Static Program Analysis

Anders Møller and Michael I. Schwartzbach

January 23, 2017

Department of Computer Science
Aarhus University, Denmark
{amoeller,mis}@cs.au.dk

Contents

Preface **iii**

1 Introduction **1**
1.1 Undecidability of Static Program Analysis 4

2 A Tiny Programming Language **7**
2.1 The Syntax of TIP 7
2.2 Example Programs 9
2.3 Control Flow Graphs 10

3 Type Analysis **13**
3.1 Types 13
3.2 Type Constraints 14
3.3 Solving Constraints 15
3.4 Slack and Limitations 16

4 Lattice Theory **19**
4.1 Example: Sign Analysis 19
4.2 Lattices 20
4.3 Constructing Lattices 22
4.4 Equations and Fixed-Points 24

5 Dataflow Analysis with the Monotone Framework **27**
5.1 Fixed-Point Algorithms 27
5.2 Example: Sign Analysis, Revisited 29
5.3 Example: Liveness 32
5.4 Example: Available Expressions 34
5.5 Example: Very Busy Expressions 38
5.6 Example: Reaching Definitions 39
5.7 Forwards, Backwards, May, and Must 40
5.8 Example: Initialized Variables 41

5.9 Example: Constant Propagation 42
5.10 Example: Interval Analysis 42
5.11 Widening 44
5.12 Narrowing 45

6 Path Sensitivity 47
6.1 Assertions 47
6.2 Branch Correlations 48

7 Interprocedural Analysis 55
7.1 Interprocedural Control Flow Graphs 55
7.2 Example: Interprocedural Sign Analysis 57

8 Control Flow Analysis 59
8.1 Closure Analysis for the λ-calculus 59
8.2 The Cubic Algorithm 60
8.3 Control Flow Graphs for Function Pointers 61
8.4 Control Flow in Object Oriented Languages 64

9 Pointer Analysis 67
9.1 Points-To Analysis 67
9.2 Andersen's Algorithm 68
9.3 Steensgaard's Algorithm 69
9.4 Interprocedural Points-To Analysis 70
9.5 Example: Null Pointer Analysis 71
9.6 Example: Shape Analysis 73
9.7 Example: Escape Analysis 76

Preface

Static program analysis is the art of reasoning about the behavior of computer programs without actually running them. This is useful not only in optimizing compilers for producing efficient code but also for automatic error detection and other tools that can help programmers. A static program analyzer is a program that reasons about the behavior of other programs. For anyone interested in programming, what can be more fun than writing programs that analyze programs?

As known from Turing and Rice, all interesting properties of the behavior of programs written in common programming languages are mathematically undecidable. This means that automated reasoning of software generally must involve approximation. It is also well known that testing may reveal errors but generally cannot show their absence. In contrast, static program analysis can – with the right kind of approximations – check all possible executions of the programs and provide guarantees about their properties. One of the key challenges when developing such analyses is how to ensure high precision and efficiency to be practically useful.

These notes present principles and applications of static analysis of programs. We cover basic type analysis, lattice theory, control flow graphs, dataflow analysis, fixed-point algorithms, narrowing and widening, path-sensitivity, interprocedural analysis and context-sensitivity, control flow analysis, and pointer analysis. A tiny imperative programming language with heap pointers and function pointers is subjected to numerous different static analyses illustrating the techniques that are presented.

We emphasize a *constraint-based approach* to static analysis where suitable constraint systems conceptually divide the analysis task into a front-end that generates constraints from program code and a back-end that solves the constraints to produce the analysis results. This approach enables separating the analysis specification, which determines its precision, from the algorithmic aspects that are important for its performance. In practice when implementing analyses, we often solve the constraints on-the-fly, as they are generated, without representing them explicitly.

We focus on analyses that are fully *automatic* (i.e., not involving programmer guidance, for example in the form of loop invariants) and *conservative* (usually meaning sound but incomplete), and we only consider Turing complete languages (like most programming languages used in ordinary software development).

The analyses that we cover are expressed using different kinds of constraint systems, each with their own constraint solvers:

- term unification constraints, with an almost-linear union-find algorithm,
- conditional subset constraints, with a cubic algorithm, and
- monotone constraints over lattices, with variations of fixpoint solvers.

The style of presentation is intended to be precise but not overly formal. The readers are assumed to be familiar with advanced programming language concepts and the basics of compiler construction.

We will see the basic tools that are required to perform static analysis of programs. Real-life applications invariably gravitate back to the techniques that we will covered, though many variations and extensions are usually required.

Two major areas will not been covered at all. The *quality* of an analysis can only be measured relatively to a suite of intended applications. It is rare that competing analyses can be formally compared, so much work in this area is concerned with performing experiments to establish the precision and efficiency of proposed analyses. The *correctness* of an analysis requires a formal semantics of the underlying programming language. Completely formal proofs of correctness of analyses are exceedingly laborious and remain mostly academic exercises. Even so, it is often possible to provide convincing informal correctness arguments.

The notes are accompanied by a web site that provides lecture slides, an implementation (in Scala) of most of the algorithms we cover, and additional exercises:

`http://cs.au.dk/˜amoeller/spa/`

Chapter 1

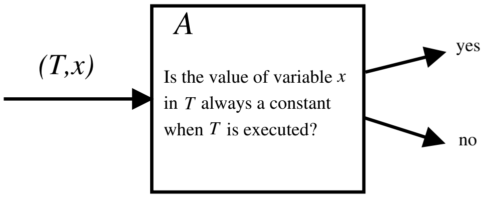
Introduction

There are many interesting questions that can be asked about a given program, for example:

- does the program terminate on every input?
- how large can the heap become during execution?
- does there exist an input that leads to a null pointer dereference, division-by-zero, or arithmetic overflow?
- are all variables initialized before they are read?
- are arrays always accessed within their bounds?
- are all assertions guaranteed to succeed?
- does the program contain dead code, or more specifically, is function `f` reachable from `main`?
- does the value of variable `x` depend on the program input?
- is it possible that the value of `x` will be read in the future?
- do `p` and `q` point to disjoint structures in the heap?
- can there be dangling references, e.g. pointers to memory that has been freed?
- are all resources properly released before the program terminates?

Such questions arise when reasoning about correctness of programs and when optimizing programs for improving their performance. Regarding correctness, programmers routinely use testing to gain confidence that their programs works as intended, but as famously stated by Dijkstra: *"Program testing can be used to show the presence of bugs, but never to show their absence."* Ideally we want guarantees about what our programs may do for all possible inputs, and we want these guarantees to be provided automatically, that is, by programs. A *program analyzer* is such a program that takes other programs as input and decides whether or not they have a given property.

Rice's theorem is a general result from 1953 that informally states that all interesting questions about the behavior of programs (written in Turing-complete programming languages[1]) are *undecidable*. This is easily seen for any special case. Assume for example the existence of an analyzer that decides if a variable in a program has a constant value. In other words, the analyzer is a program A that takes as input a program T and one of T's variables x, and decides whether or not x has a constant value whenever T is executed.


A
(T,x)
Is the value of variable x in T always a constant when T is executed?
yes
no

We could then exploit this analyzer to also decide the halting problem by using as input the following program where `TM(j)` simulates the `j`'th Turing machine on empty input:

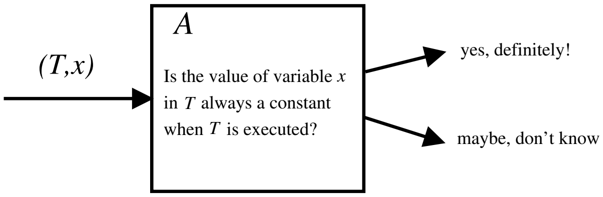
```
x = 17; if (TM(j)) x = 18;
```

Here `x` has a constant value if and only if the `j`'th Turing machine does not halt on empty input. If the hypothetical constant-value analyzer A exists, then we have a decision procedure for the halting problem, which is known to be impossible.

This seems like a discouraging result. However, our real goal is not to decide such properties but rather to solve practical problems like making the program run faster or use less space, or finding bugs in the program. The solution is to settle for *approximative* answers that are still precise enough to fuel our applications.

Most often, such approximations are *conservative* (or *safe*), meaning that all errors lean to the same side, which is determined by our intended application.

Consider again the problem of determining if a variable has a constant value. If our intended application is to perform constant propagation optimization, then the analysis may only answer *yes* if the variable really is a constant and must answer *maybe* if the variable may or may not be a constant. The trivial solution is of course to answer *maybe* all the time, so we are facing the engineering challenge of answering *yes* as often as possible while obtaining a reasonable analysis performance.


A
(T,x)
Is the value of variable x in T always a constant when T is executed?
yes, definitely!
maybe, don't know

[1] From this point on, we only consider Turing complete languages.

A different example is the question: to which variables may the pointer p point? If our intended application is to replace *p with x in order to save a dereference operation, then the analysis may only answer "&x" if p certainly must point to x and must answer "?" if this is false or the answer cannot be determined. If our intended application is instead to determine the maximal size of *p, then the analysis must reply with a possibly too large set {&x,&y,&z,...} that is guaranteed to contain all targets.

In general, all optimization applications need conservative approximations. If we are given false information, then the optimization is *unsound* and changes the semantics of the program. Conversely, if we are given trivial information, then the optimization fails to do anything.

Approximative answers may also be useful for finding bugs in programs, which may be viewed as a weak form of program verification. As a case in point, consider programming with pointers in the C language. This is fraught with dangers such as null dereferences, dangling pointers, leaking memory, and unintended aliases. Ordinary compiler technology offers little protection from pointer errors. Consider the following small program which performs every kind of error if executed with precisely 42 arguments:

```
int main(int argc, char *argv[]) {
  if (argc == 42) {
    char *p,*q;
    p = NULL;
    printf("%s",p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    free(p);
    p = (char *)malloc(100);
    p = (char *)malloc(100);
    q = p;
    strcat(p,q);
  }
}
```

The standard tools such as gcc -Wall and lint detect no errors. Finding the errors by testing might miss the errors, unless we happen to have a test case that runs the program with exactly 42 arguments. However, if we had even approximative answers to questions about null values and pointer targets, then many of the above errors could be caught statically, without actually running the program.

Exercise 1.1: Describe all the errors in the above program.

1.1 Undecidability of Static Program Analysis

(This section requires familiarity with the concept of universal Turing machines; it is not a prerequisite for the following chapters.)

The reduction from the halting problem presented above shows that some static analysis problems are undecidable, in particular the problem of deciding whether a given variable in a program has a constant value. However, termination is often the least of the concerns programmers have about whether their programs work correctly. Is the problem of deciding whether a variable has a constant value still undecidable if we are allowed to assume that our programs always terminate? And what about other desired program properties?

Using a diagonalization argument in combination with the classic proof of Rice's theorem, we can once and for all show a very strong result: *Every nontrivial*[2] *property of the behavior*[3] *of programs is undecidable, even if we only consider programs that terminate on every input.*

In other words, every attempt to build a fully automatic program analyzer, that is, a program that can check nontrivial correctness properties of the input-output behavior of other programs is doomed to fail, even if the program analyzer is allowed to assume that the given program always terminates. Approximation is inevitable.

If we model programs as deterministic Turing machines, it seems reasonable to define "correctness" as unreachability of a special *fail* state.[4] That is, on a given input, a Turing machine will eventually terminate in its accept state (intuitively returning "yes"), in its reject state (intuitively returning "no"'), in its fail state (meaning that the correctness condition has been violated), or the machine diverges (i.e., never terminates).

We can show the undecidability result using an elegant proof by contradiction. Assume P is a program that can decide whether or not the fail state is reachable in a given total Turing machine. (If the input to P is not a total Turing machine, P's output is unspecified – we only require it to correctly analyze Turing machines that always terminate.) Let us say that P halts in its accept state if and only if the fail state of the given Turing machine is unreachable, and it halts in the reject state otherwise. That is, intuitively P says "yes" if the given Turing machine works correctly, and "no" otherwise. Our goal is to show that P cannot exist.

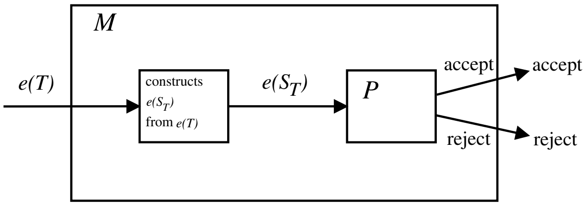
If P exists, then we can also build another Turing machine, let us call it M, that takes as input the encoding $e(T)$ of a Turing machine T and then builds the encoding $e(S_T)$ of yet another Turing machine S_T, which behaves as follows: S_T is essentially a universal Turing machine that is specialized to simulate T on

[2] A property is nontrivial if there exists at least one program that satisfies the property and at least one program that does not. Conversely, trivial properties can be decided even without looking at the input program, and all the properties we care about in practice are nontrivial.

[3] By "behavior" of programs, we mean their input-output semantics, in contrast to syntactic properties.

[4] Technically, we here restrict ourselves to safety properties; liveness properties can be addressed similarly using other models of computability.

input $e(T)$. Let w denote the input to S_T. Now S_T is constructed such that it simulates T on input $e(T)$ for at most $|w|$ moves. If the simulation ends in T's accept state, then S_T goes to its fail state. It is obviously possible to create S_T in such a way that this is the only way it can reach its fail state. If the simulation does not end in T's accept state (that is, $|w|$ moves have been made, or the simulation reaches T's reject or fail state), then S_T goes to its accept state or its reject state (which one we choose does not matter). This completes the explanation of how S_T works relative to T and w. Note that S_T never diverges, and it reaches its fail state if and only if T accepts input $e(T)$ after at most $|w|$ moves. After building $e(S_T)$, M passes it to our hypothetical program analyzer P. Assuming that P works as promised, it ends in accept if S_T is "correct", in which case we also let M halt in its accept state, and in reject otherwise, in which case M similarly halts in its reject state.



We now ask: Does M accept input $e(M)$? That is, what happens if we run M with $T = M$? If M does accept input $e(M)$, it must be the case that P accepts input $e(S_T)$, which in turn means that S_T is "correct", so its fail state is unreachable. In other words, for any input w, no matter its length, S_T does not reach its fail state. This in turn means that T does not accept input $e(T)$. However, we have $T = M$, so this contradicts our assumption that M accepts input $e(M)$. Conversely, if M rejects input $e(M)$, then P rejects input $e(S_T)$, so the fail state of S_T is reachable for some input v. This means that there must exist some w such that the fail state of S_T is reached in $|w|$ steps on input v, so T must accept input $e(T)$, and again we have a contradiction. By construction M halts in either accept or reject on any input, but neither is possible for input $e(M)$. In conclusion, the ideal program analyzer P cannot exist.

Exercise 1.2: In the above proof, the hypothetical program analyzer P is only required to correctly analyze programs that always terminate. In other words, the analyzer is allowed to assume that the given program always terminates. How can the proof be simplified if we want to prove the following weaker property? There exists no Turing machine P that can decide whether or not the fail state is reachable in a given Turing machine. (Note that the given Turing machine is now not assumed to be total.)

Chapter 2

A Tiny Programming Language

We use a tiny imperative programming language, called *TIP*, throughout the following chapters. It is designed to have a minimal syntax and yet to contain all the constructions that make static analyses interesting and challenging.

2.1 The Syntax of TIP

In this section we present the formal syntax of the TIP language, based on context-free grammars.

Expressions

The basic expressions all denote integer values:

$$
\begin{array}{rl}
E \rightarrow & \mathit{intconst} \\
\mid & \mathit{id} \\
\mid & E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid E \texttt{ > } E \mid E \texttt{ == } E \\
\mid & \texttt{(} \; E \; \texttt{)} \\
\mid & \texttt{input}
\end{array}
$$

The `input` expression reads an integer from the input stream. The comparison operators yield 0 for false and 1 for true. Pointer expressions will be added later.

Statements

The simple statements are familiar:

$$
\begin{aligned}
S \rightarrow\ & id = E\texttt{;} \\
\mid\ & \texttt{output}\ E\texttt{;} \\
\mid\ & S\ S \\
\mid\ & \\
\mid\ & \texttt{if (}E\texttt{) \{}\ S\ \texttt{\}}\ \big[\texttt{else \{}\ S\ \texttt{\}}\big]^{?} \\
\mid\ & \texttt{while (}E\texttt{) \{}\ S\ \texttt{\}}
\end{aligned}
$$

We use the notation $\big[\ldots\big]^{?}$ to indicate optional parts. In the conditions we interpret 0 as false and all other values as true. The `output` statement writes an integer value to the output stream.

