sum from the two analyses because some synchronization operations were removed by both analyses. As suggested by the class figures in the first two columns, multithreaded object analysis was more effective than reentrant lock analysis for *jlex*, *javacup*, and *cassowary*, while *pizza* and *javac* only benefited from reentrant lock analysis. In general, our analyses removed 20-40% of the static synchronization operations in the program.

The last column summarizes our runtime performance results. We present the speedup achieved by our optimizations as a percentage of the overhead of synchronization for Vortex. For *jlex, javacup*, and *cassowary*, we eliminated a significant percentage of the synchronization overhead, approaching 70% in the case of *jlex*. The absolute speedups ranged up to 5% in the case of *jlex*. Pizza did not have a significant overhead from synchronization, so no speedup was achievable. We also got no measurable speedup on *javac*.

The speedup in the case of *jlex* is due the large number of stack operations performed by this benchmark, which our analysis optimized effectively. Multithreaded analysis discovered that all of the Stack objects were thread-local, and lock analysis was successful in removing some reentrant locks in the Stack code. Most of the remaining synchronization is on DataOutputStream and BufferedOutputStream objects. Multithreaded object analysis determined that DataOutputStream was thread-local and that the most important instances of BufferedOutputStream were thread-local, but because our implementation does not yet produce specialized code for instances of BufferedOutputStream that are thread-local we were unable to take advantage of this knowledge. Implementing specialization would improve our optimization performance here.

Over 99% of *Javacup*'s synchronization comes from manipulation of strings, bitsets, stacks, hashtables, and I/O streams. Multithreaded analysis was able to remove synchronization from every method of StringBuffer, but was did not eliminate synchronization from other objects. Each of the other classes was reachable from a static variable, either in the Java library or in the *javacup* application code.

To optimize this code effectively would require three additional elements. First, we need a scalable analysis that distinguishes program creation points so that one multithreaded Hashtable does not make all Hashtables multithreaded. Our current 1-1-CFA analysis that distinguishes creation points does not scale to *javacup* or *javac*, and therefore our performance suffers for both benchmarks. Second, we need specialization to optimize different instances of the same class separately. Third, we need a more effective multithreaded analysis that can determine if a static variable is only used by one thread, rather than conservatively assuming all such variables are multithreaded.

In *Cassowary*, multithreaded analysis was able to remove synchronization from all the methods of Vector. However, the primary source of synchronization overhead was Hashtable, which was not optimized by our multithreaded analysis because it was reachable from static fields.

Although a few operations were optimized in *javac*, we did not measure any speedup in this benchmark. Since *javac* executes many operations on enclosed monitors, we expect these results to improve once we have implemented our unshared field analysis and enclosed lock analysis.

Considering that we have achieved a significant fraction of the potential speedup for several of our benchmarks although many important elements of our analyses are not yet implemented, we find these results promising.

6. Related Work

A large body of work (e.g., [ALL89] [KP98]) has focused on reducing the overhead of locking or synchronization operations. Most recently, Bacon's Thin Locks [BKM+98] reduce the overhead of Java's synchronization to a few instructions in the common case. Thin locks improve the performance of real programs by up to 70% by reducing the latency of individual synchronization operations. Our analyses complement this work by reducing the number of synchronization operations.

Diniz and Rinard [DR98] present two techniques for lock coarsening in parallelizing compilers: merging multiple locks into one, so that several objects are protected by one lock, and transforming locks that are repeatedly acquired and released within a method so that they are only acquired and released once. Their work is applicable to explicitly parallel programs; however, they do not evaluate their optimizations in this context. They do not consider thread-local locks, do not consider immutable fields as a potential source of lock nesting, and apparently can only optimize nested locks in languages like C++ where objects can be statically declared to be represented inline. Their coarsening optimizations are complementary to our work; while we can eliminate a broader class of redundant locks, their optimizations may lead to acquiring the non-redundant locks fewer times.

Another source of related work is the Concert project at the University of Illinois. To reduce the overhead of lock operations, they optimize calls from one method to another on the same receiver by eliminating the lock operation from the second method during inlining [PZC95]. They also do a lock coarsening optimization similar to that in [DR98]. Our research extends and generalizes their results by optimizing enclosing locks and thread-local objects.

Our concept of an unshared field is similar to idea of a unique pointer [M96] or unique aliasing mode [H91][NVP98]. Unlike the previous work, we find unshared fields automatically, rather than requiring annotations from the programmer. Our unshared field analysis is similar to an analysis used by Dolby to inline object fields [D97]. In order to safely inline a field, his system propagates tags to determine which fields could alias particular variables. The precision of his analysis is identical to ours given a similar analysis framework, but his work requires more strict conditions to inline a field than ours requires to identify an unshared field.

Work from the model-checking community [C98] performs shape analyses similar to ours in order to simplify models of concurrent systems. These analyses remove recursive locks and locks on thread-local objects from formal models. This allows a model checker to more easily reason about the concurrency properties of a Java program. An analysis similar to enclosing lock analysis is also performed, not to eliminate enclosed locks, but to reason about which objects might be subject to concurrent (unprotected) access. The analyses are intraprocedural, and thus are only applicable to small programs where all methods are inlined. The work does

not describe the analyses precisely, nor does it consider the potential performance improvements of removing unnecessary synchronization. Our work precisely describes a family of interprocedural analyses for removing unnecessary synchronization and provides an initial evaluation of their effects on set of benchmarks.

The Extended Static Checking System [DLN+98] allows a programmer to specify a locking protocol using code annotations. The program is then checked for simple errors such as deadlocks or race conditions. This system complements ours by focusing on the correctness of the source code, while our analyses increase the efficiency of the generated code.

7. Conclusion

This paper presented a set of interprocedural static analyses that effectively detect and eliminate unnecessary synchronization. These analyses identify excess synchronization operations due to reentrant locks, enclosed locks, and thread-local locks. A partial implementation of our analyses eliminates 30-70% of synchronization overhead on three Java benchmarks. Our optimizations support a style of programming in which synchronization code is written for software engineering objectives rather than hand-optimized for efficiency.

Acknowledgements

This work has been supported in part by a National Defense Science and Engineering Graduate Fellowship from the Department of Defense, NSF grant CCR-9503741, NSF Young Investigator Award CCR-9457767, and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software. We appreciate feedback and pointers to related work from David Grove, Martin Rinard, William Chan, Satoshi Matsuoka, members of the Vortex group, and the anonymous reviewers. We also thank the authors of our benchmarks: JavaSoft (*javac*), Philip Wadler (*pizza*), Andrew Appel (*jlex* and *javacup*), and Greg Badros (*cassowary*).

References

- [ALL89] T. E. Anderson, E. D. Lazowska and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. IEEE Transactions on Computers 38(12), December 1989, pp. 1631-1644.
- [BKM+98] D. Bacon, R. Konuru, C. Murthy, M. Serrano. Thin Locks: Featherweight Synchronization for Java. In Proceedings of the 1998 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.
- [BW88] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. Software Practice & Experience, September 1988, pp. 807-820.
- [C98] J. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. In Proceedings of the International Symposium on Software Testing and Analysis, March 1998. A more recent version is University of Hawaii ICS-TR-98-20, available at http://www.ics.hawaii.edu/~corbett/pubs.html.
- [DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In Proceedings of the Eleventh

Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1996.

- [DLN+98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Compaq SRC Research Report 159. 1998.
- [DR98] P. Diniz and M. Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-based Programs. In Journal of Parallel and Distributed Computing, Volume 49, Number 2, March 1998, pp. 218-244.
- [D97] J. Dolby. Automatic Inline Allocation of Objects. In Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1997.
- [GMS77] C. M. Geschke, J. H. Morris and E. H. Satterthwaite. Early Experiences with Mesa. Communications of the Association for Computing Machinery, 20(8), August 1977, pp. 540-553.
- [GJS96] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley, 1996.
- [GDD+97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In Proceedings of the 12th Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1997.
- [H91] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In Proceedings of the Sixth Conference on Object-Oriented Programming, Systems, Languages, and Applications, November 1991.
- [K93] D. Keppel. Tools and Techniques for Building Fast Portable Thread Packages. University of Washington Technical Report UW CSE 93-05-06, May 1993.
- [KP98] A. Krall and M. Probst. Monitors and Exceptions: How to implement Java efficiently. ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.
- [LR80] B. Lampson and D. Redell. Experience with Processes and Monitors in Mesa. In Communications of the Association for Computing Machinery 23(2), February 1980, pp. 105-117.
- [M96] N. Minsky. Towards Alias-Free Pointers. In Proceedings of the 10th European Conference on Object Oriented Programming, Linz, Austria July 1996.
- [NVP98] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In Proceedings of the 12th European Conference on Object Oriented Programming, Brussels, Belgium, July 1998.
- [PZC95] J. Plevyak, X. Zhang, and A. Chien. Obtaining Sequential Efficiency for Concurrent Object-Oriented Languages. In Proceedings of the 22nd Symposium on Principles of Programming Languages, San Francisco, CA, January 1995.
- [S88] Olin Shivers. Control-Flow Analysis in Scheme. SIGPLAN Notices, 23(7):164-174, July 1988. In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.
- [SNR+97] S. Singhal, B. Nguyen, R. Redpath, M. Fraenkel, and J. Nguyen. Building High-Performance Applications and Services in Java: An Experiential Study. IBM T.J. Watson Research Center white paper, available at http://www.ibm.com/java/education/javahipr.html. 1997.
- [SGA+98] E. G. Sirer, A. J. Gregory, N.R. Anderson, B.N. Bershad. Distributed Virtual Machines: A System Architecture for Network Computing. In Proceedings of the Eighth ACM SIGOPS European Workshop, September 1998.

Effective Synchronization Removal for Java

Erik Ruf Microsoft Research Redmond, WA 98025 erikruf@microsoft.com

Abstract

We present a new technique for removing unnecessary synchronization operations from statically compiled Java programs. Our approach improves upon current efforts based on escape analysis, as it can eliminate synchronization operations even on objects that escape their allocating threads. It makes use of a compact, equivalence-class-based representation that eliminates the need for fixed point operations during the analysis.

We describe and evaluate the performance of an implementation in the Marmot native Java compiler. For the benchmark programs examined, the optimization removes 100% of the dynamic synchronization operations in single-threaded programs, and 0-99% in multi-threaded programs, at a low cost in additional compilation time and code growth.

Introduction 1

The JavaTM programming language [GJS96] provides synchronization constructs (synchronized methods and blocks) to permit safe use of concurrently-accessed data structures. These constructs are used pervasively in both the standard libraries and the runtime system. In many cases, a large number of these operations may be safely removed without compromising program semantics, thus improving performance. Removing these operations manually may be inconvenient, error-prone or even impossible.¹

A dynamic synchronization operation in thread T on an object O is eliminable whenever no other thread T' attempts to synchronize O during the execution of the guarded code. Algorithms for automatic, static elimination of synchronization operations prove conservative approximations of this condition. Existing work falls broadly into two categories. One approach [BS96, FKR⁺00] proves that the program spawns no threads, making contention impossible and all

synchronization operations removable. This is both fast and effective, but has the disadvantages of having no effect on multithreaded programs and being unsound in the presence of notification operations.²

The second approach [ACSE99, Bla99, BH99, CGS⁺99, WR99] proves that the object O cannot escape its creating thread, and thus cannot be subject to contention. This approach fares poorly on programs (even single-threaded ones) where synchronized data structures are stored in static variables; the median synchronization removal ratio for singlethreaded programs in existing systems is below 55%. The compile-time costs of escape analysis can also be problematic: context-sensitive dependence-graph-based implementations such as [WR99] can take an hour or more to optimize programs of 10^4 statements [Rin99]. Some approaches improve performance at the cost of precision: [BH99] is context-insensitive and models only a single level of field dereferences, while [Bla99] blurs distinctions between sibling fields. [ACSE99] supplements escape analysis by eliminating synchronization operations that are always guarded by other synchronization operations. This promising extension is limited by cost/precision issues in an underlying pointer analysis, which does not scale up to programs such as javacup or javac. In general, it is difficult to assess the compile-time costs of escape-analysis-based implementations as only [Bla99] discloses analysis times.

We present a simple yet effective extension to the escape analysis approach, along with a high performance implementation technique. Our optimization achieves a superior degree of synchronization removal at a low cost in optimization time. It handles both synchronized methods and blocks, and preserves the Java synchronization, memory, and notification semantics. The distinguishing features of our approach are:

• Explicit modeling of inter-thread object flow. Instead of preserving all synchronization on escaping objects, our optimization finds cases where an object is synchronized only by a single thread (not necessarily its creating thread) during program execution, and eliminates synchronization for this case. This additional precision significantly improves the optimization in some cases, yet is obtained at little additional cost.

¹Most Java-to-bytecode translators implement the string concatenation primitive '+' via a call to a synchronized library method. Short of reimplementing strings, users cannot avoid this behavior.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. *PLDI 2000*, Vancouver, British Columbia, Canada.

Copyright 2000 ACM 1-58113-199-2/00/0006...\$5.00.

