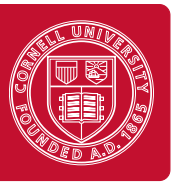
CS 4110 – Programming Languages and Logics
Lecture #2: Introduction to Semantics



What is the meaning of a program? When we write a program, we represent it using sequences of characters. But these strings are just *concrete syntax*—they do not tell us what the program actually means. It is tempting to define meaning by executing programs—either using an interpreter or a compiler. But interpreters and compilers often have bugs! We could look in a specification manual. But such manuals typically only offer an informal description of language constructs.

A better way to define meaning is to develop a formal, mathematical definition of the semantics of the language. This approach is unambiguous, concise, and—most importantly—it makes it possible to develop rigorous proofs about properties of interest. The main drawback is that the semantics itself can be quite complicated, especially if one attempts to model all of the features of a full-blown modern programming language.

There are three pedigreed ways of defining the meaning, or *semantics*, of a language:

- *Operational semantics* defines meaning in terms of execution on an abstract machine.
- *Denotational semantics* defines meaning in terms of mathematical objects such as functions.
- *Axiomatic semantics* defines meaning in terms of logical formulas satisfied during execution.

Each of these approaches has advantages and disadvantages in terms of how mathematically sophisticated they are, how easy they are to use in proofs, and how easy it is to use them to implement an interpreter or compiler. We will discuss these tradeoffs later in this course.

1 Arithmetic Expressions

To understand some of the key concepts of semantics, let us consider a very simple language of integer arithmetic expressions with variable assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer. To describe the syntactic structure of this language we will use variables that range over the following domains:

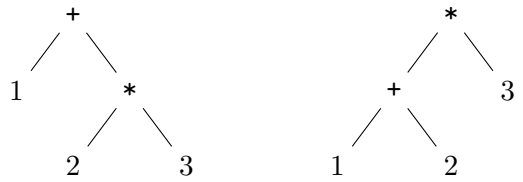
$$\begin{array}{rcl} x, y, z & \in & \textbf{Var} \\ n, m & \in & \textbf{Int} \\ e & \in & \textbf{Exp} \end{array}$$

Var is the set of program variables (e.g., *foo*, *bar*, *baz*, i, etc.). **Int** is the set of constant integers (e.g., 42, 40, 7). **Exp** is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar:

$$\begin{array}{rcl} e & ::= & x \\ & \mid & n \\ & \mid & e_1 \texttt{ + } e_2 \\ & \mid & e_1 \texttt{ * } e_2 \\ & \mid & x \texttt{ := } e_1 \texttt{ ; } e_2 \end{array}$$

Informally, the expression $x := e_1 \,;\, e_2$ means that x is assigned the value of e_1 before evaluating e_2. The result of the entire expression is the value described by e_2.

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 * 3$. One can build two abstract syntax trees:



There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes the grammar more complex and harder to understand. Another possibility is to extend the syntax to require parentheses around all addition and multiplication expressions:

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & n \\
\mid \; & (e_1 + e_2) \\
\mid \; & (e_1 * e_2) \\
\mid \; & x := e_1 \,;\, e_2
\end{aligned}
$$

However, this also leads to unnecessary clutter and complexity. Instead, we separate the "concrete syntax" of the language (which specifies how to unambiguously parse a string into program phrases) from its "abstract syntax" (which describes, possibly ambiguously, the structure of program phrases). In this course we will assume that the abstract syntax tree is known. When writing expressions, we will occasionally use parenthesis to indicate the structure of the abstract syntax tree, but the parentheses are not part of the language itself. (For details on parsing, grammars, and ambiguity elimination, see or take CS 4120.)

1.1 Representing Expressions

The syntactic structure of expressions in this language can be compactly expressed in OCaml using datatypes:

```
type exp = Var of string
         | Int of int
         | Add of exp * exp
         | Mul of exp * exp
         | Assgn of string * exp * exp
```

This closely matches the BNF grammar above. The abstract syntax tree (AST) of an expression can be obtained by applying the datatype constructors in each case. For instance, the AST of expression $2 * (foo + 1)$ is:

```
Mul(Int(2), Add(Var("foo"), Int(1)))
```

In OCaml, parentheses can be dropped when there is one single argument, so the above expression can be written as:

```
Mul(Int 2, Add(Var "foo", Int 1))
```

We could express the same structure in a language like Java using a class hierarchy, although it would be a little more complicated:

```
abstract class Expr { }
class Var extends Expr { String name; .. }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr exp1, exp2; ... }
class Mul extends Expr { Expr exp1, exp2; ... }
class Assgn extends Expr { String var, Expr exp1, exp2; .. }
```

2 Operational semantics

We have an intuitive notion of what expressions mean. For example, the $7 + (4 * 2)$ evaluates to 15, and $i := 6 + 1\,;\; 2 * 3 * i$ evaluates to 42. In this section, we will formalize this intuition precisely.

An *operational semantics* describes how a program executes on an abstract machine. A *small-step* operational semantics describes how such an execution proceeds in terms of successive reductions—here, of an expression—until we reach a value that represents the result of the computation. The state of the abstract machine is often referred to as a *configuration*. For our language a configuration must include two pieces of information:

- a *store* (also known as environment or state), which maps integer values to variables. During program execution, we will refer to the store to determine the values associated with variables, and also update the store to reect assignment of new values to variables,
- the *expression* to evaluate.

We will represent stores as partial functions from **Var** to **Int** and configurations as pairs of expressions and stores:

$$\begin{aligned} \mathbf{Store} &\triangleq \mathbf{Var} \rightharpoonup \mathbf{Int} \\ \mathbf{Config} &\triangleq \mathbf{Store} \times \mathbf{Exp} \end{aligned}$$

We will denote configurations using angle brackets. For instance, $\langle \sigma, (foo + 2) * (bar + 2) \rangle$ is a configuration where σ is a store and $(foo + 2) * (bar + 2)$ is an expression that uses two variables, foo and bar. The small-step operational semantics for our language is a relation $\rightarrow \subseteq \mathbf{Config} \times \mathbf{Config}$ that describes how one configuration transitions to a new configuration. That is, the relation $\rightarrow$ shows us how to evaluate programs one step at a time. We use infix notation for the relation $\rightarrow$. That is, given any two configurations $\langle \sigma_1, e_1 \rangle$ and $\langle \sigma_2, e_2 \rangle$, if $(\langle e_1, \sigma_1 \rangle, \langle e_2, \sigma_2 \rangle)$ is in the relation $\rightarrow$, then we write $\langle \sigma_1, e_1 \rangle \rightarrow \langle \sigma_2, e_2 \rangle$. For example, we have $\langle \sigma, (4 + 2) * y \rangle \rightarrow \langle \sigma, 6 * y \rangle$. That is, we can evaluate the configuration $\langle \sigma, (4 + 2) * y \rangle$ one step to get the configuration $\langle \sigma, 6 * y \rangle$.

Using this approach, defining the semantics of the language boils down to to defining the relation $\rightarrow$ that describes the transitions between configurations.

One issue here is that the domain of integers is infinite, as is the domain of expressions. Therefore, there is an infinite number of possible machine configurations, and an infinite number of possible single-step transitions. We need a finite way of describing an infinite set of possible transitions. We can compactly describe $\rightarrow$ using inference rules:

$$\frac{n = \sigma(x)}{\langle \sigma, x \rangle \to \langle \sigma, n \rangle}\ \textsc{Var}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 + e_2 \rangle \to \langle \sigma', e_1' + e_2 \rangle}\ \textsc{LAdd} \qquad \frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n + e_2 \rangle \to \langle \sigma', n + e_2' \rangle}\ \textsc{RAdd} \qquad \frac{p = m + n}{\langle \sigma, n + m \rangle \to \langle \sigma, p \rangle}\ \textsc{Add}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 * e_2 \rangle \to \langle \sigma', e_1' * e_2 \rangle}\ \textsc{LMul} \qquad \frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n * e_2 \rangle \to \langle \sigma', n * e_2' \rangle}\ \textsc{RMul} \qquad \frac{p = m \times n}{\langle \sigma, m * n \rangle \to \langle \sigma, p \rangle}\ \textsc{Mul}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, x := e_1 \,;\, e_2 \rangle \to \langle \sigma', x := e_1' \,;\, e_2 \rangle}\ \textsc{Assgn1} \qquad \frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n \,;\, e_2 \rangle \to \langle \sigma', e_2 \rangle}\ \textsc{Assgn}$$

The meaning of an inference rule is that if the facts above the line holds, then the fact below the line holds. The fact above the line are called *premises*; the fact below the line is called the *conclusion*. The rules without premises are *axioms*; and the rules with premises are *inductive* rules. We use the notation $\sigma[x \mapsto n]$ for the store that maps the variable x to integer n, and maps every other variable to whatever σ maps it to. More explicitly, if f is the function $\sigma[x \mapsto n]$, then we have

$$f(y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

3 Using the Semantics

Now let's see how we can use these rules. Suppose we want to evaluate the expression $(foo + 2) * (bar + 1)$ with a store σ where $\sigma(foo) = 4$ and $\sigma(bar) = 3$. That is, we want to find the transition for the configuration $\langle \sigma, (foo + 2) * (bar + 1) \rangle$. For this, we look for a rule with this form of a configuration in the conclusion. By inspecting the rules, we find that the only rule that matches the form of our configuration is LMUL, where $e_1 = foo + 2$ and $e_2 = bar + 1$ but e_1' is not yet known. We can instantiate LMUL, replacing the metavariables e_1 and e_2 with appropriate expressions.

$$\frac{\langle \sigma, foo + 2 \rangle \to \langle e_1', \sigma \rangle}{\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, e_1' * (bar + 1) \rangle}\ \textsc{LMul}$$

Now we need to show that the premise actually holds and find out what e_1' is. We look for a rule whose conclusion matches $\langle \sigma, foo + 2 \rangle \to \langle e_1', \sigma \rangle$. We find that LADD is the only matching rule:

$$\frac{\langle \sigma, foo \rangle \to \langle \sigma, e_1'' \rangle}{\langle \sigma, foo + 2 \rangle \to \langle \sigma, e_1'' + 2 \rangle}\ \textsc{LAdd}$$

We repeat this reasoning for $\langle \sigma, foo \rangle \to \langle \sigma, e_1'' \rangle$ and find that the only applicable rule is the axiom VAR:

$$\frac{\sigma(foo) = 4}{\langle \sigma, foo \rangle \to \langle \sigma, 4 \rangle}\ \textsc{Var}$$

Since this is an axiom and has no premises, there is nothing left to prove. Hence, $e_1'' = 4$ and $e_1' = 4 + 2$. We can put together the above pieces and build the following proof:

$$\dfrac{\dfrac{\dfrac{\sigma(foo) = 4}{\langle \sigma, foo \rangle \to \langle \sigma, 4 \rangle}\ \textsc{Var}}{\langle \sigma, foo + 2 \rangle \to \langle \sigma, 4 + 2 \rangle}\ \textsc{LAdd}}{\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, (4 + 2) * (bar + 1) \rangle}\ \textsc{LMul}$$

This proves that, given our inference rules, the one-step transition

$$\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, (4 + 2) * (bar + 1) \rangle$$

is derivable. The structure above is called a "proof tree" or "derivation". It is important to keep in mind that proof trees must be finite for the conclusion to be valid.

We can use a similar reasoning to find out the next evaluation step:

$$\dfrac{\dfrac{6 = 4 + 2}{\langle \sigma, 4 + 2 \rangle \to \langle \sigma, 6 \rangle}\ \textsc{Add}}{\langle \sigma, (4 + 2) * (bar + 1) \rangle \to \langle \sigma, 6 * (bar + 1) \rangle}\ \textsc{LMul}$$

And we can continue this process. At the end, we can put together all of these transitions, to get a view of the entire computation:

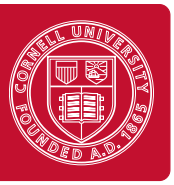
$$\begin{aligned} \langle \sigma, (foo + 2) * (bar + 1) \rangle &\to \langle \sigma, (4 + 2) * (bar + 1) \rangle \\ &\to \langle \sigma, 6 * (bar + 1) \rangle \\ &\to \langle \sigma, 6 * (3 + 1) \rangle \\ &\to \langle \sigma, 6 * 4 \rangle \\ &\to \langle \sigma, 24 \rangle \end{aligned}$$

The result of the computation is a number, 24. The machine configuration that contains the final result is the point where the evaluation stops; they are called final configurations. For our language of expressions, the final configurations are of the form $\langle \sigma, n \rangle$.

We write $\to^*$ for the reflexive and transitive closure of the relation $\to$. That is, if $\langle \sigma, e \rangle \to^* \langle \sigma', e' \rangle$ using zero or more steps, we can evaluate the configuration $\langle \sigma, e \rangle$ to $\langle \sigma', e' \rangle$. Thus, we have:

$$\langle \sigma, (foo + 2) * (bar + 1) \rangle \to^* \langle \sigma, 24 \rangle$$

CS 4110 – Programming Languages and Logics
Lecture #3: Inductive definitions and proofs



In this lecture, we will use the semantics of our simple language of arithmetic expressions,

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 * e_2 \mid x := e_1 \,;\, e_2,$$

to express useful program properties, and we will prove these properties by induction.

1 Program Properties

There are a number of interesting questions about a language one can ask: Is it deterministic? Are there non-terminating programs? What sorts of errors can arise during evaluation? Having a formal semantics allows us to express these properties precisely.

- **Determinism:** Evaluation is deterministic,

$$\forall e \in \mathbf{Exp}.\ \forall \sigma, \sigma', \sigma'' \in \mathbf{Store}.\ \forall e', e'' \in \mathbf{Exp}.$$
$$\text{if } \langle \sigma, e\rangle \to \langle \sigma', e'\rangle \text{ and } \langle \sigma, e\rangle \to \langle \sigma'', e''\rangle \text{ then } e' = e'' \text{ and } \sigma' = \sigma''.$$

- **Termination:** Evaluation of every expression terminates,

$$\forall e \in \mathbf{Exp}.\ \forall \sigma \in \mathbf{Store}.\ \exists \sigma' \in \mathbf{Store}.\ \exists e' \in \mathbf{Exp}.\ \langle \sigma, e\rangle \to^* \langle \sigma', e'\rangle \text{ and } \langle \sigma', e'\rangle \not\to,$$

  where $\langle \sigma', e'\rangle \not\to$ is shorthand for $\neg\,(\exists \sigma'' \in \mathbf{Store}.\ \exists e'' \in \mathbf{Exp}.\ \langle \sigma', e'\rangle \to \langle \sigma'', e''\rangle)$.

It is tempting to want the following soundness property,

- **Soundness:** Evaluation of every expression yields an integer,

$$\forall e \in \mathbf{Exp}.\ \forall \sigma \in \mathbf{Store}.\ \exists \sigma' \in \mathbf{store}.\ \exists n' \in \mathbf{Int}.\ \langle \sigma, e\rangle \to^* \langle \sigma', n'\rangle,$$

but unfortunately it does not hold in our language! For example, consider the totally-undefined function σ and the expression $i + j$. The configuration $\langle \sigma, i + j\rangle$ is *stuck*—it has no possible transitions—but $i + j$ is not an integer. The problem is that $i + j$ has *free variables* but σ does not contain mappings for those variables.

To fix this problem, we can restrict our attention to *well-formed* configurations $\langle \sigma, e\rangle$, where σ is defined on (at least) the free variables in e. This makes sense as evaluation typically starts with a *closed* expression. We can define the set of free variables of an expression as follows:

$$\begin{aligned}
\mathit{fvs}(x) &\triangleq \{x\} \\
\mathit{fvs}(n) &\triangleq \{\} \\
\mathit{fvs}(e_1 + e_2) &\triangleq \mathit{fvs}(e_1) \cup \mathit{fvs}(e_2) \\
\mathit{fvs}(e_1 * e_2) &\triangleq \mathit{fvs}(e_1) \cup \mathit{fvs}(e_2) \\
\mathit{fvs}(x := e_1 \,;\, e_2) &\triangleq \mathit{fvs}(e_1) \cup (\mathit{fvs}(e_2) \setminus \{x\})
\end{aligned}$$

Now we can formulate two properties that imply a variant of the soundness property above:

- **Progress:** For each expression e and store σ such that the free variables of e are contained in the domain of σ, either e is an integer or there exists a possible transition for $\langle \sigma, e \rangle$,

$$\forall e \in \mathbf{Exp}.\ \forall \sigma \in \mathbf{Store}.\\ \quad \mathit{fvs}(e) \subseteq \mathit{dom}(\sigma) \implies e \in \mathbf{Int} \text{ or } (\exists e' \in \mathbf{Exp}.\ \exists \sigma' \in \mathbf{Store}.\ \langle \sigma, e \rangle \to \langle \sigma', e' \rangle)$$

- **Preservation:** Evaluation preserves containment of free variables in the domain of the store,

$$\forall e, e' \in \mathbf{Exp}.\ \forall \sigma, \sigma' \in \mathbf{Store}.\\ \quad \mathit{fvs}(e) \subseteq \mathit{dom}(\sigma) \text{ and } \langle \sigma, e \rangle \to \langle \sigma', e' \rangle \implies \mathit{fvs}(e') \subseteq \mathit{dom}(\sigma').$$

The rest of this lecture shows how can we prove such properties using induction.

2 Inductive sets

Induction is an important concept in programming language theory. An *inductively-defined set* A is one that is described using a finite collection of axioms and inductive (inference) rules. Axioms of the form

$$\frac{}{a \in A}$$

indicate that a is in the set A. Inductive rules

$$\frac{a_1 \in A \qquad \ldots \qquad a_n \in A}{a \in A}$$

indicate that if $a_1, \ldots, a_n$ are all elements of A, then a is also an element of A.

The set A is the set of all elements that can be inferred to belong to A using a (finite) number of applications of these rules, starting only from axioms. In other words, for each element a of A, we must be able to construct a finite proof tree whose final conclusion is $a \in A$.

Example 1. The set described by a grammar is an inductive set. For instance, the set of arithmetic expressions can be described with two axioms and three inference rules:

$$\frac{}{x \in \mathbf{Exp}} \qquad \frac{}{n \in \mathbf{Exp}}$$

$$\frac{e_1 \in \mathbf{Exp} \qquad e_2 \in \mathbf{Exp}}{e_1 \texttt{ + } e_2 \in \mathbf{Exp}} \qquad \frac{e_1 \in \mathbf{Exp} \qquad e_2 \in \mathbf{Exp}}{e_1 \texttt{ * } e_2 \in \mathbf{Exp}} \qquad \frac{e_1 \in \mathbf{Exp} \qquad e_2 \in \mathbf{Exp}}{x \texttt{ := } e_1 \texttt{ ; } e_2 \in \mathbf{Exp}}$$

These axioms and rules describe the same set of expressions as the grammar:

$$e ::= x \mid n \mid e_1 \texttt{ + } e_2 \mid e_1 \texttt{ * } e_2 \mid x \texttt{ := } e_1 \texttt{ ; } e_2$$

Example 2. The natural numbers (expressed here in unary notation) can be inductively defined:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{\mathit{succ}(n) \in \mathbb{N}}$$

Example 3. The small-step evaluation relation $\to$ is an inductively defined set.

Example 4. The multi-step evaluation relation can be inductively defined:

$$\frac{}{\langle \sigma, e\rangle \to^* \langle \sigma, e\rangle}\ \textsc{Refl} \qquad \frac{\langle \sigma, e\rangle \to \langle \sigma', e'\rangle \qquad \langle \sigma', e'\rangle \to^* \langle \sigma'', e''\rangle}{\langle \sigma, e\rangle \to^* \langle \sigma'', e''\rangle}\ \textsc{Trans}$$

Example 5. The set of free variables of an expression e can be inductively defined:

$$\frac{}{y \in \mathit{fvs}(y)} \qquad \frac{y \in \mathit{fvs}(e_1)}{y \in \mathit{fvs}(e_1 + e_2)} \qquad \frac{y \in \mathit{fvs}(e_2)}{y \in \mathit{fvs}(e_1 + e_2)} \qquad \frac{y \in \mathit{fvs}(e_1)}{y \in \mathit{fvs}(e_1 * e_2)} \qquad \frac{y \in \mathit{fvs}(e_2)}{y \in \mathit{fvs}(e_1 * e_2)}$$

$$\frac{y \in \mathit{fvs}(e_1)}{y \in \mathit{fvs}(x := e_1\,;\, e_2)} \qquad \frac{y \neq x \qquad y \in \mathit{fvs}(e_2)}{y \in \mathit{fvs}(x := e_1\,;\, e_2)}$$

3 Inductive proofs

We can prove facts about elements of an inductive set using an inductive reasoning that follows the structure of the set definition.

3.1 Mathematical induction

You have probably seen proofs by induction over the natural numbers, called *mathematical induction*. In such proofs, we typically want to prove that some property P holds for all natural numbers, that is, $\forall n \in \mathbb{N}.\ P(n)$. A proof by induction works by first proving that $P(0)$ holds, and then proving for all $m \in \mathbb{N}$, if $P(m)$ then $P(m+1)$. The principle of mathematical induction can be stated succinctly as

$$P(0) \text{ and } (\forall m \in \mathbb{N}.\ P(m) \implies P(m+1)) \implies \forall n \in \mathbb{N}.\ P(n).$$

The proposition $P(0)$ is the *basis* of the induction (also called the *base case*) while $P(m) \implies P(m+1)$ is called *induction step* (or the *inductive case*). While proving the induction step, the assumption that $P(m)$ holds is called the *induction hypothesis*.

3.2 Structural induction

Given an inductively defined set A, to prove that a property P holds for all elements of A, we need to show:

1. **Base cases:** For each axiom

   $$\frac{}{a \in A},$$

   $P(a)$ holds.

2. **Inductive cases:** For each inference rule

   $$\frac{a_1 \in A \quad \ldots \quad a_n \in A}{a \in A},$$

   if $P(a_1)$ and $\ldots$ and $P(a_n)$ then $P(a)$.

Note that if the set A is the set of natural numbers from Example 2 above, then the requirements for proving that P holds for all elements of A is equivalent to mathematical induction.

If A describes a syntactic set, then we refer to induction following the requirements above as *structural induction*. If A is an operational semantics relation (such as the small-step operational semantics relation $\rightarrow$) then such an induction is called *induction on derivations*. We will see examples of structural induction and induction on derivations throughout the course.

3.3 Example: Progress

Let's consider the progress property defined above, and repeated here:

Progress: For each store σ and expression e such that the free variables of e are contained in the domain of σ, either e is an integer or there exists a possible transition for $\langle \sigma, e \rangle$:

$$\forall e \in \textbf{Exp}.\, \forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e) \subseteq \mathit{dom}(\sigma) \implies e \in \textbf{Int} \text{ or } (\exists e' \in \textbf{Exp}.\, \exists \sigma' \in \textbf{Store}.\, \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle)$$

Let's rephrase this property in terms of an explicit predicate on expressions:

$$P(e) \triangleq \forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e) \subseteq \mathit{dom}(\sigma) \implies e \in \textbf{Int} \text{ or } \left(\exists e', \sigma'.\, \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle\right)$$

The idea is to build a proof that follows the inductive structure given by the grammar:

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 * e_2 \mid x := e_1 \,;\, e_2$$

This technique is called "structural induction on e." We analyze each case in the grammar and show that $P(e)$ holds for that case. Since the grammar productions $e_1 + e_2$ and $e_1 * e_2$ and $x := e_1 \,;\, e_2$ are inductive, they are inductive steps in the proof; the cases for x and n are base cases. The proof proceeds as follows.

Proof. Let e be an expression. We will prove that

$$\forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e) \subseteq \mathit{dom}(\sigma) \implies e \in \textbf{Int} \text{ or } \left(\exists e', \sigma'.\, \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle\right)$$

by structural induction on e. We analyze several cases, one for each case in the grammar for expressions:

Case $e = x$**:** Let σ be an arbitrary store, and assume that $\mathit{fvs}(e) \subseteq \mathit{dom}(\sigma)$. By the definition of fvs we have $\mathit{fvs}(x) = \{x\}$. By assumption we have $\{x\} \subseteq \mathit{dom}(\sigma)$ and so $x \in \mathit{dom}(\sigma)$. Let $n = \sigma(x)$. By the VAR axiom we have $\langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle$, which finishes the case.

Case $e = n$**:** We immediately have $e \in \textbf{Int}$, which finishes the case.

Case $e = e_1 + e_2$**:** Let σ be an arbitrary store, and assume that $\mathit{fvs}(e) \subseteq \mathit{dom}(\sigma)$. We will assume that $P(e_1)$ and $P(e_2)$ hold and show that $P(e)$ holds. Let's expand these properties. We have

$$P(e_1) = \forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e_1) \subseteq \mathit{dom}(\sigma) \implies e_1 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e' \rangle)$$
$$P(e_2) = \forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e_2) \subseteq \mathit{dom}(\sigma) \implies e_2 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e' \rangle)$$

and want to prove:

$$P(e_1 + e_2) = \forall \sigma \in \textbf{Store}.\, \mathit{fvs}(e_1 + e_2) \subseteq \mathit{dom}(\sigma) \implies e_1 + e_2 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e' \rangle)$$

We analyze several subcases.

Subcase $e_1 = n_1$ **and** $e_2 = n_2$**:** By rule ADD, we immediately have $\langle \sigma, n_1 + n_2 \rangle \to \langle \sigma, p \rangle$, where $p = n_1 + n_2$.

Subcase $e_1 \notin$ **Int:** By assumption and the definition of fvs we have

$$fvs(e_1) \subseteq fvs(e_1 + e_2) \subseteq dom(\sigma)$$

Hence, by the induction hypothesis $P(e_1)$ we also have $\langle \sigma, e_1 \rangle \to \langle \sigma', e' \rangle$ for some e' and σ'. By rule LADD we have $\langle \sigma, e_1 + e_2 \rangle \to \langle \sigma', e' + e_2 \rangle$.

Subcase $e_1 = n_1$ **and** $e_2 \notin$ **Int:** By assumption and the definition of fvs we have

$$fvs(e_2) \subseteq fvs(e_1 + e_2) \subseteq dom(\sigma)$$

Hence, by the induction hypothesis $P(e_2)$ we also have $\langle \sigma, e_2 \rangle \to \langle \sigma', e' \rangle$ for some e' and σ'. By rule RADD we have $\langle \sigma, e_1 + e_2 \rangle \to \langle \sigma', e_1 + e' \rangle$, which finishes the case.

Case $e = e_1 * e_2$**:** . Analogous to the previous case.

Case $e = x := e_1 \,;\, e_2$**:** . Let σ be an arbitrary store, and assume that $fvs(e) \subseteq dom(\sigma)$. As above, we assume that $P(e_1)$ and $P(e_2)$ hold and show that $P(e)$ holds. Let's expand these properties. We have

$$P(e_1) = \forall \sigma.\, fvs(e_1) \subseteq dom(\sigma) \Longrightarrow e_1 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, e_1 \rangle \to \langle \sigma', e' \rangle)$$
$$P(e_2) = \forall \sigma.\, fvs(e_2) \subseteq dom(\sigma) \Longrightarrow e_2 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, e_2 \rangle \to \langle \sigma', e' \rangle)$$

and want to prove:

$$P(x := e_1 \,;\, e_2) = x := e_1 \,;\, e_2 \in \textbf{Int} \text{ or } (\exists e', \sigma'.\, \langle \sigma, x := e_1 \,;\, e_2 \rangle \to \langle \sigma', e' \rangle)$$

We analyze several subcases.

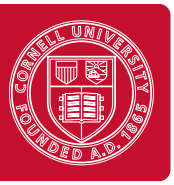
Subcase $e_1 = n_1$**:** By rule ASSGN we have $\langle \sigma, x := n_1 \,;\, e_2 \rangle \to \langle \sigma', e_2 \rangle$ where $\sigma' = \sigma[x \mapsto n_1]$.

Subcase $e_1 \notin$ **Int:** By assumption and the definition of fvs we have

$$fvs(e_1) \subseteq fvs(x := e_1 \,;\, e_2) \subseteq dom(\sigma)$$

Hence, by induction hypothesis we also have $\langle \sigma, e_1 \rangle \to \langle \sigma', e' \rangle$ for some e' and σ'. By the rule ASSGN1 we have $\langle \sigma, x := e_1 \,;\, e_2 \rangle \to \langle \sigma', x := e_1' \,;\, e_2 \rangle$, which finishes the case and the inductive proof. □

CS 4110 – Programming Languages and Logics
Lecture #4: Large-step semantics



1 Large-step operational semantics

In the last lecture we defined a semantics for our language of arithmetic expressions using a small-step evaluation relation $\rightarrow \subseteq \textbf{Config} \times \textbf{Config}$ (and its reflexive and transitive closure $\rightarrow^*$). In this lecture we will explore an alternative approach—*large-step* operational semantics—which yields the final result of evaluating an expression directly.

Defining a large-step semantics boils down to specifying a relation $\Downarrow$ that captures the evaluation of an expression. The $\Downarrow$ relation has the following type:

$$\Downarrow \subseteq (\textbf{Store} \times \textbf{Exp}) \times (\textbf{Store} \times \textbf{Int}).$$

We write $\langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle$ to indicate that $((\sigma, e), (\sigma', n)) \in \Downarrow$. In other words, the expression e with store σ evaluates in one big step to the final store σ' and integer n.