Functions

Functions take any number of arguments and return a single value:

$$F \rightarrow id\ \texttt{(}\ id\texttt{,}\ldots\texttt{,}id\ \texttt{)}\ \texttt{\{}\ \big[\ \texttt{var}\ id\texttt{,}\ldots\texttt{,}id\texttt{;}\ \big]^{?}\ S\ \texttt{return}\ E\texttt{;}\ \texttt{\}}$$

The `var` block declares a collection of local variables. Function calls are an extra kind of expression:

$$E \rightarrow id\ \texttt{(}\ E\texttt{,}\ldots\texttt{,}E\ \texttt{)}$$

Pointers

Finally, to allow dynamic memory, we introduce pointers into a heap:

$$
\begin{aligned}
E \rightarrow\ & \&id \\
\mid\ & \texttt{malloc} \\
\mid\ & {*}E \\
\mid\ & \texttt{null}
\end{aligned}
$$

The first expression creates a pointer to a variable, the second expression allocates a new cell in the heap, and the third expression dereferences a pointer value. In order to assign values to heap cells we allow another form of assignment:

$$S \rightarrow {*}id = E\texttt{;}$$

Note that pointers and integers are distinct values, so pointer arithmetic is not permitted. It is of course limiting that `malloc` only allocates a single heap cell, but this is sufficient to illustrate the challenges that pointers impose.

We also allow function pointers to be denoted by function names. In order to use those, we generalize function calls to:

$$E \rightarrow \texttt{(}E\texttt{)(}\ E\texttt{,}\ldots\texttt{,}E\ \texttt{)}$$

Function pointers serve as a simple model for objects or higher-order functions.

Programs

A program is just a collection of functions:

$P \rightarrow F \ldots F$

The final function is the main one that initiates execution. Its arguments are supplied in sequence from the beginning of the input stream, and the value that it returns is appended to the output stream. We make the notationally simplifying assumption that all declared identifiers are unique in a program, i.e. that no two different program points introduce the same identifier name.

Exercise 2.1: Argue that any program can be normalized so that all declared identifiers are unique.

TIP lacks many features known from commonly used programming languages, for example, type annotations, global variables, records, objects, nested functions, and pointer arithmetic. We will consider some of these features in exercises in later chapters.

To keep the presentation short, we deliberately have not specified all details of the TIP language, neither the syntax nor the semantics.

Exercise 2.2: Identify the under-specified parts of the TIP language, and propose meaningful choices to make it more well-defined.

2.2 Example Programs

The following TIP programs all compute the factorial of a given integer. The first one is iterative:

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

The second program is recursive:

```
rec(n) {
  var f;
  if (n==0) { f=1; }
  else { f=n*rec(n-1); }
  return f;
}
```

The third program is unnecessarily complicated:

```
foo(p,x) {                       main() {
  var f,q;                          var n;
  if (*p==0) { f=1; }               n = input;
  else {                            return foo(&n,foo);
    q = malloc;                  }
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

2.3 Control Flow Graphs

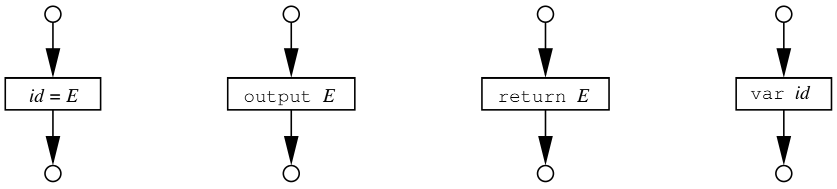
For the purpose of analysis, it is often convenient to view the program as a *control flow graph*, which is a different representation of the program source.

For now, we consider only the subset of the TIP language consisting of a single function body without pointers. A control flow graph (CFG) is a directed graph, in which *nodes* correspond to statements and *edges* represent possible flow of control. For convenience, a CFG always has a single point of entry, denoted *entry*, and a single point of exit, denoted *exit*. We may think of these as no-op statements.

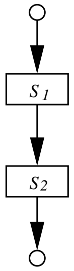
If v is a node in a CFG then $\mathit{pred}(v)$ denotes the set of predecessor nodes and $\mathit{succ}(v)$ the set of successor nodes.

Control Flow Graphs for Statements

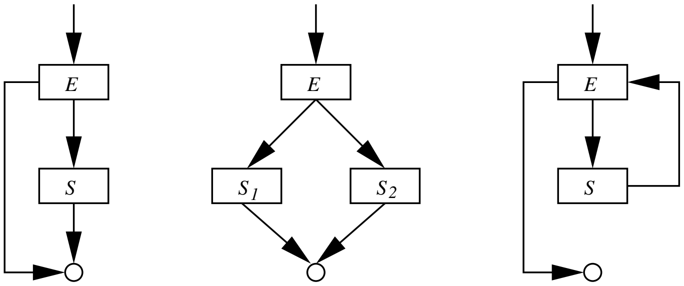
For now, we only consider simple statements, for which CFGs may be constructed in an inductive manner. The CFGs for assignments, `output`, `return` statements, and declarations look as follows:



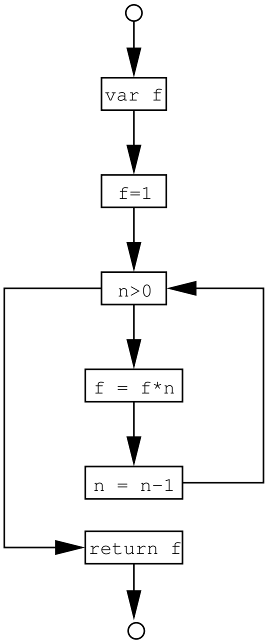
For the sequence S_1 S_2, we eliminate the exit node of S_1 and the entry node of S_2 and glue the statements together:



Similarly, the other control structures are modeled by inductive graph constructions:



Using this systematic approach, the iterative factorial function results in the following CFG:



We discuss control flow graphs for entire programs comprising multiple functions in Chapter 7.

Chapter 3

Type Analysis

Our programming language does not have explicit types, but of course the various operations are intended to be applied only to certain arguments. Specifically, the following restrictions seem reasonable:

- arithmetic operations and comparisons apply only to integers;
- only integers can be input and output of the main function;
- conditions in control structures must be integers;
- only functions can be called; and
- the unary * operator only applies to pointers.

We assume that their violation results in runtime errors. Thus, for a given program we would like to know that these requirements hold during execution. Since this is an interesting question, we immediately know that it is undecidable.

We resort to a conservative approximation: *typability*. A program is typable if it satisfies a collection of type constraints that is systematically derived from the syntax tree of the given program. This condition implies that the above requirements are guaranteed to hold during execution, but the converse is not true. Thus, our type-checker will be conservative and reject some programs that in fact will not violate any requirements during execution.

3.1 Types

We first define a language of *types* that will describe possible values:

$$
\begin{aligned}
\tau \rightarrow\ & \texttt{int} \\
\mid\ & \texttt{\&}\tau \\
\mid\ & \texttt{(}\tau\texttt{,...,}\tau\texttt{)->}\tau
\end{aligned}
$$

The type terms describe respectively integers, pointers, and function pointers. The grammar would normally generate *finite* types, but for recursive functions and data structures we need *regular* types. Those are defined as regular trees defined over the above constructors. Recall that a possibly infinite tree is regular if it contains only finitely many different subtrees.

Exercise 3.1: Show how regular types can be represented by finite automata so that two types are equal if their automata accept the same language.

3.2 Type Constraints

For a given program we generate a constraint system and define the program to be typable when the constraints are solvable. In our case we only need to consider equality constraints over regular type terms with variables. This class of constraints can be efficiently solved using the unification algorithm.

For each identifier *id* we introduce a type variable $[\![id]\!]$, and for each occurrence of a non-identifier expression E a type variable $[\![E]\!]$. Here, E refers to a concrete node in the syntax tree—not to the syntax it corresponds to. This makes our notation slightly ambiguous but simpler than a pedantically correct approach. (To avoid ambiguity, one could, for example, use the notation $[\![E]\!]_v$ where v is a unique ID of the syntax tree node.) Assuming that all declared identifiers are unique (see Exercise 2.1), there is no need to use different type variables for different occurrences of the same identifier.

The constraints are systematically defined for each construction in our language:

$intconst$:	$[\![intconst]\!] = \texttt{int}$
E_1 `op` E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \texttt{ op } E_2]\!] = \texttt{int}$
E_1`==`E_2:	$[\![E_1]\!] = [\![E_2]\!] \wedge [\![E_1\texttt{==}E_2]\!] = \texttt{int}$
`input`:	$[\![\texttt{input}]\!] = \texttt{int}$
$id = E$:	$[\![id]\!] = [\![E]\!]$
`output` E:	$[\![E]\!] = \texttt{int}$
`if (`E`)` S:	$[\![E]\!] = \texttt{int}$
`if (`E`)` S_1 `else` S_2:	$[\![E]\!] = \texttt{int}$
`while (`E`)` S:	$[\![E]\!] = \texttt{int}$
id`(`id_1`,`$\dots$`,`id_n`){` $\dots$`return` E`; }`:	$[\![id]\!] = \texttt{(}[\![id_1]\!]\texttt{,}\dots\texttt{,}[\![id_n]\!]\texttt{)->}[\![E]\!]$
id`(`E_1`,`$\dots$`,`E_n`)`:	$[\![id]\!] = \texttt{(}[\![E_1]\!]\texttt{,}\dots\texttt{,}[\![E_n]\!]\texttt{)->}[\![id\texttt{(}E_1\texttt{,}\dots\texttt{,}E_n\texttt{)}]\!]$
`(`E`)(`E_1`,`$\dots$`,`E_n`)`:	$[\![E]\!] = \texttt{(}[\![E_1]\!]\texttt{,}\dots\texttt{,}[\![E_n]\!]\texttt{)->}[\![\texttt{(}E\texttt{)(}E_1\texttt{,}\dots\texttt{,}E_n\texttt{)}]\!]$
$\&id$:	$[\![\&id]\!] = \&[\![id]\!]$
`malloc`:	$[\![\texttt{malloc}]\!] = \&\alpha$
`null`:	$[\![\texttt{null}]\!] = \&\alpha$
$*E$:	$[\![E]\!] = \&[\![*E]\!]$
$*id$=E:	$[\![id]\!] = \&[\![E]\!]$

In the above rules, each occurrence of α denotes a fresh type variable. Note that variable references and declarations do not yield any constraints and that parenthesized expression are not present in the abstract syntax.

All term constructors furthermore satisfy the general term equality axiom:

$$c(t_1, \ldots, t_n) = c'(t'_1, \ldots, t'n) \;\Rightarrow\; t_i = t'_i \text{ for each } i$$

where c is one of the term constructors, for example &.

Thus, a given program gives rise to a collection of equality constraints on type terms with variables.

Exercise 3.2: Explain each of the above type constraints.

A *solution* assigns to each type variable a type, such that all equality constraints are satisfied. The correctness claim for this algorithm is that the existence of a solution implies that the specified runtime errors cannot occur during execution.

3.3 Solving Constraints

If solutions exist, then they can be computed in almost linear time using the unification algorithm for regular terms. Since the constraints may also be extracted in linear time, the whole type analysis is quite efficient.

The complicated factorial program generates the following constraints, where duplicates are not shown:

⟦foo⟧ = (⟦p⟧,⟦x⟧)->⟦f⟧
⟦*p⟧ = int
⟦1⟧ = int
⟦p⟧ = &⟦*p⟧
⟦malloc⟧ = &α
⟦q⟧ = &⟦*q⟧
⟦f⟧ = ⟦(*p)*((x)(q,x))⟧
⟦(x)(q,x)⟧ = int
⟦input⟧ = int
⟦n⟧ = ⟦input⟧
⟦foo⟧ = (⟦&n⟧,⟦foo⟧)->⟦foo(&n,foo)⟧
⟦*p==0⟧ = int
⟦f⟧ = ⟦1⟧
⟦0⟧ = int
⟦q⟧ = ⟦malloc⟧
⟦q⟧ = &⟦(*p)-1⟧
⟦*p⟧ = int
⟦(*p)*((x)(q,x))⟧ = int
⟦x⟧ = (⟦q⟧,⟦x⟧)->⟦(x)(q,x)⟧
⟦main⟧ = ()->⟦foo(&n,foo)⟧
⟦&n⟧ = &⟦n⟧
⟦*p⟧ = ⟦0⟧

These constraints have a solution, where most variables are assigned int, except:

⟦p⟧ = &int
⟦q⟧ = &int
⟦malloc⟧ = &int
⟦x⟧ = ϕ
⟦foo⟧ = ϕ
⟦&n⟧ = &int
⟦main⟧ = ()->int

where ϕ is the regular type that corresponds to the infinite unfolding of:

$\phi = \mathtt{(\&int,}\phi\mathtt{)\text{-}>int}$

Exercise 3.3: Draw a picture of the unfolding of ϕ.

Since this solution exists, we conclude that our program is type correct. Recursive types are also required for data structures. The example program:

```
var p;
p = malloc;
*p = p;
```

creates the constraints:

$\llbracket \mathtt{p} \rrbracket = \&\alpha$
$\llbracket \mathtt{p} \rrbracket = \&\llbracket \mathtt{p} \rrbracket$

which has the solution $\llbracket \mathtt{p} \rrbracket = \psi$ where $\psi = \&\psi$. Some constraints admit infinitely many solutions. For example, the function:

```
poly(x) {
  return *x;
}
```

has type $\&\alpha\mathtt{\text{-}>}\alpha$ for any type α, which corresponds to the polymorphic behavior it displays.

3.4 Slack and Limitations

The type analysis is of course only approximate, which means that certain programs will be unfairly rejected. A simple example is:

```
bar(g,x) {
  var r;
  if (x==0) { r=g; } else { r=bar(2,0); }
  return r+1;
}

main() {
  return bar(null,1);
}
```

which never causes an error but is not typable since it among others generates constraints equivalent to:

$\mathtt{int} = \llbracket \mathtt{r} \rrbracket = \llbracket \mathtt{g} \rrbracket = \&\alpha$

which are clearly unsolvable.

Exercise 3.4: Explain the behavior of this program.

It is possible to use a more powerful polymorphic type analysis to accept the above program, but many other examples will inevitably remain rejected.

Another problem is that this type system ignores several other runtime errors, such as dereference of `null` pointers, reading of uninitialized variables, division by zero, and the more subtle *escaping stack cell* demonstrated by this program:

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

The problem is that `*p` denotes a stack cell that has *escaped* from the `baz` function. As we shall see, these problems can instead be handled by more ambitious static analyses.

Chapter 4

Lattice Theory

The technique for static analysis that we will study is based on the mathematical theory of *lattices*, which we briefly review in this chapter.

4.1 Example: Sign Analysis

As a motivating example, assume that we wish to design an analysis that can find out the possible signs of the integer values of variables and expressions in a given program. In concrete executions, values can be arbitrary integers. In contrast, our analysis considers an abstraction of the integer values by grouping them into the three categories, or *abstract values*: positive (+), negative (-), and zero (0). Similar to the analysis we considered in Chapter 3, we circumvent undecidability by introducing approximation. That is, the analysis must be prepared to handle uncertain information, in this case situations where it does not know the sign of some expression, so we add a special abstract value (?) representing “don't know”. We must also decide what information we are interested in for the cases where the sign of some expression is, for example, positive in some executions but not in others. For this example, let us assume we are interested in *definite* information, that is, the analysis should only report + for a given expression if it is certain that this expression will evaluate to a positive number in *every* execution of that expression and ? otherwise. In addition, it turns out to be beneficial to also introduce an abstract value $\bot$ for expressions whose values are not numbers (but instead, say, pointers) or have no value in any execution because they are unreachable from the program entry.

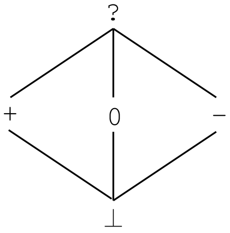
Consider this program:

```
var a,b,c;
a = 42;
b = 87;
```

```
if (input) {
  c = a + b;
} else {
  c = a - b;
}
```

Here, the analysis could conclude that `a` and `b` are positive numbers in all possible executions at the end of the program. The sign of `c` is either positive or negative depending on the concrete execution, so the analysis must report `?` for that variable.

Altogether we have an *abstract domain* consisting of the five abstract values $\{+, -, 0, ?, \bot\}$, which we can organize as follows with the least precise information at the top and the most precise information at the bottom:



The ordering reflects the fact that $\bot$ represents the empty set of integer values and `?` represents the set of all integer values.

This abstract domain is an example of a lattice. We continue the development of the sign analysis in Section 5.2, but we first need the mathematical foundation in place.

4.2 Lattices

A *partial order* is a set S equipped with a binary relation $\sqsubseteq$ where the following conditions are satisfied:

- reflexivity: $\forall x \in S : x \sqsubseteq x$
- transitivity: $\forall x, y, z \in S : x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- anti-symmetry: $\forall x, y \in S : x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$

When $x \sqsubseteq y$ we say that y is a *safe approximation* of x, or that x is *at least as precise* as y.

Let $X \subseteq S$. We say that $y \in S$ is an *upper bound* for X, written $X \sqsubseteq y$, if we have $\forall x \in X : x \sqsubseteq y$. Similarly, $y \in S$ is a *lower bound* for X, written $y \sqsubseteq X$, if $\forall x \in X : y \sqsubseteq x$. A *least* upper bound, written $\sqcup X$, is defined by:

$$X \sqsubseteq \sqcup X \land \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

Dually, a *greatest* lower bound, written $\sqcap X$, is defined by:

$$\sqcap X \sqsubseteq X \land \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

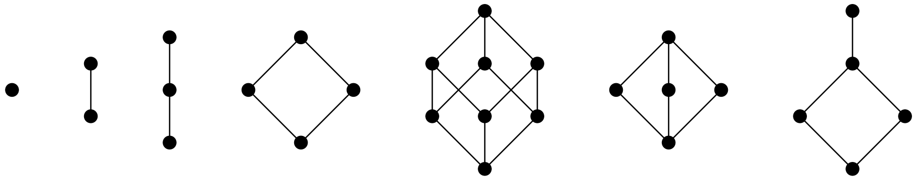
A *lattice* is a partial order in which $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$. (The literature typically calls this a *complete* lattice.) A lattice must have a unique *largest* element denoted $\top$ and a unique *smallest* element denoted $\bot$.

Exercise 4.1: Prove that $\sqcup S$ and $\sqcap S$ are the unique largest element and the unique smallest element, respectively, in S. In other words, we have $\top = \sqcup S$ and $\bot = \sqcap S$.

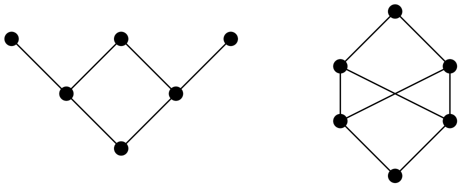
Exercise 4.2: Prove that $\sqcup S = \sqcap\emptyset$ and that $\sqcap S = \sqcup\emptyset$.

We will often look at *finite* lattices. For those the lattice requirements reduce to observing that $\bot$ and $\top$ exist and that every pair of elements x and y have a least upper bound written $x \sqcup y$ and a greatest lower bound written $x \sqcap y$.

A finite partial order may be illustrated by a Hasse diagram in which the elements are nodes and the order relation is the transitive closure of edges leading from lower to higher nodes. With this notation, all of the following partial orders are also lattices:

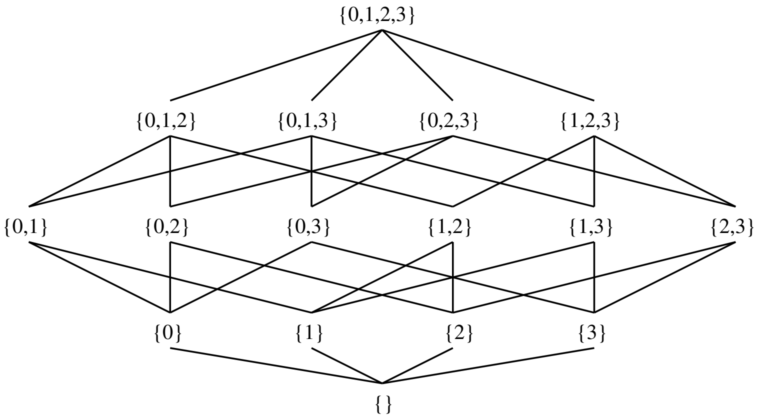


whereas these partial orders are *not* lattices:



Exercise 4.3: Why do these two diagrams not define lattices?

Every finite set A defines a lattice $(2^A, \subseteq)$, where $\bot = \emptyset$, $\top = A$, $x \sqcup y = x \cup y$, and $x \sqcap y = x \cap y$. We call this the *powerset lattice* for A. For a set with four elements, the powerset lattice looks like this:



The *height* of a lattice is defined to be the length of the longest path from $\perp$ to $\top$. For example, the above powerset lattice has height 4. In general, the lattice $(2^A, \subseteq)$ has height $|A|$.

4.3 Constructing Lattices

If $L_1, L_2, \ldots, L_n$ are lattices with finite height, then so is the *product*:

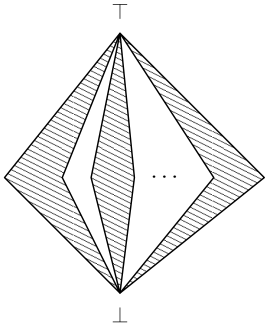
$$L_1 \times L_2 \times \ldots \times L_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in L_i\}$$

where $\sqsubseteq$ is defined pointwise. Note that $\sqcup$ and $\sqcap$ can be computed pointwise and that $\mathit{height}(L_1 \times \ldots \times L_n) = \mathit{height}(L_1) + \ldots + \mathit{height}(L_n)$.

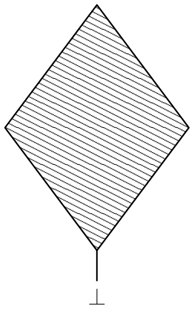
There is also a *sum* operator:

$$L_1 + L_2 + \ldots + L_n = \{(i, x_i) \mid x_i \in L_i \backslash \{\perp, \top\}\} \cup \{\perp, \top\}$$

where $\perp$ and $\top$ are as expected and $(i, x) \sqsubseteq (j, y)$ if and only if $i = j$ and $x \sqsubseteq y$. Note that $\mathit{height}(L_1 + \ldots + L_n) = \mathit{max}\{\mathit{height}(L_i)\}$. The sum operator can be illustrated as follows:

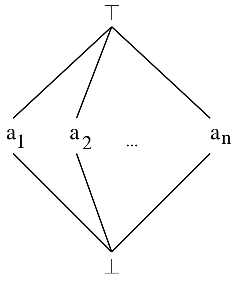


If L is a lattice, then so is $\mathit{lift}(L)$, which can be illustrated by:



and has $height(lift(L)) = height(L) + 1$ if L has finite height.

If A is a set (not necessarily a lattice), then $flat(A)$ illustrated by



is a lattice with height 2.

Exercise 4.4: Argue that $flat(A)$ can be viewed as a kind of sum lattice.

Finally, if A is a set and L is a lattice, then we obtain a *map* lattice:

$$A \mapsto L = \{[a_1 \mapsto x_1, \ldots, a_n \mapsto x_n] \mid x_i \in L\}$$

ordered pointwise: $f \sqsubseteq g \Leftrightarrow \forall a_i : f(a_i) \sqsubseteq g(a_i)$. If A is finite and L has finite height then $height(A \mapsto L) = |A| \cdot height(L)$.

Exercise 4.5: Argue that every product lattice is isomorphic to a map lattice.

Exercise 4.6: Verify the above claims about the heights of the lattices that are constructed.

The set $Sign = \{$+, -, 0, ?, $\perp\}$ with the ordering described in Section 4.1 forms a lattice that we use for describing abstract values in the sign analysis. An example of a map lattice is $StateSigns = Vars \mapsto Sign$ where $Vars$ is the set of variable names occurring in the program that we wish to analyze. Elements of this lattice describe abstract states that provide abstract values for all variables. An example of a product lattice is $ProgramSigns = StateSigns^n$ where n is the number of nodes in the CFG of the program. This lattice describes abstract states for all nodes in the program.

4.4 Equations and Fixed-Points

Continuing the sign analysis from Section 4.1, what are the signs of the variables at each line of the following program?

```
var a,b;        // 1
a = 42;         // 2
b = a + input; // 3
a = a - b;      // 4
```

We can derive a system of equations with one constraint variable for each program variable and line number from the program:

$$
\begin{aligned}
&a_1 = \mathtt{?}\\
&b_1 = \mathtt{?}\\
&a_2 = \mathtt{+}\\
&b_2 = b_1\\
&a_3 = a_2\\
&b_3 = a_2 \mathbin{\mathtt{+}} \mathtt{?}\\
&a_4 = a_3 \mathbin{\mathtt{-}} b_3\\
&b_4 = b_3
\end{aligned}
$$

The operators + and - here work on abstract values, which we return to in Section 5.2. In this constraint system, the constraint variables have values from the abstract value lattice $Sign$. We can alternatively derive the following equivalent equation system where each constraint variable instead has a value from the abstract state lattice $StateSigns$ from Section 4.3:

$$
\begin{aligned}
&x_1 = [\mathtt{a} \mapsto \mathtt{?}, \mathtt{b} \mapsto \mathtt{?}]\\
&x_2 = x_1[\mathtt{a} \mapsto \mathtt{+}]\\
&x_3 = x_2[\mathtt{b} \mapsto x_2(\mathtt{a}) \mathbin{\mathtt{+}} \mathtt{?}]\\
&x_4 = x_3[\mathtt{a} \mapsto x_3(\mathtt{a}) \mathbin{\mathtt{-}} x_3(\mathtt{b})]
\end{aligned}
$$

Notice that each equation only depends on preceding ones for this example program, so in this case the solution can be found by simple substition. However, mutually recursive equations may appear, for example, for programs that contain loops. We now show how to solve such equation systems in a general setting.

A function $f : L \to L$ is *monotone* when $\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Note that this property does not imply that f is *extensive*, that is, $\forall x \in L : x \sqsubseteq f(x)$; for example, the function that maps all inputs to $\bot$ is monotone but not extensive for lattices with more than one element.

Note that the composition of monotone functions is again monotone. Also, the definition of monotonicity generalizes naturally to functions with multiple arguments. Viewed as functions $\sqcup$ and $\sqcap$ are monotone.

We say that $x \in L$ is a *fixed-point* for f if $f(x) = x$. A *least* fixed-point x for f is a fixed-point for f where $x \sqsubseteq y$ for every fixed-point y for f.

Let L be a lattice with finite height. An *equation system* is of the form:

$$\begin{aligned}
&x_1 = F_1(x_1, \ldots, x_n)\\
&x_2 = F_2(x_1, \ldots, x_n)\\
&\vdots\\
&x_n = F_n(x_1, \ldots, x_n)
\end{aligned}$$

where x_i are variables and $F_i : L^n \to L$ is a collection of functions. If all the functions are monotone then the system has a unique least solution, which is obtained as the least fixed-point of the function $F : L^n \to L^n$ defined by:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

The central result we need is the *fixed-point theorem*. In a lattice L with finite height, every monotone function f has a unique least fixed-point given by:

$$\mathit{fix}(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

The proof of this theorem is quite simple. Observe that $\bot \sqsubseteq f(\bot)$ since $\bot$ is the least element. Since f is monotone, it follows that $f(\bot) \sqsubseteq f^2(\bot)$ and by induction that $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. Thus, we have an increasing chain:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

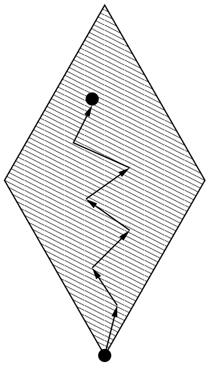
Since L is assumed to have finite height, we must for some k have that $f^k(\bot) = f^{k+1}(\bot)$. We define $\mathit{fix}(f) = f^k(\bot)$ and since $f(\mathit{fix}(f)) = f^{k+1}(\bot) = f^k(\bot) = \mathit{fix}(f)$, we know that $\mathit{fix}(f)$ is a fixed-point. Assume now that x is another fixed-point. Since $\bot \sqsubseteq x$ it follows that $f(\bot) \sqsubseteq f(x) = x$, since f is monotone and by induction we get that $\mathit{fix}(f) = f^k(\bot) \sqsubseteq x$. Hence, $\mathit{fix}(f)$ is the least fixed-point. By anti-symmetry, it is also unique.

The careful reader may have noticed that this is a *constructive* proof of the fixed-point theorem. We return to this in Section 5.1 where we consider the algorithm that can be inferred from the proof.

The time complexity of computing a fixed-point depends on three factors:

- the height of the lattice, since this provides a bound for k;
- the cost of computing f;
- the cost of testing equality.

The computation of a fixed-point can be illustrated as a walk up the lattice starting at $\bot$:



We can similarly solve systems of *inequations* of the form:

$$\begin{aligned}
&x_1 \sqsubseteq F_1(x_1,\ldots,x_n)\\
&x_2 \sqsubseteq F_2(x_1,\ldots,x_n)\\
&\vdots\\
&x_n \sqsubseteq F_n(x_1,\ldots,x_n)
\end{aligned}$$

by observing that the relation $x \sqsubseteq y$ is equivalent to $x = x \sqcap y$. Thus, solutions are preserved by rewriting the system into:

$$\begin{aligned}
&x_1 = x_1 \sqcap F_1(x_1,\ldots,x_n)\\
&x_2 = x_2 \sqcap F_2(x_1,\ldots,x_n)\\
&\vdots\\
&x_n = x_n \sqcap F_n(x_1,\ldots,x_n)
\end{aligned}$$

which is just a system of equations with monotone functions as before. Conversely, inequations of the form:

$$\begin{aligned}
&x_1 \sqsupseteq F_1(x_1,\ldots,x_n)\\
&x_2 \sqsupseteq F_2(x_1,\ldots,x_n)\\
&\vdots\\
&x_n \sqsupseteq F_n(x_1,\ldots,x_n)
\end{aligned}$$

can be rewritten into:

$$\begin{aligned}
&x_1 = x_1 \sqcup F_1(x_1,\ldots,x_n)\\
&x_2 = x_2 \sqcup F_2(x_1,\ldots,x_n)\\
&\vdots\\
&x_n = x_n \sqcup F_n(x_1,\ldots,x_n)
\end{aligned}$$

by observing that the relation $x \sqsupseteq y$ is equivalent to $x = x \sqcup y$.

Exercise 4.7: Show that $x \sqsubseteq y$ is equivalent to $x = x \sqcap y$.

Chapter 5

Dataflow Analysis with the Monotone Framework

Classical dataflow analysis, also called the *monotone framework*, starts with a CFGand a lattice L with finite height. The lattice may be fixed for all programs, or it may be *parameterized* with the given program.

To every node v in the CFG, we assign a variable $[\![v]\!]$ ranging over the elements of L. For each construction in the programming language, we then define a *dataflow constraint*, that relates the value of the variable of the corresponding node to those of other nodes (typically the neighbors).

As for type inference, we will ambiguously use the notation $[\![S]\!]$ for $[\![v]\!]$ if S is the syntax associated with v. The meaning will always be clear from the context.

We can systematically extract a collection of constraints over the variables for a given CFG. If all the constraints happen to be equations or inequations with monotone right-hand sides, then we can use the fixed-point algorithm to compute the unique least solution.

The dataflow constraints are *sound* if all solutions correspond to correct information about the program. The analysis is *conservative* since the solutions may be more or less imprecise, but computing the least solution will give the highest degree of precision.

5.1 Fixed-Point Algorithms

If the CFG has nodes $V = \{v_1, v_2, \ldots, v_n\}$, then we work in the lattice L^n. Assuming that node v_i generates the dataflow equation $[\![v_i]\!] = F_i([\![v_1]\!], \ldots, [\![v_n]\!])$, we construct the combined function $F : L^n \to L^n$ as described earlier:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

The naive algorithm that follows immediately from the fixed-point theorem is then to proceed as follows:

```
$x = (\bot, \ldots, \bot)$;
do {
 $t = x$;
 $x = F(x)$;
} while ($x \neq t$);
```

to compute the fixed-point x. Another algorithm, called *chaotic iteration*, exploits the fact that our lattice has the structure L^n to compute the fixed-point $(x_1, \ldots, x_n)$:

```
$x_1 = \bot$; $\ldots x_n = \bot$;
while ($\exists i : x_i \neq F_i(x_1, \ldots, x_n)$) {
 $x_i = F_i(x_1, \ldots, x_n)$;
}
```

The term "chaotic" comes from the fact that i is picked nondeterministically.

Exercise 5.1: Prove that chaotic iteration computes the least fixed-point of F.

Exercise 5.2: Assuming that we have a way to efficiently determine whether the loop condition holds, why is chaotic iteration better than the naive algorithm?

To obtain an efficient way to determine whether the loop condition holds in the chaotic iteration algorithm, we study further the structure of the individual constraints.

In the general case, every variable $[\![v_i]\!]$ depends on all other variables. Most often, however, an actual instance of F_i will only read the values of a few other variables. We represent this information as a map:

$$dep : V \to 2^V$$

which for each node v tells us the subset of other nodes for which $[\![v]\!]$ occurs in a nontrivial manner on the right-hand side of their dataflow equations. That is, $dep(v)$ is the set of nodes whose information may depend on the information of v. Armed with this information, we can present the *worklist* algorithm to compute the fixed-point $(x_1, \ldots, x_n)$:

```
$x_1 = \bot$; $\ldots x_n = \bot$;
$W = \{1, \ldots, n\}$;
while ($W \neq \emptyset$) {
 $i = W$.removeNext();
 $y = F_i(x_1, \ldots, x_n)$;
 if ($y \neq x_i$) {
  for ($v_j \in dep(v_i)$) $W$.add($j$);
```