 $^{^{2}}$ Removing synchronization guarding a wait, notify, or notifyAll may cause the optimized program to throw a IllegalMonitorStateException not thrown in the original program. Such operations make little sense in single-threaded code, but may be present in code fragments or libraries also intended for multi-threaded use.

• An equivalence class based representation with polymorphic summaries. Our optimization models aliasing in a flow-insensitive manner by grouping potentially aliased expressions into equivalence classes, and models synchronization behavior as attributes of these equivalence classes. The representation is constructed in a single pass without fixed point operations, and enables context-sensitive analysis and specialization via a simple mapping function. It is sufficiently compact that large programs (10⁵ statements) can be optimized without depth limiting or other explicit abstraction of nonrecursive field access paths.

This paper describes our optimization and evaluates its utility on both single- and multi-threaded programs, as well as its costs in terms of compilation time and code expansion.

2 Overview and motivation

2.1 Explicit thread modeling

In contrast with escape based techniques, which preserve synchronization on objects reachable from global state (and thus visible to multiple threads), our goal is to preserve synchronization only on objects potentially synchronized by more than one thread instance during program execution.

This relatively coarse abstraction of the problem yields benefits in several cases. It allows for the removal of global synchronization in singly-executing threads, which can arise when "helper" threads (e.g., asynchronous I/O, user interface code, or watchdog timers) are added to an otherwise single-threaded program having static instances of threadsafe data structures. Another common case involves library abstractions that safely share internal data structures (e.g., buffer pools or graphics resources) via a static lock object, but are then used only by a single thread. In the important degenerate case of purely single-threaded programs, this abstraction renders all synchronization operations removable.

Making this finer distinction requires two extensions to escape analysis. First, it is necessary to track value flow through global state, rather than merely marking globallyreachable values as "escaped." In other words, an alias analysis, rather than an escape analysis, is required. This is a simple extension, since escape analyses already model complex value flow, including aliasing, for local state. Second, the analysis must be able to identify the thread instances in which particular escaping values are synchronized. In our implementation, this is accomplished via a straightforward call-closure analysis that bounds the code executed by thread instances.

A primary limitation of this approach is that it only proves properties that hold for an object's entire lifetime. Thus, it fails to recognize cases where contention is limited to particular program scopes (e.g., fork-join parallelism) or lock scopes (e.g., enclosing synchronization).

2.2 Equivalence class based representation

Like many other systems [Bla99, CGS⁺99, CmH00, WR99], our optimization achieves context sensitivity by constructing reusable, polymorphic method summaries that can be independently applied at multiple call sites. In most existing work, aliasing is modeled via directed dependence arcs encoding "points-to" relationships, while escape properties are formulated as reachability queries over the resulting dependence graphs.

In an effort to minimize analysis time and space usage, our optimization relies on an equivalence based representation, in which potentially aliased values are forced to share common representative nodes. This choice enables single pass flow-insensitive analysis of any unit of code without the need for iteration to model flow across back arcs (e.g., loops and recursion) in the program's flow graph.

The convenience and efficiency of this representation come at a cost in precision. Within methods, the directionality of flow is lost, resulting in false aliasing between multiple values assigned into a single variable or field. A similar problem arises with recursion, where we give up context sensitivity within a recursive component to avoid iterating the analysis.

3 Algorithm

3.1 Preliminaries

The algorithm is implemented in the Marmot native compilation system for Java [FKR⁺00]. Marmot implements most Java 1.1 semantics and libraries, but does not support dynamic loading and limits the use of reflection. The resulting "closed-world" assumption enables a number of wholeprogram analyses, including the construction of a static call graph (using Rapid Type Analysis [BS96] and intraprocedural type propagation) used by our optimization.

For purposes of this exposition, the Marmot intermediate representation can be viewed as a statically-typed threeaddress format with local variables in static single assignment (SSA) form [CFRW91]. Control operations other than return and throw are irrelevant as the analysis is flow insensitive. Both synchronized blocks and methods are implemented with explicit monitorEnter and monitorExit primitives.

3.2 Phase 1: Computing thread properties

The first optimization phase identifies thread allocation sites (including those in library code, plus an artificial site for the main thread) and computes two attributes for each site:

- the set of methods potentially executed by the thread being allocated, and
- whether the allocation site (and thus the thread(s) it allocates) can be executed more than once at runtime.

At each thread allocation site, the corresponding run method(s) are derived from the thread's type and class hierarchy information. In the special case of a site t = new Thread(r), where r is an instance of Runnable, we compute r's type via intraprocedural type propagation. In many cases, r is allocated in the current method, enabling r.run to be precisely identified. A call graph closure analysis finds all methods reachable from the run method(s) and and associates them with the allocation site.

A thread allocation site is marked as multiply executed if it is in a loop, is reachable from a non-class-initialization method having multiple or multiply-executed call sites, or is reachable from a **run** method associated with a multiply executed thread allocation site. An annotation mechanism allows sites to be declared as singly-executed; we use this in library code for sites that allocate multiple threads known not to be simultaneously live.
3.3 Phase 2: Building method summaries

The second optimization phase computes

- for each global value (reference constant or static field) and its (transitive) fields and array elements, the set of allocation sites of threads potentially synchronizing the value, and
- for each method, the alias and synchronization effects of the method and its (transitive) callees.

Alias sets The optimization represents runtime values with instances of the *alias set* data structure:

```
aliasSet ::= \perp \mid
(fieldMap, synchronized, syncThreads, global).
```

The \perp case indicates a nonreference value, while the tuple case describes a reference value. The tuple elements define properties of the value:

- *fieldMap*. A mapping from fully qualified instance field names to alias sets for the corresponding field values; the distinguished fieldname **\$ELT** denotes the contents of an array object.
- *synchronized*. A boolean, true if the value may be the target of a synchronization operation.
- *syncThreads.* For escaping values, a set of thread allocation sites representing the thread instances that may synchronize the value.
- global. A boolean, true if the value can be reached from a reference constant or static field (i.e., it escapes). If true, all alias sets reachable via *fieldMap* must also have global=true. This ensures that referents of an escaping object also escape.

Alias sets support a *unification* operation that merges two alias sets in place via a union-find data structure [ASU86]. The resulting alias set's attributes are the join of the input attributes under the function, boolean, set, and boolean lattices, respectively. In addition, joining the field maps causes alias sets corresponding to fieldnames present in the domains of both maps to be unified. The unifier is also responsible for noticing when a potentially synchronized value escapes. Thus, unifying a *global* alias set with a non-*global*, *synchronized* alias set causes the *sync Threads* attribute of the result to be augmented with the set of thread allocation sites associated with the current method.³

Another operation, *new instance creation*, allows the abstraction of the aliasing and synchronization properties of an alias set. New instance creation returns an alias set isomorphic to an existing one, in which only global alias sets are shared between the old and new instances.

Alias contexts The alias context data structure models the aliasing and synchronization behavior of parameter, normal result, and exception result values transmitted between call sites and methods. It is a tuple

aliasContext ::= $\langle \langle f_0, ..., f_n \rangle, r, e \rangle$

where f_i , r, and e are alias sets corresponding to the parameter, return, and exception values. Alias contexts are used to represent the information both for methods (in which case the f_i represent formal values received from the caller, and r and e represent values returned to the caller) and for call sites (in which case the f_i represent actual values transmitted to the callee, and r and e represent values returned by the callee). We call the former use a *method context* and the latter a *site context*.

Like alias sets, alias contexts support unification and new instance creation. Unification is the pointwise extension of alias set unification to tuples. The alias context returned by new instance creation preserves (recursively) all relationships between the original f_i , r, and e.

3.3.1 Interprocedural analysis

The interprocedural analysis associates each global value with an alias set and each method with a method context. It begins by binding each static field and object constant (e.g., string literal or statically allocated array) to a new alias set with *global=true*. It also constructs initial alias sets for compiler-generated runtime data structures whose initialization is not explicit in the intermediate code (e.g., class objects, interning and reflection tables, etc).

The analysis then partitions the static call graph into strongly connected components (SCCs) and traverses them in bottom-up topological order. Processing an SCC consists of creating an initial method context object for each method in the SCC, then applying the intraprocedural analysis to each of the SCC's methods individually.

3.3.2 Intraprocedural analysis

The intraprocedural analysis ensures that any aliasing or synchronization by the method and its callees is appropriately represented in the method's context and in global alias sets. It begins by associating each formal parameter variable with the corresponding formal alias set from the method context. It then walks the method's statements, unifying alias sets using the rules in Figure 1. Because local variables obey SSA invariants, our implementation saves time and space by binding locals to new alias sets lazily upon use, and implements assignments to unbound locals by updating the binding table instead of performing unification.

Only statements that modify reference variables or values are processed. Primitive operations that induce aliasing cause the alias sets of potentially aliased expressions to be unified. For example, the assignment $\mathbf{x} \cdot \mathbf{f} = \mathbf{y}$ (where \mathbf{x} and \mathbf{y} are local variables) causes the analysis to unify \mathbf{y} 's alias set with the alias set returned by x.fieldmap(f), where x is the alias set for \mathbf{x} . Similarly, analyzing throw \mathbf{z} unifies \mathbf{z} 's alias set with those of all relevant handlers (including the returned-exception value e of the method context if \mathbf{z} could be uncaught by the method).⁴

The synchronization operations monitorEnter and monitorExit set the synchronized property of their argument alias set. In addition, if the argument alias set is global, all thread allocation sites reaching the current method are added to the argument alias set's syncThreads property.

At method invocations, the analysis constructs a site context S whose formal, return, and exception alias sets

³In effect, the optimization records the fact that a thread executes a synchronization operation on a value V at the point where V escapes, not at the point where V is synchronized. Doing so improves precision because V may not escape all threads that synchronize it.

 $^{^4 \}rm We$ can safely ignore implicit exceptions from primitives, as these are always newly constructed, unaliased objects without reference fields.

Domains

$v \in V$	local variables
$g \in G$	global values (constants, static fields)
$f \in F$	field names
$a,r,e\in A$	alias sets
$mc, sc \in C$	method, site contexts
$m,p\in M$	$\mathrm{met}\mathrm{hods}$
$s \in S$	thread creation sites
$t \in T$	types

Analysis State

$GAS: G \rightarrow A$	alias set lookup for globals
$AS: V \to A$	alias set lookup for locals
$MC: M \to C$	method context lookup
$CALLEES: M \times V \to 2^M$	method target lookup
$SCC: M \rightarrow 2^M$	SCC lookup
$TC: M \rightarrow 2^S$	thread creation site lookup

Analysis Rules

statement	action
$\begin{array}{l} v_0 = v_1 \\ v_0 = (t)v_1 \end{array}$	$\mathit{unify}(AS(v_0),AS(v_1))$
$egin{array}{ll} v&=g\ g&=v \end{array}$	unify(AS(v), GAS(g))
$v_0 = v_1.f$ $v_1.f = v_0$	$unify(AS(v_0), AS(v_1).fieldmap(f))$
$v_0 = v_1[] \ v_1[] = v_0$	$unify(AS(v_0), AS(v_1).fieldmap(\$ELT))$
$v = \phi(v_0,, v_n)$	$orall v_i \; unify(AS(v), AS(v_i))$
$v = {\tt new} \; T$	no action
$\texttt{return} \ v$	$\mathit{unify}(AS(v),r)$
throw v	$\mathit{unify}(AS(v),e)$
monitorEnter v monitorExit v	$egin{aligned} AS(v). synchronized &= true \ if \ AS(v). global \ AS(v). sync \ Threads &= \ AS(v). sync \ Threads \cup \ TC(m) \end{aligned}$
$v = p(v_0,, v_n)$	$let \ sc = \langle \langle AS(v_0),, AS(v_n) \rangle, AS(v), e \rangle$ $\forall p_i \in CALLEES(p, v_0)$ $let \ mc = MC(p_i)$ $if \ SCC(m) \neq SCC(p_i)$ $let \ mc' = newInstance(mc)$ unify(sc, mc') else unify(sc, mc)

Figure 1: Intraprocedural analysis rules for relevant statement types. The rules assume that the statements being analyzed belong to a method m with method context $\langle \langle a_0, ..., a_n \rangle, r, e \rangle$, where the AS relation maps formal variables to the corresponding a_i . This description slightly oversimplifies the handling of exceptions and assignments to locals (see text).

correspond to the actual, result, and relevant exception alias sets at the call site. It then iterates over the methods invoked by the call site, performing one of the following two operations:

- 1. Nonrecursive target. The analysis computes a new instance M' of the method context M and unifies it with the site context S.⁵ This has the combined effect of (1) reflecting callee-side aliases to the call site, and (2) propagating callee-side properties to the call site. Creating a new instance each time a method is applied prevents the accumulation of call-site-specific information in the method context, allowing context-sensitive analysis.
- 2. Recursive target. In this case, the analysis unifies the method context M and site context S. While this introduces context insensitivity at recursive call sites, it has a large performance benefit in that the analysis does not need to iterate over the entire SCC until a fixed point is reached.⁶

After a method has been analyzed, the analysis drops the reference to the local variable mapping, allowing all alias sets not escaping the method's stack frame to be reclaimed. Subsequent phases requiring information about local variables reconstitute it by reexecuting the intraprocedural analysis.⁷

3.3.3 Example

Figure 2 shows part of a toy vector class and three of its clients immediately prior to synchronization optimization. We use a Java-like syntax for the intermediate code, in which virtual calls have been statically bound, and each statement executes a single operation. In addition, explicit monitorEnter, monitorExit, and catch operations are used to implement the synchronized method SimpleVector.elementAt and the synchronized block encircling the ellipsis in method test3. The results of the first analysis phase are shown as comments: we will assume that both T1 and T2 represent single-instance thread allocation sites.