We define the relation $\Downarrow$ inductively, using inference rules:

$$\frac{}{\langle \sigma, n \rangle \Downarrow \langle \sigma, n \rangle}\ \textsc{Int} \qquad \frac{n = \sigma(x)}{\langle \sigma, x \rangle \Downarrow \langle \sigma, n \rangle}\ \textsc{Var}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow \langle \sigma', n_1 \rangle \qquad \langle \sigma', e_2 \rangle \Downarrow \langle \sigma'', n_2 \rangle \qquad n = n_1 + n_2}{\langle \sigma, e_1 + e_2 \rangle \Downarrow \langle \sigma'', n \rangle}\ \textsc{Add}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow \langle \sigma', n_1 \rangle \qquad \langle \sigma', e_2 \rangle \Downarrow \langle \sigma'', n_2 \rangle \qquad n = n_1 \times n_2}{\langle \sigma, e_1 * e_2 \rangle \Downarrow \langle \sigma'', n \rangle}\ \textsc{Mul}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow \langle \sigma', n_1 \rangle \qquad \langle \sigma'[x \mapsto n_1], e_2 \rangle \Downarrow \langle \sigma'', n_2 \rangle}{\langle \sigma, x := e_1 \,;\, e_2 \rangle \Downarrow \langle \sigma'', n_2 \rangle}\ \textsc{Assgn}$$

To illustrate the use of these rules, consider the following proof tree, which shows that evaluating $\langle \sigma, \mathit{foo} := 3 \,;\, \mathit{foo} * \mathit{bar} \rangle$ using a store σ such that $\sigma(\mathit{bar}) = 7$ yields $\sigma' = \sigma[\mathit{foo} \mapsto 3]$ and 21 as a result:

$$\dfrac{\dfrac{}{\langle \sigma, 3 \rangle \Downarrow \langle \sigma, 3 \rangle}\ \textsc{Int} \qquad \dfrac{\dfrac{}{\langle \sigma', \mathit{foo} \rangle \Downarrow \langle \sigma', 3 \rangle}\ \textsc{Var} \qquad \dfrac{}{\langle \sigma', \mathit{bar} \rangle \Downarrow \langle \sigma', 7 \rangle}\ \textsc{Var}}{\langle \sigma', \mathit{foo} * \mathit{bar} \rangle \Downarrow \langle \sigma', 21 \rangle}\ \textsc{Mul}}{\langle \sigma, \mathit{foo} := 3 \,;\, \mathit{foo} * \mathit{bar} \rangle \Downarrow \langle \sigma', 21 \rangle}\ \textsc{Assgn}$$

A closer look to this structure reveals the relation between small step and large-step evaluation: a depth-first traversal of the large-step proof tree yields the sequence of one-step transitions in small-step evaluation.

2 Equivalence of semantics

A natural question to ask is whether the small-step and large-step semantics are equivalent. The next theorem answers this question affirmatively.

Theorem (Equivalence of semantics)**.** *For all expressions e, stores σ and σ', and integers n we have:*

$$\langle \sigma, e\rangle \Downarrow \langle \sigma', n\rangle \text{ if and only if } \langle \sigma, e\rangle \to^* \langle \sigma', n\rangle$$

To streamline the proof, we will work with the following definition of the multi-step relation:

$$\frac{}{\langle \sigma, e\rangle \to^* \langle \sigma, e\rangle}\ \textsc{Refl}$$

$$\frac{\langle \sigma, e\rangle \to \langle \sigma', e'\rangle \qquad \langle \sigma', e'\rangle \to^* \langle \sigma'', e''\rangle}{\langle \sigma, e\rangle \to^* \langle \sigma'', e''\rangle}\ \textsc{Trans}$$

Proof sketch. We show each direction separately.

$\Longrightarrow$: We want to prove that the following property P holds for all expressions $e \in \mathbf{Exp}$:

$$P(e) \triangleq \forall \sigma, \sigma' \in \mathbf{Store}.\ \forall n \in \mathbf{Int}.\ \langle \sigma, e\rangle \Downarrow \langle \sigma', n\rangle \Longrightarrow \langle \sigma, e\rangle \to^* \langle \sigma', n\rangle$$

We proceed by structural induction on e. We have to consider each of the possible axioms and inference rules for constructing an expression.

Case $e = x$: Assume that $\langle \sigma, x\rangle \Downarrow \langle \sigma', n\rangle$. That is, there is some derivation in the large-step operational semantics whose conclusion is $\langle \sigma, x\rangle \Downarrow \langle \sigma, n\rangle$. There is only one rule whose conclusion matches the configuration $\langle \sigma, x\rangle$: the large-step rule VAR. Thus, we have $n = \sigma(x)$ and $\sigma' = \sigma$. By the small-step rule VAR, we also have $\langle \sigma, x\rangle \to \langle \sigma, n\rangle$. By the REFL and TRANS rules, we conclude that $\langle \sigma, x\rangle \to^* \langle \sigma, n\rangle$, which finishes the case.

Case $e = n$: Assume that $\langle \sigma, n\rangle \Downarrow \langle \sigma', n'\rangle$. There is only one rule whose conclusion matches $\langle \sigma, n\rangle$: the large-step rule INT. Thus, we have $n' = n$ and $\sigma' = \sigma$ and so $\langle \sigma, n\rangle \to^* \langle \sigma, n\rangle$ by the REFL rule.

Case $e = e_1 + e_2$: This is an inductive case. We want to prove that if $P(e_1)$ and $P(e_2)$ hold, then $P(e)$ also holds. Let's write out $P(e_1)$, $P(e_2)$, and $P(e)$ explicitly.

$$\begin{aligned} P(e_1) &= \forall n, \sigma, \sigma'.\ \langle \sigma, e_1\rangle \Downarrow \langle \sigma', n\rangle \Longrightarrow \langle \sigma, e_1\rangle \to^* \langle \sigma', n\rangle \\ P(e_2) &= \forall n, \sigma, \sigma'.\ \langle \sigma, e_2\rangle \Downarrow \langle \sigma', n\rangle \Longrightarrow \langle \sigma, e_2\rangle \to^* \langle \sigma', n\rangle \\ P(e) &= \forall n, \sigma, \sigma'.\ \langle \sigma, e_1 + e_2\rangle \Downarrow \langle \sigma', n\rangle \Longrightarrow \langle \sigma, e_1 + e_2\rangle \to^* \langle \sigma', n\rangle \end{aligned}$$

Assume that $P(e_1)$ and $P(e_2)$ hold. Also assume that there exist σ, σ' and n such that $\langle \sigma, e_1 + e_2\rangle \Downarrow \langle \sigma', n\rangle$. We need to show that $\langle \sigma, e_1 + e_2\rangle \to^* \langle \sigma', n\rangle$.
We assumed that $\langle \sigma, e_1 + e_2\rangle \Downarrow \langle \sigma', n\rangle$. This means that there is some derivation whose conclusion is $\langle \sigma, e_1 + e_2\rangle \Downarrow \langle \sigma', n\rangle$. By inspection, we see that only one rule has a conclusion of this form: the ADD rule. Thus, the last rule used in the derivation was ADD and it must be the case that $\langle \sigma, e_1\rangle \Downarrow \langle \sigma'', n_1\rangle$ and $\langle \sigma'', e_2\rangle \Downarrow \langle \sigma', n_2\rangle$ hold for some n_1 and n_2 with $n = n_1 + n_2$.

By the induction hypothesis $P(e_1)$, as $\langle \sigma, e_1 \rangle \Downarrow \langle \sigma'', n_1 \rangle$, we must have $\langle \sigma, e_1 \rangle \to^* \langle \sigma'', n_1 \rangle$. Likewise, by induction hypothesis $P(e_2)$, we have $\langle \sigma'', e_2 \rangle \to^* \langle \sigma', n_2 \rangle$. By Lemma 1 below, we have,

$$\langle \sigma, e_1 + e_2 \rangle \to^* \langle \sigma'', n_1 + e_2 \rangle,$$

and by another application of Lemma 1 we have:

$$\langle \sigma'', n_1 + e_2 \rangle \to^* \langle \sigma', n_1 + n_2 \rangle$$

Then, using the small-step ADD rule and the multi-step TRANS rule, we have:

$$\dfrac{\dfrac{n = n_1 + n_2}{\langle \sigma', n_1 + n_2 \rangle \to \langle \sigma', n \rangle}\ \textsc{Add} \qquad \dfrac{}{\langle \sigma', n \rangle \to^* \langle \sigma', n \rangle}\ \textsc{Refl}}{\langle \sigma', n_1 + n_2 \rangle \to^* \langle \sigma', n \rangle}\ \textsc{Trans}$$

Finally, by two applications of Lemma 2, we obtain $\langle \sigma, e_1 + e_2 \rangle \to^* \langle \sigma', n \rangle$, which finishes the case.

Case $e = e_1 * e_2$**.** Similar to case for $e_1 + e_2$ above.

Case $e = x := e_1; e_2$**.** Omitted. Try it as an exercise.

$\Longleftarrow$**:** We proceed by induction on the derivation of $\langle \sigma, e \rangle \to^* \langle \sigma', n \rangle$ with a case analysis on the last rule used.

Case REFL: Then $e = n$ and $\sigma' = \sigma$. We immediately have $\langle \sigma, n \rangle \Downarrow \langle \sigma, n \rangle$ by the large-step rule INT.

Case TRANS: Then $\langle \sigma, e \rangle \to \langle \sigma'', e'' \rangle$ and $\langle \sigma'', e'' \rangle \to^* \langle \sigma', n \rangle$. In this case, the induction hypothesis gives $\langle \sigma'', e'' \rangle \Downarrow \langle \sigma', n \rangle$. The result follows from Lemma 3 below.

□

Lemma 1. *If $\langle \sigma, e \rangle \to^* \langle \sigma', n \rangle$, then the following hold:*

- $\langle \sigma, e + e_2 \rangle \to^* \langle \sigma', n + e_2 \rangle$
- $\langle \sigma, e * e_2 \rangle \to^* \langle \sigma', n * e_2 \rangle$
- $\langle \sigma, n_1 + e \rangle \to^* \langle \sigma', n_1 + n \rangle$
- $\langle \sigma, n_1 * e \rangle \to^* \langle \sigma', n_1 * n \rangle$
- $\langle \sigma, x := e\,;\, e_2 \rangle \to^* \langle \sigma', x := n\,;\, e_2 \rangle$

Proof. Omitted; try it as an exercise. □

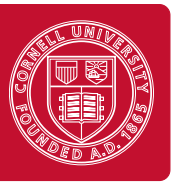
Lemma 2. *If $\langle \sigma, e \rangle \to^* \langle \sigma', e' \rangle$ and $\langle \sigma', e' \rangle \to^* \langle \sigma'', e'' \rangle$, then $\langle \sigma, e \rangle \to^* \langle \sigma'', e'' \rangle$.*

Proof. Omitted; try it as an exercise. □

Lemma 3. *If $\langle \sigma, e \rangle \to \langle \sigma'', e'' \rangle$ and $\langle \sigma'', e'' \rangle \Downarrow \langle \sigma', n \rangle$, then $\langle \sigma, e \rangle \Downarrow \langle \sigma', n \rangle$.*

Proof. Omitted; try it as an exercise. □

CS 4110 – Programming Languages and Logics
Lecture #5: The IMP Language



1 A simple imperative language

We will now consider a more realistic programming language, one where we can assign values to variables and execute control constructs such as if and while. The syntax for this imperative language, called IMP, is as follows:

arithmetic expressions	$a \in \mathbf{Aexp}$	$a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2$
boolean expressions	$b \in \mathbf{Bexp}$	$b ::=$ **true** $\mid$ **false** $\mid a_1 < a_2$
commands	$c \in \mathbf{Com}$	$c ::=$ **skip** $\mid x := a \mid c_1; c_2 \mid$ **if** b **then** c_1 **else** $c_2 \mid$ **while** b **do** c

1.1 Small-step operational semantics

We'll first give a small-step operational semantics for IMP. The configurations in this language are of the form $\langle c, \sigma \rangle$, $\langle \sigma, b \rangle$, and $\langle \sigma, a \rangle$, where σ is a store. The final configurations are of the form $\langle \sigma, skip \rangle$, $\langle \sigma, true \rangle$, $\langle \sigma, false \rangle$, and $\langle \sigma, n \rangle$. There are three different small-step operational semantics relations, one each for commands, boolean expressions, and arithmetic expressions.

$$\rightarrow_{\mathbf{Com}} \subseteq \mathbf{Com} \times \mathbf{Store} \times \mathbf{Com} \times \mathbf{Store}$$
$$\rightarrow_{\mathbf{Bexp}} \subseteq \mathbf{Bexp} \times \mathbf{Store} \times \mathbf{Bexp} \times \mathbf{Store}$$
$$\rightarrow_{\mathbf{Aexp}} \subseteq \mathbf{Aexp} \times \mathbf{Store} \times \mathbf{Aexp} \times \mathbf{Store}$$

For brevity, we will overload the symbol $\rightarrow$ and use it to refer to all of these relations. Which relation is being used will be clear from context. The evaluation rules for arithmetic and boolean expressions are similar to the ones we've seen before. However, note that since the arithmetic expressions no longer contain assignment, arithmetic and boolean expressions can not update the store.

Arithmetic expressions

$$\frac{n = \sigma(x)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, n \rangle}$$

$$\frac{\langle \sigma, a_1 \rangle \rightarrow \langle \sigma, a_1' \rangle}{\langle \sigma, a_1 + a_2 \rangle \rightarrow \langle \sigma, a_1' + a_2 \rangle} \qquad \frac{\langle \sigma, a_2 \rangle \rightarrow \langle \sigma, a_2' \rangle}{\langle \sigma, n + a_2 \rangle \rightarrow \langle \sigma, n + a_2' \rangle} \qquad \frac{p = n + m}{\langle \sigma, n + m \rangle \rightarrow \langle \sigma, p \rangle}$$

$$\frac{\langle \sigma, a_1 \rangle \rightarrow \langle \sigma, a_1' \rangle}{\langle \sigma, a_1 \times a_2 \rangle \rightarrow \langle \sigma, a_1' \times a_2 \rangle} \qquad \frac{\langle \sigma, a_2 \rangle \rightarrow \langle \sigma, a_2' \rangle}{\langle \sigma, n \times a_2 \rangle \rightarrow \langle \sigma, n \times a_2' \rangle} \qquad \frac{p = n \times m}{\langle \sigma, n \times m \rangle \rightarrow \langle \sigma, p \rangle}$$

Boolean expressions

$$\frac{\langle \sigma, a_1 \rangle \to \langle \sigma, a_1' \rangle}{\langle \sigma, a_1 < a_2 \rangle \to \langle \sigma, a_1' < a_2 \rangle} \qquad \frac{\langle \sigma, a_2 \rangle \to \langle \sigma, a_2' \rangle}{\langle \sigma, n < a_2 \rangle \to \langle \sigma, n < a_2' \rangle}$$

$$\frac{n < m}{\langle \sigma, n < m \rangle \to \langle \sigma, \textbf{true} \rangle} \qquad \frac{n \geq m}{\langle \sigma, n < m \rangle \to \langle \sigma, \textbf{false} \rangle}$$

Commands

$$\frac{\langle \sigma, e \rangle \to \langle \sigma, e' \rangle}{\langle \sigma, x := e \rangle \to \langle \sigma, x := e' \rangle} \qquad \frac{}{\langle \sigma, x := n \rangle \to \langle \sigma[x \mapsto n], \textbf{skip} \rangle}$$

$$\frac{\langle \sigma, c_1 \rangle \to \langle \sigma', c_1' \rangle}{\langle \sigma, c_1; c_2 \rangle \to \langle \sigma', c_1'; c_2 \rangle} \qquad \frac{}{\langle \sigma, \textbf{skip}; c_2 \rangle \to \langle \sigma, c_2 \rangle}$$

For if commands, we reduce the test until we get **true** or **false** and then we execute the appropriate branch:

$$\frac{\langle \sigma, b \rangle \to \langle \sigma, b' \rangle}{\langle \sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \rangle \to \langle \sigma, \textbf{if } b' \textbf{ then } c_1 \textbf{ else } c_2 \rangle}$$

$$\frac{}{\langle \sigma, \textbf{if true then } c_1 \textbf{ else } c_2 \rangle \to \langle \sigma, c_1 \rangle} \qquad \frac{}{\langle \sigma, \textbf{if false then } c_1 \textbf{ else } c_2 \rangle \to \langle \sigma, c_2 \rangle}$$

For while loops, the above strategy doesn't work (why?). Instead, we use the following rule, which can be thought of as "unrolling" the loop, one iteration at a time.

$$\frac{}{\langle \sigma, \textbf{while } b \textbf{ do } c \rangle \to \langle \sigma, \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip} \rangle}$$

We can now take a concrete program and see how it executes under the above rules. Consider we execute the program

$$\mathsf{foo} := 3; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5$$

The execution works as follows:

$$
\begin{aligned}
& \langle \sigma, \mathsf{foo} := 3; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle \\
\to\ & \langle \sigma', \textbf{skip}; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle && \text{where } \sigma' = \sigma[\mathsf{foo} \mapsto 3] \\
\to\ & \langle \sigma', \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle \\
\to\ & \langle \sigma', \textbf{if } \mathsf{foo} < 4 \textbf{ then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma', \textbf{if } 3 < 4 \textbf{ then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma', \textbf{if true then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma', \mathsf{foo} := \mathsf{foo} + 5; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle \\
\to\ & \langle \sigma', \mathsf{foo} := 3 + 5; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle \\
\to\ & \langle \sigma', \mathsf{foo} := 8; \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle \\
\to\ & \langle \sigma'', \textbf{while } \mathsf{foo} < 4 \textbf{ do } \mathsf{foo} := \mathsf{foo} + 5 \rangle && \text{where } \sigma'' = \sigma'[\mathsf{foo} \mapsto 8] \\
\to\ & \langle \sigma'', \textbf{if } \mathsf{foo} < 4 \textbf{ then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma'', \textbf{if } 8 < 4 \textbf{ then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma'', \textbf{if false then } (\mathsf{foo} := \mathsf{foo} + 5; W) \textbf{ else skip} \rangle \\
\to\ & \langle \sigma'', \textbf{skip} \rangle
\end{aligned}
$$

where W is an abbreviation for the while loop **while** foo < 4 **do** foo $:=$ foo $+ 5$.

2 Large-step operational semantics for IMP

We define large-step evaluation relations for arithmetic expressions, boolean expressions, and commands. The relation for arithmetic expressions relates an arithmetic expression and store to the integer value that the expression evaluates to. For boolean expressions, the final value is in $\mathbf{Bool} = \{\textbf{true}, \textbf{false}\}$. For commands, the final value is a store.

$$
\begin{aligned}
\Downarrow_{\mathbf{Aexp}} &\subseteq \mathbf{Aexp} \times \mathbf{Store} \times \mathbf{Int} \\
\Downarrow_{\mathbf{Bexp}} &\subseteq \mathbf{Bexp} \times \mathbf{Store} \times \mathbf{Bool} \\
\Downarrow_{\mathbf{Com}} &\subseteq \mathbf{Com} \times \mathbf{Store} \times \mathbf{Store}
\end{aligned}
$$

Again, we overload the symbol $\Downarrow$ and use it for any of these three relations; which relation is intended will be clear from context. We also use infix notation, for example writing $\langle \sigma, c \rangle \Downarrow \sigma'$ if $(c, \sigma, \sigma') \in \Downarrow_{\mathbf{Com}}$.

Arithmetic expressions.

$$
\frac{}{\langle \sigma, n \rangle \Downarrow n} \qquad \frac{\sigma(x) = n}{\langle \sigma, x \rangle \Downarrow n}
$$

$$
\frac{\langle \sigma, a_1 \rangle \Downarrow n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow n_2 \qquad n = n_1 + n_2}{\langle \sigma, a_1 + a_2 \rangle \Downarrow n} \qquad \frac{\langle \sigma, a_1 \rangle \Downarrow n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow n_2 \qquad n = n_1 \times n_2}{\langle \sigma, a_1 \times a_2 \rangle \Downarrow n}
$$

Boolean expressions.

$$\frac{}{\langle \sigma, \textbf{true} \rangle \Downarrow \textbf{true}} \qquad \frac{}{\langle \sigma, \textbf{false} \rangle \Downarrow \textbf{false}}$$

$$\frac{\langle \sigma, a_1 \rangle \Downarrow n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow n_2 \qquad n_1 < n_2}{\langle \sigma, a_1 < a_2 \rangle \Downarrow \textbf{true}} \qquad \frac{\langle \sigma, a_1 \rangle \Downarrow n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow n_2 \qquad n_1 \geq n_2}{\langle \sigma, a_1 < a_2 \rangle \Downarrow \textbf{false}}$$

Commands.

$$\textsc{Skip}\ \frac{}{\langle \sigma, \textbf{skip} \rangle \Downarrow \sigma} \qquad \textsc{Assgn}\ \frac{\langle \sigma, e \rangle \Downarrow n}{\langle \sigma, x := e \rangle \Downarrow \sigma[x \mapsto n]} \qquad \textsc{Seq}\ \frac{\langle \sigma, c_1 \rangle \Downarrow \sigma' \qquad \langle \sigma', c_2 \rangle \Downarrow \sigma''}{\langle \sigma, c_1; c_2 \rangle \Downarrow \sigma''}$$

$$\textsc{If-T}\ \frac{\langle \sigma, b \rangle \Downarrow \textbf{true} \qquad \langle \sigma, c_1 \rangle \Downarrow \sigma'}{\langle \sigma, \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rangle \Downarrow \sigma'} \qquad \textsc{If-F}\ \frac{\langle \sigma, b \rangle \Downarrow \textbf{false} \qquad \langle \sigma, c_2 \rangle \Downarrow \sigma'}{\langle \sigma, \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rangle \Downarrow \sigma'}$$

$$\textsc{While-F}\ \frac{\langle \sigma, b \rangle \Downarrow \textbf{false}}{\langle \sigma, \textbf{while}\ b\ \textbf{do}\ c \rangle \Downarrow \sigma}$$

$$\textsc{While-T}\ \frac{\langle \sigma, b \rangle \Downarrow \textbf{true} \qquad \langle \sigma, c \rangle \Downarrow \sigma' \qquad \langle \sigma', \textbf{while}\ b\ \textbf{do}\ c \rangle \Downarrow \sigma''}{\langle \sigma, \textbf{while}\ b\ \textbf{do}\ c \rangle \Downarrow \sigma''}$$

It's interesting to see that the rule for while loops does not rely on using an if command (as in the case of small-step semantics). Why does this rule work?

2.1 Command equivalence

The small-step operational semantics suggests that the loop **while** b **do** c should be equivalent to the command **if** b **then** $(c;$ **while** b **do** $c)$ **else skip**. Can we show that this indeed the case that the language is defined using the above large-step evaluation?

First, we need to to be more precise about what "equivalent commands" mean. Our formal model allows us to define this concept using large-step evaluations as follows. (One can write a similar definition using $\rightarrow^*$ in small-step semantics.)

Definition (Equivalence of commands)**.** Two commands c and c' are equivalent (written $c \sim c'$) if, for any stores σ and σ', we have

$$\langle \sigma, c \rangle \Downarrow \sigma' \iff \langle \sigma, c' \rangle \Downarrow \sigma'.$$

We can now state and prove the claim that **while** b **do** c and **if** b **then** $(c;$ **while** b **do** $c)$ **else skip** are equivalent.

Theorem. *For all* $b \in \mathbf{Bexp}$ *and* $c \in \mathbf{Com}$ *we have*

$$\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ c \sim \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ (c;\ \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ c)\ \textbf{\textit{else skip}}.$$

Proof. Let W be an abbreviation for **while** b **do** c. We want to show that for all stores σ, σ', we have:

$$\langle \sigma, W \rangle \Downarrow \sigma' \text{ if and only if } \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip} \Downarrow \sigma'$$

For this, we must show that both directions ($\Longrightarrow$ and $\Longleftarrow$) hold. We'll show only direction $\Longrightarrow$; the other is similar.

Assume that σ and σ' are stores such that $\langle \sigma, W \rangle \Downarrow \sigma'$. It means that there is some derivation that proves for this fact. Inspecting the evaluation rules, we see that there are two possible rules whose conclusions match this fact: WHILE-F and WHILE-T. We analyze each of them in turn.

- WHILE-F. The derivation must look like the following.

$$\textsc{While-F}\ \frac{\begin{array}{c}\vdots^{1}\\ \langle \sigma, b \rangle \Downarrow \textbf{false}\end{array}}{\langle \sigma, W \rangle \Downarrow \sigma}$$

  Here, we use $\vdots^{1}$ to refer to the derivation of $\langle \sigma, b \rangle \Downarrow \textbf{false}$. Note that in this case, $\sigma' = \sigma$.

  We can use $\vdots^{1}$ to derive a proof tree showing that the evaluation of **if** b **then** $(c; W)$ **else skip** yields the same final state σ:

$$\textsc{If-F}\ \frac{\begin{array}{c}\vdots^{1}\\ \langle \sigma, b \rangle \Downarrow \textbf{false}\end{array} \qquad \textsc{Skip}\ \frac{}{\langle \sigma, \textbf{skip} \rangle \Downarrow \sigma}}{\langle \sigma, \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip} \rangle \Downarrow \sigma}$$

- WHILE-T. In this case, the derivation has the following form.

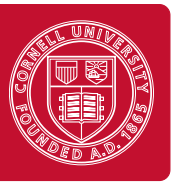
$$\textsc{While-T}\ \frac{\begin{array}{c}\vdots^{2}\\ \langle \sigma, b \rangle \Downarrow \textbf{true}\end{array} \qquad \begin{array}{c}\vdots^{3}\\ \langle \sigma, c \rangle \Downarrow \sigma''\end{array} \qquad \begin{array}{c}\vdots^{4}\\ \langle \sigma'', W \rangle \Downarrow \sigma'\end{array}}{\langle \sigma, W \rangle \Downarrow \sigma'}$$

  We can use subderivations $\vdots^{2}$, $\vdots^{3}$, and $\vdots^{4}$ to show that the evaluation of **if** b **then** $(c; W)$ **else skip** yields the same final state σ.

$$\textsc{If-T}\ \frac{\begin{array}{c}\vdots^{2}\\ \langle \sigma, b \rangle \Downarrow \textbf{true}\end{array} \qquad \textsc{Seq}\ \frac{\begin{array}{c}\vdots^{3}\\ \langle \sigma, c \rangle \Downarrow \sigma''\end{array} \qquad \begin{array}{c}\vdots^{4}\\ \langle \sigma'', W \rangle \Downarrow \sigma'\end{array}}{\langle \sigma, c; W \rangle \Downarrow \sigma'}}{\langle \sigma, \textbf{if } b \textbf{ then } (c; W) \textbf{ else skip} \rangle \Downarrow \sigma'}$$

Hence, we showed that in each of the two possible cases, the command **if** b **then** $(c; W)$ **else skip** evaluates to the same final state as the command W. □

CS 4110 – Programming Languages and Logics
Lecture #6: IMP Properties



1 Equivalence of Semantics

The small-step and large-step semantics are equivalent as captured by the following theorem.

Theorem. *For all commands c and stores σ and σ' we have*

$$\langle \sigma, c \rangle \to^* \langle \sigma', \textbf{\textit{skip}} \rangle \text{ if and only if } \langle \sigma, c \rangle \Downarrow \sigma'.$$

The proof is left as an exercise...

2 Non-Termination

For a given command c and initial state σ, the execution of the command may *terminate* with some final store σ', or it may *diverge* and never yield a final state. For example, the command

$$\textbf{while true do } \mathsf{foo} := \mathsf{foo} + 1$$

always diverges while

$$\textbf{while } 0 < \mathsf{i} \textbf{ do } \mathsf{i} := \mathsf{i} + 1$$

diverges if and only if the value of variable i in the initial state is positive.

If $\langle \sigma, c \rangle$ is a diverging configuration then there is no state σ such that

$$\langle \sigma, c \rangle \Downarrow \sigma' \qquad \text{or} \qquad \langle \sigma, c \rangle \to^* \langle \sigma', \textbf{skip} \rangle.$$

However, in small-step semantics, diverging computations generate an infinite sequence:

$$\langle \sigma, c \rangle \to \langle \sigma_1, c_1 \rangle \to \langle \sigma_2, c_2 \rangle \to \dots$$

Hence, small-step semantics allow us to state and prove properties about programs that may diverge. Later in the course, we will specify and prove properties that are of interest in potentially diverging computations.

3 Determinism

The semantics of IMP (both small-step and large-step) are *deterministic*. For example, each IMP command c and each initial store σ evaluates to at most one final store.

Theorem. *For all commands c and stores σ, σ_1, and σ_2, if $\langle \sigma, c \rangle \Downarrow \sigma_1$ and $\langle \sigma, c \rangle \Downarrow \sigma_2$ then $\sigma_1 = \sigma_2$.*

To prove this theorem, we need an induction. But structural induction on the command c will not work. (Why? Which case breaks?) Instead, we need to perform induction on the derivation of $\langle \sigma, c \rangle \Downarrow \sigma_1$. We first introduce some useful notation.

Let $\mathcal{D}$ be a derivation. We write $\mathcal{D} \Vdash y$ if $\mathcal{D}$ is a derivation of y, that is, if the conclusion of $\mathcal{D}$ is y. For example, if $\mathcal{D}$ is the following derivation

$$\dfrac{\dfrac{\overline{\langle \sigma, 6 \rangle \Downarrow 6} \qquad \overline{\langle \sigma, 7 \rangle \Downarrow 7}}{\langle \sigma, 6 \times 7 \rangle \Downarrow 42}}{\langle \sigma, \mathsf{i} := 6 \times 7 \rangle \Downarrow \sigma[\mathsf{i} \mapsto 42]}$$

then we have $\mathcal{D} \Vdash \langle \sigma, \mathsf{i} := 42 \rangle \Downarrow \sigma[\mathsf{i} \mapsto 42]$.