```
    $x_i = y$;
  }
}
```

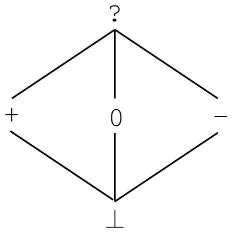
The set W is here called the worklist with operations `add` and `removeNext` for adding and (nondeterministically) removing an item.

> **Exercise 5.3**: Give an invariant that is strong enough to prove the correctness of the worklist algorithm.

Further improvements are possible. It may be beneficial to handle in separate turns the strongly connected components of the graph induced by the dep map, and the worklist set could be changed into a priority queue allowing us to exploit domain-specific knowledge about a particular dataflow problem.

5.2 Example: Sign Analysis, Revisited

Continuing the example from Section 4.1, we want to determine the sign (+,0,-) of all expressions. We start with the tiny lattice $Sign$ for describing abstract values:



The full lattice for our analysis is the map lattice:

$$Vars \mapsto Sign$$

where $Vars$ is the set of variables occurring in the given program. Each element of this lattice can be thought of as an abstract state. For each CFG node v we assign a variable $\llbracket v \rrbracket$ that denotes an abstract state that gives the sign values for all variables at the program point before the node.

The dataflow constraints model the effects on the abstract environments. For variable declarations we update accordingly:

$$\llbracket v \rrbracket = JOIN(v)\ [id_1 \mapsto \texttt{?}, \ldots, id_n \mapsto \texttt{?}]$$

For an assignment we use the constraint:

$$\llbracket v \rrbracket = JOIN(v)\ [id \mapsto eval(JOIN(v), E)]$$

and for all other nodes the constraint:

$$\llbracket v \rrbracket = JOIN(v)$$

where:

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

and $eval$ performs an abstract evaluation of expressions:

$$\begin{aligned}
&eval(\sigma, \mathit{id}) = \sigma(\mathit{id}) \\
&eval(\sigma, \mathit{intconst}) = sign(\mathit{intconst}) \\
&eval(\sigma, E_1 \; \mathsf{op} \; E_2) = \overline{\mathsf{op}}(eval(\sigma, E_1), eval(\sigma, E_2))
\end{aligned}$$

where σ is the current environment, $sign$ gives the sign of an integer constant and $\overline{\mathsf{op}}$ is an abstract evaluation of the given operator, defined by the following tables:

+	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	?
-	⊥	-	-	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

-	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	?
-	⊥	-	?	-	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0	0
-	⊥	0	+	-	?
+	⊥	0	-	+	?
?	⊥	0	?	?	?

/	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
-	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

>	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	?
-	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
-	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

It is not obvious that the right-hand sides of our constraints correspond to monotone functions. However, the $\sqcup$ operator and map updates clearly are, so it all comes down to monotonicity of the abstract operators on the sign lattice. This is best verified by a tedious manual inspection. Notice that for a lattice with n elements, monotonicity of an $n \times n$ table can be verified automatically in time $O(n^3)$.

Exercise 5.4: Describe the $O(n^3)$ algorithm for checking monotonicity of an operator given by an $n \times n$ table.

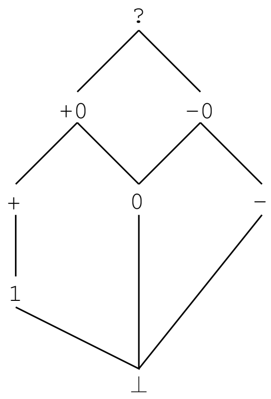
Exercise 5.5: Check that the above tables indeed define monotone operators on the $Sign$ lattice.

Exercise 5.6: Argue that these tables are the most precise possible for the *Sign* lattice, given that soundness must be preserved.

Exercise 5.7: Generate the equation system for the example program in Section 4.1, and then solve it using one of the fixed-point algorithms.

Exercise 5.8: Write a small program that leads to an equation system with mutually recursive constraints.

We lose some information in the above analysis, since for example the expression `(2>0)==1` is analyzed as `?`, which seems unnecessarily coarse. Also, `+/+` results in `?` rather than `+` since e.g. `1/2` is rounded down to zero. To handle these situations more precisely, we could enrich the sign lattice with element `1` (the constant 1), `+0` (positive or zero), and `-0` (negative or zero) to keep track of more precise abstract values:



and consequently describe the abstract operators by 8×8 tables.

Exercise 5.9: Define the six operators on the extended *Sign* lattice by means of 8×8 tables. Check that they are properly monotone.

The results of a sign analysis could in theory be used to eliminate division by zero errors by rejecting programs in which denominator expressions have sign `0` or `?`. However, the resulting analysis will probably unfairly reject too many programs to be practical, unless we add techniques such as path sensitivity (see Chapter 6).

5.3 Example: Liveness

A variable is *live* at a program point if its current value may be read during the remaining execution of the program. Clearly undecidable, this property can be approximated by a static analysis called liveness analysis (or live variables analysis).

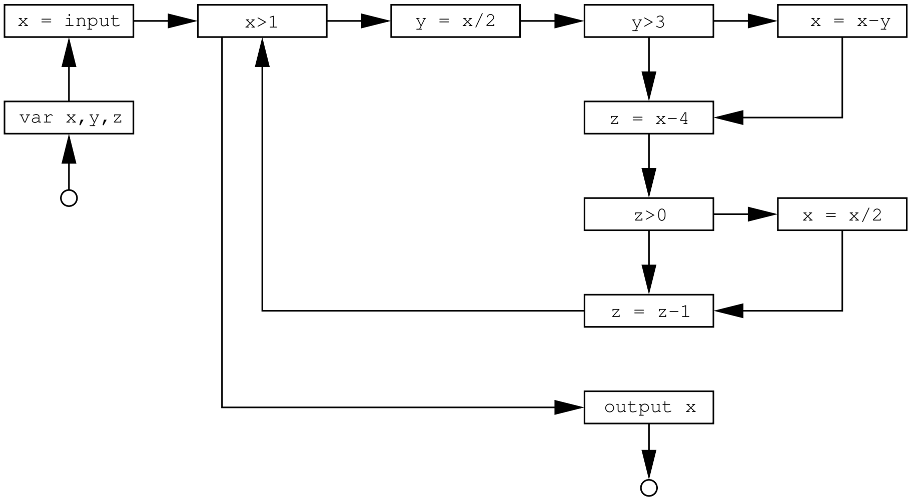
We use a powerset lattice where the elements are the variables occurring in the given program. This is an example of a *parameterized* lattice, that is, one that depends on the specific program being analyzed. For the example program:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice is thus:

$$L = (2^{\{\texttt{x},\texttt{y},\texttt{z}\}}, \subseteq)$$

The corresponding CFG looks as follows:



For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting the subset of program variables that are live at the program point *before* that node. The analysis wil be conservative, since the computed set may be too large. We use the auxiliary definition:

$$JOIN(v) = \bigcup_{w \in succ(v)} \llbracket w \rrbracket$$

For the exit node the constraint is:

$$\llbracket \mathit{exit} \rrbracket = \{\}$$

For conditions and `output` statements, the constraint is:

$$\llbracket v \rrbracket = JOIN(v) \cup vars(E)$$

For assignments, the constraint is:

$$\llbracket v \rrbracket = JOIN(v) \setminus \{\mathit{id}\} \cup vars(E)$$

For a variable declaration the constraint is:

$$\llbracket v \rrbracket = JOIN(v) \setminus \{\mathit{id}_1, \ldots, \mathit{id}_n\}$$

Finally, for all other nodes the constraint is:

$$\llbracket v \rrbracket = JOIN(v)$$

Here, $vars(E)$ denote the set of variables occurring in E. These constraints clearly have monotone right-hand sides.

Exercise 5.10: Argue that the right-hand sides of constraints define monotone functions.

The intuition is that a variable is live if it is read in the current node, or it is read in some future node unless it is written in the current node. Our example program yields these constraints:

$$\begin{array}{l}
\llbracket \texttt{var x,y,z} \rrbracket = \llbracket \texttt{x=input} \rrbracket \setminus \{\texttt{x}, \texttt{y}, \texttt{z}\} \\
\llbracket \texttt{x=input} \rrbracket = \llbracket \texttt{x>1} \rrbracket \setminus \{\texttt{x}\} \\
\llbracket \texttt{x>1} \rrbracket = (\llbracket \texttt{y=x/2} \rrbracket \cup \llbracket \texttt{output x} \rrbracket) \cup \{\texttt{x}\} \\
\llbracket \texttt{y=x/2} \rrbracket = (\llbracket \texttt{y>3} \rrbracket \setminus \{\texttt{y}\}) \cup \{\texttt{x}\} \\
\llbracket \texttt{y>3} \rrbracket = \llbracket \texttt{x=x-y} \rrbracket \cup \llbracket \texttt{z=x-4} \rrbracket \cup \{\texttt{y}\} \\
\llbracket \texttt{x=x-y} \rrbracket = (\llbracket \texttt{z=x-4} \rrbracket \setminus \{\texttt{x}\}) \cup \{\texttt{x,y}\} \\
\llbracket \texttt{z=x-4} \rrbracket = (\llbracket \texttt{z>0} \rrbracket \setminus \{\texttt{z}\}) \cup \{\texttt{x}\} \\
\llbracket \texttt{z>0} \rrbracket = \llbracket \texttt{x=x/2} \rrbracket \cup \llbracket \texttt{z=z-1} \rrbracket \cup \{\texttt{z}\} \\
\llbracket \texttt{x=x/2} \rrbracket = (\llbracket \texttt{z=z-1} \rrbracket \setminus \{\texttt{x}\}) \cup \{\texttt{x}\} \\
\llbracket \texttt{z=z-1} \rrbracket = (\llbracket \texttt{x>1} \rrbracket \setminus \{\texttt{z}\}) \cup \{\texttt{z}\} \\
\llbracket \texttt{output x} \rrbracket = \llbracket \mathit{exit} \rrbracket \cup \{\texttt{x}\} \\
\llbracket \mathit{exit} \rrbracket = \{\}
\end{array}$$

whose least solution is:

$[\![\mathit{entry}]\!] = \{\}$
$[\![\texttt{var x,y,z}]\!] = \{\}$
$[\![\texttt{x=input}]\!] = \{\}$
$[\![\texttt{x>1}]\!] = \{\texttt{x}\}$
$[\![\texttt{y=x/2}]\!] = \{\texttt{x}\}$
$[\![\texttt{y>3}]\!] = \{\texttt{x}, \texttt{y}\}$
$[\![\texttt{x=x-y}]\!] = \{\texttt{x}, \texttt{y}\}$
$[\![\texttt{z=x-4}]\!] = \{\texttt{x}\}$
$[\![\texttt{z>0}]\!] = \{\texttt{x}, \texttt{z}\}$
$[\![\texttt{x=x/2}]\!] = \{\texttt{x}, \texttt{z}\}$
$[\![\texttt{z=z-1}]\!] = \{\texttt{x}, \texttt{z}\}$
$[\![\texttt{output x}]\!] = \{\texttt{x}\}$
$[\![\mathit{exit}]\!] = \{\}$

From this information a clever compiler could deduce that y and z are never live at the same time, and that the value written in the assignment z=z-1 is never read. Thus, the program may safely be optimized into:

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

which saves the cost of one assignment and could result in better register allocation.

We can estimate the worst-case complexity of this analysis, using for example the naive algorithm from Section 5.1. We first observe that if the program has n CFG nodes and k variables, then the lattice $(2^{Vars})^n$ has height $k \cdot n$ which bounds the number of iterations we can perform. Each lattice element can be represented as a bitvector of length $k \cdot n$. For each iteration we have to perform $O(n)$ intersection, difference, or equality operations on sets of size k, which in all takes time $O(k \cdot n)$. Thus, the total time complexity is $O(k^2 \cdot n^2)$.

Exercise 5.11: What is the worst-case complexity of the liveness analysis if using the worklist algorithm?

5.4 Example: Available Expressions

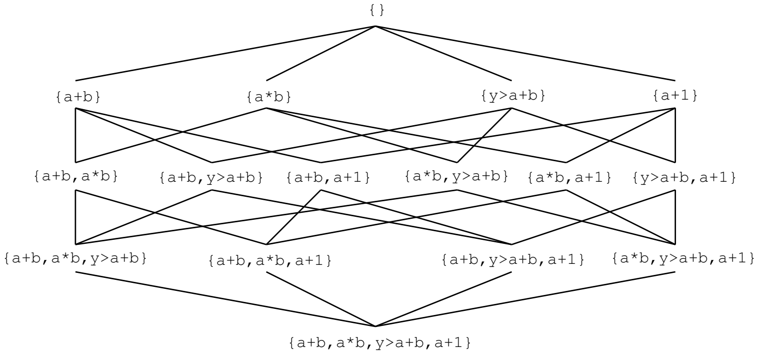
A nontrivial expression in a program is *available* at a program point if its current value has already been computed earlier in the execution. The set of available expressions for all program points can be approximated using a dataflow analysis. The lattice we use has as elements all expressions occurring in the program and is ordered by *reverse* subset inclusion. For this concrete program:

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

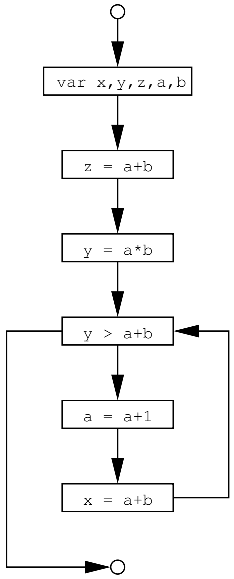
we have 4 different nontrivial expressions, so our lattice is:

$$L = (2^{\{\texttt{a+b,a*b,y>a+b,a+1}\}}, \supseteq)$$

which looks like:



The largest element of our lattice is $\emptyset$ which corresponds to the trivial information. The flow graph corresponding to the above program is:



For each CFG node v we introduce a constraint variable $\llbracket v \rrbracket$ ranging over L. Our intention is that it should contain the subset of expressions that are guaranteed always to be available at the program point after that node. For example, the expression a+b is available at the condition in the loop, but it is not available at the final assignment in the loop. Our analysis will be conservative since the computed set may be too small . The dataflow constraints are defined as follows, where we this time define:

$$\mathit{JOIN}(v) = \bigcap_{w \in pred(v)} \llbracket w \rrbracket$$

For the entry node we have the constraint:

$$\llbracket \mathit{entry} \rrbracket = \{\}$$

If v contains a condition *E* or the statement `output` *E*, then the constraint is:

$$\llbracket v \rrbracket = \mathit{JOIN}(v) \cup exps(E)$$

If v contains an assignment of the form *id*=*E*, then the constraint is:

$$\llbracket v \rrbracket = (\mathit{JOIN}(v) \cup exps(E)) \downarrow \mathit{id}$$

For all other kinds of nodes, the constraint is just:

$$\llbracket v \rrbracket = \mathit{JOIN}(v)$$

Here the function $\downarrow \mathit{id}$ removes all expressions that contain a reference to the variable *id*, and the $exps$ function is defined as:

$$\begin{aligned}
&\mathit{exps}(\mathit{intconst}) = \emptyset\\
&\mathit{exps}(\mathit{id}) = \emptyset\\
&\mathit{exps}(\mathit{input}) = \emptyset\\
&\mathit{exps}(E_1\texttt{op}E_2) = \{E_1\texttt{op}E_2\} \cup \mathit{exps}(E_1) \cup \mathit{exps}(E_2)
\end{aligned}$$

where op is any binary operator. The intuition is that an expression is available in v if it is available from all incoming edges or is computed in v, unless its value is destroyed by an assignment statement. Again, the right-hand sides of the constraints are monotone functions. For the example program, we then generate the following concrete constraints:

$$\begin{aligned}
&\llbracket \mathit{entry} \rrbracket\ = \{\}\\
&\llbracket \texttt{var x,y,z,a,b} \rrbracket\ = \llbracket \mathit{entry} \rrbracket\\
&\llbracket \texttt{z=a+b} \rrbracket\ = \mathit{exps}(\texttt{a+b}) \downarrow \texttt{z}\\
&\llbracket \texttt{y=a*b} \rrbracket\ = (\llbracket \texttt{z=a+b} \rrbracket \cup \mathit{exps}(\texttt{a*b})) \downarrow \texttt{y}\\
&\llbracket \texttt{y>a+b} \rrbracket\ = (\llbracket \texttt{y=a*b} \rrbracket \cap \llbracket \texttt{x=a+b} \rrbracket)\ \cup \mathit{exps}(\texttt{y>a+b})\\
&\llbracket \texttt{a=a+1} \rrbracket\ = (\llbracket \texttt{y>a+b} \rrbracket\ \cup \mathit{exps}(\texttt{a+1}))\downarrow \texttt{a}\\
&\llbracket \texttt{x=a+b} \rrbracket\ = (\llbracket \texttt{a=a+1} \rrbracket\ \cup \mathit{exps}(\texttt{a+b}))\downarrow \texttt{x}\\
&\llbracket \mathit{exit} \rrbracket\ = \llbracket \texttt{y>a+b} \rrbracket
\end{aligned}$$

Using the fixed-point algorithm, we obtain the minimal solution:

$$\begin{aligned}
&\llbracket \mathit{entry} \rrbracket\ = \{\}\\
&\llbracket \texttt{var x,y,z,a,b} \rrbracket\ = \{\}\\
&\llbracket \texttt{z=a+b} \rrbracket\ = \{\texttt{a+b}\}\\
&\llbracket \texttt{y=a*b} \rrbracket\ = \{\texttt{a+b}, \texttt{a*b}\}\\
&\llbracket \texttt{y>a+b} \rrbracket\ = \{\texttt{a+b}, \texttt{y>a+b}\}\\
&\llbracket \texttt{a=a+1} \rrbracket\ = \{\}\\
&\llbracket \texttt{x=a+b} \rrbracket\ = \{\texttt{a+b}\}\\
&\llbracket \mathit{exit} \rrbracket\ = \{\texttt{a+b}, \texttt{y>a+b}\}
\end{aligned}$$

which confirms our assumptions about a+b. Observe that the expressions available at the program point *before* a node v can be computed as $JOIN(v)$. With this knowledge, an optimizing compiler could systematically transform the program into a (slightly) more efficient version:

```
var x,y,z,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

while being guaranteed of preserving the semantics.