The second phase begins by assigning a new alias set $\alpha_0 = \langle \{\}, false, \{\}, true \rangle$ to the static variable SimpleVector.v, and computes the bottom-up schedule <init>, elementAt, test0, test1, test2. The method context constructed for <init> is $\langle \langle \alpha_1 \rangle, \bot, \alpha_3 \rangle$, where $\alpha_1 =$ $\langle \{\text{elements} \rightarrow \alpha_2\}, false, \{\}, false \rangle$ and α_2 and α_3 have default attributes ($\langle \{\}, false, \{\}, false \rangle$). This context indicates that the formal parameter may have a field elements described by α_2 , and there is no return value. Neither the formal, any value reachable from it, nor any thrown exception can be synchronized by <init>.

 $^5 \rm Our$ implementation folds these operations into a single, parallel traversal of M and S.

⁶Given the relative imprecision of the RTA based call graph, SCCs can sometimes be quite large (e.g., most toString methods end up in a single SCC). Iteration is further complicated by the size of the alias context data structures, and because convergence is not guaranteed (e.g., the "add to head of linked list" method will grow its list argument on each iteration). We experimented with an adaptive, iteration-based scheme that could degenerate into the direct-unification scheme described above. In most cases, the space bounds were violated before convergence was achieved, so little to no additional precision was obtained.

 $^{^7\,{\}rm Be}{\rm cause}$ all callee method contexts, even for recursive callees, are complete at reconstitution time, the nonrecursive strategy (item 1 above) is always used.

```
class SimpleVector {
  Object[] elements;
  static SimpleVector v;
  /* invoked by T1, T2 */
  static void <init>(SimpleVector this1) {
    Object[] temp = new Object[10];
    this1 elements = temp;
  }
  /* invoked by T1, T2 */
  static Object elementAt(SimpleVector this2,
                          int index) {
    monitorEnter(this2)
    try {
      Object[] elts = this2.elements;
      Object elt = elts[index];
      monitorExit(this2);
      return elt;
    catch (Throwable t) {
      monitorExit(this2);
      throw t:
    }
 }
}
/* invoked by T1 */
static void test1() {
  SimpleVector v1 = new SimpleVector;
  SimpleVector.<init>(v1);
  Object o1 = SimpleVector.elementAt(v1, 0);
}
/* invoked by T1 */
static void test2() {
  SimpleVector v2 = new SimpleVector;
  SimpleVector.<init>(v2);
  Object o2 = SimpleVector.elementAt(v2, 0);
  SimpleVector.v = v2;
3
/* invoked by T2 */
static void test3() {
  SimpleVector v3 = SimpleVector.v;
  Object o3 = SimpleVector.elementAt(v3, 0);
  monitorEnter(o3);
  try {
    monitorExit(o3);
    return;
  }
  catch (Throwable t) {
    monitorExit(o3);
    throw t;
  }
}
```

Figure 2: Example program fragments

The method context for elementAt is $\langle \langle \alpha_4, \perp \rangle, \alpha_6, \alpha_7 \rangle$, where $\alpha_4 = \langle \{\text{elements} \rightarrow \alpha_5\}, true, \{\}, false \rangle, \alpha_5 = \langle \{\text{\$ELT} \rightarrow \alpha_6\}, false, \{\}, false \rangle$, and α_6 and α_7 have default attributes. In this case, the first parameter may be synchronized, and the contents of its $\verb+elements$ array may be returned.

The intraprocedural analysis on test1 finds that the value of variable v1 may be synchronized, but does not escape either into either test1's method context or a global alias set. Analyzing the first three statements of test2 yield a similar configuration of locals, with v2 bound to $\alpha_8 = \langle \{\texttt{elements} \rightarrow \alpha_9 \}, true, \{\}, false \rangle$, $\alpha_9 = \langle \{ \text{\$ELT} \rightarrow \alpha_{10} \}, false, \{ \}, false \rangle, \text{ and o 2 bound to} \rangle$ α_{10} , where α_{10} has default attributes. The assignment SimpleVector.v = v2 unifies α_8 with α_0 , producing (due to the unification of global and a nonglobal alias sets) the alias set $\alpha_0 = \alpha_8 = \langle \{\texttt{elements} \rightarrow \alpha_9 \}, true, \{\texttt{T1}\}, true \rangle$ where $\alpha_9 = \langle \{ \text{\$ELT} \rightarrow \alpha_{10} \}, false, \{ \}, true \rangle$ and $\alpha_{10} =$ $\{ \{ \}, false, \{ \}, true \}$. At this point, we know that v and v2 may be aliases holding a value that escapes and is synchronized by a thread allocated at site T1, and that the value in o2 escapes but is not synchronized.

The analysis of test3 binds v3 to α_0 . The application of elementAt marks α_0 as synchronized under the thread allocated at T2 and binds the variable o3 to α_{10} . The synchronization of o3 causes α_{10} to be marked as synchronized, but only by T2. At the end of phase 2, the method contexts for <init> and elementAt are as given above, while the alias set for SimpleVector.v, v2, and v3 is $\alpha_0 = \langle \{\text{elements} \rightarrow \alpha_9\}, true, \{\text{T1}, \text{T2}\}, true \rangle$, where $\alpha_9 = \langle \{\text{$ELT} \rightarrow \alpha_{10}\}, false, \{\}, true \rangle$, and $\alpha_{10} = \langle \{1, true, \{\text{T2}\}, true \rangle$.

3.4 Phase 3: Specialization and transformation

The third optimization phase propagates synchronization information from call sites to callees, and uses this information to remove or simplify synchronization operations in callees. It also constructs specialized versions of methods where different call sites allow distinct simplifications.

3.4.1 Interprocedural analysis

The interprocedural analysis processes SCCs in a top-down topological order while maintaining per-SCC queues of specialization requests (in the form of $\langle method, methodContext \rangle$ pairs). The analysis iteratively executes the intraprocedural analysis over all specialization requests for methods in a given SCC until all have been satisfied.

3.4.2 Intraprocedural analysis

The intraprocedural analysis both optimizes the method body (removing or simplifying synchronization operations and redirecting calls to specialized targets) and requests the creation of specialized method bodies. Given a (method, methodContext) pair, the analysis begins by executing the intraprocedural analysis of Section 3.3.2, associating each local variable with an alias set. It then walks the method's statements, rewriting synchronization operations and calls as follows.

• Synchronization operations. An alias set is said to be *contention free* if its *syncThreads* set is empty or contains a single thread allocation site that executes at most once. Given a statement of the form monitorEnter(o) or monitorExit(o), where o has alias set o, the analysis checks to see if o is contention free. If so, it removes the statement and, if the program is multi-threaded (i.e., the analysis found a nonartificial thread allocation site), inserts a memory barrier primitive so that later optimizations will obey the Java memory semantics at this point.

• Call sites. Given a call statement, the analysis constructs a site summary S from the actual, return, and reachable exception handler alias sets. For each target method with method context M, it constructs a new instance M' of M and then walks M' and S in parallel; for each alias set m' in M' that is synchronized, the syncThreadSet attribute of the corresponding alias set s is added to the syncThreadSet attribute of m'.⁸ The updated M' is then compared with both M and the method contexts of all existing or pending specializations of the target method, under the condition that two alias sets match if their contention free status is the same. If no match is found, the method is cloned and a request to specialize the cloned method on M' is enqueued. If M' does not match M, the call is rewritten to invoke the appropriate specialized method.⁹

Marmot's intermediate representation is constructed from the Java bytecode, which uses explicit synchronization operations to implement synchronized blocks. Because bytecode verification does not prove any invariants about the use of these operations, it is up to the optimizer to find correlated groups of monitorEnter and monitorExit operations to remove.

Enter/exit correspondences that do not span method boundaries are easily handled by our optimization. Within a method, all potentially aliased objects have identical *sync*-*Threads* attributes, ensuring that all synchronization operations on a particular object will be preserved or eliminated as a whole. All of our benchmarks (and, presumably, all bytecode generated by reasonable Java front ends) have only intra-method enter/exit correspondences.

Correspondences that span multiple procedures are more difficult, as removing or preserving a synchronization operation in one method may require the removal or preservation of one or more corresponding operations in another method. Our specialization strategy handles this by aggressively specializing callees with respect to the contention status of values at call sites, ensuring that caller and (specialized) callee methods will always agree on the removal/preservation choice for any given runtime value. Less aggressive specialization strategies (in which contexts inducing differing contention properties can share a common specialization) must place additional restrictions on synchronization removal.

3.4.3 Example

We continue the example of Section 3.3.3 into the final transformation of the optimization. This phase makes no changes to test1, as the syncThreads attribute of v1's alias set (and its elements) matches that of this1 and this2's alias sets. The same is true for the invocation of <init> in test2. Since the syncThreads attribute of v2's alias set denotes multiple threads and the corresponding alias set in elementAt's context does not, test2's call to elementAt is rebound to a clone, elementAt2, with context $\langle \langle \alpha_{11}, \perp \rangle, \alpha_{13}, \alpha_{14} \rangle$, where $\alpha_{11} = \langle \{\text{elements} \rightarrow \alpha_{12}\}, true, \{\text{T1}, \text{T2}\}, false \rangle, \alpha_{12} = \langle \{\text{$ELT} \rightarrow \alpha_{13}\}, false, \{\}, false \rangle$, and α_{13} and α_{14} have default attributes. In other words, elementAt2 is a specialization of elementAt that preserves synchronization behavior on the formal parameter this2.

The call to elementAt in test3 is also retargeted to elementAt2. Local o3 is found to have the alias set $\alpha_{10} = \langle \{\}, true, \{T2\}, true \rangle$, which is synchronized, but only by a singleton thread. This means that all three synchronization operations on o3 are eliminable, so they are replaced by memory barrier primitives.

The *init* method is not processed because it has neither synchronization operations nor callees. Processing of *elementAt* finds that *this2* cannot be synchronized (recall that both invocations that passed synchronized arguments were redirected to *elementAt2*), and successfully replaces the synchronization operations on *this2* with barriers. The alias set α_{11} in the context for *elementAt2* is synchronized by two threads, causing all three synchronization operations to be preserved.

3.5 Other issues

3.5.1 Complexity

The worst-case time/space complexity of the optimization is at least exponential in program size. A method m_1 returning a new pair, both of whose arms point to the methods's argument, will have a return alias set with field map $\{\texttt{left} \rightarrow \alpha, \texttt{right} \rightarrow \alpha\}$ where α is the formal alias set. A method m_2 containing a cascade of k calls to m_1 can construct a formal alias set with a field map of size 2^k .

That said, few programs construct large recursive data structures without the use of iteration or recursion. Given that the analysis does not explore recursive paths in control flow graphs or the call graph, exponential cascades of the sort described above are rare. The method-local nature of many objects also limits duplication, as such objects do not contribute to method summaries. In practice, optimization costs are greater than linear in program size but remain manageable (> 7500 stmts/sec) even for our largest benchmarks.

3.5.2 Event notification operations

The Java threading model supports event notification via the Object.wait, Object.notify, and Object.notifyAll methods, all of which require that their this argument be locked (otherwise an exception is thrown). Preserving this behavior in the face of synchronization elimination requires some additional effort.

When a notification method is invoked on an object, a boolean *notified* attribute in the object's alias set is set to true. When the analysis finds an otherwise removable synchronization operation whose alias set has *notified=true*, it

⁸There is no need to transfer aliasing information from caller to callee, (e.g., by unifying site and method contexts) since all potentially-aliased caller-side expressions will have identical alias sets.

⁹Indirect calls require additional effort, as the call must invoke the specialized clone only for a subset of the receiver objects arriving at runtime. To handle this, new selectors (method names) are introduced at the appropriate points in the class hierarchy; these tail-call the appropriate clones with identical arguments. Such "trampolines" allow specializations to be shared at the cost of additional direct call operations. This overhead is later eliminated by the Marmot code generator, which "inlines" the tail calls into the dispatch tables and removes the trampoline method bodies.

replaces the operation with a specialized version that performs enough bookkeeping to satisfy the notification methods, without actually performing any machine-level synchronization operations.¹⁰

3.5.3 Object cloning

The method Java.lang.Object.clone returns a new object whose reference fields are aliased to the corresponding fields in the original. Representing this using the scheme described above is difficult because the analysis may add fields to the argument object long after the application of clone has been processed. Explicitly constructing aliases for all possible fields would be impractical.