Let $\mathcal{D}$ and $\mathcal{D}'$ be derivations. We say that $\mathcal{D}'$ is an *immediate subderivation* of $\mathcal{D}$ if $\mathcal{D}'$ is a derivation of one of the premises used in the final rule in the derivation $\mathcal{D}$. For example, the derivation

$$\dfrac{\overline{\langle \sigma, 6 \rangle \Downarrow 6} \qquad \overline{\langle \sigma, 7 \rangle \Downarrow 7}}{\langle \sigma, 6 \times 7 \rangle \Downarrow 42}$$

is an immediate subderivation of

$$\dfrac{\dfrac{\overline{\langle \sigma, 6 \rangle \Downarrow 6} \qquad \overline{\langle \sigma, 7 \rangle \Downarrow 7}}{\langle \sigma, 6 \times 7 \rangle \Downarrow 42}}{\langle \sigma, \mathsf{i} := 6 \times 7 \rangle \Downarrow \sigma[\mathsf{i} \mapsto 42]}$$

In a proof by induction on derivations, we assume that the property P being proved holds for all immediate subderivations, and we show that it holds of the conclusion.

Proof. As $\langle \sigma, c \rangle \Downarrow \sigma_1$, there is a derivation $\mathcal{D}_1$ such that $\mathcal{D}_1 \Vdash \langle \sigma, c \rangle \Downarrow \sigma_1$. Similarly, as $\langle \sigma, c \rangle \Downarrow \sigma_2$, there is a derivation $\mathcal{D}_2$ such that $\mathcal{D}_2 \Vdash \langle \sigma, c \rangle \Downarrow \sigma_2$.

We proceed by induction on the derivation $\mathcal{D}_1 \Vdash \langle \sigma, c \rangle \Downarrow \sigma_1$. We assume that the induction hypothesis holds for immediate subderivations of $\mathcal{D}_1$. In this case, the induction hypothesis P is:

$$P(\mathcal{D}) = \forall c \in \mathbf{Com}.\ \forall \sigma, \sigma', \sigma'' \in \mathbf{Store},\ \text{if } \mathcal{D} \Vdash \langle \sigma, c \rangle \Downarrow \sigma' \text{ and } \langle \sigma, c \rangle \Downarrow \sigma'' \text{ then } \sigma' = \sigma''.$$

We analyze the possible cases for the last rule used in $\mathcal{D}_1$.

Case SKIP: In this case

$$\mathcal{D}_1 = \dfrac{\vdots}{\langle \sigma, \mathbf{skip} \rangle \Downarrow \sigma}\ \textsc{Skip}$$

and we have $c = \mathbf{skip}$ and $\sigma_1 = \sigma$. Since the rule SKIP is the only rule that has the command **skip** in its conclusion, the last rule used in $\mathcal{D}_2$ must also be SKIP, and so we have $\sigma_2 = \sigma$ and the result holds.

Case ASSGN: In this case

$$\mathcal{D}_1 = \textsc{Assgn}\,\frac{\begin{array}{c}\vdots\\ \hline \langle \sigma, a\rangle \Downarrow n\end{array}}{\langle \sigma, x := a\rangle \Downarrow \sigma[x \mapsto n]}\,,$$

and we have $c = x := a$ and $\sigma_1 = \sigma[x \mapsto n]$. The last rule used in $\mathcal{D}_2$ must also be ASSGN, and so we have $\sigma_2 = \sigma[x \mapsto n]$ and the result holds.[1]

Case SEQ: In this case

$$\mathcal{D}_1 = \textsc{Seq}\,\frac{\begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_1'\end{array} \qquad \begin{array}{c}\vdots\\ \hline \langle \sigma_1', c_2\rangle \Downarrow \sigma_1\end{array}}{\langle \sigma, c_1; c_2\rangle \Downarrow \sigma_1}\,,$$

and we have $c = c_1; c_2$. The last rule used in $\mathcal{D}_2$ must also be SEQ, and so we have

$$\mathcal{D}_2 = \textsc{Seq}\,\frac{\begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_2'\end{array} \qquad \begin{array}{c}\vdots\\ \hline \langle \sigma_2', c_2\rangle \Downarrow \sigma_2\end{array}}{\langle \sigma, c_1; c_2\rangle \Downarrow \sigma_2}\,.$$

By the inductive hypothesis applied to the derivation $\begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_1'\end{array}$, we have $\sigma_1' = \sigma_2'$. By another application of the inductive hypothesis to $\begin{array}{c}\vdots\\ \hline \langle \sigma_1', c_2\rangle \Downarrow \sigma_1\end{array}$, we have $\sigma_1 = \sigma_2$ and the result holds.

Case IF-T: Here we have

$$\mathcal{D}_1 = \textsc{If-T}\,\frac{\begin{array}{c}\vdots\\ \hline \langle \sigma, b\rangle \Downarrow \textbf{true}\end{array} \qquad \begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_1\end{array}}{\langle \sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \Downarrow \sigma_1}\,,$$

and we have $c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2$. The last rule used in $\mathcal{D}_2$ must also be IF-T and so we have

$$\mathcal{D}_2 = \textsc{If-T}\,\frac{\begin{array}{c}\vdots\\ \hline \langle \sigma, b\rangle \Downarrow \textbf{true}\end{array} \qquad \begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_2\end{array}}{\langle \sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \Downarrow \sigma_2}\,.$$

The result holds by the inductive hypothesis applied to the derivation $\begin{array}{c}\vdots\\ \hline \langle \sigma, c_1\rangle \Downarrow \sigma_1\end{array}$.

Case IF-F: Similar to the case for IF-T.

Case WHILE-F: Straightforward, similar to the case for SKIP.

[1] Strictly speaking, we also need to argue that the evaluation of a is deterministic. In this proof we will tacitly assume deterministic evaluation of arithmetic and boolean expressions.

Case WHILE-T: Here we have

$$\mathcal{D}_1 = \text{WHILE-T}\ \dfrac{\dfrac{\vdots}{\langle \sigma, b\rangle \Downarrow \textbf{true}} \quad \dfrac{\vdots}{\langle \sigma, c_1\rangle \Downarrow \sigma_1'} \quad \dfrac{\vdots}{\langle \sigma_1', c\rangle \Downarrow \sigma_1}}{\langle \sigma, \textbf{while}\ b\ \textbf{do}\ c_1\rangle \Downarrow \sigma_1},$$

and we have $c = \textbf{while}\ b\ \textbf{do}\ c_1$. The last rule used in $\mathcal{D}_2$ must also be WHILE-T, and so we have

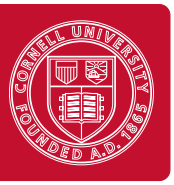
$$\mathcal{D}_2 = \text{WHILE-T}\ \dfrac{\dfrac{\vdots}{\langle \sigma, b\rangle \Downarrow \textbf{true}} \quad \dfrac{\vdots}{\langle \sigma, c_1\rangle \Downarrow \sigma_2'} \quad \dfrac{\vdots}{\langle \sigma_2', c\rangle \Downarrow \sigma_2}}{\langle \sigma, \textbf{while}\ b\ \textbf{do}\ c_1\rangle \Downarrow \sigma_2}.$$

By the inductive hypothesis applied to the derivation $\dfrac{\vdots}{\langle \sigma, c_1\rangle \Downarrow \sigma_1'}$, we have $\sigma_1' = \sigma_2'$. By another application of the inductive hypothesis, to the derivation $\dfrac{\vdots}{\langle \sigma_1', c\rangle \Downarrow \sigma_1}$, we have $\sigma_1 = \sigma_2$ and the result holds.

Note that even though $c = \textbf{while}\ b\ \textbf{do}\ c_1$ appears in the derivation of $\langle \sigma, \textbf{while}\ b\ \textbf{do}\ c_1\rangle \Downarrow \sigma_1$, we do not run in to problems, as the induction is over the *derivation*, not over the structure of the command.

□

CS 4110 – Programming Languages and Logics
Lecture #7: Denotational Semantics



We have now seen two operational models for programming languages: small-step and large-step. In this lecture, we consider a different semantic model, called *denotational semantics*.

The idea in denotational semantics is to express the meaning of a program as the mathematical function that expresses what the program computes. We can think of an IMP program c as a function from stores to stores: given an an initial store, the program produces a final store. For example, the program $\mathsf{foo} := \mathsf{bar} + 1$ can be thought of as a function that when given an input store σ, produces a final store σ' that is identical to σ except that it maps foo to the integer $\sigma(\mathsf{bar}) + 1$; that is, $\sigma' = \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{bar}) + 1]$. We will model programs as functions from input stores to output stores. As opposed to operational models, which tell us *how* programs execute, the denotational model shows us *what* programs compute.

1 A Denotational Semantics for IMP

For each program c, we write $\mathcal{C}[\![c]\!]$ for the *denotation* of c, that is, the mathematical function that c represents:

$$\mathcal{C}[\![c]\!] : \mathbf{Store} \rightharpoonup \mathbf{Store}.$$

Note that $\mathcal{C}[\![c]\!]$ is actually a partial function (as opposed to a total function), both because the store may not be defined on the free variables of the program and because program may not terminate for certain input stores. The function $\mathcal{C}[\![c]\!]$ is not defined for non-terminating programs as they have no corresponding output stores.

We will write $\mathcal{C}[\![c]\!]\sigma$ for the result of applying the function $\mathcal{C}[\![c]\!]$ to the store σ. That is, if f is the function that $\mathcal{C}[\![c]\!]$ denotes, then we write $\mathcal{C}[\![c]\!]\sigma$ to mean the same thing as $f(\sigma)$.

We must also model expressions as functions, this time from stores to the values they represent. We will write $\mathcal{A}[\![a]\!]$ for the denotation of arithmetic expression a, and $\mathcal{B}[\![b]\!]$ for the denotation of boolean expression b.

$$\mathcal{A}[\![a]\!] : \mathbf{Store} \rightharpoonup \mathbf{Int}$$
$$\mathcal{B}[\![b]\!] : \mathbf{Store} \rightharpoonup \{\mathbf{true}, \mathbf{false}\}$$

Now we want to define these functions. To make it easier to write down these definitions, we will describe (partial) functions using sets of pairs. More precisely, we will represent a partial map $f : A \rightharpoonup B$ as a set of pairs $F = \{(a, b) \mid a \in A \text{ and } b = f(a) \in B\}$ such that, for each $a \in A$, there is at most one pair of the form $(a, \_)$ in the set. Hence $(a, b) \in F$ is the same as $b = f(a)$.

We can now define denotations for IMP. We start with the denotations of expressions:

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n)\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x))\} \\
\mathcal{A}[\![a_1 + a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n = n_1 + n_2\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![\textbf{true}]\!] &= \{(\sigma, \textbf{true})\} \\
\mathcal{B}[\![\textbf{false}]\!] &= \{(\sigma, \textbf{false})\} \\
\mathcal{B}[\![a_1 < a_2]\!] &= \{(\sigma, \textbf{true}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 < n_2\} \cup \\
&\quad \{(\sigma, \textbf{false}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 \geq n_2\}
\end{aligned}$$

The denotations for commands are as follows:

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \{(\sigma, \sigma)\} \\
\mathcal{C}[\![x := a]\!] &= \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] &= \{(\sigma, \sigma') \mid \exists \sigma''.\ ((\sigma, \sigma'') \in \mathcal{C}[\![c_1]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![c_2]\!])\}
\end{aligned}$$

Note that $\mathcal{C}[\![c_1; c_2]\!] = \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$, where $\circ$ is the composition of relations, defined as follows: if $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$ then $R_2 \circ R_1 \subseteq A \times C$ is $R_2 \circ R_1 = \{(a, c) \mid \exists b \in B.\ (a, b) \in R_1 \wedge (b, c) \in R_2\}$.) If $\mathcal{C}[\![c_1]\!]$ and $\mathcal{C}[\![c_2]\!]$ are total functions, then $\circ$ is function composition.

$$\begin{aligned}
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])\}
\end{aligned}$$

But now we have a problem: the last "definition" is not really a definition because it expresses $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ in terms of itself! This is not a definition but a recursive equation. What we want is the solution to this equation.

2 Fixed points

We gave a recursive equation that the function $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ must satisfy. To understand some of the issues involved, let's consider a simpler example. Consider the following equation for a function $f : \mathbb{N} \to \mathbb{N}$.

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases} \tag{1}$$

This is not a definition for f, but rather an equation that we want f to satisfy. What function, or functions, satisfy this equation for f? The only solution to this equation is the function $f(x) = x^2$.

In general, there may be no solutions for a recursive equation (e.g., there are no functions $g : \mathbb{N} \to \mathbb{N}$ that satisfy the recursive equation $g(x) = g(x) + 1$), or multiple solutions (e.g., find two functions $g : \mathbb{R} \to \mathbb{R}$ that satisfy $g(x) = 4 \times g(\frac{x}{2})$).

We can compute solutions to such equations by building successive approximations. Each approximation is closer and closer to the solution. To solve the recursive equation for f, we start with the partial function $f_0 = \emptyset$ (i.e., f_0 is the empty relation; it is a partial function with the empty set for it's domain). We compute successive approximations using the recursive equation.

$$
\begin{aligned}
f_0 &= \emptyset \\
f_1 &= \begin{cases} 0 & \text{if } x = 0 \\ f_0(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0)\} \\
f_2 &= \begin{cases} 0 & \text{if } x = 0 \\ f_1(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0),(1,1)\} \\
f_3 &= \begin{cases} 0 & \text{if } x = 0 \\ f_2(x-1) + 2x - 1 & \text{otherwise} \end{cases} \\
&= \{(0,0),(1,1),(2,4)\} \\
&\vdots
\end{aligned}
$$

This sequence of successive approximations f_i gradually builds the function $f(x) = x^2$.

We can model this process of successive approximations using a higher-order function F that takes one approximation f_k and returns the next approximation f_{k+1}:

$$F : (\mathbb{N} \rightharpoonup \mathbb{N}) \to (\mathbb{N} \rightharpoonup \mathbb{N})$$

where

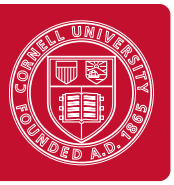
$$(F(f))(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) + 2x - 1 & \text{otherwise} \end{cases}$$

A solution to the recursive equation 1 is a function f such that $f = F(f)$. In general, given a function $F : A \to A$, we have that $a \in A$ is a *fixed point* of F if $F(a) = a$. We also write $a = \text{fix}(F)$ to indicate that a is a fixed point of F.

So the solution to the recursive equation 1 is a fixed-point of the higher-order function F. We can compute this fixed point iteratively, starting with $f_0 = \emptyset$ and at each iteration computing $f_{k+1} = F(f_k)$. The fixed point is the limit of this process:

$$
\begin{aligned}
f &= \text{fix}(F) \\
&= f_0 \cup f_1 \cup f_2 \cup f_3 \cup \ldots \\
&= \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \ldots \\
&= \bigcup_{i \geq 0} F^i(\emptyset)
\end{aligned}
$$

CS 4110 – Programming Languages and Logics
Lecture #8: Denotational Semantics Examples



Last time we defined the denotational semantics of IMP:

$$
\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n)\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x))\} \\
\mathcal{A}[\![a_1 + a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n = n_1 + n_2\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}[\![\textbf{true}]\!] &= \{(\sigma, \textbf{true})\} \\
\mathcal{B}[\![\textbf{false}]\!] &= \{(\sigma, \textbf{false})\} \\
\mathcal{B}[\![a_1 < a_2]\!] &= \{(\sigma, \textbf{true}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 < n_2\} \cup \\
&\quad \{(\sigma, \textbf{false}) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \wedge n_1 \geq n_2\} \\
\mathcal{C}[\![\textbf{skip}]\!] &= \{(\sigma, \sigma)\} \\
\mathcal{C}[\![x := a]\!] &= \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] &= \{(\sigma, \sigma') \mid \exists \sigma''.\ ((\sigma, \sigma'') \in \mathcal{C}[\![c_1]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![c_2]\!])\} \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] &= \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_2]\!]\} \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \mathit{fix}(F) \\
\text{where } F(f) &= \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma, \sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma', \sigma''.\ (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \wedge (\sigma', \sigma'') \in f\}
\end{aligned}
$$

In this lecture we'll prove Kleene's fixpoint theorem, which shows that the fixed point used to define the semantics of **while** commands exists, and work through examples of reasoning using the denotational semantics.

1 Kleene's Fixpoint Theorem

Definition (Scott Continuity)**.** A function F from U to U is said to be *Scott-continuous* if for every chain $X_1 \subseteq X_2 \subseteq \ldots$ we have $F(\bigcup_i X_i) = \bigcup_i F(X_i)$.

It is not hard to show that if F is Scott continuous, then it is also monotonic—that is, $X \subseteq Y$ implies $F(X) \subseteq F(Y)$. The proof of this fact is left as an exercise.

Theorem (Kleene Fixpoint)**.** *Let F be a Scott-continuous function. The least fixed point of F is $\bigcup_i F^i(\emptyset)$.*

Proof. Let $X = \bigcup_i F^i(\emptyset)$.

First, we will prove that X is a fixed point of F—that is, $F(X) = X$. We calculate as follows:

$$\begin{array}{rcll} F(X) & = & F(\bigcup_i F^i(\emptyset)) & \text{By definition of } X \\ & = & \bigcup_i F(F^i(\emptyset)) & \text{By Scott continuity} \\ & = & \bigcup_i F^{i+1}(\emptyset) & \\ & = & \emptyset \cup \bigcup_i F^{i+1}(\emptyset) & \\ & = & F^0(\emptyset) \cup \bigcup_i F^{i+1}(\emptyset) & \\ & = & \bigcup_i F^i(\emptyset) & \\ & = & X & \end{array}$$

Second, we will show that X is the least fixed point of F. Suppose that Y is some other arbitrary fixed point of F. By induction, we can easily show that $F^i(\emptyset) \subseteq Y$ for all i. For the base case, i is 0 and we trivially have $F^0(\emptyset) = \emptyset \subseteq Y$. For the inductive case, we assume that $F^i(\emptyset) \subseteq Y$ and prove that $F^{i+1}(\emptyset) \subseteq Y$. By our inductive hypothesis and the fact that F is monotone, we have that $F(F^{i+1}(\emptyset)) \subseteq F(Y)$. As Y is a fixed point we also have $F(Y) = Y$ and so $F^{i+1}(\emptyset) \subseteq Y$. Then, since every element of the chain

$$F^0\emptyset \subseteq F^1\emptyset \subseteq \ldots$$

is a subset of Y immediately we have that their union, $X = \bigcup_i F^i(\emptyset) \subseteq Y$. Hence, X is the least (with respect to $\subseteq$) fixed point of F. □

2 Reasoning

One of the key advantages of using denotational semantics compared to operational semantics is that proofs of equivalence can be carried out directly by simply calculating the denotations of programs and then arguing that they are identical. This is in contrast to operational techniques, where one must reason explicitly about low-level transitions and derivations involving ad hoc abstract machines.

As an example, to show that **skip**; c and c; **skip** are equivalent, we can calcuate as follows,

$$\begin{array}{rcl} \mathcal{C}[\![\textbf{skip}; c]\!] & = & \{(\sigma, \sigma'') \mid \exists \sigma'.(\sigma, \sigma') \in \mathcal{C}[\![\textbf{skip}]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}[\![c]\!]\} \\ & = & \{(\sigma, \sigma'') \mid (\sigma, \sigma'') \in \mathcal{C}[\![c]\!]\} \\ & = & \{(\sigma, \sigma'') \mid \exists \sigma'.(\sigma, \sigma') \in \mathcal{C}[\![c]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}[\![\textbf{skip}]\!]\} \\ & = & \mathcal{C}[\![c; \textbf{skip}]\!] \end{array}$$

using standard facts about partial functions, relations, and sets as convenient in the proof itself.

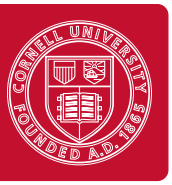
Next, consider the command $\mathcal{C}[\![\textbf{while false do } c]\!]$. By the definition of the denotational semantics, this is equal to $fix(F)$ where

$$\begin{array}{rcl} F(f) & = & \{(\sigma, \sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ & = & \{(\sigma, \sigma'') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \wedge (\sigma', \sigma'') \in f\} \end{array}$$

By the Kleene fixpoint theorem we have that $fixF = \bigcup_i F^i(\emptyset)$. It is straightforward to show by induction that since the guard is **false**, for all i we have $F^i(\emptyset) = \{(\sigma, \sigma)\}$. It follows that $fixF = \{(\sigma, \sigma)\}$, which is just $\mathcal{C}[\![\textbf{skip}]\!]$.

As an exercise, calcuate $\mathcal{C}[\![\textbf{while true do skip}]\!]$ using the same technique.

CS 4110 – Programming Languages and Logics
Lecture #9: Axiomatic Semantics



1 Introduction to axiomatic semantics

Now we turn to the third and final main style of semantics, *axiomatic semantics*. The idea in axiomatic semantics is to define meaning in terms of logical specifications that programs satisfy. This is in contrast to operational models (which show *how* programs execute) and denotational models (which show *what* programs compute). This approach to reasoning about programs and expressing program semantics was originally proposed by Floyd and Hoare and was developed further by Dijkstra and Gries.

A common way to express program specifications is in terms of pre-conditions and post-conditions:

$$\{Pre\}\ c\ \{Post\}$$

where c is a program, and Pre and $Post$ are formulas that describe properties of the program state, usually referred to as *assertions*. Such a triple is usually referred to as a *partial correctness specification* (or sometimes a "Hoare triple") and has the following meaning:

"If Pre holds before executing c, and c terminates, then $Post$ holds after c."

In other words, if we start with a store σ in which Pre holds and the execution of c with respect to σ terminates and yields a store σ', then $Post$ holds in store σ'.

Pre-conditions and post-conditions can be regarded as interfaces or contracts between the program and its clients. They help users to understand what the program is supposed to yield without needing to understand how the program executes. Typically, programmers write them as comments for procedures and functions as documentation and to make it easier to maintain programs. Such specifications are especially useful for library functions for which the source code is often not available to the users. In this case, pre-conditions and post-conditions serve as contracts between the library developers and users of the library.

However, there is no guarantee that pre-conditions and post-conditions written informally in comments are correct: the comments describe the intent of the developer, but they do not give a guarantee of correctness. Axiomatic semantics addresses this problem. It shows how to rigorously describe partial correctness statements and how to establish correctness using formal reasoning.

Note that partial correctness specifications don't ensure that the program will terminate—this is why they are called "partial". In contrast, *total correctness statements* ensure that the program terminates whenever the precondition holds. Such statements are denoted using square brackets:

$$[\ Pre\ ]\ c\ [\ Post\ ]$$

meaning:

"If Pre holds before c then c will terminate and $Post$ will hold after c."

In general a pre-condition specifies what the program expects before execution and the post-conditions specifies what guarantees the program provides (if the program terminates). Here is a simple example:

$$\{\mathsf{foo} = 0 \wedge \mathsf{bar} = i\}\ \mathsf{baz} := 0; \textbf{while}\ \mathsf{foo} \neq \mathsf{bar}\ \textbf{do}\ (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1)\ \{\mathsf{baz} = -2i\}$$

It says that if the store maps foo to 0 and bar to i before execution, then, if the program terminates, the final store will map baz to $-2i$ (i.e., -2 times the initial value of bar). Note that i is a logical variable that doesn't occur in the program and is only used to express the initial value of bar. Such variables are sometimes called *ghost variables*.

This partial correctness statement is valid. That is, it is indeed the case that if we have any store σ such that $\sigma(\mathsf{foo}) = 0$, and

$$\mathcal{C}[\![\mathsf{baz} := 0; \textbf{while}\ \mathsf{foo} \neq \mathsf{bar}\ \textbf{do}\ (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1)]\!]\sigma = \sigma',$$

then $\sigma'(\mathsf{baz}) = -2\sigma(\mathsf{bar})$.

Note that this is a *partial* correctness statement: if the pre-condition is true before c, **and** c **terminates** then the post-condition holds after c. There are some initial stores for which the program will not terminate.

The following total correctness statement is true.

$$[\mathsf{foo} = 0 \wedge \mathsf{bar} = i \wedge i \geq 0]\ \ \mathsf{baz} := 0; \textbf{while}\ \mathsf{foo} \neq \mathsf{bar}\ \textbf{do}\ (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1)\ \ [\mathsf{baz} = -2i]$$

That is, if we start with a store σ that maps foo to 0 and bar to a non-negative integer, then the execution of the command will terminate in a final store σ' with $\sigma'(\mathsf{baz}) = -2\sigma(\mathsf{bar})$.

The following partial correctness statement is not valid. (Why not?)

$$\{\mathsf{foo} = 0 \wedge \mathsf{bar} = i\}\ \mathsf{baz} := 0; \textbf{while}\ \mathsf{foo} \neq \mathsf{bar}\ \textbf{do}\ (\mathsf{baz} := \mathsf{baz} + \mathsf{foo}; \mathsf{foo} := \mathsf{foo} + 1)\ \{\mathsf{baz} = i\}$$

In the rest of our discussion of axiomatic semantics we will focus almost exclusively on partial correctness assertions.

2 Assertions

Now we turn to the following issues:

- What logic do we use for writing assertions? That is, what can we express in pre-conditions and post-condition?
- What does it mean that an assertion is valid? What does it mean that a partial correctness statement $\{Pre\}\ c\ \{Post\}$ is valid?
- How can we prove that a partial correctness statement is valid?

What can we say in pre-conditions and post-conditions? In the examples so far, we have used program variables, equality, logical variables (e.g., i), and conjunction ($\wedge$). What we allow in pre-conditions and post-conditions directly influences the sorts of program properties we can describe using partial correctness statements. We will use the set of logical formulas including comparisons

between arithmetic expressions, standard logical operators (and, or, implication, negation), as well as quantifiers (universal and existential). Assertions may also introduce logical variables that are different than the variables appearing in the program.

$$
\begin{aligned}
i, j &\in \mathbf{LVar} \\
a \in \mathbf{Aexp} &::= x \mid i \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
P, Q \in \mathbf{Assn} &::= \textsf{true} \mid \textsf{false} \mid a_1 < a_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \neg P \mid \forall i.\ P \mid \exists i.\ P
\end{aligned}
$$

Observe that the domain of boolean expressions $\mathbf{Bexp}$ is a subset of the domain of assertions. Notable additions over the syntax of boolean expression are quantifiers ($\forall$ and $\exists$). For instance, one can express the fact that variable x divides variable y using existential quantification: $\exists i.\ x \times i = y$.

3 Satisfaction and Validity

Now we would like to describe what we mean by "assertion P holds in store σ''. But to determine whether P holds or not, we need more than just the store σ (which maps program variables to their values); we also need to know the values of the logical variables. We describe those values using an interpretation I,

$$I : \mathbf{LVar} \to \mathbf{Int},$$

and define the function $\mathcal{A}_i[\![a]\!]$, which is like the denotation of expressions extended to logical variables in the obvious way:

$$
\begin{aligned}
\mathcal{A}_i[\![n]\!](\sigma, I) &= n \\
\mathcal{A}_i[\![x]\!](\sigma, I) &= \sigma(x) \\
\mathcal{A}_i[\![i]\!](\sigma, I) &= I(i) \\
\mathcal{A}_i[\![a_1 + a_2]\!](\sigma, I) &= \mathcal{A}_i[\![a_1]\!](\sigma, I) + \mathcal{A}_i[\![a_2]\!](\sigma, I)
\end{aligned}
$$

Now we can express the satisfiability of assertions as a relation $\sigma \vDash_I P$ read as "P is satisfied in store σ under interpretation I," or "store σ satisfies assertion P under interpretation I." We will write $\sigma \nvDash_I P$ whenever $\sigma \vDash_I P$ doesn't hold.

$\sigma \vDash_I \textsf{true}$	(always)
$\sigma \vDash_I a_1 < a_2$	if $\mathcal{A}_i[\![a_1]\!](\sigma, I) < \mathcal{A}_i[\![a_2]\!](\sigma, I)$
$\sigma \vDash_I P_1 \wedge P_2$	if $\sigma \vDash_I P_1$ and $\sigma \vDash_I P_2$
$\sigma \vDash_I P_1 \vee P_2$	if $\sigma \vDash_I P_1$ or $\sigma \vDash_I P_2$
$\sigma \vDash_I P_1 \Rightarrow P_2$	if $s \nvDash_I P_1$ or $\sigma \vDash_I P_2$
$\sigma \vDash_I \neg P$	if $s \nvDash_I P$
$\sigma \vDash_I \forall i.\ P$	if $\forall k \in Int.\ \sigma \vDash_{I[i \mapsto k]} P$
$\sigma \vDash_I \exists i.\ P$	if $\exists k \in Int.\ \sigma \vDash_{I[i \mapsto k]} P$

We can now say that an assertion P is valid (written $\vDash P$) if it is valid in any store, under any interpretation: $\forall \sigma, I.\ \sigma \vDash_I P$.

Having defined validity for individual assertions, we now turn to partial correctness statements. We say that a partial correctness statement $\{P\}\ c\ \{Q\}$ is satisfied in store σ and interpretation I, written $\sigma \vDash_I \{P\}\ c\ \{Q\}$, if:

$$\forall \sigma'.\ \text{if } \sigma \vDash_I P \text{ and } \mathcal{C}[\![c]\!]\sigma = \sigma' \text{ then } \sigma' \vDash_I Q$$

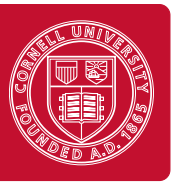
Note that this definition depends on the execution of c in the initial store σ.

Finally, we can say that a partial correctness triple is valid (written $\vDash \{P\}\ c\ \{Q\}$), if it is valid in any store and interpretation:

$$\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}.$$

Now we know what we mean when we say "assertion P holds" or "partial correctness statement $\{P\}\ c\ \{Q\}$ is valid."