We can again estimate the worst-case complexity of the analysis. We first observe that if the program has n CFG nodes and k nontrivial expressions, then the lattice has height $k \cdot n$ which bounds the number of iterations we perform. Each lattice element can be represented as a bitvector of length k. For each iteration we have to perform $O(n)$ intersection, union, or equality operations which in all takes time $O(kn)$. Thus, the total time complexity is $O(k^2n^2)$.

5.5 Example: Very Busy Expressions

An expression is *very busy* if it will definitely be evaluated again before its value changes. To approximate this property, we need the same lattice and auxiliary functions as for available expressions. For every CFG node v the variable $[\![v]\!]$ denotes the set of expressions that at the program point before the node definitely are busy. We define:

$$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$

The constraint for the exit node is:

$$[\![exit]\!] = \{\}$$

For conditions and `output` statements we have:

$$[\![v]\!] = JOIN(v) \cup exps(E)$$

For assignments the constraint is:

$$[\![v]\!] = JOIN(v) \downarrow id \ \cup exps(E)$$

For all other nodes we have the constraint:

$$[\![v]\!] = JOIN(v)$$

The intuition is that an expression is very busy if it is evaluated in the current node or will be evaluated in all future executions unless an assignment changes its value. On the example program:

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

the analysis reveals that `a*b` is very busy inside the loop. The compiler can perform *code hoisting* and move the computation to the earliest program point where it is very busy. This would transform the program into the more efficient version:

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x>0) {
  output atimesb-x;
  x = x-1;
}
output atimesb;
```

5.6 Example: Reaching Definitions

The *reaching definitions* for a given program point are those assignments that may have defined the current values of variables. For this analysis we need a powerset lattice of all assignments (really CFG nodes) occurring in the program. For the example program from before:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice becomes:

$$L = (2^{\{\texttt{x=input,y=x/2,x=x-y,z=x-4,x=x/2,z=z-1}\}}, \subseteq)$$

For every CFG node v the variable $[\![v]\!]$ denotes the set of assignments that may define values of variables at the program point after the node. We define

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

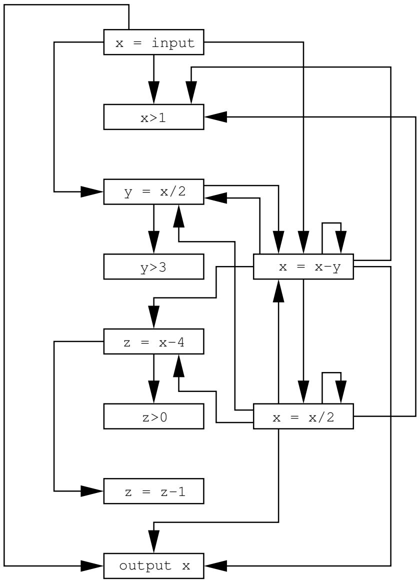
For assignments the constraint is:

$$[\![v]\!] = JOIN(v) {\downarrow} id \ \cup \{v\}$$

and for all other nodes it is simply:

$$[\![v]\!] = JOIN(v)$$

This time the $\downarrow id$ function removes all assignments to the variable *id*. This analysis can be used to construct a *def-use graph*, which is like a CFG except that edges go from definitions to possible uses. For the example program, the def-use graph is:



The def-use graph is a further abstraction of the program and is the basis of optimizations such as *dead code elimination* and *code motion*.

Exercise 5.12: Show that the def-use graph is always a subgraph of the transitive closure of the CFG.

5.7 Forwards, Backwards, May, and Must

The four classical analyses that we have seen so far can be classified in various ways. They are all just instances of the general monotone framework, but their constraints have a particular structure.

A *forwards* analysis is one that for each program point computes information about the *past* behavior. Examples of this are available expressions and reaching definitions. They can be characterized by the right-hand sides of constraints only depending on *predecessors* of the CFG node. Thus, the analysis starts at the *entry* node and moves forwards in the CFG.

A *backwards* analysis is one that for each program point computes information about the *future* behavior. Examples of this are liveness and very busy expressions. They can be characterized by the right-hand sides of constraints only depending on *successors* of the CFG node. Thus, the analysis starts at the *exit* node and moves backwards in the CFG.

A *may* analysis is one that describes information that may possibly be true and, thus, computes an *upper*/ approximation. Examples of this are liveness and reaching definitions. They can be characterized by the right-hand sides of constraints using a *union* operator to combine information.

A *must* analysis is one that describes information that must definitely be true and, thus, computes a *lower* approximation. Examples of this are available expressions and very busy expressions. They can be characterized by the right-hand sides of constraints using an *intersection* operator to combine information.

Thus, our four examples show every possible combination, as illustrated by this diagram:

	Forwards	*Backwards*
May	Reaching Definitions	Liveness
Must	Available Expressions	Very Busy Expressions

These classifications are mostly botanical in nature, but awareness of them may provide inspiration for constructing new analyses.

5.8 Example: Initialized Variables

Let us try to define an analysis that ensures that variables are initialized before they are read. This can be solved by computing for every program point the set of variables that are guaranteed to be initialized, thus our lattice is the reverse powerset of variables occurring in the given program. Initialization is a property of the past, so we need a forwards analysis. Also, we need definite information which implies a must analysis. This means that our constraints are phrased in terms of predecessors and intersections. On this basis, they more or less give themselves. For the entry node we have the constraint:

$$[\![entry]\!] = \{\}$$

for assignments we have the constraint:

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!] \cup \{id\}$$

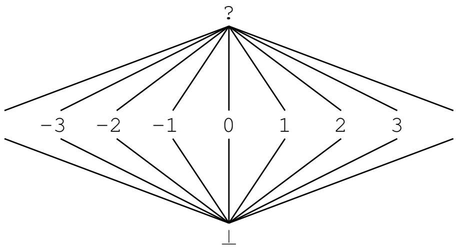
and for all other nodes the constraint:

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!]$$

The compiler could now check for every use of a variable that it is contained in the computed set of initialized variables.

5.9 Example: Constant Propagation

An analysis related to sign analysis is constant propagation, where we for every program point want to determine the variables that have a constant value. The analysis is structured just like the sign analysis, except that the basic lattice is replaced by:



and that operators are abstracted in the following manner for e.g. addition:

$\lambda n\lambda m.\texttt{if (}n=\bot\vee m=\bot\texttt{) }\{\bot\}\texttt{ else if (}n=?\vee m=?\texttt{) }\{?\}\texttt{ else }\{n{+}m\}$

Based on this analysis, an optimizing compiler could transform the program:

```
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (x < 0) { y = z-3; } else { y = 12; }
output y;
```

into:

```
var x,y,z;
x = 27;
y = input;
z = 54+y;
if (0) { y = z-3; } else { y = 12; }
output y;
```

which, following a reaching definitions analysis and a dead code elimination, can be reduced to:

```
var y;
y = input;
output 12;
```

5.10 Example: Interval Analysis

An *interval analysis* computes for every integer variable a lower and an upper bound for its possible values. Intervals are interesting analysis results, since sound answers can be used for optimizations such as array bounds checking, numerical overflow, and efficient integer representations.

This example involves a lattice of infinite height and we must use a special technique, called *widening* to ensure convergence towards a fixed-point. Increased precision may be obtained using a complementary technique, called *narrowing*.

The lattice describing a single variable is defined as:

$$\mathit{Interval} = \mathit{lift}(\{[l,h] \mid l,h \in N \ \wedge \ l \leq h\})$$

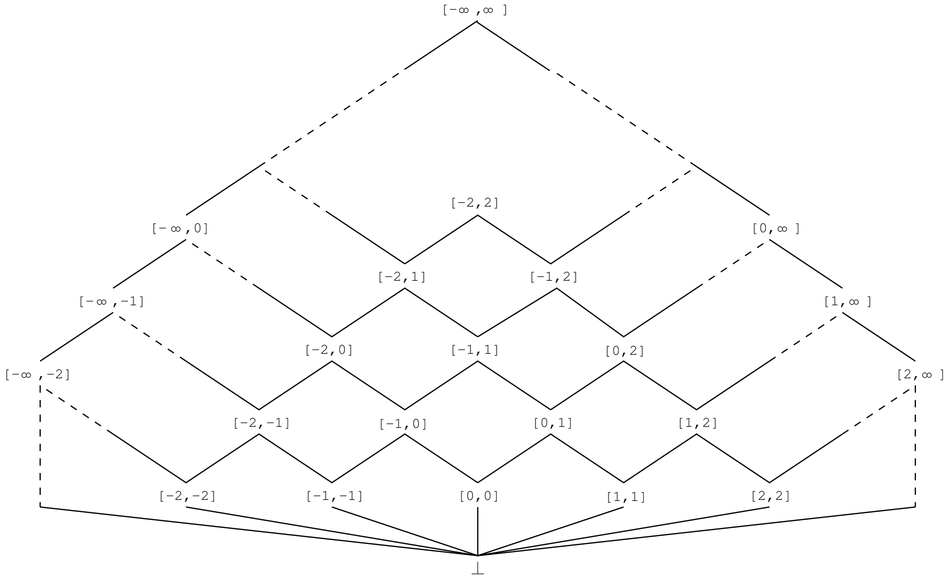
where:

$$N = \{-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty\}$$

is the set of integers extended with infinite endpoints and the order on intervals is:

$$[l_1, h_1] \sqsubseteq [l_2, h_2] \ \Leftrightarrow \ l_2 \leq l_1 \wedge h_1 \leq h_2$$

corresponding to inclusion of points. This lattice looks as follows:



It is clear that we do not have a lattice of finite height, since it contains for example the infinite chain:

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq [0,4] \sqsubseteq [0,5] \ldots$$

This carries over to the lattice we would ultimately use, namely:

$$L = \mathit{Vars} \mapsto \mathit{Interval}$$

where for the entry node we use the constant function returning the $\perp$ element:

$$[\![\mathit{entry}]\!] = \lambda x.\perp$$

for an assignment the constraint:

$$\llbracket v \rrbracket = \mathit{JOIN}(v)\ [\mathit{id} \mapsto \mathit{eval}(\mathit{JOIN}(v), E)]$$

and for all other nodes the constraint:

$$\llbracket v \rrbracket = \mathit{JOIN}(v)$$

where:

$$\mathit{JOIN}(v) = \bigsqcup_{w \in pred(v)} \llbracket w \rrbracket$$

and $eval$ performs an abstract evaluation of expressions:

$$\begin{aligned}
&eval(\sigma, \mathit{id}) = \sigma(\mathit{id})\\
&eval(\sigma, \mathit{intconst}) = [\mathit{intconst}, \mathit{intconst}]\\
&eval(\sigma, E_1\ \mathsf{op}\ E_2) = \overline{\mathsf{op}}(eval(\sigma, E_1), eval(\sigma, E_2))
\end{aligned}$$

where the abstract arithmetical operators all are defined by:

$$\overline{\mathsf{op}}([l_1, h_1], [l_2, h_2]) = [\min_{x \in [l_1, h_1], y \in [l_2, h_2]} x\ \mathsf{op}\ y, \max_{x \in [l_1, h_1], y \in [l_2, h_2]} x\ \mathsf{op}\ y]$$

For example, $\overline{+}([1, 10], [-5, 7]) = [1 - 5, 10 + 7] = [-4, 17]$.

Exercise 5.13: Argue that these definitions yield monotone operators on the $Interval$ lattice.

Exercise 5.14: Show how the abstract comparison operators can be made more precise.

The lattice has infinite height, so we are unable to use the monotone framework, since the fixed-point algorithm may never terminate. This means that for the lattice L^n the sequence of approximants:

$$F^i(\bot, \ldots, \bot)$$

need never converge. A powerful technique to address this kind of problem is introduced in the next section.

5.11 Widening

To obtain convergence of the interval analysis presented in Section 5.10 we shall use a technique called *widening* which introduces a function $w : L^n \to L^n$ so that the sequence:

$$(w \circ F)^i(\bot, \ldots, \bot)$$

now converges on a fixed-point that is larger than every $F^i(\bot, \ldots, \bot)$ and thus represents sound information about the program. The widening function w

will intuitively *coarsen* the information sufficiently to ensure termination. For our interval analysis, w is defined pointwise down to single intervals. It operates relatively to a fixed finite subset $B \subset N$ that must contain $-\infty$ and ∞. Typically, B could be seeded with all the integer constants occurring in the given program, but other heuristics could also be used. On a single interval we have:

$$w([l,h]) = [max\{i \in B \mid i \leq l\}, min\{i \in B \mid h \leq i\}]$$

which finds the best fitting interval among the ones that are allowed.

Exercise 5.15: Show that since w is an extensive monotone function and $w(Interval)$ is a finite lattice, the widening technique is guaranteed to work correctly.

5.12 Narrowing

Widening generally shoots above the target, but a subsequent technique called *narrowing* may improve the result. If we define:

$$\mathit{fix} = \bigsqcup F^i(\bot, \dots, \bot) \qquad \mathit{fixw} = \bigsqcup (w \circ F)^i(\bot, \dots, \bot)$$

then we have $\mathit{fix} \sqsubseteq \mathit{fixw}$. However, we also have that $\mathit{fix} \sqsubseteq F(\mathit{fixw}) \sqsubseteq \mathit{fixw}$, which means that a subsequent application of F may *refine* our result and still produce sound information. This technique, called *narrowing*, may in fact be iterated arbitrarily many times.

Exercise 5.16: Show that $\forall i : \mathit{fix} \sqsubseteq F^{i+1}(\mathit{fixw}) \sqsubseteq F^i(\mathit{fixw}) \sqsubseteq \mathit{fixw}$.

An example will demonstrate the benefits of these techniques. Consider this program:

```
y = 0; x = 7; x = x+1;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
}
```

Without widening, the analysis will produce the following diverging sequence of approximants for the program point after the loop:

$[\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,1]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,2]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,3]]$
$\vdots$

If we apply widening, based on the set $B = \{-\infty, 0, 1, 7, \infty\}$ seeded with the constants occurring in the program, then we obtain a converging sequence:

$$
\begin{array}{l}
[\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot] \\
[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,1]] \\
[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,7]] \\
[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,\infty]]
\end{array}
$$

However, the result for `x` is discouraging. Fortunately, a few iterations of narrowing refines the result to:

$$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,\infty]]$$

which is really the best we could hope for. Correspondingly, further narrowing has no effect. Note that the decreasing sequence:

$$\mathit{fixw} \sqsupseteq F(\mathit{fixw}) \sqsupseteq F^2(\mathit{fixw}) \sqsupseteq F^3(\mathit{fixw}) \ldots$$

is not guaranteed to converge, so heuristics must determine how many times to apply narrowing.

Chapter 6

Path Sensitivity

Until now, we have ignored the values of conditions by simply treating `if`- and `while`-statements as a nondeterministic choice between the two branches. This is called a *path insensitive* analysis as it does not distinguish different paths that lead to a given program point. This technique fails to include some information that could potentially be used in a static analysis. Consider for example the following program:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

The previous interval analysis (with widening) will conclude that after the `while`-loop the variable `x` is in the interval $[-\infty, \infty]$, `y` is in the interval $[0, \infty]$, and `z` is in the interval $[-\infty, \infty]$. However, in view of the conditionals being used, this result is too pessimistic.

6.1 Assertions

To exploit the available information, we shall extend the language with an artificial statement, `assert(`E`)`, where E is a boolean expression. This statement will abort execution at runtime if E is false and otherwise have no effect, however, we shall only insert it at places where E is guaranteed to be true. In the interval analysis, the constraints for these new statement will narrow the intervals for the various variables by exploiting information in conditionals.

In the example program, the meanings of the conditionals can be encoded by the following program transformation:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  assert(x > 0);
  z = z+x;
  if (17 > y) { assert(17 > y); y = y+1; }
  x = x-1;
}
assert(!(x > 0));
```

Constraints for a node v with an `assert` statement may trivially be given as:

$$[\![v]\!] = \mathit{JOIN}(v)$$

in which case no extra precision is gained. In fact, it requires insight into the specific static analysis to define non-trivial and sound constraints for these constructs.

For the interval analysis, extracting the information carried by general conditions, or *predicates*, such as E_1 `>` E_2 or E_1 `==` E_2 relative to the lattice elements is complicated and in itself an area of considerable study. For simplicity, let us consider conditions only of the two kinds *id* `>` E and E `>` *id*. The former kind of assertion can be handled by the constraint

$$[\![v]\!] = \mathit{JOIN}(v)[\mathit{id} \mapsto \mathit{gt}(\mathit{JOIN}(v)(\mathit{id}), \mathit{eval}(\mathit{JOIN}(v), E))]$$

where

$$\mathit{gt}([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2, \infty]$$

Exercise 6.1: Argue that this constraint for `assert` is sound and monotone.