Instead, we move the *fieldMap* attribute of the alias set data structure into a separate *contents* object that supports unification and new instance creation. Field and array element operations on alias sets are delegated to the contents object, while unify/new instance operations are performed recursively on the contents object. The Object.clone method can then be given a special method context in which the contents objects of the argument and return values are aliased, but the values themselves are not. This avoids false aliasing of the base and clone objects, but is still imprecise on field values that are immediately, strongly updated by a subclass's clone method.

3.5.4 Indirect synchronization removal

For a restricted case, our optimization is able to remove synchronization operations on objects subject to contention by multiple threads. In the Marmot runtime, an object's lock and hashcode data are stored in a corresponding, dynamically created extension object. The object extension operation must synchronize on a global lock, rather than on the object being extended, as the object's lock is not yet created.

The analysis described above does not eliminate extension synchronization in multithreaded programs because the object being synchronized (the global lock) is indeed synchronized by multiple threads. We extend the analysis by adding the alias set attributes *extended* and *extThreads*, which mirror *synchronized* and *syncThreads*, but track extension events rather than synchronization events. Extension operations on objects with contention-free *extThreads* sets are redirected to a version that does not perform synchronization.

3.5.5 Single-threaded programs

The thread-allocation-site analysis described in Section 3.2 declares a program single-threaded when it is unable to locate any thread construction sites other than that for the main thread. This knowledge allows our algorithm to avoid the insertion of memory barrier operations. It also enables the use of a garbage collector and runtime system customized for the single-threaded case.

3.5.6 Performance improvements

We lower the optimization's compile time costs by avoiding work that cannot enable the removal of synchronization

```
// a. original implementation
void f(Object obj)
  if (obj == null) {
    obj = \langle default \rangle;
  }
}
// b. modified implementation
void f(Object obj2)
  if (obj2 == null) {
    f2(\langle default \rangle);
  } else {
    f2(obj2);
}
void f2(obj) {
  . . .
}
```

Figure 3: Rewriting a method to avoid aliasing the parameter obj with the global-valued expression $\langle default \rangle$.

operations. During the second phase, we identify methods that cannot (transitively) execute synchronization operations. Such methods will never require removal of synchronization operations or retargeting of call sites, and thus can be ignored in the transformation phase. This optimization reduces costs by as much as 50%.

Another optimization lowers memory usage and reduces unification, comparison, and new instance costs by compressing method contexts. An alias set can be removed from a context if (1) it is not synchronized, (2) it is not global, (3) it only appears once in the context, and (4) all of its fields are removable. Restrictions (2) and (3) ensure that aliases are propagated from callees to callers. While the additional context traversal required by compression can increase costs on our smaller benchmarks, it reduces optimization times by as much as 30% on larger ones.

3.5.7 Avoiding false aliasing

Figure 3(a) shows source code for a common Java idiom in which a null formal parameter value is replaced with a default value prior to the execution of a method body. Our optimization assigns a common alias set to the variable obj and the expression $\langle default \rangle$. If $\langle default \rangle$ denotes a global value, the method signature for f will be marked as global. Since globals are modeled monomorphically, the alias sets of the actual parameters at all of f's call sites will be unified even though f induces no callee-side aliasing. In this case, the (otherwise convenient) bidirectional nature of unificationbased flow is problematic.

If the identity of the default value doesn't matter, the programmer can avoid this problem by constructing new default values (e.g., via new or cloning) as necessary. If identity does matter, or construction is too expensive, one can use the strategy of Figure 3(b). Binding obj via parameter passing instead of assignment keeps $\langle default \rangle$'s alias set out of the contexts of both f and f2, avoiding undesirable aliasing at call sites invoking f. For the dual case in which a global value is returned, only the new/clone approach can be used. The Marmot library uses these approaches in meth-

¹⁰The Jalapeno system [CGS⁺99] performs a similar optimization dynamically by predicating machine-level synchronization primitives on a bit in the lock object.

name	methods	stmts	dyn syncs	sync ovhd	description
javac	1,877	40,758	1.693E + 7	15.62%	javac compiling jlex 4 times
javacup	859	$21,\!657$	5.926E + 5	5.19%	javacup generating Java parser
jess	1,339	26,172	4.797E + 6	5.97%	expert system shell
jlex100	536	$15,\!698$	1.665E + 8	57.66%	jlex generating lexer for sample.lex, 100 times
marmot	8,193	$211,\!332$	1.172E + 8	10.33%	compile javac to native code
mtrt	716	16,500	7.486E + 5	1.49%	multithreaded ray tracer
multimarmot	8,225	212,160	1.183E + 8	9.99%	multithreaded compile of javac
plasma	1,038	17,857	4.159E + 4	0.01%	constrained plasma field simulation/visualization
slice	1,059	$18,\!697$	1.388E + 4	0.02%	viewer for 2D slices of 3D radiology data
volano	741	$13,\!085$	$4.623E{+7}$	5.52%	chat room simulator

Figure 4: Benchmark programs. Method and statement counts were performed on the intermediate form just prior to application of the synchronization optimization algorithm.

name	sync operations			
	original	opt		
		complete	partial	
javac	1.693E+7	0	3,740	
javacup	5.926E + 5	0	0	
jess	4.797E+6	0	0	
jlex100	1.665E + 8	0	0	
marmot	1.172E + 8	0	0	
mtrt	7.486E + 5	948	0	
multimarmot	1.183E + 8	7.810E + 7	0	
plasma	4.159E + 4	3,188	0	
slice	1.388E + 4	8,664	0	
volano	4.623E + 7	4.610E + 5	0	

Figure 5: Dynamic synchronization measurements.

ods of the String and StringBuffer classes when the null value is replaced by the string "null". It also returns clones of string literals in some contexts where returning a single value causes undesirable aliasing.

4 Results

4.1 Benchmark programs

We tested our algorithm on five single-threaded and five multi-threaded programs, described in Figure 4. Most of these programs are well known. Marmot is the bytecode-to-native-code compiler described in $[FKR^+00]$, while multiMarmot is a version of marmot reconfigured to perform per-method optimizations (amounting to approximately 25% of total compilation time) in two parallel threads. Plasma and slice are modified versions of publicdomain applet code.¹¹ Volano is the VolanoMarkTM 1.0 networking benchmark; we optimized both the client and server but report results only for the client.

The method and statement counts were performed after unreachable methods (in both the benchmark program and the libraries) were removed by a "treeshake" pass. The "synchronization overhead" column approximates the fraction of execution time spent performing synchronization operations in the unoptimized program. We computed this value by measuring the average cost of an executing an empty synchronized block (7.5E-8 seconds, or 58 machine cycles), multiplying it by the number of dynamic synchronization operations, and dividing by the unoptimized exe-



Figure 6: Fraction of synchronization operations removed

cution time.¹² Interestingly, the single-threaded programs execute far more synchronization operations as a function of running time than the multi-threaded programs do.

Testing was performed on a dual-processor 770Mhz Intel Pentium III workstation with 512MB of memory under Windows 2000 Professional. All results are the mean of multiple executions; standard deviations were nominal.

4.2 Synchronization removal

Figure 5 shows dynamic synchronization counts for the original and optimized versions of the benchmark applications. The "partial" category refers to operations that were only partially removed to preserve the notification semantics (c.f., section 3.5.2); only javac had removals of this sort.

Figure 6 presents the fraction of synchronizations removed in each of three scenarios. The leftmost column of each bar represents our optimization with all methods treated as executing in all threads, restricting removals to those enabled by escape analysis. As an escape analysis, our system is roughly comparable to existing work, except on javac, where it does much better, and jess, where it fails almost completely due to an imprecision in the call graph causing false aliasing with a static. The central column rep-

¹¹Available from the author.

 $^{^{12}}$ This figure overestimates the cost of recursive synchronization (no machine level lock is required) and underestimates the cost of initial synchronization (a lock object must be allocated) and contention (queue operations are required). The estimate does not account for secondary effects due to caches, missed optimizations, etc.

name	execution time				
	original	opt	gcopt		
javac	8.13	6.44	5.97		
javacup	0.86	0.76	0.66		
jess	6.03	5.63	5.12		
jlex100	21.66	8.37	8.09		
marmot	85.10	71.35	61.83		
mtrt	3.78	3.73	3.73		
multimarmot	88.85	80.67	80.67		
plasma	22.41	22.51	22.51		
slice	4.64	4.64	4.64		
volano	6.29	6.29	6.29		

Figure 7: Execution time measurements (user+kernel time in seconds).



resents the optimization with thread information enabled. This version achieved 100% elimination in single threaded code and improvements over our escape analysis in plasma and slice.

The rightmost column represents a rough upper bound on the degree of synchronization removal possible using techniques that prove an object to be synchronizable by at most one thread during the object's lifetime. We computed this value by instrumenting the library to count the number of objects synchronized by more than one thread during execution, and assuming that all other synchronizations were removable. In mtrt, the optimization improved upon the bound because some synchronization operations referencing multiply-synchronized objects were found to be removable (c.f., section 3.5.4). In multimarmot, worker threads performing per-method optimizations never contend for permethod data, but since that data is reached from a shared symbol table, a large number of unnecessary synchronization operations are preserved. All three scenarios fare poorly on volano, where 98% of the synchronization takes place on BufferedInputStream objects that are synchronized by multiple threads. 13

4.3 Execution time

Figure 7 presents execution times for unoptimized and optimized versions of the benchmark programs. For programs found to be single-threaded by our analysis, we examined two strategies. The first performs synchronization elimination only, while the second passes a threading flag to the code generator and runtime system, enabling the use of memory allocation and collection primitives specialized for the single-threaded case.

In Figure 8, each speedup result is divided into three segments. The lower segment represents an estimated speedup computed from the measured synchronization counts and the average-case synchronization cost described in Section 4.1. Together, the lower two segments represent the speedup measured when our optimization is applied. This value exceeds the estimate because synchronization removal enables additional optimization.¹⁴ The sum of all three segments is the speedup measured when the optimization and the single-threaded flag are enabled. In cases where nontrivial speedup is achieved, the additional optimizations account for a significant fraction of the improvement (the majority of the improvement in 4 of 6 cases).

Other than multimarmot, which improved by 10%, the multi-threaded benchmarks did not become faster as a result of synchronization removal. Mtrt performs all of its synchronization as part of loading its data file, which represents a small fraction of the overall computation. All of the synchronization in plasma and slice occurs in the AWT libraries; the inner loops of the applets are floating point computations that do not perform synchronization. No performance improvement was obtained on volano, as very few synchronization operations were removed.

4.4 Static costs

Figure 9 presents various static measures of our optimization. The absolute costs of the synchronization analysis were quite low (seconds), and represented only a small fraction of overall compilation time.¹⁵ At the same time, by shrinking method sizes (removing synchronization code) and increasing method counts (generating specialized methods), the optimization significantly altered the costs of subsequent phases of the Marmot optimizing compiler. Overall compilation times fell by 79% in javacup, but rose by 20% in multimarmot. With the exception of javac, which contains notification operations, the single-threaded programs did not require specialization. For multithreaded programs, the average number of specializations per method ranged from .08 (mtrt) to .20 (volano).

The optimization's effect on the amount of code generated¹⁶ varied greatly. In some cases, the removal of synchronization code (which Marmot always inlines) more than compensated for the addition of specialized methods and any

¹³An analysis tracking relationships between locks may be able to remove these synchronizations, which appear to be guarded by an escaping, but less frequently synchronized, DataInputStream object.

¹⁴In the single-threaded case, memory barriers are eliminated, enabling a small amount of additional load caching. Most of the benefit comes from additional inlining made possible (under Marmot's sizebased heuristics) by reductions in method sizes when synchronization code is removed.

¹⁵It is worth noting that the analysis allocates a large amount of storage while analyzing a method, much of which becomes dead when the method summary is constructed. Not surprisingly, the optimization performs better under Marmot's generational garbage collector than under its copying collector.

 $^{^{16} \}rm The$ "code growth" column in Figure 9 refers only to executable code generated for the user program and Java libraries. It does not include C or assembly runtime code, static data, or static metadata.

name	opt	frac of	comp	specs	tramps	code
	time	comp	time			growth
	(sec)	time	change			
javac	4.17	6.75%	3.45%	3	0	9.88%
javacup	1.01	2.75%	-79.15%	0	0	-21.76%
jess	1.59	4.55%	-14.05%	0	0	6.82%
jlex100	0.56	3.28%	-8.79%	0	0	-1.03%
marmot	22.00	5.41%	12.99%	0	0	20.35%
mtrt	0.86	3.85%	2.73%	58	49	-0.93%
multimarmot	28.03	6.46%	20.38%	1198	775	4.32%
plasma	1.11	4.04%	10.59%	132	124	8.73%
slice	1.16	3.90%	18.35%	152	124	12.43%
volano	0.73	3.91%	10.69%	148	86	7.8%

Figure 9: Static statistics. Optimization time includes the cost of call graph construction.

additional inlining enabled by method size decreases. For the single-threaded programs, almost all of the code size increase is attributable to the inlining of allocation operations under the single-threaded storage management regime.

5 Related work

This section describes work not addressed in the introduction or in the text.