CS 4110 – Programming Languages and Logics
Lecture #10: Hoare Logic



1 Hoare Logic

How do we show that a partial correctness statement $\{P\}\ c\ \{Q\}$ holds? We know that $\{P\}\ c\ \{Q\}$ is valid if it holds for all stores and interpretations: $\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}$. Furthermore, showing that $\sigma \vDash_I \{P\}\ c\ \{Q\}$ requires reasoning about the execution of command c (that is, $\mathcal{C}[\![c]\!]$), as indicated by the definition of validity.

It turns out that there is an elegant way of deriving valid partial correctness statements, without having to reason about stores, interpretations, and the execution of c. We can use a set of inference rules and axioms, called *Hoare* rules, to directly derive valid partial correctness statements. The set of rules forms a proof system known as Hoare logic.

$$\textsc{Skip}\ \frac{}{\{P\}\ \textbf{skip}\ \{P\}} \qquad \textsc{Assign}\ \frac{}{\{P[a/x]\}\ x := a\ \{P\}}$$

$$\textsc{Seq}\ \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}} \qquad \textsc{If}\ \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

$$\textsc{While}\ \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

The assertion P in the rule for while loops is essentially a loop invariant; it is an assertion that holds before and after each iteration, as shown in the premise of the rule. Therefore, it is both a pre-condition for the loop (because it holds before the first iteration); and also a post-condition for the loop (because it holds after the last iteration). The fact that P is both a pre- and post-condition for the while loop is reflected in the conclusion of the rule.

There is one more rule, the rule of consequence, which allows to strengthen pre-conditions and weaken post-conditions:

$$\textsc{Consequence}\ \frac{\vDash (P \Rightarrow P') \qquad \{P'\}\ c\ \{Q'\} \qquad \vDash (Q' \Rightarrow Q)}{\{P\}\ c\ \{Q\}}$$

These set of Hoare rules represent an inductive definition for a set of partial correctness statements $\{P\}\ c\ \{Q\}$. We will say that $\{P\}\ c\ \{Q\}$ is a theorem in Hoare logic, written $\vdash \{P\}\ c\ \{Q\}$, if we can build a finite proof tree for it.

2 Soundness and Completeness

At this point we have two kinds of partial correctness assertions:

- valid partial correctness statements $\vDash \{P\}\ c\ \{Q\}$, which hold for all stores and interpretations, according to the semantics of c, and
- Hoare logic theorems $\vdash \{P\}\ c\ \{Q\}$, that is, partial correctness statements that can be derived using the axioms and rules of Hoare logic.

The question is how do these sets relate to each other? More precisely, we have to answer two questions. First, is each Hoare logic theorem guaranteed to be valid partial correctness triple? In other words,

$$\text{does} \quad \vdash \{P\}\ c\ \{Q\} \quad \text{imply} \quad \vDash \{P\}\ c\ \{Q\}?$$

The answer is yes, and it shows that Hoare logic is sound. Soundness is important because it says that Hoare logic doesn't allow us to derive partial correctness assertions that actually don't hold. The proof of soundness requires induction on the derivations in $\vdash \{P\}\ c\ \{Q\}$ (we omit this proof).

The second question refers to the expressiveness and power of Hoare rules: can we always build a Hoare logic proof for each valid assertion? In other words,

$$\text{does} \quad \vDash \{P\}\ c\ \{Q\} \quad \text{imply} \quad \vdash \{P\}\ c\ \{Q\}?$$

The answer is a qualified yes: if $\vDash \{P\}\ c\ \{Q\}$ then there is a proof of $\{P\}\ c\ \{Q\}$ using the rules of Hoare logic, provided there are proofs for the validity of assertions that occur in the rule of consequence $\vDash (P \Rightarrow P')$ and $\vDash (Q' \Rightarrow Q)$. This result is known as the *relative completeness of Hoare logic* and is due to Cook (1974).

3 Example: Factorial

As an example illustrating how we can use Hoare logic to verify the correctness of a program, consider a program that computes the factorial of a number n:

$$\begin{array}{l}\{\mathsf{x} = n \wedge n > 0\} \\ \quad \mathsf{y} := 1; \\ \quad \textbf{while}\ \mathsf{x} > 0\ \textbf{do}\ \{ \\ \qquad \mathsf{y} := \mathsf{y} * \mathsf{x}; \\ \qquad \mathsf{x} := \mathsf{x} - 1 \\ \quad \} \\ \{\mathsf{y} = n!\}\end{array}$$

Because the derivation for this proof is somewhat large, we will go through the reasoning used to construct it step by step.

At the top level, the program is a sequence of an assignment and a loop. To use the SEQ rule, we need to find an assertion that holds after the assignment and before the loop. Examining the rule for while loops, we see that the assertion before the loop must be an invariant for the loop.

Inspecting the loop we see that it builds the factorial up in y starting with n, then multiplying it by $n-1$, then $n-2$, etc. At each iteration, x contains the next value multiplied into y, that is:

$$\mathsf{y} = n * (n-1) * ... * (\mathsf{x}+1)$$

If we multiply both sides of this equality by x! and re-write the equality we get $\mathsf{x}! * \mathsf{y} = n!$, which is an invariant for the loop. However, to make the proof go through, we need a slightly stronger invariant:

$$I = \mathsf{x}! * \mathsf{y} = n! \wedge \mathsf{x} \geq 0$$

Having identified a suitable loop invariant, let us take a step back and review where we are. We want to prove that our overall partial correctness specification is valid. To do this, we need to show two facts:

$$\{\mathsf{x} = n \wedge n > 0\}\ \mathsf{y} := 1\ \{I\} \tag{1}$$

$$\{I\}\ \textbf{while}\ \mathsf{x} > 0\ \textbf{do}\ \{\mathsf{y} := \mathsf{y} * \mathsf{x}; \mathsf{x} := \mathsf{x} - 1\}\ \{\mathsf{y} = n!\} \tag{2}$$

After showing that both (1) and (2) hold, we can use the rule SEQ to obtain the desired result.

To show (1), we use the ASSIGN axiom and obtain the following: $\{I[1/\mathsf{y}]\}\ \mathsf{y} := 1\ \{I\}$. Expanding this out, we obtain:

$$\{\mathsf{x}! * 1 = n! \wedge \mathsf{x} \geq 0\}\ y := 1\ \{\mathsf{x}! * \mathsf{y} = n! \wedge \mathsf{x} \geq 0\}$$

With the following implication,

$$\mathsf{x} = n \wedge n > 0 \implies \mathsf{x}! * 1 = n! \wedge \mathsf{x} \geq 0,$$

(which can be shown by an easy calculation) we obtain (1) using the rule CONSEQUENCE.

Now let us prove (2). To use the WHILE rule, we need to show that I is an invariant for the loop:

$$\{I \wedge \mathsf{x} > 0\}\ \mathsf{y} := \mathsf{y} * \mathsf{x}; \mathsf{x} := \mathsf{x} - 1\ \{I\} \tag{3}$$

We will show this by going backwards through the sequence of assignments:

$$\{(\mathsf{x}-1)! * \mathsf{y} = n! \wedge (\mathsf{x}-1) \geq 0\}\ \mathsf{x} := \mathsf{x} - 1\ \{I\} \tag{4}$$

$$\{(\mathsf{x}-1)! * \mathsf{y} * \mathsf{x} = n! \wedge (\mathsf{x}-1) \geq 0\}\ \mathsf{y} := \mathsf{y} * \mathsf{x}\ \{(\mathsf{x}-1)! * \mathsf{y} = n! \wedge (\mathsf{x}-1) \geq 0\} \tag{5}$$

Then, using the following implication:

$$I \wedge \mathsf{x} > 0 \implies (\mathsf{x}-1)! * \mathsf{y} * \mathsf{x} = n! \wedge (\mathsf{x}-1) \geq 0$$

we obtain (3) using CONSEQUENCE, (4), and (5). Thus, I is an invariant for the loop and so by WHILE we obtain,

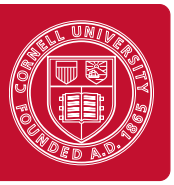
$$\{I\}\ \textbf{while}\ \mathsf{x} > 0\ \textbf{do}\ \{\mathsf{y} := \mathsf{y} * \mathsf{x}; \mathsf{x} := \mathsf{x} - 1\}\ \{I \wedge \mathsf{x} \leq 0\}$$

To finish the proof, we just have to show

$$\begin{aligned} & I \wedge \mathsf{x} \leq 0 \implies \mathsf{y} = n! \\ i.e., \quad & \mathsf{x}! * \mathsf{y} = n! \wedge \mathsf{x} \geq 0 \wedge \mathsf{x} \leq 0 \implies \mathsf{y} = n! \end{aligned}$$

which holds as $\mathsf{x} \geq 0$ and $\mathsf{x} \leq 0$ implies $\mathsf{x} = 0$ and so $\mathsf{x}! = 1$. The result follows by CONSEQUENCE.

CS 4110 – Programming Languages and Logics
Lecture #11: Relative Completeness



1 Relative Completeness

In the last lecture, we discussed the issue of completeness—i.e., whether it is possible to derive every valid partial correctness specification using the axioms and rules of Hoare logic. Unfortunately, if treated as a pure deductive system, Hoare logic cannot be complete. To see why, consider the following partial correctness specifications:

$$\{\textbf{true}\}\ \textbf{skip}\ \{P\} \qquad\qquad \{\textbf{true}\}\ c\ \{\textbf{false}\}$$

The first is valid if and only if the assertion P is valid while the second is valid if and only if the command c does not halt.

It turns out that the culprit is the CONSEQUENCE rule,

$$\textsc{Consequence}\ \frac{\vDash (P \Rightarrow P') \qquad \{P'\}\ c\ \{Q'\} \qquad \vDash (Q' \Rightarrow Q)}{\{P\}\ c\ \{Q\}}$$

which includes two premises about the validity of implications between the assertions involved.

Although we cannot have a complete proof system for first-order formulas, Hoare logic does enjoy the property stated in the following theorem:

Theorem. $\forall P, Q \in \textbf{Assn}, c \in \textbf{Com}.\ \vDash \{P\}\ c\ \{Q\}$ *implies* $\vdash \{P\}\ c\ \{Q\}$.

This result, due to Cook (1974), is known as the *relative completeness* of Hoare logic. It says that Hoare logic is no more incomplete than the language of assertions—i.e., if we had an oracle that could decide the validity of assertions, then we could decide the validity of partial correctness specifications.

2 Weakest Liberal Preconditions

Cook's proof of relative completeness depends on the notion of *weakest liberal preconditions*. Given a command c and a postcondition Q the weakest liberal precondition is the weakest assertion P such that $\{P\}\ c\ \{Q\}$ is a valid triple. Here, "weakest" means that any other valid precondition implies P. That is, P most accurately describes input states for which c either does not terminate or ends up in a state satisfying Q.

Formally, an assertion P is a weakest liberal precondition of c and Q if:

$$\forall \sigma, I.\ \sigma \vDash_I P \iff (\mathcal{C}[\![c]\!]\ \sigma) \text{ undefined } \vee (\mathcal{C}[\![c]\!]\sigma) \vDash_I Q$$

We write $wlp(c, Q)$ for the weakest liberal precondition of command c and postcondition Q. From left-to-right, the formula above states that $wlp(c, Q)$ is a valid precondition: $\vDash \{P\}\ c\ \{Q\}$. The

right-to-left implication says it is the weakest valid precondition: if another assertion R satisfies $\vDash \{R\}\ c\ \{Q\}$, then R implies P. It can be shown that weakest liberal preconditions are unique modulo equivalence.

We can calculate the weakest liberal precondition of a command as follows:

$$\begin{array}{rcl} wlp(\textbf{skip}, P) & = & P \\ wlp((\mathsf{x} := a, P) & = & P[a/\mathsf{x}] \\ wlp((c_1; c_2), P) & = & wlp(c_1, wlp(c_2, P)) \\ wlp(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, P) & = & (b \implies wlp(c_1, P)) \wedge (\neg b \implies wlp(c_2, P)) \end{array}$$

The definition of $wlp(\textbf{while}\ b\ \textbf{do}\ c, P)$ is slightly more complicated—it encodes the weakest liberal precondition for each iteration of the loop. To give the intuition, first define the weakest liberal precondition for a loop that termintes in i steps as follows:

$$\begin{array}{rcl} F_0(P) & = & \textbf{true} \\ F_{i+1}(P) & = & (\neg b \implies P) \wedge (b \implies wlp(c, F_i(P))) \end{array}$$

We can then express the weakest liberal precondition using an infinitary conjunction:

$$wlp(\textbf{while}\ b\ \textbf{do}\ c, P) = \bigwedge_i F_i(P)$$

See Winskel Chapter 7 for the details of how to encode the weakest liberal precondition for a while loop as an ordinary assertion.

To check that our definition is correct, we can prove (how?) that it yields a valid partial correctness specification:

Lemma 1.

$$\begin{array}{l} \forall c \in \mathbf{Com}, Q \in \mathbf{Assn}. \\ \quad \vDash \{wlp(c, Q)\}\ c\ \{Q\}\ \textit{and}\ \forall R \in \mathbf{Assn}.\ \vDash \{R\}\ c\ \{Q\}\ \textit{implies}\ (R \implies wlp(c, Q)) \end{array}$$

It is not hard to prove that it also yields a provable specification:

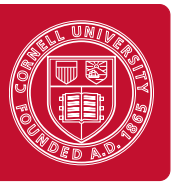
Lemma 2.

$$\forall c \in \mathbf{Com}, Q \in \mathbf{Assn}.\ \vdash \{wlp(c, Q)\}\ c\ \{Q\}$$

Relative completeness follows by a simple argument:

Proof Sketch. Let c be a command and let P and Q be assertions such that the partial correctness specification $\{P\}\ c\ \{Q\}$ is valid. By Lemma 1 we have $\vDash P \implies wlp(c, Q)$. By Lemma 2 we have $\vdash \{wlp(c, Q)\}\ c\ \{Q\}$. We conclude $\vdash \{P\}\ c\ \{Q\}$ using the CONSEQUENCE rule. □

CS 4110 – Programming Languages and Logics
Lecture #13: λ-calculus



Lambda calculus (or λ-calculus) was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s to describe functions in an unambiguous and compact manner. Many real languages are based on the lambda calculus, including Lisp, Scheme, Haskell, and ML. A key characteristic of these languages is that functions are values, just like integers and booleans are values: functions can be used as arguments to functions, and can be returned from functions.

The name "lambda calculus" comes from the use of the Greek letter lambda (λ) in function definitions. (The letter lambda has no significance.) "Calculus" means a method of calculating by the symbolic manipulation of expressions.

Intuitively, a function is a rule for determining a value from an argument. Some examples of functions in mathematics are

$$f(x) = x^3$$
$$g(y) = y^3 - 2y^2 + 5y - 6.$$

1 Syntax

The pure λ-calculus contains just function definitions (called *abstractions*), variables, and function *application* (i.e., applying a function to an argument). If we add additional data types and operations (such as integers and addition), we have an *applied* λ-calculus. In the following text, we will sometimes assume that we have integers and addition in order to give more intuitive examples.

The syntax of the pure λ-calculus is defined as follows.

$e ::=$	x	variable
	$\mid \lambda x.\, e$	abstraction
	$\mid e_1\ e_2$	application

An abstraction $\lambda x.\, e$ is a function: variable x is the *argument*, and expression e is the *body* of the function. Note that the function $\lambda x.\, e$ doesn't have a name. Assuming we have integers and arithmetic operations, the expression $\lambda x.\, x^2$ is a function that takes an argument x and returns the square of x.

An application $e_1\ e_2$ requires that e_1 is (or evaluates to) a function, and then applies the function to the expression e_2. For example, $(\lambda x.\, x^2)\ 5$ is, intuitively, equal to 25, the result of applying the squaring function $\lambda x.\, x^2$ to 5.

Here are some examples of lambda calculus expressions.

$\lambda x.\, x$	a lambda abstraction called the *identity function*
$\lambda x.\, (f\ (g\ x))$	another abstraction
$(\lambda x.\, x)\ 42$	an application
$\lambda y.\, \lambda x.\, x$	an abstraction that ignores its argument and returns the identity function

Lambda expressions extend as far to the right as possible. For example $\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, x\ (\lambda y.\, y)$, and is not the same as $(\lambda x.\, x)\ (\lambda y.\, y)$. Application is left associative. For example $e_1\ e_2\ e_3$ is the same as $(e_1\ e_2)\ e_3$. In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.

1.1 Variable binding and α-equivalence

An occurrence of a variable in an expression is either *bound* or *free*. An occurrence of a variable x in a term is bound if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identifiers are bound.

Consider the following term:

$$\lambda x.\, (x\ (\lambda y.\, y\ a)\ x)\ y$$

Both occurrences of x are bound, the first occurrence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a program has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed program in lambda calculus is a closed term.

The symbol λ is a *binding operator*, as it binds a variable within some scope (i.e., some part of the expression): variable x is bound in e in the expression $\lambda x.\, e$.

The name of bound variables is not important. Consider the mathematical integrals $\int_0^7 x^2 dx$ and $\int_0^7 y^2 dy$. They describe the same integral, even though one uses variable x and the other uses variable y in their definition. The meaning of these integrals is the same: the bound variable is just a placeholder. In the same way, we can change the name of bound variables without changing the meaning of functions. Thus $\lambda x.\, x$ is the same function as $\lambda y.\, y$. Expressions e_1 and e_2 that differ only in the name of bound variables are called *α-equivalent*, sometimes written $e_1 =_\alpha e_2$.

1.2 Higher-order functions

In lambda calculus, functions are values: functions can take functions as arguments and return functions as results. In the pure lambda calculus, every value is a function, and every result is a function!

For example, the following function takes a function f as an argument, and applies it to the value 42.

$$\lambda f.\, f\ 42$$

This function takes an argument v and returns a function that applies its own argument (a function) to v.

$$\lambda v.\, \lambda f.\, (f\ v)$$

2 Semantics

2.1 β-equivalence

Application $(\lambda x.\, e_1)\ e_2$ applies the function $\lambda x.\, e_1$ to e_2. In some ways, we would like to regard the expression $(\lambda x.\, e_1)\ e_2$ as equivalent to the expression e_1 where every (free) occurrence of x is replaced with e_2. For example, we would like to regard $(\lambda x.\, x^2)\ 5$ as equivalent to 5^2.

We write $e_1\{e_2/x\}$ to mean expression e_1 with all free occurrences of x replaced with e_2. There are several different notations to express this substitution, including $[x \mapsto e_2]e_1$ (used by Pierce), $[e_2/x]e_1$ (used by Mitchell), and $e_1[e_2/x]$ (used by Winskel).

Using our notation, we would like expressions $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$ to be equivalent.

We call this equivalence, between $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$, is called *β-equivalence*. Rewriting $(\lambda x.\, e_1)\ e_2$ into $e_1\{e_2/x\}$ is called a *β-reduction*. Given a lambda calculus expression, we may, in general, be able to perform β-reductions. This corresponds to executing a lambda calculus expression.

There may be more than one possible way to β-reduce an expression. Consider, for example, $(\lambda x.\, x+x)\ ((\lambda y.\, y)\ 5)$. We could use β-reduction to get either $((\lambda y.\, y)\ 5)+((\lambda y.\, y)\ 5)$ or $(\lambda x.\, x+x)\ 5$. The order in which we perform β-reductions results in different semantics for the lambda calculus.

2.2 Call-by-value

Call-by-value (or CBV) semantics makes sure that functions are only called on values. That is, given an application $(\lambda x.\, e_1)\ e_2$, CBV semantics makes sure that e_2 is a value before calling the function.

So, what is a value? In the pure lambda calculus, any abstraction is a value. Remember, an abstraction $\lambda x.\, e$ is a function; in the pure lambda calculus, the only values are functions. In an applied lambda calculus with integers and arithmetic operations, values also include integers. Intuitively, a value is an expression that can not be reduced/executed/simplified any further.

We can give small step operational semantics for call-by-value execution of the lambda calculus. Here, v can be instantiated with any value (e.g., a function).

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e \to e'}{v\ e \to v\ e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}$$

We can see from these rules that, given an application $e_1\ e_2$, we first evaluate e_1 until it is a value, then we evaluate e_2 until it is a value, and then we apply the function to the value—a β-reduction.

Let's consider some examples. (These examples use an applied lambda calculus that also includes reduction rules for arithmetic expressions.)

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 &\to (\lambda x.\, \lambda y.\, y\ x)\ 7\ \lambda x.\, x+1\\ &\to (\lambda y.\, y\ 7)\ \lambda x.\, x+1\\ &\to (\lambda x.\, x+1)\ 7\\ &\to 7+1\\ &\to 8\end{aligned}$$

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) &\to (\lambda f.\, f\ 7)\ ((\lambda y.\, y)\ (\lambda y.\, y))\\ &\to (\lambda f.\, f\ 7)\ (\lambda y.\, y)\\ &\to (\lambda y.\, y)\ 7\\ &\to 7\end{aligned}$$

2.3 Call-by-name

Call-by-name (or CBN) semantics applies the function as soon as possible. The small step operational semantics are a little simpler, as they do not need to ensure that the expression to which a function is applied is a value.

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \to e_1\{e_2/x\}}$$

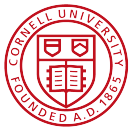
Let's consider the same examples we used for CBV.

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 &\to (\lambda y.\, y\ (5+2))\ \lambda x.\, x+1\\ &\to (\lambda x.\, x+1)\ (5+2)\\ &\to (5+2)+1\\ &\to 7+1\\ &\to 8\end{aligned}$$

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) &\to ((\lambda x.\, x\ x)\ \lambda y.\, y)\ 7\\ &\to ((\lambda y.\, y)\ (\lambda y.\, y))\ 7\\ &\to (\lambda y.\, y)\ 7\\ &\to 7\end{aligned}$$

Note that the answers are the same, but the order of evaluation is different. (Later we will see languages where the order of evaluation is important, and may result in different answers.)

CS 4110 – Programming Languages and Logics
Lecture #14: More λ-calculus



1 Lambda calculus evaluation

There are many different evaluation strategies for the λ-calculus. The most permissive is *full* β *reduction*, which allows any *redex*—i.e., any expression of the form $(\lambda x.\ e_1)\ e_2$—to step to $e_1\{e_2/x\}$ at any time. It is defined formally by the following small-step operational semantics rules:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'} \qquad \frac{e_1 \to e_1'}{\lambda x.\ e_1 \to \lambda x.\ e_1'} \qquad \beta\ \frac{}{(\lambda x.\ e_1)\ e_2 \to e_1\{e_2/x\}}$$

The *call by value* (CBV) strategy enforces a more restrictive strategy: it only allows an application to reduce after its argument has been reduced to a value (i.e., a λ-abstraction) and does not allow evaluation under a λ. It is described by the following small-step operational semantics rules (here we show a left-to-right version of CBV):

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \beta\ \frac{}{(\lambda x.\ e_1)\ v_2 \to e_1\{v_2/x\}}$$

Finally, the *call by name* (CBN) strategy allows an application to reduce even when its argument is not a value but does not allow evaluation under a λ. It is described by the following small-step operational semantics rules:

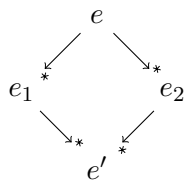
$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \beta\ \frac{}{(\lambda x.\ e_1)\ e_2 \to e_1\{e_2/x\}}$$

2 Confluence

It is not hard to see that the full β reduction strategy is non-deterministic. This raises an interesting question: does the choices made during the evaluation of an expression affect the final result? The answer turns out to be no: full β reduction is *confluent* in the following sense:

Theorem (Confluence). *If $e \to^* e_1$ and $e \to^* e_2$ then there exists e' such that $e_1 \to^* e'$ and $e_2 \to^* e'$.*

Confluence can be depicted graphically as follows:



Confluence is often also called the Church-Rosser property.

3 Substitution

Each of the evaluation relations for λ-calculus has a β defined in terms of a substitution operation on expressions. Because the expressions involved in the substitution may share some variable names (and because we are working up to α-equivalence) the definition of this operation is slightly subtle and defining it precisely turns out to be tricker than might first appear.

As a first attempt, consider an obvious (but incorrect) definition of the substitution operator. Here we are substituting e for x in some other expression:

$$\begin{array}{rcll} y\{e/x\} & = & \left\{ \begin{array}{ll} e & \text{if } y = x \\ y & \text{otherwise} \end{array} \right. & \\ (e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) & \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\} & \text{where } y \neq x \end{array}$$

Unfortunately this definition produces the wrong results when we substitute an expression with free variables under a λ. For example,

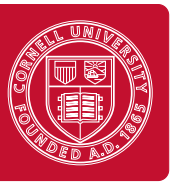
$$(\lambda y.x)\{y/x\} = (\lambda y.y)$$

To fix this problem, we need to revise our definition so that when we substitute under a λ we do not accidentally bind variables in the expression we are substituting. The following definition correctly implements *capture-avoiding substitution*:

$$\begin{array}{rcll} y\{e/x\} & = & \left\{ \begin{array}{ll} e & \text{if } y \neq x \\ y & \text{otherwise} \end{array} \right. & \\ (e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) & \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.(e_1\{e/x\}) & \text{where } y \neq x \text{ and } y \notin \mathit{fv}(e) \end{array}$$

Note that in the case for λ-abstractions, we require that the bound variable y be different from the variable x we are substituting for and that y not appear in the free variables of e, the expression we are substituting. Because we work up to α-equivalence, we can always pick y to satisfy these side conditions. For example, to calculate $(\lambda z.x\ z)\{(w\ y\ z)/x\}$ we first rewrite $\lambda z.x\ z$ to $\lambda u.x\ u$ and then apply the substitution, obtaining $\lambda u.(w\ y\ z)\ u$ as the result.

CS 4110 – Programming Languages and Logics
Lecture #15: De Bruijn, Combinators, Encodings



1 de Bruijn Notation

One way to avoid the tricky interaction between free and bound names in the substitution operator is to pick a representation for expressions that doesn't have any names at all! Intuitively, we can think of a bound variable is just a pointer to the λ that binds it. For example, in $\lambda x.\lambda y.y\ x$, the y points to the first λ and the x points to the second λ.

So-called *de Bruijn* notation uses this idea as the representation for λ expressions. Here is the grammar for λ expressions in de Bruijn notation:

$$e ::= n \mid \lambda.e \mid e\ e$$

Variables are represented by integers n that refer to (the index of) their binder while $lambda$-abstractions have the form $\lambda.e$. Note that the the variable bound by the abstraction is not named—i.e., the representation is nameless.

As examples, here are several terms written using standard notation and in de Bruijn notation:

Standard	de Bruijn
$\lambda x.x$	$\lambda.0$
$\lambda z.z$	$\lambda.0$
$\lambda x.\lambda y.x$	$\lambda.\lambda.1$
$\lambda x.\lambda y.\lambda s.\lambda z.x\ s\ (y\ s\ z)$	$\lambda.\lambda.\lambda.\lambda.3\ 1\ (2\ 1\ 0)$
$(\lambda x.x\ x)\ (\lambda x.x\ x)$	$(\lambda.0\ 0)\ (\lambda.0\ 0)$
$(\lambda x.\lambda x.x)\ (\lambda y.y)$	$(\lambda.\lambda.0)\ (\lambda.0)$

To represent a λ-expression that contains free variables in de Bruijn notation, we need a way to map the free variables to integers. We will work with respect to a map Γ from variables to integers called a *context*. As an example, if Γ maps x to 0 and y to 1, then the de Bruijn representation of $x\ y$ with respect to Γ is $0\ 1$, while the representation of $\lambda z.\ x\ y\ z$ with respect to Γ is $\lambda.\ 1\ 2\ 0$. Note that in this second example, because we have gone under a λ, we have shifted the integers representing x and y up by one to avoid capturing them.

In general, whenever we work de Bruijn representations of expressions containing free variables (i.e., when working with respect to a context Γ) we will need to modify the indices of those variables. For example, when we substitute an expression containing free variables under a λ, we will need to shift the indices up so that they continue to refer to the same numbers with respect to Γ after the substitution as they did before. For example, if we substitute $0\ 1$ for the variable bound by the outermost λ in $\lambda.\lambda.1$ we should get $\lambda.\lambda.2\ 3$, not $\lambda.\lambda.0\ 1$. We will use an auxiliary function

that shifts the indices of free variables above a cutoff c up by i:

$$\begin{array}{rcl} \uparrow_c^i (n) & = & \begin{cases} n & \text{if } n < c \\ n+i & \text{otherwise} \end{cases} \\ \uparrow_c^i (\lambda.e) & = & \lambda.(\uparrow_{c+1}^i e) \\ \uparrow_c^i (e_1\ e_2) & = & (\uparrow_c^i\ e_1)\ (\uparrow_c^i\ e_2) \end{array}$$

The cutoff keeps track of the variables that were bound in the original expression and so should not be shifted as the shifting operator walks down the structure of an expression. The cutoff is 0 initially.

Using this shifting function, we can define substitution as follows:

$$\begin{array}{rcl} n\{e/m\} & = & \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\ (\lambda.e_1)\{e/m\} & = & \lambda.e_1\{(\uparrow_0^1 e)/m+1\})) \\ (e_1\ e_2)\{e/m\} & = & (e_1\{e/m\})\ (e_1\{e/m\}) \end{array}$$

Note that when we go under a λ we increase the index of the variable we are substituting for and shift the free variables in the expression e up by one.

The β rule for terms in de Bruijn notation is as follows:

$$\beta \frac{}{(\lambda.e_1)\ e_2 \rightarrow \uparrow_0^{-1} (e_1\{\uparrow_0^1 e_2/0\})}$$

That is, we substitute occurrences of 0, the index of the variable being bound by the λ, with e_2 shifted up by one. Then we shift the result down by one to ensure that any free variables in e_1 continue to refer to the same things after we remove the λ.