Negated condition are handled in similar fashions, and all other conditions are given the trivial, but sound identity constraint.

With this refinement, the interval analysis of the above example will conclude that after the `while`-loop the variable `x` is in the interval $[-\infty..0]$, `y` is in the interval $[0, 17]$, and `z` is in the interval $[0, \infty]$.

Exercise 6.2: Discuss how more conditions may be given non-trivial constraints for `assert` to improve analysis precision further.

6.2 Branch Correlations

The use of `assert` statements at conditional branches provides a simple kind of path sensitivity called *control sensitivity*, however it is insufficient for reasoning about correlations of branches in programs. Here is a typical example:

```
 if (condition) {
  open();
  flag = 1;
} else {
  flag = 0;
}
...
if (flag) {
  close();
}
```

We here assume that `open` and `close` are built-in functions for opening and closing a specific file. The file is initially closed, and the "..." consists of statements that do not call `open` or `close` or modify `flag`. We wish to design an analysis that can check that `close` is only called if the file is currently open.

As a starting point, we use this lattice for modeling the open/closed status of the file:

$$L = (2^{\{\mathsf{open},\mathsf{closed}\}}, \subseteq)$$

For every CFG node v the variable $[\![v]\!]$ denotes the possible status of the file at the program point after the node. For `open` and `close` statements the constraints are:

$$[\![\texttt{open()}]\!] = \{\mathsf{open}\}$$

$$[\![\texttt{close()}]\!] = \{\mathsf{closed}\}$$

For the entry node, we define:

$$[\![\mathit{entry}]\!] = \{\mathsf{closed}\}$$

and for every other node, which does not modify the file status, the constraint is simply:

$$[\![v]\!] = \mathit{JOIN}(v)$$

where JOIN is defined as usual for a forward, may analysis:

$$\mathit{JOIN}(v) = \bigcup_{w \in \mathit{pred}(v)} [\![w]\!]$$

In the example program, the `close` function is clearly called if and only if `open` is called, but the current analysis fails to discover this.

Exercise 6.3: Write the constraints being produced for the example program and show that the solution for $[\![\texttt{flag}]\!]$ (the node for the last `if` condition) is $\{\mathsf{open}, \mathsf{closed}\}$.

Arguing that the program has the desired property obviously involves the `flag` variable, which the lattice above ignores. So, we can try with a slightly more sophisticated lattice – a product lattice that keeps track of both the status of the file and the value of the flag:

$$L' = (2^{\{\mathsf{open},\mathsf{closed}\}}, \subseteq) \times (2^{\{\texttt{flag}=0, \texttt{flag}\neq 0\}}, \subseteq)$$

Additionally, we insert `assert` to make sure that conditionals are not ignored:

```
if (condition) {
  assert(condition);
  open();
  flag = 1;
} else {
  assert(!condition);
  flag = 0;
}
...
if (flag) {
  assert(flag);
  close();
} else {
  assert(!flag);
}
```

This is still insufficient, though. At the program point after the first `if-else` statement, the analysis only knows that `open` *may* have been called and `flag` *may* be 0.

Exercise 6.4: Specify the constraints that fit with the L' lattice. Then show that the analysis produces the lattice element $(2^{\{\mathsf{open},\mathsf{closed}\}}, 2^{\{\texttt{flag}=0, \texttt{flag}\neq 0\}})$ for the first node after the the first `if-else` statement.

The present analysis is also called an *independent attribute analysis* as the abstract value of the file is independent of the abstract value of the boolean flag. What we need is a *relational* analysis that can keep track of relations between variables. This can be achieved by generalizing the analysis to maintain *multiple* abstract states per program point. If L is the original lattice as defined above, we replace it by

$$L'' = P \mapsto L$$

where P is a finite set of *path contexts*. A path context is here a predicate over the program state. (For instance, a condition expression in TIP defines such a predicate.) In general, each statement is then analyzed in $|P|$ different path contexts, each describing a set of paths that lead to the statement. For the example above, we can use $P = \{\texttt{flag} = 0, \texttt{flag} \neq 0\}$.

The constraints for `open`, `close`, and *entry* are:

$$[\![\texttt{open()}]\!] = \lambda p.\{\mathsf{open}\}$$

$$[\![\texttt{close()}]\!] = \lambda p.\{\mathsf{closed}\}$$

$$\llbracket \mathit{entry} \rrbracket = \lambda p.\{\textsf{closed}\}$$

The constraints for assignments make sure that `flag` gets special treatment:

$$\llbracket \texttt{flag = 0} \rrbracket = [\texttt{flag} = 0 \mapsto \bigcup_{p \in P} \mathit{JOIN}(v)(p),\ \texttt{flag} \neq 0 \mapsto \emptyset]$$

$$\llbracket \texttt{flag = } n \rrbracket = [\texttt{flag} \neq 0 \mapsto \bigcup_{p \in p} \mathit{JOIN}(v)(p),\ \texttt{flag} = 0 \mapsto \emptyset]$$

$$\llbracket \texttt{flag = } E \rrbracket = \lambda q. \bigcup_{p \in P} \mathit{JOIN}(v)(p)$$

Here, n is an *intconst* other than `0` and E is a non-*intconst* expression, and JOIN is defined pointwise:

$$\mathit{JOIN}(v)(p) = \bigcup_{w \in \mathit{pred}(v)} \llbracket w \rrbracket(p)$$

The situation $\llbracket v \rrbracket(p) = \emptyset$ corresponds to p being an infeasible path context at v. For `assert`, we also give special treatment to `flag`:

$$\llbracket \texttt{assert(flag)} \rrbracket = [\texttt{flag} \neq 0 \mapsto \mathit{JOIN}(v)(\texttt{flag} \neq 0),\ \texttt{flag} = 0 \mapsto \emptyset]$$

Notice the small but important difference from the constraint for the case `flag = 1`. As before, the case for negated expressions is similar.

Exercise 6.5: Give an appropriate constraint for `assert(!flag)`.

Finally, for any other node v, including other `assert` nodes, the constraint keeps the dataflow information for different path contexts apart but otherwise simply propagates and joins the information:

$$\llbracket v \rrbracket = \lambda p.\mathit{JOIN}(v)(p)$$

Although this is sound, we could make more precise constraints for `assert` nodes by recognizing other patterns that fit into the abstraction given by the lattice.

For our specific program, the following constraints are generated:

$$
\begin{array}{l}
\llbracket \mathit{entry} \rrbracket = \lambda p.\{\textsf{closed}\} \\
\llbracket \texttt{condition} \rrbracket = \llbracket \mathit{entry} \rrbracket \\
\llbracket \texttt{assert(condition)} \rrbracket = \llbracket \texttt{condition} \rrbracket \\
\llbracket \texttt{open()} \rrbracket = \lambda p.\{\textsf{open}\} \\
\llbracket \texttt{flag = 1} \rrbracket = \big[\texttt{flag} \neq 0 \mapsto \bigcup_{p \in P} \llbracket \texttt{open()} \rrbracket(p),\ \texttt{flag} = 0 \mapsto \emptyset\big] \\
\llbracket \texttt{assert(!condition)} \rrbracket = \llbracket \texttt{condition} \rrbracket \\
\llbracket \texttt{flag = 0} \rrbracket = \big[\texttt{flag} = 0 \mapsto \bigcup_{p \in P} \llbracket \texttt{assert(!condition)} \rrbracket(p),\ \texttt{flag} \neq 0 \mapsto \emptyset\big] \\
\llbracket \texttt{...} \rrbracket = \lambda p.\big(\llbracket \texttt{flag = 1} \rrbracket(p) \cup \llbracket \texttt{flag = 0} \rrbracket(p)\big) \\
\llbracket \texttt{flag} \rrbracket = \llbracket \texttt{...} \rrbracket
\end{array}
$$

$$
\begin{aligned}
&[\![\texttt{assert(flag)}]\!] = [\texttt{flag} \neq 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} \neq 0),\ \texttt{flag} = 0 \mapsto \emptyset]\\
&[\![\texttt{close()}]\!] = \lambda p.\{\mathsf{closed}\}\\
&[\![\texttt{assert(!flag)}]\!] = [\texttt{flag} = 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} = 0),\ \texttt{flag} \neq 0 \mapsto \emptyset]\\
&[\![\mathit{exit}]\!] = \lambda p.\big([\![\texttt{close()}]\!](p) \cup [\![\texttt{assert(!flag)}]\!](p)\big)
\end{aligned}
$$

The minimal solution is, for each $[\![v]\!](p)$:

	$\texttt{flag} = 0$	$\texttt{flag} \neq 0$
$[\![\mathit{entry}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$
$[\![\texttt{condition}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$
$[\![\texttt{assert(condition)}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$
$[\![\texttt{open()}]\!]$	$\{\mathsf{open}\}$	$\{\mathsf{open}\}$
$[\![\texttt{flag = 1}]\!]$	$\emptyset$	$\{\mathsf{open}\}$
$[\![\texttt{assert(!condition)}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$
$[\![\texttt{flag = 0}]\!]$	$\{\mathsf{closed}\}$	$\emptyset$
$[\![\texttt{...}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{open}\}$
$[\![\texttt{flag}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{open}\}$
$[\![\texttt{assert(flag)}]\!]$	$\emptyset$	$\{\mathsf{open}\}$
$[\![\texttt{close()}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$
$[\![\texttt{assert(!flag)}]\!]$	$\{\mathsf{closed}\}$	$\emptyset$
$[\![\mathit{exit}]\!]$	$\{\mathsf{closed}\}$	$\{\mathsf{closed}\}$

The analysis produces the lattice element $[\texttt{flag} = 0 \mapsto \{\mathsf{closed}\}, \texttt{flag} \neq 0 \mapsto \{\mathsf{open}\}]$ for the program point after the first `if-else` statement. The constraint for the `assert(flag)` statement will eliminate the possibility that the file is closed at that point. This ensures that `close` is only called if the file is open, as desired.

Exercise 6.6: For the present example, the basic lattice L is a defined as a powerset of a finite set A. Show that $P \mapsto 2^A$ is isomorphic to $2^{P \times A}$. (This explains why such analyses are called *relational*.)

Exercise 6.7: Describe a variant of the example program above where the present analysis would be improved if combining it with constant propagation.

In general, the program analysis designer is left with the choice of P. Often P consists of combinations of predicates that appear in conditionals in the program. This quickly results in an exponential blow-up: for k predicates, each statement may need to be analyzed in 2^k different path contexts. In practice, however, there is usually much redundancy in these analysis steps. Thus, in addition to the challenge of reasoning about the `assert` predicates relative to the lattice elements, it requires a considerable effort to avoid too many redundant computations in path sensitive analysis. One approach is *iterative refinement* where P is initially a single universal path context, which is then iteratively refined by adding relevant predicates until either the desired properties can be established or disproved or the analysis is unable to select relevant predicates and hence gives up.

Exercise 6.8: Assume that we change the rule for open from

$$[\![\texttt{open()}]\!] = \lambda p.\{\texttt{open}\}$$

to

$$[\![\texttt{open()}]\!] = \lambda p.\ \text{if } JOIN(v)(p) = \emptyset \text{ then } \emptyset \text{ else } \{\texttt{open}\}$$

Argue that this is sound and for some programs more precise than the original rule.

Exercise 6.9: The following is a variant of the previous example program:

```
if (condition) {
  flag = 1;
} else {
  flag = 0;
}
...
if (condition) {
  open();
}
...
if (flag) {
  close();
}
```

Show how a path sensitive analysis can prove for this variant that close is called if and only if open has been called.

Exercise 6.10: Construct yet another variant of the open/close example program where the desired property can only be established with a choice of P that includes a predicate that does *not* occur as a conditional expression in the program source. (Such a program may be challenging to handle with the iterative refinement technique.)

Chapter 7

Interprocedural Analysis

So far, we have only analyzed the body of a single function, which is called an *intraprocedural* analysis. We now consider *interprocedural* analysis of whole programs containing multiple functions and function calls.

7.1 Interprocedural Control Flow Graphs

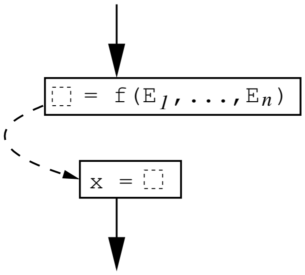
We use the subset of the TIP language containing functions, but still ignore pointers. As we shall see, the CFG for an entire program is then quite simple to obtain. It becomes more complicated when adding function pointers, which we discuss in later chapters.

First we construct the CFGs for all individual function bodies as usual. All that remains is then to glue them together to reflect function calls properly. We need to take care of parameter passing, return values, and values of local variables across calls. For simplicity we assume that all function calls are performed in connection with assignments:

```
x = f(E₁,...,Eₙ);
```

Exercise 7.1: Show how any program can be rewritten to have this form by introducing new temporary variables.

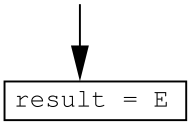
In the CFG, we represent each function call statement using *two* nodes: a *call node* representing the connection from the caller to the entry of f, and an *after-call node* for the exit of f back to the caller:



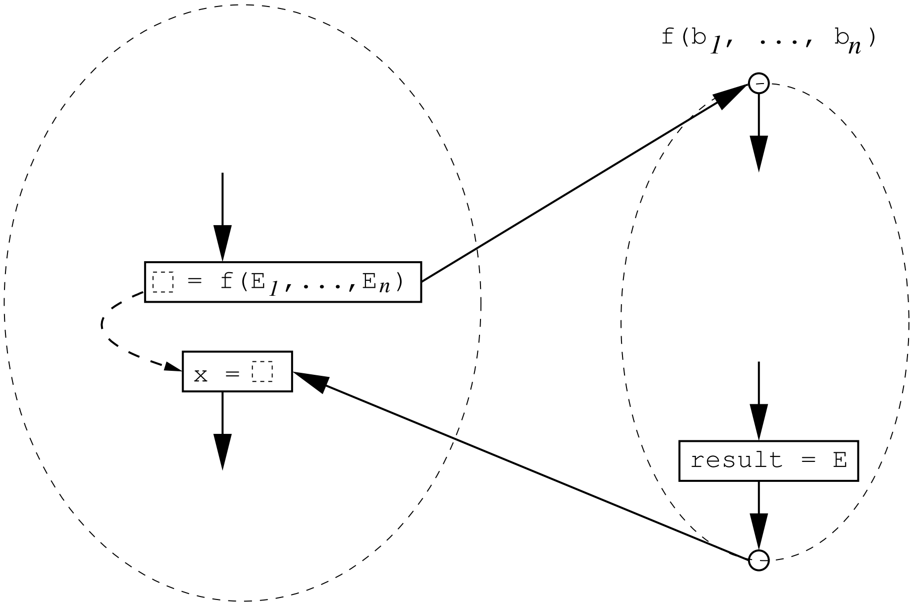
Next, we represent each `return` statement

```
return E;
```

as an assignment using a special variable named `result`:



We can now glue together the caller and the callee as follows:



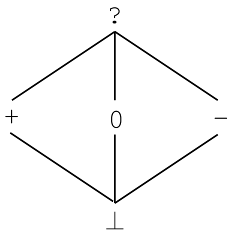
The connection between the call node and its after-call node is represented by a special edge, which we need for propagating abstract values for local variables of the caller.

With this interprocedural CFG in place, we can apply the monotone framework. Examples are given in the following sections.

Exercise 7.2: How many edges may the interprocedural CFG contain in a program with n CFG nodes?

7.2 Example: Interprocedural Sign Analysis

Recall the intraprocedural sign analysis from Sections 4.1 and 5.2. This analysis models values with the lattice *Sign*:



and abstract states are represented by the map lattice $Vars \mapsto Sign$.

To make the analysis interprocedural, we define constraints for function entries and exits. For an entry node v of a function `f(`$\mathtt{b}_1$`, ..., `$\mathtt{b}_n$`)` we consider the abstract states for all callers $pred(v)$ and model the passing of parameters $\mathtt{E}_1, \ldots, \mathtt{E}_n$:

$$[\![v]\!] \;= \bigsqcup_{w \in pred(v)} \bot[\mathtt{b}_1 \mapsto eval([\![w]\!], \mathtt{E}_1), \ldots, \mathtt{b}_n \mapsto eval([\![w]\!], \mathtt{E}_n)]$$

For an after-call node v that stores the return value in the variable `x` and where v' is the accompanying call node, the dataflow can be modeled by the following constraint:

$$[\![v]\!] \;= [\![v']\!][\mathtt{x} \mapsto [\![w]\!](\mathtt{result})] \quad \text{where } w \in pred(v)$$

We use the fact that an after-call node v has precisely one predecessor $w \in pred(v)$. The constraint obtains the abstract values of the local variables from the call node v' and the abstract value of `result` from w.

In a backward analysis, one would consider the call node and the function exit node rather than the function entry node and the after-call node. Also notice that we exploit the fact that the variant of the TIP language we consider here does not have global variables, a heap, nested functions, nor higher-order functions.