5.1 Synchronization optimizations

[DR96, DR97] describe schemes for aggregating multiple critical regions guarded by the same lock into a single, larger critical region, and for replacing multiple lock objects with a single lock that guards all of the subobjects' operations. These techniques reduce the number of lock operations performed at the risk of reducing parallelism due to coarser lock granularity. [Tse95] automatically restructures parallel programs to replace barrier synchronization with less expensive operations, or to remove it entirely. In both cases, the transformations were developed for a particular style of thread synchronization produced by a parallelizing compiler, so it is not clear that they are sound for general monitor synchronization as in Java.

Another way to reduce the runtime cost of synchronization operations is to implement them more efficiently. The IBM "thin locks" work [BKMS98] and the Marmot lock implementation [FKR⁺00] are examples of fast locking mechanisms.

5.2 Related analyses

The construction of abstract summary functions for use in interprocedural analysis dates back at least to the "functional approach" of [SP81]. Alias analysis based on equivalence classes and unification was introduced in [Ste96b, Ste96a]. Recent work in the context of summarybased pointer analysis includes [CRL99, CmH00] and the summary-based escape analyses [Bla99, CGS⁺99, WR99] discussed in the introduction. [FRD00] explores a combination of equivalence-class-based analysis and procedure summaries that supports higher-order procedures.

6 Future work

While our optimization did very well on single-threaded benchmarks and had some success in the multithreaded case, there is much work to be done for multithreaded programs. Our optimization treats all threads as though they run for the duration of program execution, while many programs (including multimarmot) use fork/join strategies in which thread lifetimes are far shorter. This suggests the pursuit of more temporally sensitive strategies. Another open issue is the treatment of threads themselves; all existing analysis, including ours, treat all data reachable from thread objects as escaping. More powerful techniques are required to show that some state held in instance variables of threads remains unaliased.

Our flow analysis is fragile in the presence of cycles in the call graph (e.g., jess). Possible improvements include enhancing the Marmot static call graph analysis via context sensitive techniques or modeling of polymorphic data structures. Alternatively, we could avoid the use of a static call graph by encoding method dispatch into types and using the instantiation constraint based flow analysis technique of [FRD00]. A third option would allow limited use of sets of *aliasSet* to represent values in cases where unification yields false aliases [SH97].

The high speed of our analysis opens several opportunities. One possibility is to use the analysis as a preprocessing phase to reduce the cost of a more precise model. Another is an iterative, pessimistic call graph optimization in the style of [HH98], in which the analysis is used to model class sets, which are used to improve the call graph, enabling reanalysis, etc., until convergence is achieved.

We plan to apply equivalence-class-based summarization techniques to other interprocedural problems such as stack allocation, memory disambiguation, and type propagation.

Finally, we believe it is important to continue the search for and the development of additional, more realistic multithreaded programs for use in the design and testing of optimizations.

7 Conclusion

We have described an effective, efficient technique for statically removing unnecessary synchronization operations from Java programs. The distinguishing features of this approach are

• the use of thread closure and alias analyses rather than escape analysis, enabling more precise modeling of value flow in the face of global variables and multiple threads, and • the use of equivalence class based method summaries, enabling simple, fast, non-fixed-point, context-sensitive analysis and transformation.

Our optimization handles both synchronized methods and blocks, and preserves the Java synchronization, memory, and notification semantics. Our experiments, performed in the context of an optimizing compiler, demonstrate improvements in dynamic synchronization counts and execution time in both single- and multi-threaded programs, at a reasonable cost in compilation time and code growth.

Acknowledgments

We thank the members of the Advanced Programming Languages group at Microsoft Research for their support of the Marmot compiler infrastructure, Bjarne Steensgaard for finding unifier bugs, and the anonymous referees for their suggestions.

References

- [ACSE99] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In SAS'99, LNCS. Springer-Verlag, September 1999.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA, 1986.
- [BH99] J. Bogda and U. Hölzle. Removing unnecessary synchronizations in Java. In Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), November 1999.
- [BKMS98] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 258-268, June 1998.
- [Bla99] B. Blanchet. Escape analysis for object oriented languages. application to Java. In Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), November 1999.
- [BS96] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings OOP-SLA '96, ACM SIGPLAN Notices*, pages 324-341, October 1996. Published as Proceedings OOPSLA '96, ACM SIGPLAN Notices, volume 31, number 10.
- [CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991.
- [CGS⁺99] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99), November 1999.
- [CmH00] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation, June 2000.

- [CRL99] R. Chatterjee, B. G. Rynder, and W. A. Landi. Relevant context inference. In Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 133-146, January 1999.
- [DR96] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized objectbased programs. In Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, LNCS 1239, pages 285-299, August 1996.
- [DR97] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 187-200, 1997.
- [FKR⁺00] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. Software: Practice and Experience, 30(3):199-232, March 2000.
- [FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation, 2000.
- [GJS96] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.
- [HH98] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 97-105, June 1998.
- [Rin99] M. Rinard. Personal communication. 1999.
- [SH97] M. Shapiro and S. Horwitz. Fast and accurate flowinsensitive points-to analysis. In Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 1-14, January 1997.
- [SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In Program Flow Analysis: Theory and Applications, chapter 7, pages 189–284. Prentice-Hall, 1981.
- [Ste96a] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In International Conference on Compiler Construction, number 1060 in Lecture Notes in Computer Science, pages 136-150, April 1996.
- [Ste96b] B. Steensgaard. Points-to analysis in almost linear time. In Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 32-41, January 1996.
- [Tse95] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pages 144–155, July 1995.
- [WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOP-SLA '99), November 1999.

Encapsulating Objects with Confined Types

Christian Grothoff Jens Palsberg Jan Vitek S³ Lab, Department of Computer Sciences, Purdue University

 $\{grothoff, palsberg, jv\}@cs.purdue.edu$

ABSTRACT

Object-oriented languages provide little support for encapsulating objects. Reference semantics allows objects to escape their defining scope. The pervasive aliasing that ensues remains a major source of software defects. This paper introduces Kacheck/J a tool for inferring object encapsulation properties in large Java programs. Our goal is to develop practical tools to assist software engineers, thus we focus on simple and scalable techniques. Kacheck/J is able to infer *confinement* for Java classes. A class and its subclasses are confined if all of their instances are encapsulated in their defining package. This simple property can be used to identify accidental leaks of sensitive objects. The analysis is scalable and efficient; Kacheck/J is able to infer confinement on a corpus of 46,000 classes (115 MB) in 6 minutes.

1. INTRODUCTION

Object-oriented languages rely on reference semantics to allow sharing of objects. Sharing occurs when an object is accessible to different clients; an object is aliased when it is accessible from the same client through different access paths. Sharing is both a powerful tool and a source of subtle program defects. A potential consequence of aliasing is that methods invoked on an object may depend on each other in a manner not anticipated by designers of those objects, and updates in one sub-system can affect apparently unrelated sub-systems, undermining the reliability of the program.

While object-oriented languages provide linguistic support for protecting access to variables, methods, and even entire classes, they fail to provide any systematic way of protecting objects. A class may well declare some variable private and yet return the contents of that variable from a public method. In other words, object-oriented languages protect the state of individual objects, but cannot guarantee the integrity of systems of interacting objects. They lack a notion of an *encapsulation boundary* that would ensure that references to 'protected' objects do not escape. The goal of this paper is to experiment with pragmatic notions of encapsulation in order to provide software engineers with tools to guide them in the design of robust systems. To this end, we focus on simple models of encapsulation that can easily be understood. We deliberately ignore more powerful escape analyses [2, 3, 9] which are sensitive to small source code changes and return results that may be difficult to interpret. Of course, the tradeoff is that our analysis will sometime deem an object as 'escaping' when a more precise analysis would discover that this is not the case.

We have chosen to investigate *confined types* [5] as they give rise to a form of encapsulation that is both simple to understand and that can be checked with little cost. The basic idea underlying confined types is the following:

Objects of a confined type are encapsulated in their defining package.

Thus, if a class is confined, instances of that class and all of its subclasses cannot be manipulated by code belonging to other packages. In terms of aliasing, confinement allows aliases within a package but prevents them from spreading to other packages as illustrated graphically in Figure 1.





The definition of confinement in [5] requires explicit annotations and thus pre-supposes that software is designed with confinement in mind. In this work we take a different approach: Kacheck/J infers confinement in existing Java packages. We begin with the following controversial thesis:

Thesis: All package-scoped classes in Java programs should be confined.

Furthermore, we show that a majority of large Java applications were written such that confinement would hold for package-scoped classes. In other words, confinement is a natural property to expect of package-scoped classes and one that should be enforced by compilers. To validate our hypothesis we gathered a large number of Java programs (46,000 class files—to the best of our knowledge the largest such benchmark suite) and implemented the Kacheck/J tool to infer confinement properties of these classes. The results of our analysis show that without any change to source programs, 3,998 classes (or 25% of the package-scoped classes) are confined. Furthermore, we found that if one adds features to Java that address the lack of generic container types, then the number of confined types can be increased to over 4,800. Finally, we were surprised to discover that with appropriate tool support, the number of confined classes can be well over 14,500 for that same benchmark suite (or 32%of all classes). Even though we can agree that there are valid uses of package-scoped classes that break confinement, we feel that these uses should be flagged and treated specially rather than the converse.

While more powerful program analysis may yield higher numbers of confined classes, especially if a whole-program approach is taken, our current numbers are already surprisingly high. Another pleasant surprise is that these results can be obtained efficiently. The average time to analyze a class file is less than 8 ms (or about 350 s for the whole benchmark suite) for a tool entirely written in Java running on stock hardware.

The contributions of this paper are:

- 1. A simpler and less restrictive set of confinement rules than in [5] (Section 3).
- 2. A constraint-based confinement analysis (Section 4).
- 3. A presentation of the Kacheck/J confinement inference tool (Section 5).
- 4. Confinement results for a large-scale benchmark suite (Section 6).
- 5. A discussion of refactorings aimed at improving confinement as well as proposals for better language support for confinement (Section 6).

2. AN EXAMPLE OF CONFINEMENT

In modern object-oriented programming languages such as Java, confinement can be achieved by a disciplined use of built-in access control mechanisms and some simple coding idioms. We will give a simple motivating example and use it to discuss our analysis.

Consider the class HashtableEntry used within the implementation of Hashtable in the Sun JDK's java.util package. The access modifier for this class is set to default access, which, in Java, means that the class is scoped by its defining package. HashtableEntry instances are used to implement linked lists of elements which have the same hash code modulo table size. They are a prime example of an internal data structure which is only relevant to one particular implementation of a hashtable and that should not escape the context of Hashtable and definitely not of the defining package java.util. Yet how can we be sure that code outside of the package cannot get access to a HashtableEntry object? Since HashtableEntry is package-scoped we need not worry that outside code will create instances of the class. But in case a public method were to return a HashtableEntry object or a public field held a reference to such an object, outside code would be able to cast that reference to Object and either store it or use it as an argument. The implementation of Hashtable itself could cast a HashtableEntry object to some public superclass, and then expose a reference to the object. It is likely that a programmer would consider such a scenario to be the result of a programming error, and a good programmer would be careful and prevent such confinement breaches. One can view this as an escape problem: can references to instances of a package-scoped class escape their enclosing package? If not, then the objects of such a class are said to be *encapsulated* in the package. In the example at hand, HashtableEntry is indeed encapsulated as programmers have carefully avoided exposing them to code outside of the java.util package.

Kacheck/J discovers potential confinement violations and returns a list of confined types for each package analyzed by the tool. For instance, in the above example, the expected result of the analysis would be that HashtableEntry is confined to the package java.util, while Hashtable is not since it has been declared public. The analysis relies on access modifiers of classes, fields and methods, along with results of a simple intra-procedural analysis of the bytecode of all methods defined in the enclosing package (this part of the analysis performs confinement checks). Furthermore, for package-scoped classes, the code of inherited methods is also analyzed (this part of the analysis performs the socalled anonymity checks). Figure 2 illustrates the checks performed by the tool.



Figure 2: Analysis overview. All classes in the enclosing package, java.util in this case, are checked for confinement. Parent classes of confined classes (e.g. Object) are checked for anonymity. Client code need not be checked.

The analysis is modular since only one package needs to be considered at a time; this turns out to be a key feature for scalability. Furthermore, since client code is not required when checking confinement, it is possible to use Kacheck/J on library code.

In fact, our analysis infers that the class HashtableEntry is not confined because the method clone is invoked on one of its instances. The Hashtable's clone method clones all of the entries in the table. The problem with clone is that it returns a copy of the receiver cast to Object. Manual inspection of the code reveals that each invocation of HashtableEntry.clone() is immediately followed by a cast to HashtableEntry. Thus instances of the class do not escape. But our analysis is not precise enough to discover that. A simple and efficient fix is to refactor the code by replacing HashtableEntry.clone() with a new method clone_that returns a HashtableEntry. This refactoring is simple enough and has the advantage of removing unnecessary type casts.

Simplifying Assumptions

Kacheck/J operates under some simplifying assumptions which we detail here.