To illustrate how this works consider the following example, which we wrote as $(\lambda u.\lambda v.u\ x)\ y$ in standard notation. We will work with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

$$\begin{array}{rl} & (\lambda.\lambda.1\ 2)\ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda.1\ 2)\{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda.1\ 2)\{2/0\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{(\uparrow_0^1 2)/(0+1)\}) \\ = & \uparrow_0^{-1} \lambda.((1\ 2)\{3/1\}) \\ = & \uparrow_0^{-1} \lambda.(1\{3/1\})\ (2\{3/1\}) \\ = & \uparrow_0^{-1} \lambda.3\ 2 \\ = & \lambda.2\ 1 \end{array}$$

which, in standard notation (with respect to Γ), is the same as $\lambda v.y\ x$.

2 Combinators

Yet another way to avoid the issues having to do with free and bound variable names in the λ-calculus is to work with closed expressions or *combinators*. It turns out that just using two combinators, S, K, and application, we can encode the entire λ-calculus.

Here are the evaluation rules for S, K, as well as a third combinator I, which will also be useful:

$$\begin{aligned}
&\mathsf{K}\ x\ y \to x \\
&\mathsf{S}\ x\ y\ z \to x\ z\ (y\ z) \\
&\mathsf{I}\ x \to x
\end{aligned}$$

Equivalently, here are their definitions as closed λ-expressions:

$$\begin{aligned}
\mathsf{K} &= \lambda x.\lambda y.\ x \\
\mathsf{S} &= \lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z) \\
\mathsf{I} &= \lambda x.\ x
\end{aligned}$$

It is not hard to see that I is not needed—it can be encoded as S K K.

To show how these combinators can be used to encode the λ-calculus, we have to define a translation that takes an arbitrary closed λ-calculus expression and turns it into a combinator term that behaves the same during evaluation. This translation is called *bracket abstraction*. It proceeds in two steps. First, we define a function $[x]$ that takes a combinator term M possibly containing free variables and builds another term that behaves like $\lambda x.M$, in the sense that $([x]\ M)\ N \to M\{N/x\}$ for every term N:

$$\begin{aligned}
[x]\ x &= \mathsf{I} \\
[x]\ N &= \mathsf{K}\ N && \text{where } x \notin fv(N) \\
[x]\ N_1\ N_2 &= \mathsf{S}\ ([x]\ N_1)\ ([x]\ N_2)
\end{aligned}$$

Second, we define a function $(e)*$ that maps a λ-calculus expression to a combinator term:

$$\begin{aligned}
(x)* &= x \\
(e_1\ e_2)* &= (e_1)*\ (e_2)* \\
(\lambda x.e)* &= [x]\ (e)*
\end{aligned}$$

As an example, the expression $\lambda x.\lambda y.\ x$ is translated as follows:

$$\begin{aligned}
& (\lambda x.\lambda y.\ x)* \\
= \ & [x]\ (\lambda y.\ x)* \\
= \ & [x]\ ([y]\ x) \\
= \ & [x]\ (\mathsf{K}\ x) \\
= \ & (\mathsf{S}\ ([x]\ \mathsf{K})\ ([x]\ x)) \\
= \ & \mathsf{S}\ (\mathsf{K}\ \mathsf{K})\ \mathsf{I}
\end{aligned}$$

We can check that this behaves the same as our original λ-expression by seeing how it evaluates when applied to arbitrary expressions e_1 and e_2.

$$\begin{aligned}
& (\lambda x.\lambda y.\ x)\ e_1\ e_2 \\
= \ & (\lambda y.\ e_1)\ e_2 \\
= \ & e_1
\end{aligned}$$

and

$$\begin{aligned}
& (\mathsf{S}\ (\mathsf{K}\ \mathsf{K})\ \mathsf{I})\ e_1\ e_2 \\
= \ & (\mathsf{K}\ \mathsf{K}\ e_1)\ (\mathsf{I}\ e_1)\ e_2 \\
= \ & \mathsf{K}\ e_1\ e_2 \\
= \ & e_1
\end{aligned}$$

3 λ-calculus encodings

The pure λ-calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure λ-calculus. We can however encode objects, such as booleans, and integers.

3.1 Booleans

Let us start by encoding constants and operators for booleans. That is, we want to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as expected. For example:

$$\begin{aligned} \mathsf{AND}\ \mathsf{TRUE}\ \mathsf{FALSE} &= \mathsf{FALSE} \\ \mathsf{NOT}\ \mathsf{FALSE} &= \mathsf{TRUE} \\ \mathsf{IF}\ \mathsf{TRUE}\ e_1\ e_2 &= e_1 \\ \mathsf{IF}\ \mathsf{FALSE}\ e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \mathsf{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \mathsf{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

Thus, both TRUE and FALSE are functions that take two arguments; TRUE returns the first, and FALSE returns the second. We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\, \text{if } b = \mathsf{TRUE} \text{ then } t \text{ else } f.$$

The definitions for TRUE and FALSE make this very easy.

$$\mathsf{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

Definitions of other operators are also straightforward.

$$\begin{aligned} \mathsf{NOT} &\triangleq \lambda b.\, b\ \mathsf{FALSE}\ \mathsf{TRUE} \\ \mathsf{AND} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ b_2\ \mathsf{FALSE} \\ \mathsf{OR} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ \mathsf{TRUE}\ b_2 \end{aligned}$$

3.2 Church numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

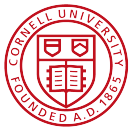
$$\begin{aligned} \overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &= \lambda f.\, \lambda x.\, f\ x \\ \overline{2} &= \lambda f.\, \lambda x.\, f\ (f\ x) \\ \mathsf{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x) \end{aligned}$$

In the definition for SUCC, the expression $n\ f\ x$ applies f to x n times (assuming that variable n is the Church encoding of the natural number n). We then apply f to the result, meaning that we apply f to x $n + 1$ times.

Given the definition of SUCC, we can easily define addition. Intuitively, the natural number $n_1 + n_2$ is the result of apply the successor function n_1 times to n_2.

$$\mathsf{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \mathsf{SUCC}\ n_2$$

CS 4110 – Programming Languages and Logics
Lecture #17: Programming in the λ-calculus



1 Nontermination

Consider the expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, which we will refer to as omega for brevity. Let's try evaluating omega.

$$\begin{array}{rcl} \mathsf{omega} & = & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \to & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & = & \mathsf{omega} \end{array}$$

Evaluating omega never reaches a value! It is an infinite loop!

What happens if we use omega as an actual argument to a function? Consider the following program.

$$(\lambda x.(\lambda y.y))\ \mathsf{omega}$$

If we use CBV semantics to evaluate the program, we must reduce omega to a value before we can apply the function. But omega never evaluates to a value, so we can never apply the function. Thus, under CBV semantics, this program does not terminate. If we use CBN semantics, we can apply the function immediately, without needing to reduce the actual argument to a value:

$$(\lambda x.(\lambda y.y))\ \mathsf{omega} \to_{\mathrm{CBN}} \lambda y.y$$

CBV and CBN are common evaluation orders; many functional programming languages use CBV semantics. Later we will see the call-by-need strategy, which is similar to CBN in that it does not evaluate actual arguments unless necessary but is more efficient.

2 Recursion

We can write nonterminating functions, as we saw with omega. We can also write recursive functions that terminate. However, we need to develop techniques for expressing recursion.

Let's consider how we would like to write the factorial function.

$$\mathsf{FACT} \triangleq \lambda n.\, \mathsf{IF}\ (\mathsf{ISZERO}\ n)\ 1\ (\mathsf{TIMES}\ n\ (\mathsf{FACT}\ (\mathsf{PRED}\ n)))$$

In slightly more readable notation, this is just:

$$\mathsf{FACT} \triangleq \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathsf{FACT}\ (n-1)$$

Here, as in the definition above, the name FACT is simply meant to be shorthand for the expression on the right-hand side of the equation. But FACT appears on the right-hand side of the equation as well! This is not a definition, it's a recursive equation.

2.1 Recursion Removal Trick

We can perform a "trick" to define a function FACT that satisfies the recursive equation above. First, let's define a new function FACT′ that looks like FACT, but takes an additional argument f. We assume that the function f will be instantiated with FACT′ itself.

$$\mathsf{FACT}' \triangleq \lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\ f\ (n-1))$$

Note that when we call f, we pass it a copy of itself, preserving the assumption that the actual argument for f will be FACT′. Now we can define the factorial function FACT in terms of FACT′.

$$\mathsf{FACT} \triangleq \mathsf{FACT}'\ \mathsf{FACT}'$$

Let's try evaluating FACT on an integer.

$$\begin{aligned}
\mathsf{FACT}\ 3 &= (\mathsf{FACT}'\ \mathsf{FACT}')\ 3 && \text{Definition of FACT} \\
&= ((\lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\ f\ (n-1)))\ \mathsf{FACT}')\ 3 && \text{Definition of FACT}' \\
&\to (\lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\mathsf{FACT}'\ \mathsf{FACT}'\ (n-1)))\ 3 && \text{Application to FACT}' \\
&\to \textbf{if } 3 = 0 \textbf{ then } 1 \textbf{ else } 3 \times (\mathsf{FACT}'\ \mathsf{FACT}'\ (3-1)) && \text{Application to } n \\
&\to 3 \times (\mathsf{FACT}'\ \mathsf{FACT}'\ (3-1)) && \text{Evaluating } \textbf{if} \\
&\to \dots \\
&\to 3 \times 2 \times 1 \times 1 \\
&\to^* 6
\end{aligned}$$

So we now have a technique for writing a recursive function f: write a function f' that explicitly takes a copy of itself as an argument, and then define $f \triangleq f'\ f'$.

2.2 Fixed point combinators

There is another way of writing recursive functions: we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point. We saw this technique earlier in the course when we defined the denotational semantics for **while** loops.

Let's consider the factorial function again. The factorial function FACT is a fixed point of the following function.

$$G \triangleq \lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\ (n-1))$$

Recall that if g if a fixed point of G, then we have $G\ g = g$. So if we had some way of finding a fixed point of G, we would have a way of defining the factorial function FACT.

There are a number of "fixed point combinators," and the (infamous) Y combinator is one of them. Thus, we can define the factorial function FACT to be simply Y G, the fixed point of G. The Y combinator is defined as

$$\mathsf{Y} \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators. Note how similar its defnition is to omega.

We'll use a slight variant of the Y combinator, Z, which is easier to use under CBV. (What happens when we evaluate Y G under CBV?). The Z combinator is defined as

$$\mathsf{Z} \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y))$$

Let's see it in action, on our function G. Define FACT to be $\mathsf{Z}\ G$ and calculate as follows:

$$\begin{array}{lll}
 & \mathsf{FACT} & \\
= & \mathsf{Z}\ G & \\
= & (\lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y)))\ G & \text{Definition of } \mathsf{Z} \\
\to & (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y)) & \\
\to & G\ (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) & \\
= & (\lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\ (n-1))) & \\
 & \quad (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) & \text{definition of } G \\
\to & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 & \\
 & \quad \textbf{else } n \times ((\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y)\ (n-1)) & \\
=_\beta & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\lambda y.\, (\mathsf{Z}\ G)\ y)\ (n-1) & \\
=_\beta & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\mathsf{Z}\ G\ (n-1)) & \\
= & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\mathsf{FACT}\ (n-1)) &
\end{array}$$

There are many (indeed infinitely many) fixed-point combinators. Here's a cute one:

$$\mathsf{Y}_k \triangleq (\mathsf{L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L})$$

where

$$\mathsf{L} \triangleq \lambda abcdefghijklmnopqstuvwxyzr.\, (r\ (t\ h\ i\ s\ i\ s\ a\ f\ i\ x\ e\ d\ p\ o\ i\ n\ t\ c\ o\ m\ b\ i\ n\ a\ t\ o\ r))$$

To gain some more intuition for fixed-point combinators, let's derive a fixed-point combinator that was originally discovered by Alan Turing. Suppose we have a higher order function f, and want the fixed point of f. We know that $\Theta\ f$ is a fixed point of f, so we have

$$\Theta\ f = f\ (\Theta\ f).$$

This means, that we can write the following recursive equation:

$$\Theta = \lambda f.\, f\ (\Theta\ f).$$

Now we can use the recursion removal trick we described earlier. Define Θ' as $\lambda t.\, \lambda f.\, f\ (t\ t\ f)$, and Θ as $\Theta'\ \Theta'$. Then we have the following equalities:

$$\begin{aligned}
\Theta &= \Theta'\ \Theta' \\
&= (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ \Theta' \\
&\to \lambda f.\, f\ (\Theta'\ \Theta'\ f) \\
&= \lambda f.\, f\ (\Theta\ f)
\end{aligned}$$

Let's try out the Turing combinator on our higher order function G that we used to define FACT.

This time we will use CBN evaluation.

$$
\begin{aligned}
\mathsf{FACT} &= \Theta\ G \\
&= ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f)))\ G \\
&\to (\lambda f.\, f\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ f))\ G \\
&\to G\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ G) \\
&= G\ (\Theta\ G) && \text{for brevity} \\
&= (\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\ (n-1)))\ (\Theta\ G) && \text{Definition of } G \\
&\to \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\Theta\ G)\ (n-1)) \\
&= \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathsf{FACT}\ (n-1))
\end{aligned}
$$

3 Definitional translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus. We now consider definitional translation, where we define the meaning of language constructs by translation to another language. This is a form of denotational semantics, but instead of the target being mathematical objects, it is a simpler programming language (such as λ-calculus). Note that definitional translation does not necessarily produce clean or efficient code; rather, it defines the meaning of the source language constructs in terms of the target language.

For each language construct, we will define an operational semantics directly, and then give an alternate semantics by translation to a simpler language. We will start by introducing *evaluation contexts*, which make it easier to present the new language features succinctly.

3.1 Evaluation contexts

Recall the syntax and CBV operational semantics for the lambda calculus:

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \\
v &::= \lambda x.\, e
\end{aligned}
$$

$$
\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}
$$

Of the operational semantics rules, only the β-reduction rule told us how to "reduce" an expression; the other two rules tell us the order to evaluate expressions—e.g., evaluate the left hand side of an application to a value first, then evaluate the right hand side of an application to a value. The operational semantics of many of the languages we will consider have this feature: there are two kinds of rules, *congruence rules* that specify evaluation order, and the *computation rules* that specify the "interesting" reductions.

Evaluation contexts are a simple mechanism that separates out these two kinds of rules. An evaluation context E (sometimes written $E[\cdot]$) is an expression with a "hole" in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the "hole") in place of a subexpression. Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the

language. The following grammar defines evaluation contexts for the pure CBV λ-calculus.

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

We write $E[e]$ to mean the evaluation context E where the hole has been replaced with the expression e. The following are examples of evaluation contexts, and evaluation contexts with the hole filled in by an expression.

$$\begin{array}{ll} E_1 = [\cdot]\ (\lambda x.\, x) & E_1[\lambda y.\, y\ y] = (\lambda y.\, y\ y)\ \lambda x.\, x \\ E_2 = (\lambda z.\, z\ z)\ [\cdot] & E_2[\lambda x.\, \lambda y.\, x] = (\lambda z.\, z\ z)\ (\lambda x.\, \lambda y.\, x) \\ E_3 = ([\cdot]\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) & E_3[\lambda f.\, \lambda g.\, f\ g] = ((\lambda f.\, \lambda g.\, f\ g)\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) \end{array}$$

Using evaluation contexts, we can define the evaluation semantics for the pure CBV λ-calculus with just two rules, one for evaluation contexts, and one for β-reduction.

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}$$

Note that the evaluation contexts for the CBV λ-calculus ensure that we evaluate the left hand side of an application to a value, and then evaluate the right hand side of an application to a value before applying β-reduction.

We can specify the operational semantics of CBN λ-calculus using evaluation contexts:

$$E ::= [\cdot] \mid E\ e \qquad \frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \to e_1\{e_2/x\}}$$

We'll see the benefit of evaluation contexts as we see languages with more syntactic constructs.

3.2 Multi-argument functions and currying

Our syntax for functions only allows function with a single argument: $\lambda x.\, e$. We could define a language that allows functions to have multiple arguments.

$$e ::= x \mid \lambda x_1, \ldots, x_n.\, e \mid e_0\ e_1\ \ldots\ e_n$$

Here, a function $\lambda x_1, \ldots, x_n.\, e$ takes n arguments, with names x_1 through x_n. In a multi argument application $e_0\ e_1\ \ldots\ e_n$, we expect e_0 to evaluate to an n-argument function, and $e_1, \ldots, e_n$ are the arguments that we will give the function.

We can define a CBV operational semantics for the multi-argument λ-calculus as follows.

$$E ::= [\cdot] \mid v_0\ \ldots\ v_{i-1}\ E\ e_{i+1}\ \ldots\ e_n \qquad \frac{e \to e'}{E[e] \to E[e']}$$

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda x_1, \ldots, x_n.\, e_0)\ v_1\ \ldots\ v_n \to e_0\{v_1/x_1\}\{v_2/x_2\}\ldots\{v_n/x_n\}}$$

The evaluation contexts ensure that we evaluate a multi-argument application $e_0\ e_1\ \ldots\ e_n$ by evaluating each expression from left to right down to a value.

Now, the multi-argument λ-calculus isn't any more expressive that the pure λ-calculus. We can show this by showing how any multi-argument λ-calculus program can be translated into an equivalent pure λ-calculus program. We define a translation function $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure λ-calculus. That is, if e is a multi-argument lambda calculus expression, $\mathcal{T}[\![e]\!]$ is a pure λ-calculus expression.

We define the translation as follows.

$$\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x_1, \ldots, x_n.\, e]\!] &= \lambda x_1.\ \ldots\ \lambda x_n.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_0\ e_1\ e_2\ \ldots\ e_n]\!] &= (\ldots((\mathcal{T}[\![e_0]\!]\ \mathcal{T}[\![e_1]\!])\ \mathcal{T}[\![e_2]\!])\ldots\ \mathcal{T}[\![e_n]\!])
\end{aligned}$$

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*. Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C. We could describe this function, using mathematical notation for domains of functions, as being an element of $A \times B \to C$. Currying this function produces a function that is an element of $A \to (B \to C)$. That is, the curried version of the function takes an argument from domain A, and returns a function that takes an argument from domain B and produces a result of domain C.

CS 4110 – Programming Languages and Logics
Lecture #18: More Definitional Translation and Continuations

In the last lecture we introduced a general framework for defining language features by translation. This lecture presents several additional example translations (for products, let-expressions, and laziness, and mutable references); discusses correctness; and introduces continuations.

0.1 Products and let

A product is a pair of expressions (e_1, e_2). If e_1 and e_2 are both values, then we regard the product as also being a value. (That is, we cannot further evaluate a product if both elements are values.) Given a product, we can access the first or second element using the operators $\#1$ and $\#2$ respectively. That is, $\#1\ (v_1, v_2) \to v_1$ and $\#2\ (v_1, v_2) \to v_2$. (Other common notation for projection includes π_1 and π_2, and fst and snd.)

The syntax of λ-calculus extended with products and let expressions is defined as follows.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \\ &\quad \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \\ &\quad \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ v &::= \lambda x.\, e \mid (v_1, v_2) \end{aligned}$$

Note that values in this language are either functions or pairs of values.

We define a small-step CBV operational semantics for the language using evaluation contexts.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e_2$$

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}$$

$$\frac{}{\#1\ (v_1, v_2) \to v_1} \qquad \frac{}{\#2\ (v_1, v_2) \to v_2}$$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \to e\{v/x\}}$$

Next, we define an equivalent semantics by translation to the pure CBV λ-calculus.

$$
\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![(e_1, e_2)]\!] &= (\lambda x.\, \lambda y.\, \lambda f.\, f\ x\ y)\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\#1\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, x) \\
\mathcal{T}[\![\#2\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, y) \\
\mathcal{T}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= (\lambda x.\, \mathcal{T}[\![e_2]\!])\ \mathcal{T}[\![e_1]\!]
\end{aligned}
$$

Note that we encode a pair (e_1, e_2) as a value that takes a function f, and applies f to v_1 and v_2, where v_1 and v_2 are the result of evaluating e_1 and e_2 respectively. The projection operators pass a function to the encoding of pairs that selects either the first or second element as appropriate. Also note that the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is equivalent to the application $(\lambda x.\, e_2)\ e_1$.

1 Laziness

In previous lectures we defined semantics for both the call-by-name λ-calculus and the call-by-value λ-calculus. It turns out that we can translate a call-by-name program into a call-by-value program. In CBV, arguments to functions are evaluated before the function is applied; in CBN, functions are applied as soon as possible. In the translation, we delay the evaluation of arguments by wrapping them in a function. This is called a *thunk*: wrapping a computation in a function to delay its evaluation.

Since arguments to functions are turned into thunks, when we want to use an argument in a function body, we need to evaluate the thunk. We do so by applying the thunk (which is simply a function); it doesn't matter what we apply the thunk to, since the thunk's argument is never used.

$$
\begin{aligned}
\mathcal{T}[\![x]\!] &= x\ (\lambda y.\, y) \\
\mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ (\lambda z.\, \mathcal{T}[\![e_2]\!]) \qquad\qquad z \text{ is not a free variable of } e_2
\end{aligned}
$$

2 References

We can also introduce constructs for creating, reading, and updating memory locations, also called *references*. The resulting language is still a functional language (since functions are first-class values), but expressions can have side-effects, that is, they can modify state. The syntax of this language is defined as follows.

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \ell \\
v &::= \lambda x.\, e \mid \ell
\end{aligned}
$$

Expression $\mathsf{ref}\ e$ creates a new memory location (like a malloc), and sets the initial contents of the location to (the result of) e. The expression $\mathsf{ref}\ e$ itself evaluates to a memory location ℓ.

Think of a location as being like a pointer to a memory address. The expression $!e$ assumes that e evaluates to a memory location, and $!e$ evaluates to the current contents of the memory location. Expression $e_1 := e_2$ assumes that e_1 evaluates to a memory location ℓ, and updates the contents of ℓ with (the result of) e_2. Locations ℓ are not intended to be used directly by a programmer: they are not part of the *surface syntax* of the language, the syntax that a programmer would write. They are introduced only by the operational semantics.

We define a small-step CBV operational semantics. We use configurations $\langle \sigma, e \rangle$, where e is an expression, and σ is a map from locations to values.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \mathsf{ref}\ E \mid !E \mid E := e \mid v := E \qquad \frac{\langle \sigma, e \rangle \to \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \to \langle \sigma', E[e'] \rangle}$$

$$\beta\text{-REDUCTION}\ \frac{}{\langle \sigma, (\lambda x.\, e)\ v \rangle \to \langle \sigma, e\{v/x\} \rangle} \qquad \text{ALLOC}\ \frac{}{\langle \sigma, \mathsf{ref}\ v \rangle \to \langle \sigma[\ell \mapsto v], \ell \rangle}\ \ell \notin dom(\sigma)$$

$$\text{DEREF}\ \frac{}{\langle \sigma,\ !\ell \rangle \to \langle \sigma, v \rangle}\ \sigma(\ell) = v \qquad \text{ASSIGN}\ \frac{}{\langle \sigma, \ell := v \rangle \to \langle \sigma[\ell \mapsto v], v \rangle}$$

References do not add any expressive power to the λ-calculus: it is possible to translate λ-calculus with references to the pure λ-calculus. Intuitively, this is achieved by explicitly representing the store, and threading the store through the evaluation of the program. The details are left as an exercise.

3 Adequacy of translation

In each of the previous translations, we defined a semantics for the source language (using evaluation contexts and small-step rules) and the target language (by translation). We would like to be able to show that the translation is correct—that is, that it preserves the meaning of source programs.

More precisely, we would like an expression e in the source language to evaluate to a value v if and only if the translation of e evaluates to a value v' such that v' is "equivalent to" v. What exactly it means for v' to be "equivalent to" v will depend on the translation. Sometimes, it will mean that v' is literally the translation of v; other times, it will mean that v' is merely related to the translation of v by some equivalence.

One tricky issue is that in general, there can be many ways to define equivalences on functions. One way is to say that two functions are equivalent if they agree on the result when applied to any value of a base type (e.g., integers or booleans). The idea is that if two functions disagree when passed a more complex value (say, a function), then we could write a program that uses these functions to produce functions that disagree on values of base types.

There are two criteria for a translation to be *adequate*: soundness and completeness. For clarity, let's suppose that $\mathbf{Exp}_{\text{src}}$ is the set of source language expressions, and that $\to_{\text{src}}$ and $\to_{\text{trg}}$ are the evaluation relations for the source and target languages respectively. A translation is sound if every target evaluation represents a source evaluation:

$$\text{Soundness: } \forall e \in \mathbf{Exp}_{\text{src}}.\ \text{if } \mathcal{T}[\![e]\!] \to^*_{\text{trg}} v' \text{ then } \exists v.\ e \to^*_{\text{src}} v \text{ and } v' \text{ equivalent to } v$$

A translation is complete if every source evaluation has a target evaluation.

$$\text{Completeness: } \forall e \in \mathbf{Exp}_{\text{src}}.\ \text{if } e \to^*_{\text{src}} v \text{ then } \exists v'.\ \mathcal{T}[\![e]\!] \to^*_{\text{trg}} v' \text{ and } v' \text{ equivalent to } v$$

4 Continuations

In each of the preceding translations, the control structure of the source language was translated directly into the corresponding control structure in the target language. For example:

$$\mathcal{T}[\![\lambda x.\, e]\!] = \lambda x.\, \mathcal{T}[\![e]\!]$$
$$\mathcal{T}[\![e_1\ e_2]\!] = \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!]$$

This style of translation works well when the source language is similar to the target language. However, when the control structures of the source and target languages differ considerable, it doesn't work as well.

Continuations are a programming technique that may be used directly by a programmer, or used in program transformations by a compiler. Because they make the control flow of the program explicit, they can be used to overcome discrepancies between source and target languages in definitional translation. They can also be used to define the semantics of control-flow constructs such as exceptions.

Intuitively, a continuation represents "the rest of the program." Consider the program

$$\mathsf{if}\ \mathsf{foo} < 10\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar}$$

and consider the evaluation of the expression $\mathsf{foo} < 10$. When we finish evaluating this subexpression, we will evaluate the if statement, and then evaluate the appropriate branch. The *continuation* of the subexpression $\mathsf{foo} < 10$ is the rest of the computation that will occur after we evaluate the subexpression. We can write this continuation as a function that takes the result of the subexpression:

$$(\lambda y.\ \mathsf{if}\ y\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar})\ (\mathsf{foo} < 10)$$

The evaluation order and result of this program will be the same as the original expression; the difference is that we extracted the continuation of the subexpression in to a function.

The nice thing about continuations is that it makes the control explicit, and this is especially useful in the case of functional programs, where control is not explicit otherwise. In fact, we can rewrite a program to make continuations more explicit. Let's consider another program, and convert it so that continuations are explicit

$$(\lambda x.\, x)\ ((1 + 2) + 3) + 4$$

We'll start by defining a continuation for the outermost evaluation context, which takes a value, and applies the identity function to it.

$$k_0 = \lambda v.\, (\lambda x.\, x)\ v$$

The evaluation context that is evaluated next-to-last takes a value, adds 4 to it, and then passes the result to k_0.

$$k_1 = \lambda a.\, k_0\ (a + 4)$$

Likewise, for the next evaluation contexts.

$$k_2 = \lambda b.\, k_1\ (b + 3)$$
$$k_3 = \lambda c.\, k_2\ (c + 2)$$

The program itself is now equivalent to $k_3\ 1$. Since $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is just syntactic sugar for $(\lambda x.\, e')\ e$, we can actually rewrite the above as

$$\begin{array}{l}\mathsf{let}\ c = 1\ \mathsf{in}\\ \mathsf{let}\ b = c + 2\ \mathsf{in}\\ \mathsf{let}\ a = b + 3\ \mathsf{in}\\ \mathsf{let}\ v = a + 4\ \mathsf{in}\\ (\lambda x.\, x)\ v\end{array}$$

This is fairly close to some machine instructions of the form:

$$\begin{array}{l}\mathsf{set}\ c, 1\\ \mathsf{add}\ b, c, 2\\ \mathsf{add}\ a, b, 3\\ \mathsf{add}\ v, a, 4\\ \mathsf{call\ id}, v\end{array}$$

Using continuations, functions can be transformed into "functions that don't return"—i.e., functions that take, besides the usual arguments, an additional argument representing a continuation. When the function finishes, it invokes the continuation on its result, instead of returning the result to its caller. Writing functions in this way is usually referred to as Continuation-Passing Style, or CPS for short. For instance, the CPS version of factorial looks like the following:

$$\mathsf{FACT}_{cps} = \mathsf{Y}\ \lambda f.\, \lambda n, k.\, \mathsf{if}\ n = 0\ \mathsf{then}\ k\ 1\ \mathsf{else}\ f\ (n - 1)\ (\lambda v.\, k\ (n * v))$$

Note that the last thing that code in FACT_{cps} does is call a function (either k or f), and does not do anything with the result.

Continuation-passing style is an important concept in the compilation of functional languages and is used as an intermediate compiler representation (it has been used in compilers for Scheme, ML, etc). The main advantage is that CPS makes the control flow explicit and makes it easier to translate functional code to machine code where control is explicit (in the form of sequences of machine instructions and jumps). For instance, a CPS call can be easily translated into a jump to the invoked method, since the invoked function does not return the control.

4.1 CPS translation

We can translate λ-calculus programs into continuation-passing style. We define a translation function $\mathcal{CPS}[\![\cdot]\!]$, which takes a CBV λ-calculus expression, and translates the expression to a CBV λ-calculus expression in continuation-passing style.