Chapter 8

Control Flow Analysis

If we introduce higher-order functions, objects, or function pointers into the programming language, then control flow and dataflow suddenly become intertwined. At each call site, it is no longer trivial to see which code is being called. The task of *control flow analysis* is to approximate conservatively the interprocedural control flow graph for such languages.

8.1 Closure Analysis for the λ-calculus

Control flow analysis in its purest form can best be illustrated by the classical λ-calculus:

$$\begin{aligned} E &\rightarrow \lambda id.E \\ &\;\mid\; id \\ &\;\mid\; E\;E \end{aligned}$$

and later we shall generalize this technique to the full TIP language. For simplicity we assume that all λ-bound variables are distinct. To construct a CFG for a term in this calculus, we need to compute for every expression *E* the set of *closures* to which it may evaluate. A closure can be modeled by a symbol of the form λid that identifies a concrete λ-abstraction. This problem, called *closure analysis*, can be solved using a variation of the monotone framework. However, since the intraprocedural control flow is trivial in this language, we might as well perform the analysis directly on the AST.

The lattice we use is the powerset of closures occurring in the given term ordered by subset inclusion. For every syntax tree node v we introduce a constraint variable $[\![v]\!]$ denoting the set of resulting closures. For an abstraction $\lambda id.E$ we have the constraint:

$$\lambda id \in [\![\lambda id.E]\!]$$

(the function may certainly evaluate to itself) and for an application E_1E_2 the *conditional* constraint:

$$\lambda id \in [\![E_1]\!] \Rightarrow \big([\![E_2]\!] \subseteq [\![id]\!] \;\wedge\; [\![E]\!] \subseteq [\![E_1E_2]\!]\big)$$

for every closure $\lambda id.E$ (the actual argument may flow into the formal argument, and the value of the function body is among the possible results of the function call).

Exercise 8.1: Show how the resulting constraints can be transformed into standard monotone inequations and solved by a fixed-point computation.

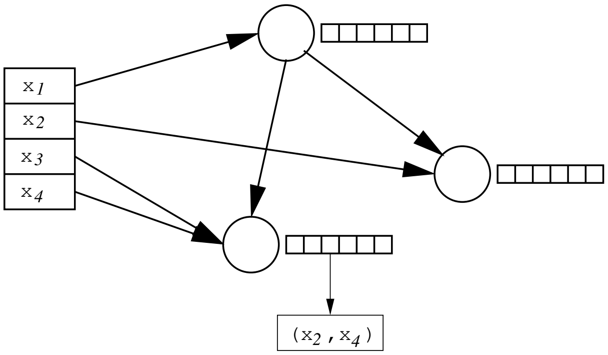
8.2 The Cubic Algorithm

The constraints for closure analysis are an instance of a general class that can be solved in cubic time. Many problems fall into this category, so we will investigate the algorithm more closely.

We have a finite set of *tokens* $\{t_1, \ldots, t_k\}$ and a finite set of *variables* $x_1, \ldots, x_n$ whose values are sets of tokens. Our task is to read a collection of *constraints* of the form $t \in x$ or $t \in x \Rightarrow y \subseteq z$ and produce the minimal solution.

Exercise 8.2: Show that a unique minimal solution exists, since solutions are closed under intersection.

The algorithm is based on a simple data structure. Each variable is mapped to a node in a directed acyclic graph (DAG). Each node has an associated bitvector belonging to $\{0, 1\}^k$, initially defined to be all 0's. Each bit has an associated list of pairs of variables, which is used to model conditional constraints. The edges in the DAG reflect inclusion constraints. The bitvectors will at all times directly represent the minimal solution. An example graph may look like:



Constraints are added one at a time. A constraint of the form $t \in x$ is handled by looking up the node associated with x and setting the corresponding bit to 1. If its list of pairs was not empty, then an edge between the nodes corresponding to y and z is added for every pair (y, z) and the list is emptied. A constraint of the form $t \in x \Rightarrow y \subseteq z$ is handled by first testing if the bit corresponding to t in the node corresponding to x has value 1. If this is so, then an edge between the nodes corresponding to y and z is added. Otherwise, the pair (y, z) is added to the list for that bit.

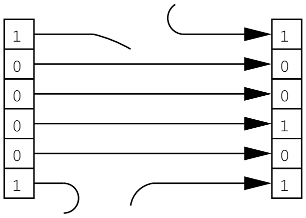
If a newly added edge forms a cycle, then all nodes on that cycle are merged into a single node, which implies that their bitvectors are unioned together and their pair lists are concatenated. The map from variables to nodes is updated accordingly. In any case, to reestablish all inclusion relations we must propagate the values of each newly set bit along all edges in the graph.

To analyze the time complexity this algorithm, we assume that the numbers of tokens and variables are both $O(n)$. This is clearly the case in closure analysis of a program of size n.

Merging DAG nodes on cycles can be done at most $O(n)$ times. Each merger involves at most $O(n)$ nodes and the union of their bitvectors is computed in time at most $O(n^2)$. The total for this part is $O(n^3)$.

New edges are inserted at most $O(n^2)$ times. Constant sets are included at most $O(n^2)$ times, once for each $t \in x$ constraint.

Finally, to limit the cost of propagating bits along edges, we imagine that each pair of corresponding bits along an edge are connected by a tiny bitwire. Whenever the source bit is set to 1, that value is propagated along the bitwire which then is broken:



Since we have at most n^3 bitwires, the total cost for propagation is $O(n^3)$. Adding up, the total cost for the algorithm is also $O(n^3)$. The fact that this seems like a lower bound as well is referred to as the *cubic time bottleneck*.

The kinds of constraints covered by this algorithm is a simple case of the more general *set constraints*, which allows richer constraints on sets of finite terms. General set constraints are also solvable but in time $O(2^{2^n})$.

8.3 Control Flow Graphs for Function Pointers

Consider now our tiny language where we allow functions pointers. For a computed function call:

$E \rightarrow$ (*E*)(*E*$_1$,...,*E*$_n$)

we cannot see from the syntax which functions may be called. A coarse but sound CFG could be obtained by assuming that *any* function with the right number of arguments could be called. However, we can do much better by performing a control flow analysis. Note that a function call $\mathit{id}\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$ may be seen as syntactic sugar for the general notation $\texttt{(}\mathit{id}\texttt{)(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$.

Our lattice is the powerset of the set of tokens containing *id* for every function name *id*, ordered by subset inclusion. For every syntax tree node v we introduce a constraint variable $[\![v]\!]$ denoting the set of functions or function pointers to which v could evaluate. For a constant function name *id* we have the constraint:

$$\mathit{id} \in [\![\mathit{id}]\!]$$

for assignments *id*=*E* we have the constraint:

$$[\![E]\!] \subseteq [\![\mathit{id}]\!]$$

and, finally, for computed function calls we have for every definition of a function f with arguments $a_1,\ldots,a_n$ and return expression E' the constraint:

$$f \in [\![E]\!] \;\Rightarrow\; \big([\![E_i]\!] \subseteq [\![a_i]\!] \;\wedge\; [\![E']\!] \subseteq [\![\texttt{(}E\texttt{)(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!]\big)$$

A still more precise analysis could be obtained if we restricted ourselves to typable programs and only generated constraints for those functions f for which the call would be type correct.

Given this inferred information, we construct the CFG as before but with edges between a call site and all possible target functions according to the control flow analysis. Consider the following example program:

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

The control flow analysis generates the following constraints:

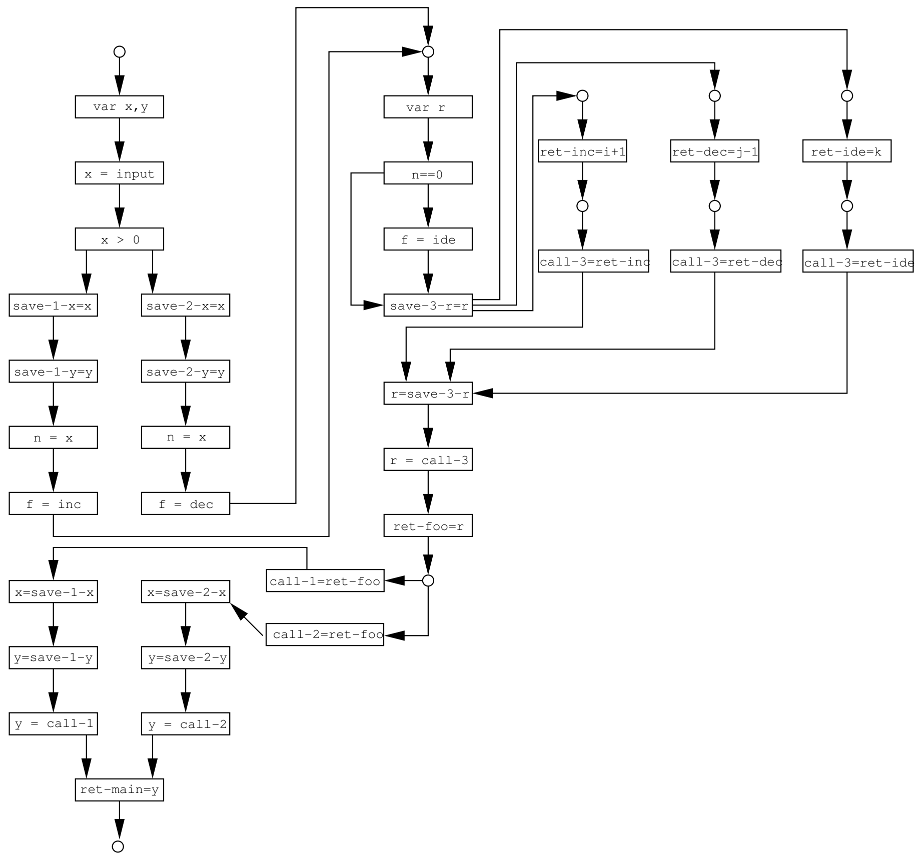
$\mathtt{inc} \in [\![\mathtt{inc}]\!]$
$\mathtt{dec} \in [\![\mathtt{dec}]\!]$

$$
\begin{array}{l}
\texttt{ide} \in \llbracket\texttt{ide}\rrbracket \\
\llbracket\texttt{ide}\rrbracket \subseteq \llbracket\texttt{f}\rrbracket \\
\llbracket\texttt{(f)(n)}\rrbracket \subseteq \llbracket\texttt{r}\rrbracket \\
\texttt{inc} \in \llbracket\texttt{f}\rrbracket \Rightarrow \llbracket\texttt{n}\rrbracket \subseteq \llbracket\texttt{i}\rrbracket \ \wedge\ \llbracket\texttt{i+1}\rrbracket \subseteq \llbracket\texttt{(f)(n)}\rrbracket \\
\texttt{dec} \in \llbracket\texttt{f}\rrbracket \Rightarrow \llbracket\texttt{n}\rrbracket \subseteq \llbracket\texttt{j}\rrbracket \ \wedge\ \llbracket\texttt{j-1}\rrbracket \subseteq \llbracket\texttt{(f)(n)}\rrbracket \\
\texttt{ide} \in \llbracket\texttt{f}\rrbracket \Rightarrow \llbracket\texttt{n}\rrbracket \subseteq \llbracket\texttt{k}\rrbracket \ \wedge\ \llbracket\texttt{k}\rrbracket \subseteq \llbracket\texttt{(f)(n)}\rrbracket \\
\llbracket\texttt{input}\rrbracket \subseteq \llbracket\texttt{x}\rrbracket \\
\llbracket\texttt{foo(x,inc)}\rrbracket \subseteq \llbracket\texttt{y}\rrbracket \\
\llbracket\texttt{foo(x,dec)}\rrbracket \subseteq \llbracket\texttt{y}\rrbracket \\
\texttt{foo} \in \llbracket\texttt{foo}\rrbracket \\
\texttt{foo} \in \llbracket\texttt{foo}\rrbracket \Rightarrow \llbracket\texttt{x}\rrbracket \subseteq \llbracket\texttt{n}\rrbracket \ \wedge\ \llbracket\texttt{inc}\rrbracket \subseteq \llbracket\texttt{f}\rrbracket \ \wedge\ \llbracket\texttt{r}\rrbracket \subseteq \llbracket\texttt{foo(x,inc)}\rrbracket \\
\texttt{foo} \in \llbracket\texttt{foo}\rrbracket \Rightarrow \llbracket\texttt{x}\rrbracket \subseteq \llbracket\texttt{n}\rrbracket \ \wedge\ \llbracket\texttt{dec}\rrbracket \subseteq \llbracket\texttt{f}\rrbracket \ \wedge\ \llbracket\texttt{r}\rrbracket \subseteq \llbracket\texttt{foo(x,dec)}\rrbracket
\end{array}
$$

The nonempty values of the least solution are:

$$
\begin{array}{l}
\llbracket\texttt{inc}\rrbracket = \{\texttt{inc}\} \\
\llbracket\texttt{dec}\rrbracket = \{\texttt{dec}\} \\
\llbracket\texttt{ide}\rrbracket = \{\texttt{ide}\} \\
\llbracket\texttt{f}\rrbracket = \{\texttt{inc}, \texttt{dec}, \texttt{ide}\} \\
\llbracket\texttt{foo}\rrbracket = \{\texttt{foo}\}
\end{array}
$$

On this basis, we can construct the following monovariant interprocedural CFG for the program:



which then can be used as basis for subsequent interprocedural static analyses.

8.4 Control Flow in Object Oriented Languages

A language with function pointers or higher-order functions must use the kind of control flow analysis described in the previous sections to obtain a reasonably precise CFG. For common object-oriented languages, such as Java or C#, it is also useful, but the added structure provided by the class hierarchy and the type system permits some simpler alternatives. In the object-oriented setting the question is which method implementations may be executed at a given method invocation site:

```
x.m(a,b,c)
```

The simplest solution is to scan the class library and select any method named `m` whose signature accepts the types of the actual arguments. A better choice, called *Class Hierarchy Analysis (CHA)*, is to consider only the part of the class hierarchy that is spanned by the declared type of `x`. A further refinement, called

Rapid Type Analysis (RTA), is to restrict further to the classes of which objects are actually allocated. Yet another technique, called *Variable Type Analysis (VTA)*, performs *intraprocedural* control flow analysis while making conservative assumptions about the remaining program.

These techniques are of course much faster than full-blown control flow analysis, and for real-life programs they are often sufficiently precise.

Chapter 9

Pointer Analysis

The final extension of the TIP language introduces simple pointers and dynamic memory. Since our toy version of `malloc` only allocates a single cell, we cannot build arbitrary structures in the heap. However, the main problems with pointers are amply represented in the language fragment that we consider.

9.1 Points-To Analysis

The most important information that must be obtained is the set of possible cells that the pointers may point to. There are of course arbitrarily many possible locations during execution, so we must select some finite representatives. The canonical choice is to introduce an abstract cell *id* for every variable named *id* and an abstract cell `malloc-i`, where `i` is a unique index, for each occurrence of a `malloc` operation in the program. We use $Cell$ to denote the set of such cells for the given program, and we use Loc to denote the set of abstract locations of the cells, written &$c \in Loc$ for every $c \in Cell$.

The first points-to analyses that we shall study are flow-insensitive. The end result of such an analysis is a function pt that for each pointer variable p returns the set $pt(p)$ of cells it may point to. We must of course perform a conservative analysis, so these sets will in general be too large.

Given this information, many other facts can be approximated. If we wish to know whether pointer variables p and q *may* be aliases, then a safe answer is obtained by checking whether $pt(p) \cap pt(q)$ is nonempty.

An almost-trivial analysis, called *address taken*, is to use all possible cells, except that *id* is only included if the expression &*id* occurs in the given program. This only suffices for very simple applications, so more ambitious approaches are usually preferred. If we restrict ourselves to typable programs, then any points-to analysis could be improved by removing those locations whose types are not equal to that of the pointer variable.

9.2 Andersen's Algorithm

One approach to points-to analysis is quite similar to control flow analysis. For each cell *c* we introduce a constraint variable $[\![c]\!]$ ranging over sets of locations.

The analysis assumes that the program has been normalized so that every pointer manipulation is of one of the six kinds:

1) *id* = `malloc`
2) id_1 = &id_2
3) id_1 = id_2
4) id_1 = *id_2
5) *id_1 = id_2
6) *id* = `null`

> **Exercise 9.1**: Show how this normalization can be performed systematically by introducing fresh temporary variables.