Sealed packages

We assume that all classes of a package are known at analysis time. This assumption is important for the analysis results as a new class may break confinement of pre-existing classes (e.g. creating a HashtableEntry and returning it from a public method). In Java, user code may load new classes which declare themselves a member of a package. There are several possible approaches here. For example, packages loaded from a jar file may be declared sealed [14, 16], in which case no user class can be added to that package. Another solution would be to add support for incremental confinement analysis as part of bytecode verification.

Reflection

The analysis assumes that reflection does not violate language access control. In other words, it assumes that the semantics of private, protected and default access modifiers are respected by the reflection mechanisms. This assumption can be violated by changing the settings of the Java Security Manager. This may result in additional confinement breaches.

Native code

Native methods are not checked by Kacheck/J and may breach confinement. We assume that native methods defined in the current package do not directly breach confinement, while we make no assumptions about the behavior of native methods defined in other packages. Manual inspection of a large number of native methods indicates that this assumption is reasonable. Furthermore, we assume that native code in other parts of the system does not violate the semantics of the language by ignoring access control declarations.

3. CONFINED TYPES

The goal of confinement is to satisfy the following soundness property:

Soundness: An object of confined type is encapsulated in the scope of that type.

In [5], the granularity of confinement is the package. Thus, no instance of a confined type may escape the package in which that type is defined. We say that instances of a confined class are encapsulated in their defining package.

Confinement is enforced by two sets of constraints. The first set of constraints, *confinement rules*, apply to the enclosing package, the package in which the confined class is defined. These rules track values of confined types and ensure that they are neither exposed in public members, nor widened to non-confined types. The second set of constraints, socalled *anonymity rules*, applies to methods inherited by the confined classes, potentially including library code, and ensures that these methods do not leak a reference to the distinguished variable **this** which may refer to an object of confined type.

In this section, we adapt the rules of Bokowski and Vitek to inference of confinement. The new rules are both simpler and less restrictive (i.e., more classes can be shown confined), while remaining sound. As in the original paper, the rules presented here do not require a closed-world assumption. Confinement inference is performed at the package level. The rules assume that all classes in a package are known and, for confined classes, that their superclasses are available.

3.1 Anonymity Rules

Enforcing confinement relies on tracking the spread of encapsulated objects within a package and preventing them from crossing package boundaries. We have chosen to track encapsulated objects via their type. Thus, a confinement breach will occur as soon as a value of a confined type can escape its package. Since we track types, widening a value from a confined type to a non-confined type is a violation of the confinement property.

Anonymity rules apply to inherited methods which may (but do not have to) reside in classes outside of the enclosing package. The goal of this set of rules is to prevent a method from leaking a reference to the distinguished **this** pointer. The motivation for these rules is that if **this** refers to an encapsulated object, returning or storing it amounts to hidden widening. Thus, we say that a method is *anonymous* if the following three rules hold.

$\mathcal{A}1$	An anonymous method cannot widen this to a non-confined type.
$\mathcal{A}2$	An anonymous method cannot be native .
$\mathcal{A}3$	Methods invoked on this must be anonymous.

Figure 3: Anonymity rules.

The first rule prevents an inherited method from storing or returning **this** unless the static type of **this** also happens to be confined. The second rule ensures that **native** methods are never anonymous. While rules A1 and A2 are direct anonymity violations, the rule A3 tracks transitive violations. The call mentioned in rule A3 depends on the dynamic type of **this** (the target of the call). Thus, anonymity of methods is determined in relation to a specific type.

3.2 Confinement Rules

Confinement rules are applied to all classes of a package. A class is *confined* if it satisfies the five rules of Figure 4.

$\mathcal{C}1$	All methods invoked on a confined type must be anonymous.
$\mathcal{C}2$	A confined type cannot be public.
C3	A confined type cannot appear in the type of a pub- lic (or protected) field or the return type of a public (or protected) method of a non-confined type.
$\mathcal{C}4$	Subtypes of a confined type must be confined.
C5	A confined type cannot be widened to a non- confined type.

Figure 4: Confinement rules.

Rule C1 ensures that no inherited method invoked on a confined type will leak the **this** pointer. This rule does not preclude a confined type from *inheriting* non-anonymous methods, as long as they are never called. Rule C2 prevents public classes from being confined. Rule C3 ensures that no exposed member (private or protected) is of a confined type. This applies to all non-confined types in the package. Rule C4 prevents non-confined classes (or interfaces) from extending confined types. Finally, rule C5 prevents values of confined type from being cast to non-confined types.

Exceptions are a case of widening which is not explicitly listed in these rules. Instead, we consider that **throw** widens its argument to the class **Throwable**, which is declared public and thus violates rule C5.

Our confinement rules do not forbid packages from having native code, but rule A2 explicitly states that native methods are not anonymous. The motivation for this design choice is that while the developer of a package may be expected to manually inspect native code in the current package, it would be difficult to check native code of parent classes belonging to standard libraries. Furthermore, uses of this that violate A1 are usually not perceived as bad behavior for native code. Essentially, we assume that native code within the enclosing package is, to some extent, trusted.

4. CONSTRAINT-BASED ANALYSIS

We use a constraint-based program analysis to infer method anonymity and confinement. Constraint-based analyses have previously been used for a wide variety of purposes, including type inference and flow analysis. Constraint-based analysis proceeds in two steps:

- $1. \quad {\rm Generate\ a\ system\ of\ constraints\ from\ program\ text}.$
- 2. Solve the constraint system.

The solution to the constraint system is the desired information. In our case, constraints are of the following forms:

A constraint not-anon(methodId) asserts that the method

methodId is *not* anonymous; similarly, not-conf(classId) asserts that the class classId is *not* confined. The remaining four forms of constraints denote logical implications. For example, not-anon(A.m()) \Rightarrow not-conf(C) is read "if method m in class A is not anonymous then class C will not be confined."

We generate constraints from the program text in a straightforward manner. The example of Figure 5 illustrates the generation of constraints. For each syntactic construct, we have indicated in comments the associated rule from Section 3. Figure 6 details the constraints that are generated for that example. A complete description of the constraints generated from Java bytecode is given in Appendix A.

public o	class A {		
	A a;		
	<pre>public A m() {</pre>		
	a = this;	11	(A1)
	<pre>new B().t(this);</pre>	11	(A1)
	return this:	11	(A1)
	}		. ,
	native void o():	11	(A2)
3		<i>''</i>	(112)
class B	extends A {		
CIUDD D	word $\pm (\Lambda_{2})$		
	$\begin{array}{c} x p(f) \\ z \\ $	11	(12)
	recurn chis.m();	//	(AS)
	}		
	public A getD() {	<i>,,</i>	(01)
	return new D().p();	//	(CI)
2	}		
}			(
public o	class C {	11	(C2)
	<pre>public D getD() {</pre>	//	(C3)
	return new D();		
	}		
	<pre>public D d = new D();</pre>	//	(C3)
}			
class D	extends B {	//	(C4)
	A getA() {		
	<pre>this.t(this);</pre>	11	(C5)
	a = new D();	11	(C5)
	<pre>return new D();</pre>	11	(C5)
	}		
ŀ			

Figure 5: Example program.

All our constraints are ground Horn clauses. Our solution procedure computes the set of clauses not-conf(classId) that are either immediate facts or derivable via logical implication. This computation can be done in linear time.

Control Flow Analysis

The rule C1 poses a control flow problem as it mandates that only methods that are actually invoked on a confined type need to be anonymous. Any conservative control flow analysis can be used to yield a set of candidate methods. We have chosen to perform a simple flow insensitive analysis that is practical and precise enough for our purposes.

Case	Constraint	Explanation
$(\mathcal{A}1)$	$not-conf(\mathtt{A}) \Rightarrow not-anon(\mathtt{A.m()})$	this widened to A
$(\mathcal{A}2)$	not-anon(A.o())	o is native
$(\mathcal{A}3)$	$not-anon(A.m()) \Rightarrow not-anon(B.p())$	B.p() calls m() with this being the receiver object
$(\mathcal{C}1)$	$not-anon(D.p()) \Rightarrow not-conf(D)$	p() invoked on a D-object
$(\mathcal{C}2)$	not-conf(C)	class C declared to be public
$(\mathcal{C}3)$	$not-conf(\mathtt{C}) \Rightarrow not-conf(\mathtt{D})$	public method C.getD() has return type D; public field C.d has type D
$(\mathcal{C}4)$	$not-conf(D) \Rightarrow not-conf(B)$	D extends B
$(\mathcal{C}5)$	$not-conf(\mathtt{A}) \Rightarrow not-conf(\mathtt{D})$	D widened to A

Figure 6: The constraints generated from the example in Figure 5.

Since, by definition, confined types cannot be invoked from outside of their defining package and cannot be widened to non-confined types, the analysis only needs to record methods invoked on instances of a confined type. Thus, only invocations of the type x.m(), where the type of x is confined, need to be retained. This forms the root set for the control flow analysis. Transitive calls from within a confined method in this root set (e.g. this.m()) are recorded by anonymity rule A3. The type of x in x.m() is determined as the union of the most general type inferred during bytecode verification with all subtypes of that type that are ever widened to it.

The analysis does not attempt to perform dead-code detection, so while the method that includes an invocation such as a.m() may be dead, we will nevertheless add m to the root set. This simplifies the analysis but costs some precision. Doing dead code detection would also lead to analysis results that are much more sensitive to changes in the source program. We strongly believe that the results of confinement inference should be stable in the face of trivial changes to the source program and that any changes should have only local effects.

5. IMPLEMENTING KACHECK/J

Although the confinement and anonymity rules have been described as source level constraints, we have chosen to implement Kacheck/J as a bytecode analyzer. The main advantage of working at the bytecode level is the large number of class files freely available. The implementation of Kacheck/J leverages the Open Virtual Machine project's bytecode verification framework.

In OVM, bytecode verification has been implemented using the flyweight pattern. For each of the 200 bytecode instructions defined in the Java Virtual Machine Specification, the OVM verifier creates an **Instruction** object that is responsible for computing the effect this instruction will have on an abstract state. Verification is a simple fixed-point iteration. The verification starts with an initial state which includes the instruction pointer, operand stack and variables. The verifier follows all possible control flows within the method.

This flyweight approach allows us to use the OVM bytecode verifier as a static analysis engine. We generate constraints by subclassing only 9 of the 200 Instruction objects. These special purpose instructions perform some simple checks and record basic facts about the program execution. For instance, the **areturn** instruction checks if **this** is used as return value, and if so, it reports that **this** is widened to the return type of the method. The **invoke** instructions record dependencies like the use of **this** as an argument or when a method is invoked on **this**.

Overall, the following changes were applied to the verifier:

- In non-static methods, local variable 0 (this) is tracked.
- Uses of this are recorded.
- All widenings are recorded.
- Types of thrown exceptions are recorded.

Widenings are captured by intercepting subtype checks. Anonymity checks only require slight modifications to the code that simulates the nine instructions: a check is added to record operations on this. See the Appendix A for details.

The flow analysis computes the implication chains for each potentially confined type T_1 , such that

$$T_2 \Rightarrow (A \Rightarrow)^* A \Rightarrow T_1$$

is collapsed to

$$T_2 \Rightarrow T_1.$$

The constraints of the form T and $T \Rightarrow T$ are solved immediately while they are recorded.

The code specific to confined types (including verbose reporting of violations) is about 5,600 lines. The code reused from OVM (including class loading) is about 25,000 lines of code. The current version of the OVM is about 74,000 lines of code.

Example

Figure 7 gives an example of a chain of constraints that results in classes being not confined. Mind that the tool reorders parts of the solving process, while here only the final chain of constraints is explained.

The method P.nonAnon() is not anonymous because it widens this to java.lang.Object, which is a non-confined class

(because it is public). This will generate a constraint of type $C \Rightarrow A$:

 $not-conf(Object) \Rightarrow not-anon(P.nonAnon())$

The invocation of nonAnon in nonAnonInd with this as the receiver generates a constraint of the type $A \Rightarrow A$:

\Rightarrow not-anon(B.nonAnonInd())

The method **nonAnonInd**() is invoked on C. By rule C1 a constraint of the type $A \Rightarrow C$ is generated:

$$not-anon(B.nonAnonInd()) \Rightarrow not-conf(C)$$

As C extends B, a constraint of the type $C \Rightarrow C$ is generated by rule C4:

$$not-conf(C) \Rightarrow not-conf(B)$$

Solving this constraint system will result in B and C being non-confined (and P and X cannot be confined either because they are public).

<pre>public class P {</pre>		
<pre>public Object nonAnon() {</pre>		
return this;	11	(1)
}		
}		
class B extends P {		
<pre>public Object nonAnonInd() {</pre>		
return this.nonAnon();	11	(2)
}		
}		
class C extends B {	11	(3)
}		
public class X {		
<pre>public Object invocation() {</pre>		
return new C().nonAnonInd():	11	(4)
}		
}		
} }		

Figure 7: Sample constraint chain.

6. **RESULTS**

Kacheck/J has been evaluated on a large data set. This section gives an overview of the benchmark programs and presents the results of the analysis. We also discuss extensions of Kacheck/J, coding idioms for confinement and improved language support.