Let's consider a translation from λ-calculus with pairs and integers. The syntax of the source language is as follows.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

The translation $\mathcal{CPS}[\![e]\!]$ will produce a function that whose argument is the continuation to which to pass the result. That is, for all expressions e, the translation is of the form $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$, where k is a continuation. We will both assume and guarantee that for any expression e, the translation $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$ will apply k to the result of evaluating e.

For convenience, instead of writing $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$ we write $\mathcal{CPS}[\![e]\!]\ k = \ldots$.

$$
\begin{aligned}
\mathcal{CPS}[\![n]\!]\ k &= k\ n \\
\mathcal{CPS}[\![e_1 + e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda n. \mathcal{CPS}[\![e_2]\!]\ (\lambda m. k\ (n+m))) && n \text{ is not a free variable of } e_2
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{CPS}[\![(e_1, e_2)]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda v. \mathcal{CPS}[\![e_2]\!]\ (\lambda w. k\ (v, w))) && v \text{ is not a free variable of } e_2 \\
\mathcal{CPS}[\![\#1\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v. k\ (\#1\ v)) \\
\mathcal{CPS}[\![\#2\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v. k\ (\#2\ v))
\end{aligned}
$$

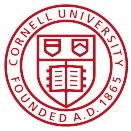
$$
\begin{aligned}
\mathcal{CPS}[\![x]\!]\ k &= k\ x \\
\mathcal{CPS}[\![\lambda x. e]\!]\ k &= k\ (\lambda x. \lambda k'. \mathcal{CPS}[\![e]\!]\ k') && k' \text{ is not a free variable of } e \\
\mathcal{CPS}[\![e_1\ e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda f. \mathcal{CPS}[\![e_2]\!]\ (\lambda v. f\ v\ k)) && f \text{ is not a free variable of } e_2
\end{aligned}
$$

We translate a function $\lambda x. e$ to a function that takes an additional argument k', which is the continuation after the function application. That is, k' is the continuation to which we hand the result of evaluating the function body. In function application, we see that in addition to the actual argument, we also give the continuation as the additional argument.

Let's see an example translation and execution...

$$
\begin{aligned}
\mathcal{CPS}[\![(\lambda a. a + 6)\ 7]\!]\ \mathsf{ID} &= \mathcal{CPS}[\![(\lambda a. a + 6)]\!]\ \ (\lambda f. \mathcal{CPS}[\![7]\!]\ (\lambda v. f\ v\ \mathsf{ID})) \\
&= (\lambda f. \mathcal{CPS}[\![7]\!]\ (\lambda v. f\ v\ \mathsf{ID}))\ (\lambda a, k'. \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a]\!]\ (\lambda n. \mathcal{CPS}[\![6]\!]\ (\lambda m. k'\ (m+n)))) \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a]\!]\ (\lambda n. (\lambda m. k'\ (m+n))\ 6)) \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a) \\
&\to (\lambda v. (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a)\ v\ \mathsf{ID})\ 7 \\
&\to (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a)\ 7\ \mathsf{ID} \\
&\to (\lambda n. (\lambda m. \mathsf{ID}\ (m+n))\ 6)\ 7 \\
&\to (\lambda m. \mathsf{ID}\ (m+7))\ 6 \\
&\to \mathsf{ID}\ (6+7) \\
&\to \mathsf{ID}\ 13 \\
&\to 13
\end{aligned}
$$

CS 4110 – Programming Languages and Logics
Lecture #19: Simply Typed λ-calculus



A *type* is a collection of computational entities that share some common property. For example, the type **int** represents all expressions that evaluate to an integer, and the type **int** $\to$ **int** represents all functions from integers to integers. The Pascal subrange type `[1..100]` represents all integers between 1 and 100.

Types can be thought of as describing computations succinctly and approximately: types are a *static* approximation to the run-time behaviors of terms and programs. Type systems are a lightweight formal method for reasoning about behavior of a program. Uses of type systems include: naming and organizing useful concepts; providing information (to the compiler or programmer) about data manipulated by a program; and ensuring that the run-time behavior of programs meet certain criteria.

In this lecture, we'll consider a type system for the lambda calculus that ensures that values are used correctly; for example, that a program never tries to add an integer to a function. The resulting language (lambda calculus plus the type system) is called the *simply-typed lambda calculus*.

1 Simply-typed lambda calculus

The syntax of the simply-typed lambda calculus is similar to that of untyped lambda calculus, with the exception of abstractions. Since abstractions define functions tht take an argument, in the simply-typed lambda calculus, we explicitly state what the type of the argument is. That is, in an abstraction $\lambda x\!:\!\tau.\, e$, the τ is the expected type of the argument.

The syntax of the simply-typed lambda calculus is as follows. It includes integer literals n, addition $e_1 + e_2$, and the *unit value* $()$. The unit value is the only value of type **unit**.

expressions	$e ::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid ()$
values	$v ::= \lambda x\!:\!\tau.\, e \mid n \mid ()$
types	$\tau ::= \textbf{int} \mid \textbf{unit} \mid \tau_1 \to \tau_2$

The operational semantics of the simply-typed lambda calculus are the same as the untyped lambda calculus. For completeness, we present the CBV small step operational semantics here.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E$$

$$\textsc{Context}\ \frac{e \to e'}{E[e] \to E[e']}$$

$$\beta\textsc{-reduction}\ \frac{}{(\lambda x\!:\!\tau.\, e)\ v \to e\{v/x\}}$$

$$\textsc{Add}\ \frac{}{n_1 + n_2 \to n}\ n = n_1 + n_2$$

1.1 The typing relation

The presence of types does not alter the evaluation of an expression at all. So what use are types?

We will use types to restrict what expressions we will evaluate. Specifically, the type system for the simply-typed lambda calculus will ensure that any *well-typed* program will not get *stuck*. A term e is stuck if e is not a value and there is no term e' such that $e \to e'$. For example, the expression $42 + \lambda x.\, x$ is stuck: it attempts to add an integer and a function; it is not a value, and there is no operational rule that allows us to reduce this expression. Another stuck expression is $()\ 47$, which attempts to apply the unit value to an integer.

We introduce a relation (or *judgment*) over *typing contexts* (or *type environments*) Γ, expressions e, and types τ. The judgment

$$\Gamma \vdash e : \tau$$

is read as "e has type τ in context Γ".

A typing context is a sequence of variables and their types. In the typing judgment $\Gamma \vdash e : \tau$, we will ensure that if x is a free variable of e, then Γ associates x with a type. We can view a typing context as a partial function from variables to types. We will write $\Gamma, x : \tau$ or $\Gamma[x \mapsto \tau]$ to indicate the typing context that extends Γ by associating variable x with with type τ. The empty context is sometimes written $\emptyset$, or often just not written at all. For example, we write $\vdash e : \tau$ to mean that the closed term e has type τ under the empty context.

Given a typing environment Γ and expression e, if there is some τ such that $\Gamma \vdash e : \tau$, we say that e *is well-typed under context* Γ; if Γ is the empty context, we say e *is well-typed*.

We define the judgment $\Gamma \vdash e : \tau$ inductively.

$$\text{T-INT}\ \frac{}{\Gamma \vdash n : \textbf{int}} \qquad \text{T-ADD}\ \frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}} \qquad \text{T-UNIT}\ \frac{}{\Gamma \vdash () : \textbf{unit}}$$

$$\text{T-VAR}\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \text{T-ABS}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \text{T-APP}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

An integer n always has type **int**. Expression $e_1 + e_2$ has type **int** if both e_1 and e_2 have type **int**. The unit value $()$ always has type **unit**.

Variable x has whatever type the context associates with x. Note that Γ must contain an association for x in order for the judgment $\Gamma \vdash x : \tau$ to hold, that is, $x \in \mathit{dom}(\Gamma)$. The abstraction $\lambda x : \tau.\, e$ has the function type $\tau \to \tau'$ if the function body e has type τ' under the assumption that x has type τ. Finally, an application $e_1\ e_2$ has type τ' provided that e_1 is a function of type $\tau \to \tau'$, and e_2 is an argument of the expected type, i.e., of type τ.

To type check an expression e, we attempt to construct a derivation of the judgment $\vdash e : \tau$, for some type τ. For example, consider the program $(\lambda x : \textbf{int}.\, x + 40)\ 2$. The following is a proof that $(\lambda x : \textbf{int}.\, x + 40)\ 2$ is well-typed.

$$\text{T-APP}\ \frac{\text{T-ABS}\ \dfrac{\text{T-ADD}\ \dfrac{\text{T-VAR}\ \dfrac{}{x : \textbf{int} \vdash x : \textbf{int}} \qquad \text{T-INT}\ \dfrac{}{x : \textbf{int} \vdash 40 : \textbf{int}}}{x : \textbf{int} \vdash x + 40 : \textbf{int}}}{\vdash \lambda x : \textbf{int}.\, x + 40 : \textbf{int} \to \textbf{int}} \qquad \text{T-INT}\ \dfrac{}{\vdash 2 : \textbf{int}}}{\vdash (\lambda x : \textbf{int}.\, x + 40)\ 2 : \textbf{int}}$$

1.2 Type soundness

We mentioned above that the type system ensures that any well-typed program does not get stuck. We can state this property formally.

Theorem (Type soundness)**.** *If* $\vdash e : \tau$ *and* $e \to^* e'$ *and* $e' \not\to$ *then* e' *is a value and* $\vdash e' : \tau$*.*

We will prove this theorem using two lemmas: *preservation* and *progress*. Intuitively, preservation says that if an expression e is well-typed, and e can take a step to e', then e' is well-typed. That is, evaluation preserves well-typedness. Progress says that if an expression e is well-typed, then either e is a value, or there is an e' such that e can take a step to e'. That is, well-typedness means that the expression cannot get stuck. Together, these two lemmas suffice to prove type soundness.

1.2.1 Preservation

Lemma (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \to e'$ *then* $\vdash e' : \tau$*.*

Proof. Assume $\vdash e : \tau$ and $e \to e'$. We need to show $\vdash e' : \tau$. We will do this by induction on the derivation of $e \to e'$.

Consider the last rule used in the derivation of $e \to e'$.

- ADD

  Here $e \equiv n_1 + n_2$, and $e' = n$ where $n = n_1 + n_2$, and $\tau = \textbf{int}$. By the typing rule T-INT, we have $\vdash e' : \textbf{int}$ as required.

- β-REDUCTION

  Here, $e \equiv (\lambda x : \tau'.\, e_1)\, v$ and $e' \equiv e_1\{v/x\}$. Since e is well-typed, we have derivations showing $\vdash \lambda x : \tau'.\, e_1 : \tau' \to \tau$ and $\vdash v : \tau'$. There is only one typing rule for abstractions, T-ABS, from which we know $x : \tau' \vdash e_1 : \tau$. By the substitution lemma (see below), we have $\vdash e_1\{v/x\} : \tau$ as required.

- CONTEXT

  Here, we have some context E such that $e = E[e_1]$ and $e' = E[e_2]$ for some e_1 and e_2 such that $e_1 \to e_2$. Since e is well-typed, we can show by induction on the structure of E that $\vdash e_1 : \tau_1$ for some τ_1. By the inductive hypothesis, we thus have $\vdash e_2 : \tau_1$. By the context lemma (see below) we have $\vdash E[e_2] : \tau$ as required.

□

Additional lemmas we used in the proof above.

Lemma (Substitution)**.** *If* $x : \tau' \vdash e : \tau$ *and* $\vdash v : \tau'$ *then* $\vdash e\{v/x\} : \tau$*.*

Lemma (Context)**.** *If* $\vdash E[e] : \tau$ *and* $\vdash e : \tau'$ *and* $\vdash e' : \tau'$ *then* $\vdash E[e'] : \tau$*.*

1.2.2 Progress

Lemma (Progress)**.** *If $\vdash e : \tau$ then either e is a value or there exists an e' such that $e \to e'$.*

Proof. We proceed by induction on the derivation of $\vdash e : \tau$.

- T-VAR

  This case is impossible, since a variable is not well-typed in the empty environment.

- T-UNIT, T-INT, T-ABS

  Trivial, since e must be a value.

- T-ADD

  Here $e \equiv e_1 + e_2$ and $\vdash e_i : \textsf{int}$ for $i \in \{1, 2\}$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \to e_i'$.

  If e_1 is not a value, then by CONTEXT, $e_1 + e_2 \to e_1' + e_2$. If e_1 is a value and e_2 is not a value, then by CONTEXT, $e_1 + e_2 \to e_1 + e_2'$. If e_1 and e_2 are values, then, it must be the case that they are both integer literals, and so, by ADD, we have $e_1 + e_2 \to n$ where n equals e_1 plus e_2.

- T-APP

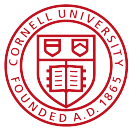
  Here $e \equiv e_1\ e_2$ and $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \to e_i'$.

  If e_1 is not a value, then by CONTEXT, $e_1\ e_2 \to e_1'\ e_2$. If e_1 is a value and e_2 is not a value, then by CONTEXT, $e_1\ e_2 \to e_1\ e_2'$. If e_1 and e_2 are values, then, it must be the case that e_1 is an abstraction $\lambda x : \tau'.\, e'$, and so, by β-REDUCTION, we have $e_1\ e_2 \to e'\{e_2/x\}$.

□

Clearly, not all expressions in the untyped lambda calculus are well-typed. Indeed, type soundness implies that any lambda calculus program that gets stuck is not well-typed. But are there programs that do not get stuck that are not well-typed? Unfortunately, the answer is yes. In particular, because the simply-typed lambda calculus requires us to specify a type for function arguments, any given function can only take arguments of one type. Consider, for example, the identity function $\lambda x.\, x$. This function may be applied to any argument, and it will not get stuck. However, we must provide a type for the argument. If we specify $\lambda x : \textsf{int}.\, x$, then this function can only accept integers, and the program $(\lambda x : \textsf{int}.\, x)\ ()$ is not well-typed, even though it does not get stuck. Indeed, in the simply-typed lambda calculus, there is a different identity function for each type.

CS 4110 – Programming Languages and Logics
Lectures #21: Normalization



1 Introduction

A limitation of the simply-typed lambda-calculus is that we can no longer write recursive functions. Consider the nonterminating expression $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. What type does it have? Let's suppose that the type of $\lambda x.\, x\ x$ is $\tau \to \tau'$. But $\lambda x.\, x\ x$ is applied to itself! So that means that the type of $\lambda x.\, x\ x$ is the argument type τ. So we have that τ must be equal to $\tau \to \tau'$. There is no such type for which this equality holds. (At least, not in this type system...)

This means that every well-typed program in the simply-typed lambda calculus terminates. Formally:

Theorem (Normalization)**.** *If* $\vdash e : \tau$ *then there exists a value* v *such that* $e \to^* v$.

The rest of this lecture is devoted to proving this theorem.

2 Notation

For simplicity, we'll work with the simply-typed lambda calculus over unit,

$$\begin{array}{rcl} e & ::= & x \mid () \mid \lambda x : \tau.\, e \mid e_1\ e_2 \\ v & ::= & () \mid \lambda x : \tau.\, e \\ \tau & ::= & \textbf{unit} \mid \tau_1 \to \tau_2 \end{array}$$

with the standard call-by value semantics:

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

$$\textsc{Context}\ \frac{e \to e'}{E[e] \to E[e']} \qquad \beta\textsc{-reduction}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}$$

3 A First Attempt

As a first attempt toward proving normalization, let us try a proof by structural induction on e. We will need the following lemmas, all of which are standard. Each of these lemmas can be proved by straightforward induction on the typing derivation. We leave the proofs as exercises.

Lemma (Inversion)**.**

- *If* $\Gamma \vdash x : \tau$ *then* $\Gamma(x) = \tau$
- *If* $\Gamma \vdash \lambda x : \tau_1.\, e : \tau$ *then* $\tau = \tau_1 \to \tau_2$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$.

- *If* $\Gamma \vdash e_1\ e_2 : \tau$ *then* $\Gamma \vdash e_1 : \tau' \to \tau$ *and* $\Gamma \vdash e_2 ty \tau'$.

Lemma (Canonical Forms)**.**

- *If* $\Gamma \vdash v :$ ***unit*** *then* $v = ()$
- *If* $\Gamma \vdash v : \tau_1 \to \tau_2$ *then* $v = \lambda x : \tau_1 . e$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$.

Now let us attempt to prove prove the main theorem.

Theorem (Normalization)**.** *If* $\vdash e : \tau$ *then there exists a value* v *such that* $e \to^* v$.

Proof. By structural induction on e.

Case $e = x$**:**
By inversion, we have that the empty typing context maps x to τ, which is a contradiction. Hence, the case vacuously holds.

Case $e = ()$**:**
Immediate since e is already a value.

Case $e = \lambda x : \tau . e$**:**
Immediate since e is already a value.

Case $e = e_1\ e_2$**:**
By inversion we have $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. Hence, by induction hypothesis there exist v_1 and v_2 such that that $e_1 \to^* v_1$ and $e_2 \to^* v_2$. Moreover, by canonical forms we have that $v_1 = \lambda x : \tau' . e'$. Hence, $v_1\ v_2 \to e'\{v_2/x\}$.

At this point we would *like* to apply the induction hypothesis to $e'\{v_2/x\}$ to show that it also evaluates to a value, but doing this would *not* be valid—the induction hypothesis only applies to immediate subexpressions of e! Moreover, we cannot get around this by using the other induction principles we have seen before, such as induction on the size of the expression or on the typing derivation—these induction hypotheses do not apply to $e'\{v_2/x\}$ either! □

We need a different proof technique.

4 Logical Relations

To prove normalization, we will employ a technique invented by Tait in 1967 called a *logical relation*. The idea in a logical relation proof is to define a predicate on expressions indexed on types that implies the property we want.

At base types this set will simply contain all expressions satisfying the property. At function types, we will require that the property be preserved whenever we apply the function to an argument of appropriate type that also has the property.

More formally, we define the following predicate $R_\tau(e)$ inductively on τ. We use e halts as an abbreviation for exists v such that $e \to^* v$.

Definition (Logical Relation)**.**

- $R_{\textbf{unit}}(e)$ iff $\vdash e : \textbf{unit}$ and e halts.
- $R_{\tau_1 \to \tau_2}(e)$ iff $\vdash e : \tau_1 \to \tau_2$ and e halts, and for every e' such that $R_{\tau_1}(e')$ we have $R_{\tau_2}(e\ e')$.

Normalization then follows from the next few lemmas.

The first states the correspondence between R_τ and halting.

Lemma 1. *If $R_\tau(e)$ then e halts.*

The proof is straightforward as halting is built into each case of the definition of the logical relation.

The second states that all closed well-typed expressions satisfy the predicate at their type.

Lemma 2. *If $\vdash e : \tau$ then $R_\tau(e)$*

To prove the first, we will need the following lemma:

Lemma 3. *If $\vdash e : \tau$ and $e \to e'$ then $R_\tau(e)$ iff $R_\tau(e')$.*

We leave the proof of this lemma as an exercise.

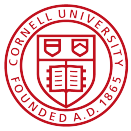
Returning to Lemma 2, we strengthen the induction hypothesis to allow a non-empty typing context:

Lemma 4. *If $x_1 : \tau_1 \dots x_k : \tau_k \vdash e : \tau$, and v_1 to v_k are values such that $\vdash v_1 : \tau_1$ to $\vdash v_k : \tau_k$ and $R_{\tau_1}(v_1)$ to $R_{\tau_k}(v_k)$, then $R_\tau(e\{v_1/x_1\} \dots \{v_k/x_k\})$.*

Proof. By structural induction on e.

- **Case** $e = x$**:**
  By inversion we have that $x = x_i$ and $\tau = \tau_i$. By definition, $e\{v_1/x_1\} \dots \{v_k/x_k\} = v_i$. We have $R_{\tau_i}(v_i)$ by assumption.
- **Case** $e = ()$**:**
  By inversion we have that $\tau = \textbf{unit}$. By definition, $e\{v_1/x_1\} \dots \{v_k/x_k\} = ()$. We obtain $R_{\textbf{unit}}(())$ by the definition of the logical relation as $\vdash () : \textbf{unit}$ and $()$ halts.
- **Case** $e = \lambda x : \tau'.\, e'$**:**
  By inversion we have $\tau = \tau' \to \tau''$ and $x_1 : \tau_1 \dots x_k : \tau_k, x : \tau' \vdash e' : \tau''$. We immediately have that $(\lambda x : \tau'.\, e')\{v_1/x_1\} \dots \{v_k/x_k\}$ halts since it is already a value. Let e'' be an arbitrary expression such that $R_{\tau'}(e'')$. By definition of the logical relation we have $\vdash e'' : \tau'$ and e'' halts. So there exists a v'' such that $e'' \to^* v''$. By Lemma 3 we have that $R_{\tau'}(v'')$. By the induction hypothesis, we have $R_{\tau''}(e'\{v_1/x_1\} \dots \{v_k/x_k\}\{v''/x\})$. Hence, by the definition of the operational semantics and the previous lemma again we also have $R_{\tau''}(e\{v_1/x_1\} \dots \{v_k/x_k\})\ e'')$. Therefore by the definition of the logical relation we have $R_{\tau' \to \tau''}(e\{v_1/x_1\} \dots \{v_k/x_k\})$ as required.
- **Case** $e = e_1\ e_2$**:**
  By inversion we have $x_1 : \tau_1 \dots x_k : \tau_k \vdash e_1 : \tau' \to \tau''$ and $x_1 : \tau_1 \dots x_k : \tau_k \vdash e_1 : \tau'$ and $\tau = \tau''$. By induction hypothesis we have $R_{\tau' \to \tau''}(e_1\{v_1/x_1\} \dots \{v_k/x_k\})$ and $R_{\tau'}(e_2\{v_1/x_1\} \dots \{v_k/x_k\})$. By Lemma 1 we have that $e_1\{v_1/x_1\} \dots \{v_k/x_k\}$ halts. By the definition of the logical relation we have $R_{\tau''}(e_1\{v_1/x_1\} \dots \{v_k/x_k\}\ e_2\{v_1/x_1\} \dots \{v_k/x_k\})$, which is $R_{\tau''}((e_1\ e_2)\{v_1/x_1\} \dots \{v_k/x_k\})$, as required. □

CS 4110 – Programming Languages and Logics
Lectures #22: Advanced Types



1 Overview

In this lecture we will extend the simply-typed λ-calculus with several features we saw earlier in the course, including products, sums, and references, as well as one new one.

1.1 Products

We have previously seen how to encode *products* into untyped λ-calculus.

$$e ::= \cdots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$
$$v ::= \cdots \mid (v_1, v_2)$$

We defined congruence rules that determine the order of evaluation, using the following evaluation contexts.

$$E ::= \cdots \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E$$

We also defined two computation rules that determin how the pairing constructor and destructors interact.

$$\frac{}{\#1\ (v_1, v_2) \to v_1} \qquad \frac{}{\#2\ (v_1, v_2) \to v_2}$$

In simply-typed λ-calculus, the type of a product expression (or a *product type*) is a pair of types, written $\tau_1 \times \tau_2$. The typing rules for the product constructors and destructors are as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

Note the similarities between these rules and the proof rules for conjunction in natural deduction. We will examine this relationship closely later in the course.

1.2 Sums

The next example, *sums*, are dual to products. Intuitively, a product holds two values, one of type τ_1, and one of type τ_2, while a sum holds a single value that is either of type τ_1 or of type τ_2. The type of a sum is written $\tau_1 + \tau_2$. There are two constructors for sums, corresponding to whether we are constructing a sum with a value of τ_1 or a value of τ_2.

$$e ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3$$
$$v ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v$$

There are congruence rules that determine the order of evaluation, as defined by the following evaluation contexts.

$$E ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ E \mid \mathsf{inr}_{\tau_1+\tau_2}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3$$

There are also two computation rules that that show how the constructors and destructors interact.

$$\frac{}{\mathsf{case}\ \mathsf{inl}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \to e_2\ v} \qquad \frac{}{\mathsf{case}\ \mathsf{inr}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \to e_3\ v}$$

The type of a sum expression (or a *sum type*) is written $\tau_1 + \tau_2$. The typing rules for the sum constructors and destructor are the following.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \to \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ e_1 \mid e_2 : \tau}$$

Let's see an example of a program that uses sum types.

$$\begin{array}{l}\mathsf{let}\ f = \lambda a : \mathbf{int} + (\mathbf{int} \to \mathbf{int}).\ \mathsf{case}\ a\ \mathsf{of}\ (\lambda y.\ y + 1) \mid (\lambda g.\ g\ 35)\ \mathsf{in} \\ \mathsf{let}\ h = \lambda x : \mathbf{int}.\ x + 7\ \mathsf{in} \\ f\ (\mathsf{inr}_{\mathbf{int}+(\mathbf{int}\to\mathbf{int})}\ h)\end{array}$$

The function f takes argument a, which is a sum—that is, the actual argument for a will either be a value of type **int** or a value of type **int** $\to$ **int**. We destruct the sum value with a case statement, which must be prepared to take either of the two kinds of values that the sum may contain. In this instance, we end up applying f to a value of type **int** $\to$ **int** (i.e., a value injected into the right type of the sum), so the entire program ends up evaluating to 42.

1.3 References

Next we consider mutable references. Recall the syntax and semantics for references.

$$\begin{aligned} e &::= \cdots \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \ell \\ v &::= \cdots \mid \ell \\ E &::= \cdots \mid \mathsf{ref}\ E \mid !E \mid E := e \mid v := E \end{aligned}$$

$$\textsc{Alloc}\ \frac{}{\langle \sigma, \mathsf{ref}\ v\rangle \to \langle \sigma[\ell \mapsto v], \ell\rangle}\ \ell \notin dom(\sigma) \qquad \textsc{Deref}\ \frac{}{\langle \sigma,\ !\ell\rangle \to \langle \sigma, v\rangle}\ \sigma(\ell) = v$$

$$\textsc{Assign}\ \frac{}{\langle \sigma, \ell := v\rangle \to \langle \sigma[\ell \mapsto v], v\rangle}$$

To extend the type system, we add a new type, τ **ref**, to stand for the type of a location that contains a value of type τ. For example the expression $\mathsf{ref}\ 7$ has type **int ref**, since it evaluates to a location that contains a value of type **int**. Dereferencing a location of type τ **ref** results in a value of type τ, so $!e$ has type τ if e has type τ **ref**. And for assignment $e_1 := e_2$, if e_1 has type τ **ref**, then e_2 must have type τ.

$$\tau ::= \cdots \mid \tau\ \mathbf{ref}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsf{ref}\ e : \tau\ \textbf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \textbf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \textbf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

Note that there is no typing rule for location values. What should the type of a location value ℓ be? Clearly, it should be of type τ **ref**, where τ is the type of the value contained in location ℓ. But how do we know what value is contained in location ℓ? We could directly examine the store, but this would not be inefficient. In addition, examining the store directly may not give us a conclusive answer! Consider, for example, a store σ and location ℓ where $\sigma(\ell) = \ell$; what is the type of ℓ?

Instead, we introduce *store typings* to track the types of values stored in locations. Store typings are partial functions from locations to types. We use metavariable Σ to range over store typings. Our typing relation now becomes a relation over 4 entities: typing contexts, store typings, expressions, and types. We write $\Gamma, \Sigma \vdash e : \tau$ when expression e has type τ under typing context Γ and store typing Σ.

Our new typing rules for references are as follows. (Typing rules for other constructs are modified to take a store typing in the obvious way.)

$$\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textsf{ref}\ e : \tau\ \textbf{ref}} \quad \frac{\Gamma, \Sigma \vdash e : \tau\ \textbf{ref}}{\Gamma, \Sigma \vdash\ !e : \tau} \quad \frac{\Gamma, \Sigma \vdash e_1 : \tau\ \textbf{ref} \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau} \quad \frac{}{\Gamma, \Sigma \vdash \ell : \tau\ \textbf{ref}}\ \Sigma(\ell) = \tau$$

So, how do we state type soundness? Our type soundness theorem for simply-typed lambda calculus said that if $\Gamma \vdash e : \tau$ and $e \to^* e'$ then e' is not stuck. But our operational semantics for references now has a store, and our typing judgment now has a store typing in addition to a typing context. We need to adapt the definition of type soundness appropriately. to do so, we define what it means for a store to be well-typed with respect to a typing context.

Definition. Store σ is *well-typed* with respect to typing context Γ and store typing Σ, written $\Gamma, \Sigma \vdash \sigma$, if $dom(\sigma) = dom(\Sigma)$ and for all $\ell \in dom(\sigma)$ we have $\Gamma, \Sigma \vdash \sigma(\ell) : \Sigma(\ell)$.

We can now state type soundness for our language with references.

Theorem (Type soundness)**.** *If $\cdot, \Sigma \vdash e : \tau$ and $\cdot, \Sigma \vdash \sigma$ and $\langle e, \sigma \rangle \to^* \langle e', \sigma' \rangle$ then either e' is a value, or there exists e'' and σ'' such that $\langle e', \sigma' \rangle \to \langle e'', \sigma'' \rangle$.*

We can prove type soundness for our language using the same strategy as for the simply-typed lambda calculus: using the preservation and progress lemmas. The progress lemma can be easily adapted for the semantics and type system for references. Adapting preservation is a little more involved, since we need to describe how the store typing changes as the store evolves. The rule ALLOC extends the store σ with a fresh location ℓ, producing store σ'. Since $dom(\Sigma) = dom(\sigma) \neq dom(\sigma')$, it means that we will not have σ' well-typed with respect to typing store Σ.

Since the store can increase in size during the evaluation of the program, we also need to allow the store typing to grow as well.