For each of these pointer manipulations we then generate constraints:

$$\begin{array}{rl}
id\ \texttt{= malloc}: & \texttt{\&malloc-i} \in [\![id]\!] \\
id_1\ \texttt{=}\ \&id_2: & \&id_2 \in [\![id_1]\!] \\
id_1\ \texttt{=}\ id_2: & [\![id_2]\!] \subseteq [\![id_1]\!] \\
id_1\ \texttt{=}\ {*}id_2: & \&\alpha \in [\![id_2]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![id_1]\!] \\
{*}id_1\ \texttt{=}\ id_2: & \&\alpha \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![\alpha]\!]
\end{array}$$

The last two constraints are generated for every location $\&\alpha \in Loc$, but we need in fact only consider those whose addresses are actually taken in the given program. The `null` assignment is ignored, since it corresponds to the trivial constraint $\emptyset \subseteq [\![id]\!]$. Since these constraints match the requirements of the cubic algorithm, they can be solved in time $O(n^3)$. The resulting points-to function is defined as:

$$pt(p) = \{\alpha \in Cell \mid \&\alpha \in [\![p]\!]\}$$

Consider the following example program:

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Andersen's algorithm generates these constraints:

$\&\texttt{malloc-1} \in [\![\texttt{p}]\!]$
$[\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$
$[\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!]$
$\&\alpha \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\texttt{z}]\!] \subseteq [\![\alpha]\!]$
$[\![\texttt{q}]\!] \subseteq [\![\texttt{p}]\!]$
$\&\texttt{y} \in [\![\texttt{q}]\!]$
$\&\alpha \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\alpha]\!] \subseteq [\![\texttt{x}]\!]$
$\&\texttt{z} \in [\![\texttt{p}]\!]$

The nonempty values in the least solution are:

$pt(\texttt{p}) = \{\texttt{malloc-1}, \texttt{y}, \texttt{z}\}$
$pt(\texttt{q}) = \{\texttt{y}\}$

which gives a quite precise result. Note that while this algorithm is flow insensitive, the directionality of the constraints implies that the dataflow is still modeled with some accuracy.

9.3 Steensgaard's Algorithm

A popular alternative is Steensgaard's algorithm, which performs a coarser analysis essentially by viewing assignments as being bidirectional. The analysis can be expressed elegantly using term unification. We use a term variable $[\![c]\!]$ for every cell c and a term constructor $\&t$ representing a pointer to t. (Notice the change in notation compared to Section 9.2: here, $[\![c]\!]$ is a term variable and does not directly denote a set of locations.)

$$\begin{array}{rl}
id\ \texttt{= malloc}: & [\![id]\!] = \&[\![\texttt{malloc-i}]\!] \\
id_1\ \texttt{=}\ \&id_2: & [\![id_1]\!] = \&[\![id_2]\!] \\
id_1\ \texttt{=}\ id_2: & [\![id_1]\!] = [\![id_2]\!] \\
id_1\ \texttt{=}\ {*}id_2: & [\![id_2]\!] = \&\alpha \;\wedge\; [\![id_1]\!] = \alpha \\
{*}id_1\ \texttt{=}\ id_2: & [\![id_1]\!] = \&\alpha \;\wedge\; [\![id_2]\!] = \alpha
\end{array}$$

Each α denotes a fresh term variable.

As usual, term constructors satisfy the general term equality axiom:

$$\&\alpha_1 = \&\alpha_2 \;\Rightarrow\; \alpha_1 = \alpha_2$$

The resulting points-to function is defined as:

$$pt(p) = \{t \in Cell \mid [\![p]\!] = \&[\![t]\!]\}$$

For the previous example program, Steensgaard's algorithm generates the following constraints:

$[\![\texttt{p}]\!] = \&[\![\texttt{malloc-1}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{z}]\!]$

$$\begin{array}{ll}
[\![\mathtt{p}]\!] = \&\alpha_1 & [\![\mathtt{z}]\!] = \alpha_1 \\
[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!] & \\
[\![\mathtt{q}]\!] = \&[\![\mathtt{y}]\!] & \\
[\![\mathtt{x}]\!] = \alpha_2 & [\![\mathtt{p}]\!] = \&\alpha_2 \\
[\![\mathtt{p}]\!] = \&[\![\mathtt{z}]\!] &
\end{array}$$

This in turn implies that:

$$pt(\mathtt{p}) = pt(\mathtt{q}) = \{\mathtt{malloc\text{-}1}, \mathtt{y}, \mathtt{z}\}$$

which is less precise than Andersen's algorithm, but using the faster algorithm.

9.4 Interprocedural Points-To Analysis

If function pointers are distinguished from other pointers, then we can perform an interprocedural points-to analysis by first computing an interprocedural CFG as described earlier and then running either Andersen's or Steensgaard's algorithm. If, however, function pointers may have indirect references as well then we need to perform the control flow analysis and the points-to analysis simultaneously to resolve for example the function call:

```
(***x)(1,2,3);
```

To express the combined algorithm, we make the syntactic simplification that all function calls are of the form:

id_1 `=` `(`id_2`)(`a_1`,`$\ldots$`,`a_n`);`

where id_i and a_i are variables. Similarly, all return expressions are assumed to be just variables.

Exercise 9.2: Show how to perform these simplifications in a systematic manner.

Andersen's algorithm is already similar to control flow analysis, and it can simply be extended with the appropriate constraints. A reference to a constant function f generates the constraint:

$$f \in [\![f]\!]$$

The computed function call generates the constraint

$$f \in [\![id_2]\!] \Rightarrow \big([\![a_1]\!] \subseteq [\![x_1]\!] \ \wedge\ \ldots\ \wedge\ [\![a_n]\!] \subseteq [\![x_n]\!] \ \wedge\ [\![id]\!] \subseteq [\![id_1]\!]\big)$$

for every occurrence of a function definition

f`(`x_1`,`$\ldots$`,`x_n`)` `{`$\ldots$`return` id`;` `}`

This will maintain the precision of the control flow analysis.

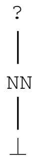
9.5 Example: Null Pointer Analysis

We are now also able to define an analysis that detects `null` dereferences. Specifically, we want to ensure that **id* is only executed when *id* is not `null`. Let us consider intraprocedural analysis, so we can ignore function calls.

As before, we assume that the program is normalized, so that all pointer manipulations are of these kinds:

1) id = `malloc`
2) id_1 = &id_2
3) id_1 = id_2
4) id_1 = *id_2
5) *id_1 = id_2
6) id = `null`

The basic lattice we use, called $Null$, is:



where `NN` means *definitely not* `null` and `?` represents values that may be `null`. We then form the following map lattice:

$$Cell \mapsto Null$$

For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting an element from the map lattice. We shall use each constraint variable to describe an abstract state for the program point immediately after the node.

For all nodes that do not involve pointer operations we have the constraint:

$$[\![v]\!] = JOIN(v)$$

where

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

For a heap load operation id_1 = *id_2 we need to model the change of the program variable id_1. Our abstraction has a single abstract cell for id_1. With the assumption of intraprocedural analysis, that abstract cell represents a single concrete cell. (With an interprocedural analysis, we would need to take into account that each stack frame at runtime has an instance of the variable.) For the expression *id_2 we can ask the points-to analysis for the possible cells $pt(id_2)$. With these observations, we can give a constraint for heap load operations:

$$id_1 \texttt{ = *}id_2\texttt{:} \quad [\![v]\!] = load(JOIN(v), id_1, id_2)$$

where

$$load(\sigma, id_1, id_2) = \sigma[id_1 \mapsto \bigsqcup_{\alpha \in pt(id_2)} \sigma(\alpha)]$$

Similar reasoning gives constraints for the other operations that affect pointer variables:

$$\begin{aligned}
id\ \texttt{= malloc}: \quad & [\![v]\!] = JOIN(v)[id \mapsto \texttt{NN}, \texttt{malloc-i} \mapsto \texttt{?}] \\
id_1\ \texttt{= \&}id_2: \quad & [\![v]\!] = JOIN(v)[id_1 \mapsto \texttt{NN}] \\
id_1\ \texttt{=}\ id_2: \quad & [\![v]\!] = JOIN(v)[id_1 \mapsto JOIN(v)(id_2)] \\
id\ \texttt{= null}: \quad & [\![v]\!] = JOIN(v)[id \mapsto \texttt{?}]
\end{aligned}$$

Exercise 9.3: Explain why the above four constraints are monotone and sound.

For a heap store operation *id_1 = id_2 we need to model the change of whatever id_1 points to. That may be multiple abstract cells, namely t$pt(id_1)$. Moreover, each abstract heap cell `malloc-i` may describe multiple concrete cells. In the constraint for heap store operations, we must therefore join the new abstract value into the existing one for each affected cell in $pt(id_1)$:

$$\texttt{*}id_1\ \texttt{=}\ id_2: \quad [\![v]\!] = store(JOIN(v), id_1, id_2)$$

where

$$store(\sigma, id_1, id_2) = \sigma \quad \underset{\alpha \in pt(id_1)}{[\alpha \mapsto} \quad \sigma(\alpha) \sqcup \sigma(id_2)\ ]$$

The situation we here see at heap store operations where we model an assignment by joining the new abstract value into the existing one is called a *weak update*. In contrast, in a *strong update* the new abstract value overwrites the existing one, which we see in the null pointer analysis at all operations that modify program variables. Strong updates are obviously more precise than weak updates in general, but it may require more elaborate analysis abstractions to detect situations where strong update can be applied soundly.

After performing the null pointer analysis of a given program, a pointer dereference *id at a program point v is guaranteed to be safe if

$$JOIN(v)(id) = \texttt{NN}$$

The precision of this analysis depends of course on the quality of the underlying points-to analysis.

Consider the following buggy example program:

```
p = malloc;
q = &p;
n = null;
*q = n;
*p = n;
```

Andersen's algorithm computes the following points-to sets:

$$\begin{aligned}
&pt(\mathtt{p}) = \{\mathtt{malloc\text{-}1}\}\\
&pt(\mathtt{q}) = \{\mathtt{p}\}\\
&pt(\mathtt{n}) = \emptyset
\end{aligned}$$

Based on this information, the null pointer analysis generates the following constraints:

$$\begin{aligned}
&[\![\mathtt{p=malloc}]\!] &&= \bot[\mathtt{p} \mapsto \mathtt{NN}, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]\\
&[\![\mathtt{q=\&p}]\!] &&= [\![\mathtt{p=malloc}]\!][\mathtt{q} \mapsto \mathtt{NN}]\\
&[\![\mathtt{n=null}]\!] &&= [\![\mathtt{q=\&p}]\!][\mathtt{n} \mapsto \mathtt{?}]\\
&[\![\mathtt{*q=n}]\!] &&= [\![\mathtt{n=null}]\!][\mathtt{p} \mapsto [\![\mathtt{n=null}]\!](\mathtt{p}) \sqcup [\![\mathtt{n=null}]\!](\mathtt{n})]\\
&[\![\mathtt{*p=n}]\!] &&= [\![\mathtt{*q=n}]\!][\mathtt{malloc\text{-}1} \mapsto [\![\mathtt{*q=n}]\!](\mathtt{malloc\text{-}1}) \sqcup [\![\mathtt{*q=n}]\!](\mathtt{n})]
\end{aligned}$$

for which the least solution is:

$$\begin{aligned}
&[\![\mathtt{p=malloc}]\!] &&= [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \bot, \mathtt{n} \mapsto \bot, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]\\
&[\![\mathtt{q=\&p}]\!] &&= [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{n} \mapsto \bot, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]\\
&[\![\mathtt{n=null}]\!] &&= [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{n} \mapsto \mathtt{?}, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]\\
&[\![\mathtt{*q=n}]\!] &&= [\mathtt{p} \mapsto \mathtt{?}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{n} \mapsto \mathtt{?}, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]\\
&[\![\mathtt{*p=n}]\!] &&= [\mathtt{p} \mapsto \mathtt{?}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{n} \mapsto \mathtt{?}, \mathtt{malloc\text{-}1} \mapsto \mathtt{?}]
\end{aligned}$$

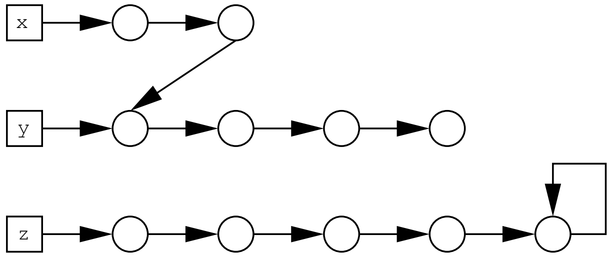
By inspecting this information, an analysis could statically detect that when `*q=n` is evaluated, which is immediately after `n=null`, the variable `q` is definitely non-`null`. On the other hand, when `*p=n` is evaluated, we cannot rule out the possibility that `p` may contain `null`.

Exercise 9.4: Show an alternative constraint for heap load operations using weak update, together with an example program where the modified analysis then gives a result that is less precise than the analysis presented above.

Exercise 9.5: Show an (unsound) alternative constraint for heap store operations using strong update, together with an example program where the modified analysis then gives a wrong result.

9.6 Example: Shape Analysis

So far, we have viewed the heap as an amorphous structure and only answered questions about stack based variables. The heap can be analyzed in more detail using *shape analysis*. Note that we can produce interesting heaps, even though the `malloc` operation only allocates a single heap cell. An example of a non-trivial heap is:



where x, y, and z are program variables. We will seek to answer questions about disjointness of the structures contained in program variables. In the example above, x and y are not disjoint whereas y and z are.

Shape analysis requires a more ambitious lattice of *shape graphs*, which are directed graphs in which the nodes are the abstract cells for the given program and the edges correspond to possible pointers. Shape graphs are ordered by inclusion of their sets of edges. Thus, $\perp$ is the graph without edges and $\top$ is the completely connected graph. Formally, our lattice is then

$$2^{Cell \times Cell}$$

ordered by the usual subset inclusion. For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting a shape graph that describes all possible stores after that program point. For the nodes corresponding to the various pointer manipulations we have the constraints:

$$\begin{array}{rl}
id \texttt{ = malloc}: & [\![v]\!] = JOIN(v)\!\downarrow\! id \;\cup\; \{(id, \texttt{malloc-i})\} \\
id_1 \texttt{ = \&}id_2: & [\![v]\!] = JOIN(v)\!\downarrow\! id_1 \;\cup\; \{(id_1, id_2)\} \\
id_1 \texttt{ = } id_2: & [\![v]\!] = assign(JOIN(v), id_1, id_2) \\
id_1 \texttt{ = *}id_2: & [\![v]\!] = load(JOIN(v), id_1, id_2) \\
\texttt{*}id_1 \texttt{ = } id_2: & [\![v]\!] = store(JOIN(v), id_1, id_2) \\
id \texttt{ = null}: & [\![v]\!] = JOIN(v)\!\downarrow\! id
\end{array}$$

and for all other nodes the constraint:

$$[\![v]\!] = JOIN(v)$$

where

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

$$\sigma\!\downarrow\! x = \{(s,t) \in \sigma \mid s \neq x\}$$

$$assign(\sigma, x, y) = \sigma\!\downarrow\! x \;\cup \{(x,t) \mid (y,t) \in \sigma\}$$

$$load(\sigma, x, y) = \sigma\!\downarrow\! x \;\cup \{(x,t) \mid (y,s) \in \sigma, (s,t) \in \sigma\}$$

$$store(\sigma, x, y) = \sigma \;\cup \{(s,t) \mid (x,s) \in \sigma, (y,t) \in \sigma\}$$

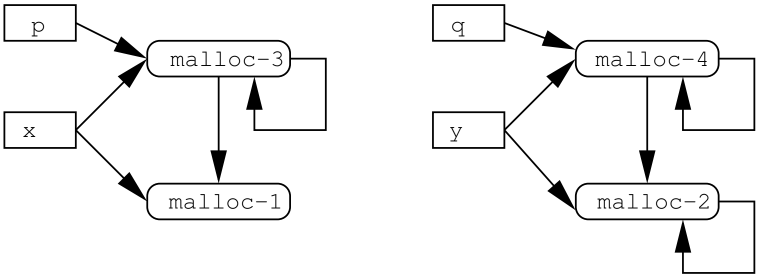
Notice that the constraint for heap store operations uses weak update.

Exercise 9.6: Explain the above constraints.

Consider now the following program:

```
var x,y,n,p,q;
x = malloc; y = malloc;
*x = null; *y = y;
n = input;
while (n>0) {
  p = malloc; q = malloc;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

After the loop, the analysis produces the following shape graph:



From this result we can safely conclude that x and y will always be disjoint.

Note that our shape analysis also computes a flow sensitive points-to map that for each program point v is defined by:

$$pt(p) = \{t \mid (p,t) \in [\![v]\!]\}$$

This analysis is more precise than Andersen's algorithm, but clearly also more expensive to perform. As an example, consider the program:

```
x = &y;
x = &z;
```

After these statements, Andersen's algorithm would predict that $pt(\mathtt{x}) = \{\mathtt{y},\mathtt{z}\}$ whereas the shape analysis computes $pt(\mathtt{x}) = \{\mathtt{z}\}$ for the final program point. This flow sensitive points-to information could be used to boost the null pointer analysis. However, an initial flow insensitive points-to analysis would still be required to construct a CFG for programs using function pointers. Conversely, if we have another points-to analysis, then it may be used to boost the precision of the shape analysis by restricting the locations considered in the *load* and *store* functions. Alternatively, we could perform on-the-fly points-to and control flow analysis during a dataflow analysis by suitably augmenting the lattice.

9.7 Example: Escape Analysis

We earlier lamented the *escaping stack cell* error displayed by the program:

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

which was beyond the scope of the type system. Having performed the simple shape analysis, we can easily perform an *escape analysis* to catch such errors. We just need to check that the possible cells for return expressions in the shape graph cannot reach arguments or variables defined in the function itself, since all other locations must then necessarily reside in earlier frames on the invocation stack.