6.1 The Purdue Benchmark Suite

The Purdue Benchmark Suite (PBS) consists of 33 Java programs and libraries of varying size, purpose and origin. The entire suite contains 46,165 classes (or 115 MB of bytecode) and 1,771 packages. To the best of our knowledge the PBS is the largest such collection of Java programs. Most of the benchmarks are freely available and can be obtained from the Kacheck/J web page.

Name	Description					
Aglets	Mobile agent toolkit	ag				
AlgebraDB	Relational database	db				
Bloat	Purdue bytecode optimizer	bl				
Denim	Design tool	de				
Forte	Integrated dev. environment	fo				
GFC	Graphic foundation classes	gf				
GJ	Java compiler	gj				
HyperJ	IBM composition framework	hj				
JAX	Packaging tool	ja				
JDK 1.1.8	Library code (Sun)	j1				
JDK 1.2.2	Library code (Sun)	j2				
JDK 1.3.0	Library code (IBM)	j3				
JDK 1.3.1	Library code (Sun)	j4				
JavaSeal	Mobile agent system	js				
Jalapeno 1.1	Java JIT compiler	jp				
JPython	Python implementation	ју				
JTB	Purdue Java tree builder	jb				
JTOpen	IBM toolbox for Java	jt				
Kawa	Scheme compiler	kw				
OVM	Java virtual machine	o4				
Ozone	ODBMS	OZ				
Rhino	Javascript interpreter	$^{\rm rh}$				
SableCC	Java to HTML translator	\mathbf{sc}				
Satin	Toolkit from Berkeley	\mathbf{sa}				
Schroeder	Audio editor	$^{\rm sh}$				
Soot	Bytecode optimizer framework	so				
Symjpack	Symbolic math package	sy				
Tomcat	Java servlet reference impl.	tc				
Toba	Bytecode-to-C translator	to				
Voyager	Distributed object system	vy				
Web Server	Java Web Server	ws				
Xerces	XML parser	xe				
Zeus	Java/XML data binding	ze				

Figure 8: The Purdue Benchmark Suite (PBS v1.0).

Figure 9 gives an overview of the sizes, in number of classes, for each program or library that is part of the PBS. Appendix B provides additional data about the benchmarks. Our largest benchmarks, over 2,000 classes each, are Forte, JDK 1.2.2, JDK 1.3.*, Ozone, Voyager and JTOpen. Ozone and Forte are applications, while the others are libraries. The number of package-scoped classes is indicated in light gray for each application. This number is an upper bound for the number of confined classes; public classes can not be confined.



Figure 9: Benchmark characteristics: program sizes.

Figure 10 relates the proportion of package-scoped members to package-scoped classes. Package-scoped members are fields and methods that are declared to have either private or default access. Most coding disciplines encourage the use of package-scoped methods and package-scoped classes. Not surprisingly, programs that were designed with reuse in mind, such as libraries and frameworks, are better-written than one-shot applications. For instance, the Aglet workbench and JTOpen, both libraries, exhibit high degrees of encapsulation. Forte is noteworthy because even though it is an application, it has over 50% package-scoped classes and members. Compilers and optimizers written in an objectoriented style, such as Bloat, Toba and Soot, have high numbers of package-scoped classes because of the many classes used to represent syntactic elements or individual bytecode instructions. At the other extreme, we have applications like Jax and Kawa which have almost no package-scoped classes. It is also worth noting the increase in encapsulation between different versions of the JDK. The percentage of package-scoped classes doubled between JDK1.1.8 and JDK1.3.1, while the absolute number of classes tripled.



Figure 10: Benchmark characteristics: member encapsulation.

Coding style has an impact on confinement. While the relation between package-scoped classes and confined types is obvious, there is a more subtle connection between packagescoped members and confined types: public and protected methods can return potentially confined types. So it is reasonable to expect that programs with low proportions of package-scoped members will also have comparatively fewer confined types.

6.2 Confined Types

Running Kacheck/J over the PBS yields 3,998 confined classes, 25% of the package-scoped classes are confined. Figure 11 shows confined classes in percentage of all classes. The numbers are broken down per program with confined inner classes in light gray. Raw numbers are given in Appendix B.

There are 6 programs where more than 40% of the packagescoped types are confined (db, gf, jy, jb, jp, o4). It is interesting to note that these programs have very little in common: they are a mix of libraries (gf), frameworks (o4) and applications (db, jy, jb, jp). Their ratio of package-scoped classes and their sizes vary widely. Indeed, manual inspection of the programs indicates that programming style is essential to confinement. For example, in early versions of OVM and Kacheck/J, unit tests were systematically stored



Figure 11: Confined types.

in a sub-package of the current package. Some methods and classes were declared public only to allow testing of the code. This in turn prevented many classes from being confined. The large number of confined inner classes in OVM (o4) comes from the objects representing bytecode instructions nested in an instruction set class. For Jalapeno, the high confinement ratio (153 classes out of 994) is partially the result of the single package structure of the program.



Figure 12: Confinement and package-scoping.

Quite predictably, programs with very few package-scoped classes (e.g. ja, kw, sh, gf) end up with few confined classes. Figure 12 shows the relationship between package-scoped classes and confined classes. The variability in this figure is quite high. For instance, libraries like Aglets (ag) which have very high ratios of package-scoped members and classes still perform quite poorly with only 13 classes being confined out of 410. Why does this happen? There can be two explanations: either the classes are really confined and our analysis is simply not powerful enough to discover that this is the case, or our original assumption that package-scoped classes are naturally confined is wrong. The first case leads to the question of how to improve our analysis. The second case raises the question of whether we can refactor the code to make them confined. To answer these questions, we start with a discussion of confinement violations.

6.3 Confinement Violations

Confinement breaches are caused by a small number of widely used programming idioms. For any violation Kacheck/J returns a textual representation of the implication chain that caused the violation. We give examples of the main causes for classes not being confined.

6.3.1 Anonymity Violations

The top three anonymity violations (accounting for 133 nonconfined classes) in the entire JDK come from methods in the AWT library which register the current object for notification. The method addImpl is representative:

6.3.2 Widening to superclass

Widening to a superclass is among the most frequent kind of confinement breach. For instance, Kacheck/J signals the following widening in the Aglet benchmark:

```
com/ibm/aglets/tahiti/SecurityPermissionEditor:
```

```
- illegal widening to:
```

- com/ibm/aglets/tahiti/PermissionEditor

PermissionEditor is an abstract superclass of the non-public **SecurityPermissionEditor**. **PermissionEditor** is the part of the interface that is exported outside the package.

6.3.3 Widening in Containers

A large number of violations comes from the use of container classes in Java. Data structures such as vectors and hashtables always take arguments of type Object, thus any use of a container will entail widening to the most generic super type. For instance, Kacheck/J reports that NativeLibrary, an inner class of ClassLoader, is not confined.

```
java/lang/ClassLoader$NativeLibrary:
Illegal Widening to java/lang/Object
```

The error occurs because an instance of NativeLibrary is stored in a vector:

```
systemNativeLibraries.addElement(lib);
```

As such, this violation may indicate a security problem. The internals of class loaders should really be encapsulated. Inspection of the code reveals that the **Vector** in which the object is stored is private.

After a little more checking it is obvious that the vector does not escape from its defining class. But this requires inspection of the source code and only remains true only until the next patch is applied to the class. This example shows the usefulness of tools such as Kacheck/J as they can direct the attention of software engineers towards potential security breaches or software defects.

6.3.4 Anonymous Inner Classes

This violation occurs frequently when inner classes are used to implement call-backs. For example in Aglets the Mouse-Listener class is public. Thus, the following code violates confinement of the anonymous inner class.

```
MouseListener mlistener = new MouseAdapter() {
   public void mouseEntered(MouseEvent e)
      { ... } };
```

Similar situations occur with package-scoped classes that implement public interfaces. They are package-scoped to protect their members, but are exported outside of the package.

6.4 Confinement with Generics

In Java, vectors, hashtables and other containers are omnipresent. Every time an object is stored in a container, its type is widened to **Object** leading to a widening violation for the object's class. If Java supported proper parametric polymorphism, the large majority of the violations would disappear (there can be a few heterogenous data structures, but they seem be the exception).

In order to try to assess the impact of generics, without rewriting all of the programs in the PBS, we modified Kacheck/J to ignore widening violations linked to containers. This is done by ignoring all widenings to Object that occur in calls to methods of classes java.util. Figure 13 gives the percentages of confined classes without generic violations; we call these classes Generic-Confined (GC). The light gray bars show the original number of confined classes. The dark grey bars show the effect of adding genericity. The number of confined types increases by 875 (over all programs in the PBS).



Figure 13: Generic-confined types.

These results should be viewed with caution because they can represent an overestimate of the potential gains since we do not guarantee that the container instances are packagescoped.

6.5 Inferring Access Modifiers

The low number of confined classes in some of the benchmarks is surprising. Looking at the access modifiers of classes in these benchmarks, the reason is immediately clear. For example, in Kawa, out of 443 classes, only 5 are packagescoped. Similarly, many benchmarks contain methods and or fields that are declared as public and thus prevent certain types from being confined. Are these access modes the tightest possible, or are they sometimes randomly chosen? To answer this question we infer the tightest access modes during analysis and then use the inferred modes for confinement checking. Figure 14 shows the result of this analysis. Classes that become confined with modifier inference are called *Confinable* (CA). With mode inference, the number of confinable classes jumps to 13,064 for the entire PBS. Furthermore if we combine confinable and generics, we obtain 14,591 Generic-Confinable classes.



Figure 14: Confinable types.

Figure 15 relates the results of this new analysis to the original number of package-scoped classes. It is quite telling to see that Jax and Kawa, which were applications with the lowest number of confined classes suddenly have about 40% of their classes confinable.



Figure 15: Confinable types and package-scoping.

Of course, using this option on library code may yield an overestimate of the potential gains as some classes that are never used from within the library can be made packagescoped, even though client code requires access to these classes. Nevertheless, the results give a good indication of the potential gains.

6.6 Hierarchical Packages

Our last experiment involves changing the semantics of the Java package mechanism. Currently, Java has a flat package namespace; that is to say, even though package names can be nested, there is no semantics in this nesting. This creates a dilemma between data abstraction and modularity. Good design practice suggests that applications be split into packages according to functional characteristics of the code. On the other hand, creating packages forces certain classes to become public even if those classes should not be used by clients of the program. From a confinement perspective, we could say more packages result in fewer confined classes. One extreme is Jalapeno, which is structured as a single package. This diminishes the usefulness of the confinement property.

To evaluate the impact of the package structure on confinement, we modified Kacheck/J to use a hierarchical package model. The general idea is that package-access would be extended to neighbor packages. We introduce a definition of scope that we call *n*-package-scoped. *n*-package-scoped limits access to classes in packages that are less than *n* nodes in the tree of package names away from the defining package. For example, the class java.util.HashtableEntry would be visible for java.lang.System for n = 2. The unnamed package is defined to have distance ∞ from all other packages, making a *n*-package-scoped class a.A invisible for b.B regardless of the choice of *n*.

Figure 16 shows the cumulative improvements yielded by increasing the proximity threshold n. With n = 9 most programs are treated as a single package and the benefits are 3,691 additional confined classes. The largest increase in confined classes comes from the Voyager benchmark with 813 new confined classes. The most important increment is at n = 3 with 2,679 additional confined classes. This threshold value allows classes to access package-scoped members (and classes) of sibling classes.



Figure 16: Hierarchical packages.

6.7 Coding for Confinement

Our results clearly point to containers as one source of confinement violations. We considered using generic extensions of Java (such as GJ) to increase confinement. Unfortunately, the homogeneous translation strategies adopted by most of these extensions imply that at the bytecode level, code written with GJ is translated back to code that uses the standard Java container classes. Thus, it is not possible for Kacheck/J to verify that classes stored in generic containers remain confined. Heterogeneous translation strategies have the drawback of causing code duplication. Fortunately, it is possible to achieve the desired result with some coding techniques. The basic idea is to use the adapter pattern to wrap an unconfined object around each confined object that must be stored in a container. A confined implementation of a hashtable could provide an interface Entry with two methods boolean equal(Entry e) and int hashCode(). In the package that contains the confined class C, the programmer would define an implementation RealEntry of Entry with a package-scoped constructor that takes the key and value (where for example the value has the type of the confined class) and package-scoped accessor methods. The Hashtable itself would only be able to access the public methods defined in Entry.

The cost of this change would be the creation of the extra Entry object that might not be required by other implementations of Hashtable. On the other hand, to access a key-value pair, this implementation only requires one cast (Entry to the RealEntry to access key and value), where the default implementation requires a cast on key and value. For other containers, the tradeoffs may be worse.

```
public interface Entry {
    public boolean equal(Entry e);
    public int hashCode(); }
public class Hashtable {
    public void put(Entry e) {...}
    public Entry get(Entry e) {...} }
class MyEntry implements Entry {
    ConfinedKey key;
    ConfinedValue val;
    public boolean equal(Entry e) {...}
    public int hashCode() {...} }
```

Figure 17: Example Hashtable interface.