Lemma (Preservation)**.** *If $\Gamma, \Sigma \vdash e : \tau$ and $\Gamma, \Sigma \vdash \sigma$ and $\langle e, \sigma \rangle \to \langle e', \sigma' \rangle$ then there exists some $\Sigma' \supseteq \Sigma$ such that $\Gamma, \Sigma' \vdash e' : \tau$ and $\Gamma, \Sigma' \vdash \sigma'$.*

We write $\Sigma' \supseteq \Sigma$ to mean that for all $\ell \in dom(\Sigma)$ we have $\Sigma(\ell) = \Sigma'(\ell)$. This makes sense if we think of partial functions as sets of pairs: $\Sigma \equiv \{(\ell, v) \mid \ell \in dom(\Sigma) \wedge \Sigma(\ell) = v\}$. Note that the preservation lemma states simply that there is some store type $\Sigma' \supseteq \Sigma$, but does not specify what

exactly that store typing is. Intuitively, Σ' will either be Σ, or Σ extended with a newly allocated location.

Interestingly, references are enough to recover Turing completeness. For example, to implement a recursive function f we can initialize a reference cell containing a dummy value for f and then "backpatch" it with the actual definition. For example, here is an implementation of the familiar factorial function, written using **let** expressions, conditionals, and natural numbers for clarity.

$$\begin{array}{l}\textbf{let}\ r = \mathsf{ref}\ \lambda x.\, 0\ \textbf{in} \\ r := \lambda x\!:\!\textbf{int}.\ \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x \times\ !r\ (x-1)\end{array}$$

This trick is known as "Landin's knot" after its inventor.

1.4 Fixpoints

Another way to obtain fixpoints in the simply-typed lambda calculus is to simply add a new primitive fix to the language. The evaluation rules for the new primitive mimic the behavior of the fixpoint combinators we saw previously.

We extend the syntax with the new primitive operator. Intuitively, fix e is the fixed-point of the function e. Note that fix v is *not* a value.

$$e ::= \cdots \mid \mathsf{fix}\ e$$

We extend the operational semantics for the new operator. There is a new evaluation context, and a new axiom.

$$E ::= \cdots \mid \mathsf{fix}\ E \qquad\qquad \frac{}{\mathsf{fix}\ \lambda x\!:\!\tau.\, e \to e\{(\mathsf{fix}\ \lambda x\!:\!\tau.\, e)/x\}}$$

Note that we can define the letrec $x\!:\!\tau = e_1$ in e_2 construct in terms of the fix operator.

$$\mathsf{letrec}\ x\!:\!\tau = e_1\ \mathsf{in}\ e_2 \triangleq \mathsf{let}\ x = \mathsf{fix}\ \lambda x\!:\!\tau.\, e_1\ \mathsf{in}\ e_2$$

The typing rule for fix is left as an exercise.

Returning to our trusty factorial example, the following program implements the factorial function using the fix operator.

$$\mathsf{FACT} \triangleq \mathsf{fix}\ \lambda f\!:\!\textbf{int} \to \textbf{int}.\ \lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ n \times (f\ (n-1))$$

Note that we can write non-terminating computations for any type: the expression fix $\lambda x\!:\!\tau.\, x$ has type τ, and does not terminate.

Although the fix operator is normally used to define recursive functions, it can be used to find fixed points of any type. For example, consider the following expression.

$$\begin{array}{l}\mathsf{fix}\ \lambda x\!:\!(\textbf{int} \to \textbf{int}) \times (\textbf{int} \to \textbf{int}).\ (\lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ (\#2\ x)\ (n-1), \\ \qquad\qquad\qquad\qquad \lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\#1\ x)\ (n-1))\end{array}$$

This expressiondefines a pair of mutually recursive functions; the first function returns true if and only if its argument is even; the second function returns true if and only if its argument is odd.

1.5 Exceptions

Many programming langauges provide support for throwing and catching exceptions. We can model an extremely simple form of exceptions by extending the simply-typed λ-calculus with a single exception representing an error. We first extend the syntax of the language,

$$e ::= \ldots \mid \textbf{error} \mid \textbf{try}\ e\ \textbf{with}\ e$$

and then add new evaluation contexts,

$$E ::= \cdots \mid \textbf{try}\ E\ \textbf{with}\ e$$

and rules for propagating and catching exceptions:

$$\frac{}{E[\textbf{error}] \to \textbf{error}} \qquad \frac{}{\textbf{try error with}\ e \to e} \qquad \frac{}{\textbf{try}\ v\ \textbf{with}\ e \to v}$$

The typing rule for exceptions allows them to take *any* type, while the typing rule for try-with expressions requires both sub-expressions to have the same type:

$$\frac{}{\Gamma \vdash \textbf{error} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{try}\ e_1\ \textbf{with}\ e_2 : \tau}$$

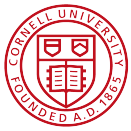
The first typing rule is extremely flexible, allowing errors to be thrown anywhere in a program. However, it is not hard to see that it causes the progress lemma to become false: the expression **error** is not a value but is stuck. Fortunately, we can prove the following weaker version, which is still strong enough to prove a useful form of type soundness.

Lemma (Progress)**.** *If $\vdash e : \tau$ then e is a value or e is **error** or there exists e' such that $e \to e'$.*

The preservation theorem remains unchanged.
The actual soundness theorem is as follows:

Theorem 1 (Soundness)**.** *If $\vdash e : \tau$ and $e \to^* e'$ and $e' \not\to$ then either e is a value or e is **error**.*

CS 4110 – Programming Languages and Logics
Lecture #23: Polymorphism



1 Parametric polymorphism

Polymorphism (Greek for "many forms") is the ability for code to be used with values of different types. For example, a polymorphic function is one that can be invoked with arguments of different types. A polymorphic datatype is one that can contain elements of different types.

There are several different kinds of polymorphism that are commonly used in modern programming languages.

- *Subtype polymorphism* allows a term to have many types using the subsumption rule, which allows a value of type τ to masquerade as a value of type τ' provided that τ is a subtype of τ'.
- *Ad-hoc polymorphism* usually refers to code that appears to be polymorphic to the programmer, but the actual implementation is not. For example, languages with *overloading* allow the same function name to be used with functions that take different types of parameters. Although it looks like a polymorphic function to the code that uses it, there are actually multiple function implementations (none being polymorphic) and the compiler invokes the appropriate one. Ad-hoc polymorphism is a dispatch mechanism: the type of the arguments is used to determine (either at compile time or run time) which code to invoke.
- *Parametric polymorphism* refers to code that is written without knowledge of the actual type of the arguments; the code is parametric in the type of the parameters. Examples include polymorphic functions in ML and Java generics.

In this lecture we will consider parametric polymorphism in detail. As a motivating example, suppose we are working in the simply-typed λ-calculus, and consider a "doubling" function for integers that takes a function f, and an integer x, applies f to x, and then applies f to the result.

$$\mathsf{doubleInt} \triangleq \lambda f\!:\!\mathbf{int} \to \mathbf{int}.\ \lambda x\!:\!\mathbf{int}.\ f\ (f\ x)$$

We could also write a double function for booleans. Or for functions over integers. Or for any other type...

$$\begin{aligned}\mathsf{doubleBool} &\triangleq \lambda f\!:\!\mathbf{bool} \to \mathbf{bool}.\ \lambda x\!:\!\mathbf{bool}.\ f\ (f\ x)\\ \mathsf{doubleFn} &\triangleq \lambda f\!:\!(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int}).\ \lambda x\!:\!\mathbf{int} \to \mathbf{int}.\ f\ (f\ x)\\ &\vdots\end{aligned}$$

In the simply-typed λ-calculus, if we want to apply the doubling operation to different types of arguments in the same program, we need to write a new function for each type. This violates a fundamental principle of software engineering:

> **Abstraction Principle:** *Every major piece of functionality in a program should be implemented in just one place in the code. When similar functionality is provided by distinct pieces of code, the two should be combined into one by abstracting out the varying parts.*

In the doubling functions above, the varying parts are the types. We need a way to abstract out the type of the doubling operation, and later instantiate it with different concrete types.

1.1 Polymorphic λ-calculus

We can extend the simply-typed λ-calculus with abstraction over types. The resulting system is known by two names: *polymorphic λ-calculus* and *System F*.

A *type abstraction* is a new expression, written $\Lambda X.\, e$, where Λ is the upper-case form of the Greek letter lambda, and X is a *type variable*. A *type application*, written $e_1\ [\tau]$, *instantiaties* a type application at a particular type.

When a type abstraction meets a type application during evaluation, we substitute the free occurrences of the type variable with the type. Importantly, instantiation does not require the program to keep run-time type information, or to perform type checks at run-time; it is just used as a way to statically check type safety in the presence of polymorphism.

1.2 Syntax and operational semantics

The syntax of the polymorphic λ-calculus is given by the following grammar.

$$\begin{aligned} e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e\ [\tau] \\ v &::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda X.\, e \end{aligned}$$

The evaluation rules are the same as for the simply-typed λ-calculus, as well as two new rules for evaluating type abstractions and applications.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E\ [\tau]$$

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x\!:\!\tau.\, e)\ v \to e\{v/x\}}$$

$$\text{TYPE-REDUCTION}\ \frac{}{(\Lambda X.\, e)\ [\tau] \to e\{\tau/X\}}$$

To illustrate, consider a simple example. In this language, the polymorphic identity function is written as

$$\mathsf{ID} \triangleq \Lambda X.\, \lambda x\!:\!X.\, x$$

We can apply the polymorphic identity function to **int**, yielding the identity function on integers.

$$(\Lambda X.\, \lambda x\!:\!X.\, x)\ [\mathbf{int}] \to \lambda x\!:\!\mathbf{int}.\, x$$

We can apply ID to other types as well:

$$(\Lambda X.\, \lambda x\!:\!X.\, x)\ [\mathbf{int} \to \mathbf{int}] \to \lambda x\!:\!\mathbf{int} \to \mathbf{int}.\, x$$

1.3 Type system

We also need to provide a type for the new type abstraction. The type of $\Lambda X.\, e$ is $\forall X.\, \tau$, where τ is the type of e, and may contain the type variable X. Intuitively, we use this notation because we can instantiate the type expression with any type for X: for any type X, expression e can have the type τ (which may mention X).

Type checking expressions is slightly different than before. Besides the type environment Γ (which maps variables to types), we also need to keep track of the set of type variables Δ. This is to ensure that a type variable X is only used in the scope of an enclosing type abstraction $\Lambda X.\, e$. Thus, typing judgments are now of the form $\Delta, \Gamma \vdash e : \tau$, where Δ is a set of type variables, and Γ is a typing context. We also use an additional judgment $\Delta \vdash \tau$ ok to ensure that type τ uses only type variables from the set Δ.

$$\frac{}{\Delta, \Gamma \vdash n : \textsf{int}} \qquad \frac{}{\Delta, \Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau\ \textsf{ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Delta \cup \{X\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda X.\, e : \forall X.\, \tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall X.\, \tau' \quad \Delta \vdash \tau\ \textsf{ok}}{\Delta, \Gamma \vdash e\ [\tau] : \tau'\{\tau/X\}}$$

$$\frac{}{\Delta \vdash X\ \textsf{ok}}\ X \in \Delta \qquad \frac{}{\Delta \vdash \textsf{int}\ \textsf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \textsf{ok} \quad \Delta \vdash \tau_2\ \textsf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \textsf{ok}} \qquad \frac{\Delta \cup \{X\} \vdash \tau\ \textsf{ok}}{\Delta \vdash \forall X.\, \tau\ \textsf{ok}}$$

2 Programming in Polymorphic λ-Calculus

Now we consider a number of examples of programming in the polymorphic λ-calculus.

2.1 Doubling

Let's consider the doubling operation again. We can write a polymorphic doubling operation as

$$\textsf{double} \triangleq \Lambda X.\, \lambda f : X \to X.\, \lambda x : X.\, f\ (f\ x).$$

The type of this expression is

$$\forall X.\, (X \to X) \to X \to X$$

We can instantiate this on a type, and provide arguments. For example,

$$\begin{aligned} \textsf{double}\ [\textsf{int}]\ (\lambda n : \textsf{int}.\, n + 1)\ 7 &\to (\lambda f : \textsf{int} \to \textsf{int}.\, \lambda x : \textsf{int}.\, f\ (f\ x))\ (\lambda n : \textsf{int}.\, n + 1)\ 7 \\ &\to^* 9 \end{aligned}$$

2.2 Self Application

Recall that in the simply-typed λ-calculus, we had no way of typing the expression $\lambda x.\, x\ x$. In the polymorphic λ-calculus, however, we can type this expression if we give it a polymorphic type and instantiate it appropriately.

$$\vdash \quad \lambda x{:}\forall X.\, X \to X.\, x\ [\forall X.\, X \to X]\ x \quad : \quad (\forall X.\, X \to X) \to (\forall X.\, X \to X)$$

2.3 Sums and Products

We can encode sums and products in polymorphic λ-calculus without adding any additional types! The encodings are based on the Church encodings from untyped λ-calculus.

$$\begin{array}{rl}
\tau_1 \times \tau_2 & \triangleq \forall R.\, (\tau_1 \to \tau_2 \to R) \to R \\
(\cdot, \cdot) & \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1, \lambda v_2 : T_2.\, \Lambda R.\, \lambda p : (T_1 \to T_2 \to R).\, p\ v_1\ v\ 2 \\
\pi_1 & \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v : T_1 \times T_2.\, v\ [T_1]\ (\lambda x : T_1.\, \lambda y : T_2.\, x) \\
\pi_2 & \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v : T_1 \times T_2.\, v\ [T_2]\ (\lambda x : T_1.\, \lambda y : T_2.\, y) \\
 & \\
\mathsf{unit} & \triangleq \forall R.\, R \to R \\
() & \triangleq \Lambda R.\, \lambda x : R.\, x \\
 & \\
\tau_1 + \tau_2 & \triangleq \forall R.(\tau_1 \to R) \to (\tau_2 \to R) \to R \\
\mathsf{inl} & \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_1 : T_1.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_1\ v_1 \\
\mathsf{inr} & \triangleq \Lambda T_1.\, \Lambda T_2.\, \lambda v_2 : T_2.\, \Lambda R.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, b_2\ v_2 \\
\mathsf{case} & \triangleq \Lambda T_1.\, \Lambda T_2.\, \Lambda R.\, \lambda v : T_1 + T_2.\, \lambda b_1 : T_1 \to R.\, \lambda b_2 : T_2 \to R.\, v\ [R]\ b_1\ b_2 \\
 & \\
\mathsf{void} & \triangleq \forall R.\, R
\end{array}$$

3 Type Erasure

The semantics presented above explicitly passes type. In an implementation, one often wants to eliminate types for efficiency. The following translation "erases" the types from a polymorphic λ-calculus expression.

$$\begin{array}{rcll}
\mathit{erase}(x) & = & x & \\
\mathit{erase}(\lambda x{:}\tau.\, e) & = & \lambda x.\, \mathit{erase}(e) & \\
\mathit{erase}(e_1\ e_2) & = & \mathit{erase}(e_1)\ \mathit{erase}(e_2) & \\
\mathit{erase}(\Lambda X.\, e) & = & \lambda z.\, \mathit{erase}(e) & \text{where } z \text{ is fresh for } e \\
\mathit{erase}(e\ [\tau]) & = & \mathit{erase}(e)\ (\lambda x.\, x) &
\end{array}$$

The following theorem states that the translation is adequate.

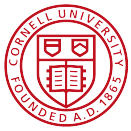
Theorem (Adequacy)**.** *For all expressions e and e', we have $e \to e'$ iff $\mathit{erase}(e) \to \mathit{erase}(e')$.*

4 Type Inference

The type reconstruction problem asks whether, for a given untyped λ-calculus expression e' there exists a well-typed System F expression e such that $\mathit{erase}(e) = e'$. It was shown to be undecidable

by Wells in 1994. See Chapter 23 of Pierce for further discussion, as well as restrictions for which type reconstruction is decidable.

CS 4110 – Programming Languages and Logics
Lecture #24: Type Inference



1 Polymorphism in OCaml

In languages lik OCaml, programmers don't have to annotate their programs with $\forall X.\ \tau$ or $e\ [\tau]$. Both are automatically inferred by the compiler, although the programmer can specify types explicitly if desired.

For example, we can write

```
let double f x = f (f x)
```

and Ocaml will figure out that the type is

```
('a → 'a) → 'a → 'a
```

which is roughly equivalent to

$$\forall A.\ (A \to A) \to A \to A$$

We can also write

```
double (fun x → x+1) 7
```

and Ocaml will infer that the polymorphic function `double` is instantiated on the type `int`.

The polymorphism in ML is not, however, exactly like the polymorphism in System F. ML restricts what types a type variable may be instantiated with. Specifically, type variables can not be instantiated with polymorphic types. Also, polymorphic types are not allowed to appear on the left-hand side of arrows—i.e., a polymorphic type cannot be the type of a function argument. This form of polymorphism is known as *let-polymorphism* (due to the special role played by `let` in ML), or *prenex polymorphism*. These restrictions ensure that *type inference* is decidable.

An example of a term that is typable in System F but not typable in ML is the self-application expression $\lambda x.\ x\ x$. Try typing

```
fun x → x x
```

in the top-level loop of Ocaml, and see what happens...

2 Type Inference

In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: $\lambda x : \tau.\ e$. These annotations are used in the typing rule for functions.

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\ e : \tau \to \tau'}$$

Suppose that we didn't want to provide type annotations for function arguments. We would need to guess a τ to put into the type context.

Can we still type check our program without these type annotations? For the simply typed-lambda calculus (and many of the extensions we have considered so far), the answer is yes: we can *infer* (or *reconstruct*) the types of a program.

Let's consider an example to see how this type inference could work.

$$\lambda a.\, \lambda b.\, \lambda c.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

Since the variable b is used in an addition, the type of b must be **int**. The variable a must be some kind of function, since it is applied to the expression $b+1$. Since a has a function type, the type of the expression $b+1$ (i.e., **int**) must be a's argument type. Moreover, the result of the function application ($a\ (b+1)$) is used as the test of a conditional, so it had better be the case that the result type of a is also **bool**. So the type of a should be **int** $\to$ **bool**. Both branches of a conditional should return values of the same type, so the type of c must be the same as the type of b, namely **int**.

We can write the expression with the reconstructed types:

$$\lambda a : \mathbf{int} \to \mathbf{bool}.\, \lambda b : \mathbf{int}.\, \lambda c : \mathbf{int}.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

2.1 Constraint-based typing

We now present an algorithm that, given a typing context Γ and an expression e, produces a set of *constraints*—equations between types (including type variables)—that must be satisfied in order for e to be well-typed in Γ. We introduce *type variables*, which are just placeholders for types. We let metavariables X and Y range over type variables. The language we will consider is the lambda calculus with integer constants and addition. We assume that all function definitions contain a type annotation for the argument, but this type may simply be a type variable X.

$$\begin{aligned} e &::= x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ \tau &::= \mathbf{int} \mid X \mid \tau_1 \to \tau_2 \end{aligned}$$

To formally define type inference, we introduce a new typing relation:

$$\Gamma \vdash e : \tau \mid C$$

Intuitively, if $\Gamma \vdash e : \tau \mid C$, then expression e has type τ provided that every constraint in the set C is satisfied.

We define the judgment $\Gamma \vdash e : \tau \mid C$ with inference rules and axioms. When read from bottom to top, these inference rules provide a procedure that, given Γ and e, calculates τ and C such that $\Gamma \vdash e : \tau \mid C$.

$$\text{CT-VAR}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \mid \emptyset} \qquad\qquad \text{CT-INT}\ \frac{}{\Gamma \vdash n : \mathbf{int} \mid \emptyset}$$

$$\text{CT-ADD}\ \frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \mathbf{int} \mid C_1 \cup C_2 \cup \{\tau_1 = \mathbf{int}, \tau_2 = \mathbf{int}\}}$$

$$\text{CT-ABS}\ \frac{\Gamma, x:\tau_1 \vdash e:\tau_2 \mid C}{\Gamma \vdash \lambda x:\tau_1.\, e:\tau_1 \to \tau_2 \mid C}$$

$$\text{CT-APP}\ \frac{\Gamma \vdash e_1:\tau_1 \mid C_1 \quad \Gamma \vdash e_2:\tau_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}}{\Gamma \vdash e_1\ e_2 : X \mid C'}\ X \text{ fresh}$$

Note that we must be careful with the choice of type variables—in particular, the type variable in the rule CT-APP must be chosen appropriately.

2.2 Unification

So what does it mean for a set of constraints to be satisfied? To answer this question, we define *type substitutions* (or just *substitutions*, when it's clear from context). A type substitution is a finite map from type variables to types. For example, we write $[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$ for the substitution that maps type variable X to **int**, and type variable Y to **int** $\to$ **int**. Note that the same variable may occur in both the domain and range of a substitution. In that case, the intention is that the substitutions are performed simultaneously. For example the substitution $[X \mapsto \textbf{int}, Y \mapsto (\textbf{int} \to X)]$ maps Y to **int** $\to X$.

More formally, we define substitution of type variables as follows.

$$\sigma(X) = \begin{cases} \tau & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases}$$
$$\sigma(\textbf{int}) = \textbf{int}$$
$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

Note that we don't need to worry about avoiding variable capture, since there are no constructs in the language that bind type variables. If we had polymorphic types $\forall X.\ \tau$ from the polymorphic lambda calculus, we would need to be concerned with this.

Given two substitutions σ and σ', we write $\sigma \circ \sigma'$ for their composition: $(\sigma \circ \sigma')(\tau) = \sigma(\sigma'(\tau))$.

2.2.1 Unification

Constraints are of the form $\tau = \tau'$. We say that a substitution σ *unifies* constraint $\tau = \tau'$ if $\sigma(\tau) = \sigma(\tau')$. We say that substitution σ *satisfies* (or *unifies*) set of constraints C if σ unifies every constraint in C.

For example, the substitution $\sigma = [X \mapsto \textbf{int}, Y \mapsto (\textbf{int} \to \textbf{int})]$ unifies the constraint

$$X \to (X \to \textbf{int}) \ = \ \textbf{int} \to Y$$

since

$$\sigma(X \to (X \to \textbf{int})) \quad = \quad \textbf{int} \to (\textbf{int} \to \textbf{int}) \quad = \quad \sigma(\textbf{int} \to Y)$$

So to solve a set of constraints C, we need to find a substitution that unifies C. More specifically, suppose that $\Gamma \vdash e : \tau \mid C$; a solution for (Γ, e, τ, C) is a pair $\sigma, \tau')$ such that σ satisfies C and $\sigma(\tau) = \tau'$. If there are no substitutions that satisfy C, then we know that e is not typeable.

2.2.2 Unification algorithm

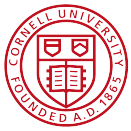
To calculate solutions to constraint sets, we use the idea, due to Hindley and Milner, of using *unification* to check that the set of solutions is non-empty, and to find a "best" solution (from which all other solutions can be easily generated). The unification algorithm is defined as follows:

$$
\begin{aligned}
\mathit{unify}(\emptyset) &= [] \qquad \text{(the empty substitution)} \\
\mathit{unify}(\{\tau = \tau'\} \cup C') &= \text{if } \tau = \tau' \text{ then} \\
&\qquad \mathit{unify}(C') \\
&\quad \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then} \\
&\qquad \mathit{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau'] \\
&\quad \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then} \\
&\qquad \mathit{unify}(C'\{\tau/X\}) \circ [X \mapsto \tau] \\
&\quad \text{else if } \tau = \tau_o \to \tau_1 \text{ and } \tau' = \tau'_o \to \tau'_1 \text{ then} \\
&\qquad \mathit{unify}(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\}) \\
&\quad \text{else} \\
&\qquad \textit{fail}
\end{aligned}
$$

The check that X is not a free variable of the other type ensures that the algorithm doesn't produce a cyclic substitution (e.g., $X \mapsto (X \to X)$), which doesn't make sense with the finite types we currently have.

The unification algorithm always terminates. (How would you go about proving this?) Moreover, it produces a solution if and only if a solution exists. The solution found is the most general solution, in the sense that if $\sigma = \mathit{unify}(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = (\sigma'' \circ \sigma)$.

CS 4110 – Programming Languages and Logics
Lecture #26: Records and Subtyping



1 Records

We have previously seen binary products, i.e., pairs of values. Binary products can be generalized in a straightforward way to n-ary products, also called *tuples*. For example, $\langle 3, (), \mathsf{true}, 42\rangle$ is a 4-ary tuple containing an integer, a unit value, a boolean value, and another integer. Its type is $\mathbf{int} \times \mathbf{unit} \times \mathbf{bool} \times \mathbf{int}$.

Records are a generalization of tuples. We annotate each field of record with a *label*, drawn from some set of labels $\mathcal{L}$. For example, $\{\mathsf{foo} = 32, \mathsf{bar} = \mathsf{true}\}$ is a record value with an integer field labeled foo and a boolean field labeled bar. The type of the record value is written $\{\mathsf{foo} : \mathbf{int}, \mathsf{bar} : \mathbf{bool}\}$. We extend the syntax, operational semantics, and typing rules of the call-by-value lambda calculus to support records.

$$
\begin{array}{l}
l \in \mathcal{L} \\
e ::= \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\
v ::= \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
\tau ::= \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{array}
$$

We add new evaluation contexts to evaluate the fields of records.

$$E ::= \cdots \mid \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \mid E.l$$

We also add a rule to access the field of a location.

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}$$

Finally, we add new typing rules for records. Note that the order of labels is important: the type of the record value $\{\mathsf{lat} = -40, \mathsf{long} = 175\}$ is $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$, which is different from $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$, the type of the record value $\{\mathsf{long} = 175, \mathsf{lat} = -40\}$. In many languages with records, the order of the labels is not important; indeed, we will consider weakening this restriction in the next section.

$$\frac{\forall i \in 1..n. \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

2 Subtyping

Subtyping is a key feature of object-oriented languages. It was first introduced in the SIMULA languages by the Norwegian researchers Dahl and Nygaard.

The principle of subtyping is as follows. If τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$, and also sometimes as $\tau_1 <: \tau_2$), then a program can use a value of type τ_1 whenever it would use a value of type τ_2. If $\tau_1 \leq \tau_2$, then τ_1 is sometimes referred to as the subtype, and τ_2 as the supertype.

We can express the principle of subtyping in a typing rule, often referred to as the "subsumption typing rule" (since the supertype subsumes the subtype).

$$\textsc{Subsumption}\ \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

This rule says that if e has type τ and τ is a subtype of τ', then e also has type τ'. Recall that we provided an intuition for a type as a set of computational entities that share some common property. Type τ is a subtype of type τ' is every computational entity in the set for τ can be regarded as a computational entity in the set for τ'.

So what types are in a subtype relation? We will define inference rules and axioms for the subtype relation $\leq$. The subtype relation is both reflexive and transitive. These properties are intuitive if we think of subtyping as a subset relation. We add inference rules that express this.

$$\frac{}{\tau \leq \tau} \qquad\qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

2.1 Subtyping for records

Consider records and record types. A record consists of a set of labeled fields. Its type includes the types of the fields in the record. Let's define the type **Point** to be the record type $\{\mathsf{x} : \textbf{int}, \mathsf{y} : \textbf{int}\}$, that contains two fields x and y, both integers. That is:

$$\textbf{Point} = \{\mathsf{x} : \textbf{int}, \mathsf{y} : \textbf{int}\}.$$

Lets also define

$$\textbf{Point3D} = \{\mathsf{x} : \textbf{int}, \mathsf{y} : \textbf{int}, \mathsf{z} : \textbf{int}\}$$

as the type of a record with three integer fields x, y and z. Because **Point3D** contains all of the fields of **Point**, and those have the same type as in **Point**, it makes sense to say that **Point3D** is a subtype of **Point**—i.e., **Point3D** $\leq$ **Point**.

Think about any code that used a value of type **Point**. This code could access the fields x and y, and that's pretty much all it could do with a value of type **Point**. A value of type **Point3D** has these same fields, x and y, and so any piece of code that used a value of type **Point** could instead use a value of type **Point3D**.

We can write a subtyping rule for records that allows the subtype to have more fields than the supertype. This is sometimes called "width" subtyping for records.

$$\frac{}{\{l_1 : \tau_1, \ldots, l_{n+k} : \tau_{n+k}\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}\ k \geq 0$$

But why not let the corresponding fields be in a subtyping relation? For example, if $\tau_1 \leq \tau_2$ and $\tau_3 \leq \tau_4$, then is $\{\mathsf{foo} : \tau_1, \mathsf{bar} : \tau_3\}$ a subtype of $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$? (Note that this is only correct because the fields of records are immutable—more on this when we consider subtyping rules for references.) Also, why not relax the requirement that the order of fields be the same? The following rule allows both "depth" and "permutation" subtyping for records (along with the "width" subtyping rule we saw before).

$$\text{S-RECORD}\ \frac{\forall i \in 1..n.\ \exists j \in 1..m.\quad l'_i = l_j\ \ \wedge\ \ \tau_j \leq \tau'_i}{\{l_1 : \tau_1, \ldots, l_m : \tau_m\} \leq \{l'_1 : \tau'_1, \ldots, l'_n : \tau'_n\}}$$

2.2 Top

Many languages a type $\top$ (pronounced "top") that is a supertype of every other type.

$$\text{S-TOP}\ \frac{}{\tau \leq \top}$$

The $\top$ type can be used to model types such as Java's `Object`.

2.3 Subtyping for sums and products

Like records, we can extend the subtyping relation to handle products and sums.

$$\text{S-PRODUCT}\ \frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2} \qquad \text{S-SUM}\ \frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 + \tau_2 \leq \tau'_1 + \tau'_2}$$

2.4 Subtyping for functions

Consider two function types $\tau_1 \to \tau_2$ and $\tau'_1 \to \tau'_2$. What are the subtyping relations between τ_1,, τ_2, τ'_1,, and τ'_2 that should be satisfied in order for $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$ to hold?