6.8 **Runtime Performance**

All benchmarks were performed on a Pentium III 800 with 256 MB of RAM running Linux 2.2.19 with IBM JDK 1.3. Except for the JDK tests (j1, j2, j3, j4) all running times include loading and analyzing required parts of the Sun JDK 1.3.1 libraries. The longest running time is that of JDK 1.3.1 which consists of 7,037 classes and is analyzed in 41 seconds. On average, Kacheck/J needs 7.5 ms per class. Figure 18 summarizes the cost of confinement checking, detailed timings are in the appendix.



Figure 18: Running times in ms (log10 scale).

7. RELATED WORK

Reference semantics permeate object-oriented programming languages, and the issue of controlling aliasing has been the

Figure 19: Confinement violation C1.

focus of numerous papers in the recent years [12, 11, 8, 1, 15, 10, 13, 7]. We will discuss briefly the most relevant work.

Bokowski and Vitek [5] introduced the notion of confined types. In their paper, confined types are explicitly declared. The implication is that software must be designed and implemented with confinement in mind. Their paper discussed an implementation of a source-level confinement checker based on Bokowski's CoffeeStrainer [4]. Kacheck/J infers confinement from existing Java code. The main difference between that work and the present paper lies in the definition of anonymity. The most interesting confinement breach is hidden widening of confined types to public types that can occur with inherited methods (rule C1).

Consider the example of Figure 19. Intra-procedural analysis would not reveal that (new NotConf()).violation() will widen NotConfined to Parent. So, Bokowski and Vitek chose to rely on explicit anonymity declarations and added an additional anonymity constraint:

 $\mathcal{A}4$ Anonymity declarations must be preserved when overriding methods.

Thus, once a method is declared anonymous, all overriding definitions of that method have to abide by the constraints. When inferring anonymity, the rule $\mathcal{A}4$ is not necessary. The goal of $\mathcal{A}4$ was to ensure that anonymity of a method is independent from the result of method lookup. If anonymity of methods is inferred, dynamic binding can be taken into account.

```
public class A { // A is not confined
    Object m() {
        // m() is anonymous in relation to C
        // but not in relation to B
        return null; }
        public Object n() {
            return new C().m(); } }
class B extends A { // B is not confined
        Object m() { // m() is not anonymous
            return this; } }
class C extends A{} // C is confined
```

Figure 20: Anonymity need not be preserved in all subtypes.

Figure 20 shows a confined class C that extends a class A. The method A.m() meets all anonymity criteria except for rule $\mathcal{A}4$. The violation of that rule occurs in class B, because B extends A and redefines m() with an implementation that returns this. The key point to notice here is that the anonymity violation cannot occur if the dynamic type of this is A. We say the method A.m() is anonymous *in relation* to C, but not in relation to B.

Another difference between the old and the new anonymity rules is that we allow widening of the **this** reference to other confined types. The old rules forbid returning **this** or using **this** as an argument completely. The new rules allow such cases, if the type of the return value or the argument is again a confined type. An example is shown in figure 21, which is a minimal variation of figure 19. In this case the new rules would allow both classes to be confined.

In [15], flexible alias protection is presented as a means to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing-mode declarations specify constraints on the sharing of references. The mode **rep** protects *representation objects* from exposure. In essence, **rep** objects belong to a single owner object and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode **arg** marks argument objects which do not belong to the current owner, and therefore may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles. Clarke, Potter, and Noble [7] have formalized representation containment by means of ownership types.

Hogg's Islands [11] and Almeida's Balloons [1] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from [15] is that both proposals strive for full encapsulation, that is, all objects reachable from an owner are protected from aliasing. This is equivalent to declaring everything inside an Island or Balloon as rep. This is restrictive, since it prevents many common programming styles; it is not possible to mix protected and unprotected objects as done with flexible alias protection and confined types. Hogg's proposal extends Smalltalk-80 with sharing annotations but it has neither been implemented nor formally validated. Almeida did present an abstract interpretation algorithm to decide if a class meets his balloon invariants, but it was also not implemented so far. Balloon types are similar to confined types in that they only require an analysis of the code of the balloon type and not of the whole program.

Boyland, Noble and Retert [6] introduced capabilities as a uniform system to describe restrictions imposed on references. Their system can model many of the different modifiers used to address the aliasing problem, such as immutable, unique, readonly or borrowed. They also model a notion of anonymous references, which is different from the one used in this paper. Their system of access rights cannot be used to model confined types, mainly because it lacks support for modeling package-scoped access.

Kent and Maung [13] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time. In the field of static program analysis, a number of techniques have been developed. Static escape analyses such as the ones proposed by Blanchet [2] and others [3, 9] provide much more precise results than our technique, but come at a higher analysis cost. They often require whole program analyses, and are sensitive to small changes in the source code. More than anything, their results can be hard to interpret for a programmer; knowing that an object escapes may not be enough to have an idea how to re-engineer the code to avoid such an occurrence.

8. CONCLUSION

We have presented the Kacheck/J tool for inferring confinement in Java programs and used the tool to analyze over 46,000 classes. The results of the analysis are surprisingly high, about 25% of all package-scoped classes and interfaces are confined. Furthermore, we discovered that many of the confinement violations are caused by the use of container classes and thus might be solved by extending Java with genericity, this would increase confinement to 30%. The biggest surprise was the number of violations due to badly chosen access modifiers. After inferring tighter access modifiers, 45% of all package-scoped classes were confined. We expect that these numbers will rise even further once programmers start to write code with confinement in mind..

Confinement is an important property. It bounds aliasing of encapsulated objects to the defining package of their class, and helps in re-engineering object-oriented software by exposing potential software defects, or at least making, often subtle, dependencies visible. We have demonstrated that inferring confined types is fast and scalable. Kacheck/J is available from

http://gecko.cs.purdue.edu/kacheck/



Acknowledgments

This work is supported by grants from DAAD, Lockheed-Martin, CERIAS, and NSF (CCR–9734265). The authors wish to thank Ben Titzer and Theodore Witkamp for developing the modifier inference package, and Mario Südholt as well as the members of IFIP WG2.4 for helpful comments. Some of the programs from the Purdue Benchmark Suite originate from the Ashes suite, we wish to thank the Sable research group at McGill for making these available.

9. **REFERENCES**

- Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In ECOOP Proceedings, June 1997.
- [2] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 34(10) of ACM SIGPLAN Notices, pages 20–34, Denver, CO, October 1999. ACM Press.
- [3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications, volume 34(10) of ACM SIGPLAN Notices, pages 35–46, Denver, CO, October 1999. ACM Press.
- [4] Boris Bokowski. CoffeeStrainer: Statically-checked constraints on the definition and use of types in Java. In *Proceedings of ESEC/FSE'99*, Toulouse, France, September 1999.
- [5] Boris Bokowski and Jan Vitek. Confined Types. In Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, USA, November 1999.
- [6] John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In ECOOP'01 — Object-Oriented Programming, 15th European Conference, number to appear in Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.
- [7] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In OOPSLA '98 Conference Proceedings, volume 33(10) of ACM SIGPLAN Notices, pages 48–64. ACM, October 1998.
- [8] D. Detlefs, K. Rustan M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.
- [9] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 226-229, La Jolla, California, June 21-23, 1995.
- [10] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using Sandwich Types. In *Proceedings of the 2nd Types in Compilation workshop*, volume LNCS 1473, Kyoto, Japan, March 1998. Springer Verlag.
- [11] John Hogg. Islands: Aliasing protection in object-oriented languages. In OOPSLA Proceedings, November 1991.
- [12] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. OOPS Messenger, 3(2), April 1992.
- [13] S.J.H. Kent and I. Maung. Encapsulation and Aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*. Prentice Hall, 1995.
- [14] Sun Microsystems. Support for extensions and applications in the version 1.2 of the Java platform. 2000.
- [15] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, volume 1543 of *LNCS*, Brussels, Belgium, July 20 - 24 1998. Springer-Verlag.
- [16] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed calls in Java packages. In OOPSLA '2000 Conference Proceedings, ACM SIGPLAN Notices. ACM, October 2000.

APPENDIX A. CONSTRAINT GENERATION

In this section we present which opcodes generate which constraints for confined types.

InvokeStatic

- If this occurs in the argument list, record widening of this to the type T of the matching argument in the current method m. This generates the constraint: $C \Rightarrow A$ where C is not-conf(T) and A is not-anon(m).
- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C' \Rightarrow C$ where C' is not-conf(T') and C is not-conf(T).

Areturn, Putfield, Putstatic, Aastore

- If the variable that is returned or stored is **this**, record widening of **this** to the declared type T' (the return type, type of the field or the type of the array). This generates a constraint $A \Rightarrow C$ where C is $\mathsf{not-conf}(T')$ and A is $\mathsf{not-anon}(m)$ with m being the current method.
- If the variable that is used is an object but not this and has inferred type T, record widening to the corresponding declared type T'. This generates constraints C ⇒ C' where C is not-conf(T') and C is not-conf(T).

InvokeInterface, InvokeVirtual, InvokeSpecial

- If this occurs in the argument list, record widening of this to the type T of the matching argument in the current method m. This generates the constraint: C ⇒ A where C is not-conf(T) and A is not-anon(m).
- If the call is of the form this.n(), calling a method n from method m on this, record method invocation distinguishing between invokevirtual, invokeinterface and invokespecial. This generates the constraint A ⇒ A' where A is not-anon(n) and A' is not-anon(m).
- If the call is not on this but of the form a.n(), record an invocation on type T where T is the inferred type of a. This generates the constraint $A \Rightarrow C$ where A is not-anon(n) and C is not-conf(T).
- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C \Rightarrow C'$ where C is not-conf(T') and C is not-conf(T).

Athrow

- If the variable that is thrown is this, record widening of this to Throwable. This generates a constraint C ⇒ A where C is not-conf(Throwable) and A is not-anon(m) with m being the current method. Because the condition not-conf(Throwable) is always true, a primitive constraint A can be used, too.
- If the thrown variable is an object but not this and has inferred type T, record widening to Throwable. This generates a constraint $C \Rightarrow C'$ where C is again always true (not-conf(Throwable)) and C' is not-conf(T).

Call Propagation

A call to method m on a type T must generate additional constraints for all subtypes S_i of T that are widened to T.

Benchmark	Classes		Place	Opcodoc	Confinement				Time	
	All	Public	Inner	1 Kgs	Opcodes	С	GC	CA	GCA	(ms)
Aglets	410	193	133	18	107846	13	15	60	66	4979
AlgebraDB	161	130	9	6	51218	20	24	81	97	3009
Bloat	282	150	127	17	84212	10	17	29	39	3623
Denim	949	684	271	63	288140	65	71	187	211	9463
Forte	6535	3053	3769	192	1123362	306	437	1149	1346	37565
GFC	153	143	8	15	58003	5	5	58	58	3284
GJ	338	202	189	12	105323	27	27	51	52	4245
HyperJ	1007	862	70	26	211269	32	38	193	212	6711
JAX	255	255	0	9	97932	0	0	99	104	3790
JDK 1.1.8	1704	1423	29	80	917132	71	96	712	744	13103
JDK 1.2.2	4338	2655	1365	130	958619	527	603	1062	1173	23463
JDK 1.3.0	5438	3326	1780	176	1180406	581	685	1297	1476	29336
JDK 1.3.1	7037	4569	2043	213	2010305	756	891	2126	2344	41304
JPython	214	134	35	7	103094	40	45	90	107	4107
JTB	158	150	1	6	48900	4	4	8	8	3009
JTOpen	3022	1439	557	52	1048704	438	467	1049	1113	23950
Jalapeno 1.1	994	730	132	29	255436	155	159	543	549	6770
JavaSeal	75	56	19	9	34933	1	2	14	17	2685
Kawa	443	438	100	6	68733	1	1	177	177	3910
OVM	763	391	539	26	89975	313	313	427	428	6072
Ozone	2442	1705	490	112	447984	93	221	754	920	13245
Rhino	95	67	1	5	51752	11	15	28	33	3201
SableCC	342	290	47	8	45621	3	5	24	28	3470
Satin	938	559	455	48	194985	48	52	206	218	7955
Schroeder	108	103	7	2	41422	0	1	6	7	3270
Soot	721	302	79	6	65137	45	47	90	92	5622
Symjpack	194	125	0	11	73465	8	10	53	89	3559
Toba	762	327	79	11	98993	53	55	102	104	6020
Tomcat	1271	916	221	93	286368	65	109	377	448	8918
Voyager	5667	4430	1305	294	996077	208	295	1268	1442	34082
Web Server	1024	787	52	60	370664	51	72	255	301	9308
Xerces	622	508	125	35	233919	22	47	221	279	6038
Zeus	604	517	74	39	180437	20	38	237	278	5640
Total	46165	30277	13555	1771	10917301	3998	4873	13064	14591	347567

Figure 22: Statistics for the benchmarks. C is Confined, GC is Generic-Confined, CA is Confinable and GCA is Genrice-Confinable.