Consider the following expression:

$$G \triangleq \lambda f : \tau'_1 \to \tau'_2.\, \lambda x : \tau'_1.\, f\ x.$$

This function has type

$$(\tau'_1 \to \tau'_2) \to \tau'_1 \to \tau'_2.$$

Now suppose we had a function $h : \tau_1 \to \tau_2$ such that $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$. By the subtyping principle, we should be able to give h as an argument to G, and G should work fine. Suppose that v is a value of type τ'_1. Then $G\ h\ v$ will evaluate to $h\ v$, meaning that h will be passed a value of type τ_1. Since h has type $\tau_1 \to \tau_2$, it must be the case that $\tau'_1 \leq \tau_1$. (What could go wrong if $\tau_1 \leq \tau'_1$?)

Furthermore, the result type of $G\ h\ v$ should be of type τ'_2 according to the type of G, but $h\ v$ will produce a value of type τ_2, as indicated by the type of h. So it must be the case that $\tau_2 \leq \tau'_2$.

Putting these two pieces together, we get the typing rule for function types.

$$\text{S-FUNCTION}\ \frac{\tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2}$$

Note that the subtyping relation between the argument and result types in the premise are in different directions! The subtype relation for the result type is in the same direction as for the conclusion (primed version is the supertype, non-primed version is the subtype); it is in the opposite direction for the argument type. We say that subtyping for the function type is *covariant* in the result type, and *contravariant* in the argument type.

2.5 Subtyping for references

Suppose we have a location l of type τ **ref**, and a location l' of type τ' **ref**. What should the relationship be between τ and τ' in order to have τ **ref** $\leq \tau'$ **ref**?

Let's consider the following program R, that takes a location x of type τ' **ref** and reads from it.

$$R \triangleq \lambda x{:}\tau' \textbf{ ref}.\ !x$$

The program R has the type τ' **ref** $\to \tau'$. Suppose we gave R the location l as an argument. Then $R\ l$ will look up the value stored in l, and return a result of type τ (since l is type τ **ref**. Since R is meant to return a result of type τ' **ref**, we thus want to have $\tau \leq \tau'$.

So this suggests that subtyping for reference types is covariant.

But now consider the following program W, which takes a location x of type τ' **ref**, a value y of type τ', and writes y to the location.

$$W \triangleq \lambda x{:}\tau' \textbf{ ref}.\, \lambda y{:}\tau'.\, x := y$$

This program has type τ' **ref** $\to \tau' \to \tau'$. Suppose we have a value v of type τ', and consider the expression $W\ l\ v$. This will evaluate to $l := v$, and since l has type τ **ref**, it must be the case that v has type τ, and so $\tau' \leq \tau$. This suggests that subtyping for reference types is contravariant!

In fact, subtyping for reference types must be *invariant*: reference type τ **ref** is a subtype of τ' **ref** if and only if $\tau = \tau'$. Indeed, to be sound, subtyping for any mutable location must be invariant. Interestingly, in the Java programming language, arrays are mutable locations but have covariant subtyping!

Suppose that we have two classes Person and Student such that Student extends Person (that is, Student is a subtype of Person). The following Java code is accepted, since an array of Student is a subtype of an array of Person, according to Java's covariant subtyping for arrays.

```
Person[ ] arr = new Student[ ] { new Student("Alice") };
```

This is fine as long as we only read from arr. The following code executes without any problems, since arr[0] is a Student which is a subtype of Person.

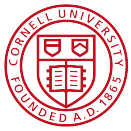
```
Person p = arr[0];
```

However, the following code, which attempts to update the array, has some issues.

```
arr[0] = new Person("Bob");
```

Even though the assignment is well-typed, it attempts to assign an object of type Person into an array of Students! In Java, this produces an ArrayStoreException, indicating that the assignment to the array failed.

CS 4110 – Programming Languages and Logics
Lecture #27: Existential Types



1 Modules

Simple languages, such as C and FORTRAN, often have a single global namespace. This causes problems in large programs due to name collisions—i.e., two different programmers (or pieces of code) using the same name for different purposes—are likely. In addition, it often leads to situations where multiple components of a program are more tightly coupled, since one component may use a name defined by the other.

Modular programming addresses these issues. A *module* is a collection of named entities that are related to each other in some way. Modules provide separate namespaces: different modules have different name spaces, and so can freely use names without worrying about name collisions.

Typically, a module can choose what names and entities to export (i.e., which names to allow to be used outside of the module), and what to keep hidden. The exported entities are declared in an *interface*, and the interface typically does not export details of the implementation. This means that different modules can implement the same interface in different ways. Also, by hiding the details of module implementation, and preventing access to these details except through the exported interface, programmers of modules can be confident that code invariants are not broken.

Packages in Java are a form of modules. A package provides a separate namespace (we can have a class called `Foo` in package `p1` and package `p2` without any conflicts). A package can hide details of its implementation by using private and package-level visibility.

How do we access the names exported by a module? Given a module m that exports an entity names x, common syntax for accessing x is $m.x$. Many languages also provide a mechanism to use all exported names of a module using shorter notation—e.g., "Open m", or "import m", or "using m".

2 Existential types

In this section, we will extend the simply-typed lambda calculus with *existential types* (and records). An existential type is written $\exists X.\ \tau$, where type variable X may occur in τ. If a value has type $\exists X.\ \tau$, it means that it is a pair $\{\tau', v\}$ of a type τ' and a value v, such that v has type $\tau\{\tau'/X\}$.

We introduce a language construct to create existential values, and a construct to use existential values. The syntax of the new language is given by the following grammar.

$$
\begin{aligned}
e &::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\
&\quad \mid \{\ l_1 = e_1, \ldots, l_n = e_n\ \} \mid e.l \\
&\quad \mid \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 \\
v &::= n \mid \lambda x{:}\tau.\, e \mid \{\ l_1 = v_1, \ldots, l_n = v_n\ \} \mid \mathsf{pack}\ \{\tau_1, v\}\ \mathsf{as}\ \exists X.\ \tau_2 \\
\tau &::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid \{\ l_1{:}\tau_1, \ldots, l_n{:}\tau_n\ \} \mid \exists X.\ \tau
\end{aligned}
$$

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, $\mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2$, is often called *packing*, and the construct to use an existential value is called *unpacking*. Before we present the operational semantics and typing rules, let's see an example to get an intuition for packing and unpacking.

Here we create an existential value that implements a counter, without revealing details of its implementation.

$$\begin{array}{l}\mathsf{let}\ counterADT = \\ \qquad \mathsf{pack}\ \{\mathbf{int}, \{\ \mathsf{new} = 0, \mathsf{get} = \lambda i\!:\!\mathbf{int}.\ i, \mathsf{inc} = \lambda i\!:\!\mathbf{int}.\ i+1\ \}\ \} \\ \qquad \mathsf{as}\ \exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \} \\ \mathsf{in}\ \ldots\end{array}$$

The abstract type name is **Counter**, and its concrete representation is **int**. The type of the variable $counterADT$ is $\exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \}$. We can use the existential value $counterADT$ as follows.

$$\begin{array}{l}\mathsf{unpack}\ \{C, c\} = counterADT\ \mathsf{in} \\ \mathsf{let}\ y = c.\mathsf{new}\ \mathsf{in} \\ c.\mathsf{get}\ (c.\mathsf{inc}\ (c.\mathsf{inc}\ y))\end{array}$$

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type

$$\exists \mathbf{Counter}.\ \{\mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\}.$$

Why do we do this? Without this annotation, we would not know which occurrences of the witness type are intended to be replaced with the type variable, and which are intended to be left as the witness type.

In the counter example above, the type of expressions $\lambda i : \mathbf{int}.\ i$ and $\lambda i : \mathbf{int}.\ i + 1$ are both $\mathbf{int} \to \mathbf{int}$, but one is the implementation of get, of type $\mathbf{Counter} \to \mathbf{int}$ and the other is the implementation of inc, of type $\mathbf{Counter} \to \mathbf{Counter}$.

We now define the operational semantics for existentials. We add two new evaluation contexts, and one evaluation rule for unpacking an existential value.

$$E ::= \cdots \mid \mathsf{pack}\ \{\tau_1, E\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = E\ \mathsf{in}\ e$$

$$\frac{}{\mathsf{unpack}\ \{X, x\} = (\mathsf{pack}\ \{\tau_1, v\}\ \mathsf{as}\ \exists Y.\ \tau_2)\ \mathsf{in}\ e \to e\{v/x\}\{\tau_1/X\}}$$

The typing rules ensure that existential values are used correctly.

$$\frac{\Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\} \quad \Delta \vdash \exists X.\ \tau_2\ \mathsf{ok}}{\Delta, \Gamma \vdash \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 : \exists X.\ \tau_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \exists X.\ \tau_1 \quad \Delta \cup \{X\}, \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta, \Gamma \vdash \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable X does *not* appear free in τ_2. This rules out programs such as,

$$\begin{array}{l}\mathsf{let}\ m = \\ \quad\quad \mathsf{pack}\ \{\mathbf{int}, \{a = 5, f = \lambda x\!:\!\mathbf{int}.x + 1\}\}\ \mathsf{as}\ \exists X.\ \{a\!:\!X, f\!:\!X \to X\} \\ \mathsf{in} \\ \mathsf{unpack}\ \{X, x\} = m\ \mathsf{in}\ x.f\ x.a\end{array}$$

where the type of $(f.x\ x.a)$ has X free.

3 Church Encoding

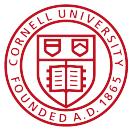
It turns out that we can encode existentials in System F! The idea is to use a Church encoding, where an existential value is a function that takes a type and then calls the continuation

$$\begin{array}{rcl}\exists X.\ \tau & \triangleq & \forall Y.\ (\forall X.\ \tau \to Y) \to Y \\ \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 & \triangleq & \Lambda Y.\ \lambda f : (\forall X.\tau_2 \to Y).\ f\ [\tau_1]\ e \\ \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 & \triangleq & e_1\ [\tau_2]\ (\Lambda X.\lambda x : \tau_1.\ e_2)\end{array}$$

where e_1 has type $\exists X.\tau_1$ and e_2 has type τ_2

For further details see Pierce, Chapter 24.

CS 4110 – Programming Languages and Logics
Lecture #27: Recursive Types



1 Introduction

Many programming languages have the ability to define recursive data types. For example, suppose we want to define binary trees with integer data at the nodes. In Java we can write

```
class Tree {
   Tree leftChild, rightChild;
   int data;
}
```

A binary tree is an object of this class. In OCaml we can write

```
type tree = Leaf | Node of tree * tree * int
```

These types are *recursive* because they are defined in terms of themselves.

In the simply-typed lambda calculus, we do not yet have any mechanism to define recursive types. We would like the type **tree** to satisfy

$$\mathit{tree} = \mathbf{unit} + \mathbf{int} \times \mathit{tree} \times \mathit{tree},$$

In other words, we would like **tree** to be a solution of the equation

$$\alpha = \mathbf{unit} + \mathbf{int} \times \alpha \times \alpha$$

However, no such solution exists among the types we have seen so far.

How might we augment our set of types to include solutions to such recursive type equations? There are two basic approaches, called the *equirecursive* and *isorecursive* approach, respectively.

2 Equirecursive Types

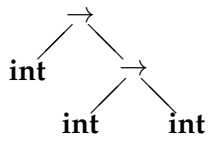
By unwinding the equation above, we can see that

$$\begin{aligned}
\alpha &= \mathbf{unit} + \mathbf{int} \times \alpha \times \alpha \\
&= \mathbf{unit} + \mathbf{int} \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha) \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha) \\
&= \mathbf{unit} + \mathbf{int} \times (\mathbf{unit} + \mathbf{int} \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha) \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha)) \times \\
&\qquad\qquad (\mathbf{unit} + \mathbf{int} \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha) \times (\mathbf{unit} + \mathbf{int} \times \alpha \times \alpha)) \\
&= \cdots
\end{aligned}$$

At each level, we have a finite type with the type variable α appearing at some of the leaves, and we obtain the next level by substituting the right-hand side of the equation for α. This gives a sequence of deeper and deeper finite trees, where each successive tree is a substitution instance of the previous tree.

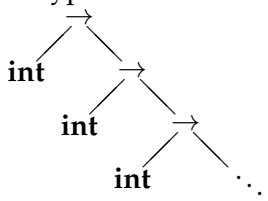
If we take the limit of this process, we have an infinite tree. We can think of this as an infinite labeled graph whose nodes are labeled with the type constructors $\times$, $+$, **int**, and **unit**. This is very much like an ordinary type expression, except that it is infinite. There are no more α's, because we have substituted for all of them all the way down. This infinite tree is a solution of our equation, and this is what we take as the type **tree**.

More generally, over standard type constructors such as $\rightarrow$, $\times$, $+$, **unit**, and **int**, we can form the set of (finite) types inductively in the usual way. Each such type can be regarded as a finite labeled tree. For example, the type **int** $\rightarrow$ **int** $\rightarrow$ **int** can be viewed as the labeled tree



Now let us add some infinite types. These are infinite labeled trees that respect the arities of the constructors; that is, if the constructor is binary (such as $\times$ or $\rightarrow$), any node labeled with that constructor must have exactly two children; and if the constructor is nullary, such at **unit**, then any node labeled with that symbol must be a leaf. Within these constraints, the tree may be infinite.

A (finite or infinite) expression with only finitely many subexpressions (up to isomorphism) is called *regular*. For example, the infinite type



is regular, since it has only two subexpressions up to isomorphism, namely itself and **int**. The limit of the unwinding of the equation above, which we took to be the type **tree**, is also regular; it has exactly five subexpressions up to isomorphism, namely **tree**, **unit**, **tree** $\times$ **tree** $\times$ **int**, **tree** $*$ **tree**, and **int**.

Regular trees are all we need to provide solutions to finite systems of type equations. Suppose we have n type equations in n variables:

$$\begin{aligned} \alpha_1 &= \tau_1 \\ &\vdots \\ \alpha_n &= \tau_n, \end{aligned} \tag{1}$$

where each τ_i is a finite type over the type constructors and type variables $\alpha_1 \ldots \alpha_n$. This system has a solution $\sigma_1 \ldots \sigma_n$ in which each σ_i is a regular tree. Moreover, if no right-hand side is a variable, then the solution is unique.

2.1 The μ Constructor

We can specify the infinite solutions to systems of type equations using a finite syntax involving a new type constructor μ, the *fixpoint type constructor*. If we have an equation $\alpha = \tau$ such that the right-hand side is not α, then there is a unique solution, which is a finite or infinite regular tree.

The solution will be infinite if α occurs in τ and will be finite (in fact it will just be τ) if α does not occur in τ. We denote this unique solution by $\mu\alpha.\,\tau$.

Syntactically, μ acts as a binding operator in type expressions as λ does in λ-terms, with the same notions of scope, free and bound variables, α-conversion, and substitution.

Since $\mu\alpha.\,\tau$ is a solution to $\alpha = \tau$, we have

$$\mu\alpha.\,\tau = \tau\{\mu\alpha.\,\tau/\alpha\}.$$

For example, to get a **tree** type satisfying our original equation, we can define

$$\textbf{tree} \triangleq \mu\alpha.\,\textbf{unit} + \textbf{int} \times \alpha \times \alpha.$$

The solutions $\sigma_1 \ldots \sigma_n$ to any finite system of equations can be expressed in terms of μ. For example, suppose τ_1 and τ_2 are finite type expressions over the type variables α_1, α_2 such that neither τ_1 nor τ_2 is a variable. The system

$$\alpha_1 = \tau_1 \qquad\qquad \alpha_2 = \tau_2$$

has a unique solution σ_1, σ_2 specified by

$$\sigma_1 = \mu\alpha_1.\,(\tau_1\{\mu\alpha_2.\,\tau_2/\alpha_2\}) \qquad\qquad \sigma_2 = \mu\alpha_2.\,(\tau_2\{\mu\alpha_1.\,\tau_1/\alpha_1\}).$$

Mutually recursive type declarations arise quite often in practice. For example, consider the following Java class definitions for `Node` and `Edge`:

```
class Node {
   Edge[] inEdges, outEdges;
}
class Edge {
   Node source, sink;
}
```

Note that `Node` refers to `Edge` and vice versa. So we must take a mutual fixpoint.

2.2 Typing Rules

In the equirecursive view, since $\mu\alpha.\,\tau = \tau\{\mu\alpha.\,\tau/\alpha\}$, the typing rules are simple:

$$\mu\text{-INTRO}\ \frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash e : \mu\alpha.\,\tau} \qquad\qquad \mu\text{-ELIM}\ \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}$$

Equivalently, we can just allow substitution of equals for equals in type expressions.

3 Isorecursive Types

There is another approach to recursive types, the *isorecursive* approach. Here we do not have any infinite types, but rather the expression $\mu\alpha.\,\tau$ is itself a type. In this approach, $\mu\alpha.\,\tau$ and $\tau\{\mu\alpha.\,\tau/\alpha\}$ are considered distinct (but isomorphic) types.

The step of substituting $\mu\alpha.\,\tau$ for α in τ is called *unfolding*, and the reverse operation is called *folding*. The conversion of elements between these two types is accomplished by explicit **fold** and **unfold** operations.

$$\begin{array}{rcl} \mathbf{unfold}_{\mu\alpha.\,\tau} & : & \mu\alpha.\,\tau \;\rightarrow\; \tau\{\mu\alpha.\,\tau/\alpha\} \\ \mathbf{fold}_{\mu\alpha.\,\tau} & : & \tau\{\mu\alpha.\,\tau/\alpha\} \;\rightarrow\; \mu\alpha.\,\tau \end{array}$$

We will often suppress the subscripts when there is no ambiguity. In this view, the equality symbol in a recursive equation is not really an equality, but an isomorphism.

3.1 Typing Rules

In the isorecursive view, the typing rules consist of a pair of introduction and elimination rules for μ-types that explicitly mention **fold** and **unfold**:

$$\mu\text{-INTRO}\;\frac{\Gamma \vdash e : \tau\{\mu\alpha.\,\tau/\alpha\}}{\Gamma \vdash \mathbf{fold}\; e : \mu\alpha.\,\tau} \qquad\qquad \mu\text{-ELIM}\;\frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \mathbf{unfold}\; e : \tau\{\mu\alpha.\,\tau/\alpha\}}$$

3.2 Operational Semantics

With isorecursive types, we also need to augment the operational semantics. We only need one rule:

$$\frac{}{\mathbf{unfold}\;(\mathbf{fold}\; e) \rightarrow e}$$

Intuitively, to access data in a recursive type $\mu\alpha.\,\tau$, we need to unfold it first; but the only way that values of type $\mu\alpha.\,\tau$ could have been created in the first place is via a **fold**.

3.3 An Example

Suppose we want to write a program to add a list of numbers. The list type is a recursive type, which we can define as $\mathbf{intlist} \triangleq \mu\alpha.\,\mathbf{unit} + \mathbf{int} \times \alpha$. Now suppose we want to add up the elements of an **intlist**. This will be a recursive function, so we would need to take a fixpoint. In the body of this function, we would like to do a case on the **intlist** ℓ. But to do a case, we need a sum type, and ℓ is a μ-type, so we will have to unfold it first. (OCaml does this automatically when it sees a match.) So the body would be

$$\begin{array}{l} \mathsf{case}\;\mathbf{unfold}\;\ell\;\mathsf{of} \\ \quad (\lambda u : \mathbf{unit}.\,0) \\ \quad \mid (\lambda p : \mathbf{int} \times \mathbf{intlist}.\,(\#1\;p) + f\;(\#2\;p)) \end{array}$$

This is just the same code that you would write in OCaml, except we have broken out some of the things that Ocaml hides for you. In particular, we have explicitly shown the recursion in the definition of the **intlist** type and the **unfold** that is needed to get the exploded view of the type.

4 Equirecursive vs. Isorecursive

Programming languages deal with recursive types in different ways. Java and Modula-3 take the equirecursive approach, in which the folded and unfolded types are considered equal, and the **fold**/**unfold** operations are just the identity functions. Recursive types and their unfoldings are fully substitutable for each other.

```
class E {
   String x;
   E e;
   public String toString() {
      return e.e.e.e.e.e.e.e.e.e.e.e.e.e.e.e.x;
   }
}
```

On the other hand, the ML family, CLU, and C use isorecursive types, in which $\mu\alpha.\,\tau$ and $\tau\{\mu\alpha.\,\tau/\alpha\}$ are considered different (but isomorphic) types, and the casting operators **fold** and **unfold** are required to go between them. CLU uses "up" and "down" instead of **fold** and **unfold**. In OCaml, the **unfold** operator is performed automatically and implicitly by the "match" and "let" statements and the pattern-matching mechanism. The type constructors in a recursive datatype definition, applied to arguments, act as **fold** operations.

```
# type tree = Leaf | Node of tree * tree * int;;
type tree = Leaf | Node of tree * tree * int
# Node (Leaf, Leaf, 4);;
- : tree = Node (Leaf, Leaf, 4)
```

5 Numbers as a Recursive Type

We started with primitive types **unit**, **boolean**, and **int**. We have already seen that the type **boolean** can be represented as **unit** $+$ **unit** with values true and false represented by left and right injections respectively.

Now that we have recursive types, we no longer need to take **int** as primitive, but we can define it as a recursive type. A natural number is either 0 or a successor of a natural number. Thus we can take

$$\begin{aligned}\textbf{nat} &\triangleq \mu\alpha.\,\textbf{unit} + \alpha\\ 0 &\triangleq \textbf{fold}\ (\mathsf{inl}_{\textbf{nat}}\ ())\\ 1 &\triangleq \textbf{fold}\ (\mathsf{inr}_{\textbf{nat}}\ 0)\\ 2 &\triangleq \textbf{fold}\ (\mathsf{inr}_{\textbf{nat}}\ 1),\end{aligned}$$

and so on. We can use the recursive type **nat** to code up all of the usual arithmetic, and all these operations are well-typed. For example, the successor function would be

$$(\lambda x : \textbf{nat}.\,\textbf{fold}\ (\mathsf{inr}_{\textbf{nat}}\ x)) : \textbf{nat} \to \textbf{nat}.$$

So all we really need as primitive types and type constructors are **unit**, recursive types, products, and sums. With these we can build all the other types like natural numbers, integers, lists, trees, floating point numbers, and so on.

6 Self-Application and Ω

Recall the *paradoxical combinator* Ω defined by

$$\omega \triangleq \lambda x.\, xx \qquad\qquad \Omega \triangleq \omega\, \omega.$$

We can now give these terms recursive types, provided we insert some folding and unfolding. Since x is applied as a function, it must have some kind of function type, say $\sigma \to \tau$. But since it is applied to itself as an argument, it must also have type σ. This seems to indicate that the type of x must satisfy the equation $\sigma = \sigma \to \tau$. The recursive type $\mu\alpha.\, (\alpha \to \tau)$ appears to be in order (here τ can be anything).

To actually apply x to x, we have to unfold it. The type of **unfold** x is

$$\textbf{unfold}\ x : (\mu\alpha.\, (\alpha \to \tau)) \to \tau.$$

This is a function with domain $\mu\alpha.\, (\alpha \to \tau)$, which is the type of x, so we can apply it to x. The type of the result (**unfold** x) x is τ. Thus the fully typed ω term is

$$\omega \triangleq (\lambda x : \mu\alpha.\, (\alpha \to \tau).\, (\textbf{unfold}\ x)\ x)\ :\ (\mu\alpha.\, (\alpha \to \tau)) \to \tau.$$

If we now fold this, we get

$$\textbf{fold}\ \omega : \mu\alpha.\, (\alpha \to \tau).$$

Therefore, we can apply ω as a function to **fold** ω, and the result is

$$\omega\ (\textbf{fold}\ \omega) : \tau.$$

This is the same as the original Ω term, but with explicit folding and unfolding.

We can do this in OCaml:

```
# type u = Fold of (u -> u);;
type u = Fold of (u -> u)
# let omega = fun x -> match x with Fold f -> f x;;
val omega : u -> u = <fun>
# omega (Fold omega);;
...runs forever until you hit control-c
```

So we were finally able to introduce nontermination. But the point is that it passed typechecking, so the program was well-typed.

7 Untyped to Typed λ-Calculus

In fact, with recursive types, we can type everything in the pure untyped lambda calculus. Every λ-term can be applied as a function to any other λ-term, so every λ-term (with appropriate folds and unfolds inserted) has type

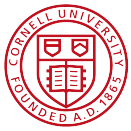
$$U \triangleq \mu\alpha.\, \alpha \to \alpha$$

The translation is

$$\begin{aligned}
[\![x]\!] &\triangleq x \\
[\![e_0\, e_1]\!] &\triangleq (\mathbf{unfold}\ [\![e_0]\!])\ [\![e_1]\!] \\
[\![\lambda x.\, e]\!] &\triangleq \mathbf{fold}\ \lambda x : U.\, [\![e]\!].
\end{aligned}$$

Note that every untyped term maps to a term of type U.

CS 4110 – Programming Languages and Logics
Lecture #29: Propositions as Types



1 Propositions as Types

There is a deep connection between type systems and formal logic. This connection, known as propositions-as-types, was recognized by early 20th century mathematicians and later developed substantially by Haskell Curry and William Howard. Although it is usually formulated in terms of simple type systems (or System F) and proof systems like natural deduction, the connection is actually quite robust and has been extended to many other systems including classical logic. It continues to bear fruit today: recent work by Abramsky, Pfenning, Wadler, and others has developed a connection between the session types used in concurrent process calculi and linear logic.

The main intuitions for propositions-as-types comes from thinking of proofs constructively. For example, the proof rule for introducting a conjunction $\phi \wedge \psi$,

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \wedge\text{-INTRO}$$

can be thought of as a function that takes a proof of ϕ and a proof of ψ and builds a proof of $\phi \wedge \psi$. This is a significant departure from classical logic, which has rules such as excluded middle or double-negation elimination,

$$\frac{}{\Gamma \vdash \psi \vee \neg\psi} \text{ EXCLUDED MIDDLE}$$

that do not have an obvious constructive interpretation.

Propositions-as-types recongizes that each constructive proof can be turned into a program that witnesses the proof, as summarized by the following table.

Type Systems		Formal Logic	
τ	Type	ϕ	Formula
τ	Inhabited type	ϕ	Theorem
e	Well-typed expression	π	Proof

Hence, for every proof in first-order logic, we can obtain a well-typed expression in λ-calculus, and vice versa.

2 Natural Deduction

To illustrate propositions-as-types formally, we begin by reviewing natural deduction—a proof system for first-order logic. The syntax of first-order logic formulas is as follows,

$$\phi ::= \top \mid \bot \mid P \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \to \psi \mid \neg\phi \mid \forall x.\, \phi$$

where P ranges over propositional variables. We will let negation $\neg P$ be an abbreviation for $P \to \bot$.

The proof rules for natural deduction are as follows:

$$\frac{}{\Gamma, \phi \vdash \phi}\ \textsc{Axiom}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\text{-INTRO} \qquad \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ \to\text{-ELIM}$$

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ \wedge\text{-INTRO} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\text{-ELIM1} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\text{-ELIM2}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}\ \vee\text{-INTRO1} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}\ \vee\text{-INTRO2} \qquad \frac{\Gamma \vdash \phi \vee \psi \qquad \Gamma \vdash \phi \to \chi \qquad \Gamma \vdash \psi \to \chi}{\Gamma \vdash \chi}\ \vee\text{-ELIM}$$

$$\frac{\Gamma, P \vdash \phi}{\Gamma \vdash \forall P.\ \phi}\ \forall\text{-INTRO2} \qquad \frac{\Gamma \vdash \forall P.\ \phi}{\Gamma \vdash \phi\{\psi/P\}}\ \forall\text{-ELIM}$$

Note that some rules from classical logic, such as excluded middle,

$$\frac{}{\Gamma \vdash P \vee \neg P}$$

are not included, nor are they derivable from these rules.

3 System F Type System

It should be obvious that these proof rules bear a close correspondence to the type systems we have been developing over the class few weeks. Here are the typing rules for System F, annotated with the same labels:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Axiom}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x{:}\sigma.\, e{:}\tau}\ \to\textsc{-intro} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ \to\textsc{-elim}$$

$$\frac{\Gamma \vdash e_1{:}\sigma \qquad \Gamma \vdash e_2{:}\tau}{\Gamma \vdash (e_1, e_2){:}\sigma \times \tau}\ \wedge\textsc{-intro} \qquad \frac{\Gamma \vdash e{:}\sigma \times \tau}{\Gamma \vdash \#1\ e : \sigma}\ \wedge\textsc{-elim1} \qquad \frac{\Gamma \vdash e{:}\sigma \times \tau}{\Gamma \vdash \#2\ e : \tau}\ \wedge\textsc{-elim2}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathit{inl}_{\sigma+\tau}\ e{:}\sigma + \tau}\ \vee\textsc{-intro1} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{inr}_{\sigma+\tau}\ e{:}\sigma + \tau}\ \vee\textsc{-intro2}$$

$$\frac{\Gamma \vdash e\sigma + \tau \qquad \Gamma \vdash e_1{:}\sigma \to \rho \qquad \Gamma \vdash e_2{:}\tau \to \rho}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ e_1\ e_2{:}\rho}\ \vee\textsc{-elim}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau}\ \forall\textsc{-intro2} \qquad \frac{\Gamma \vdash e{:}\forall\alpha.\ \tau}{\Gamma \vdash e\ [\sigma]{:}\tau\{\sigma/\alpha\}}\ \forall\textsc{-elim}$$

We can summarize the relationship between formulas and types using the following table:

Type Systems		**Formal Logic**	
$\to$	Function	$\to$	Implication
$\times$	Product	$\wedge$	Conjunction
$+$	Sum	$\vee$	Disjunction
$\forall$	Universal	$\forall$	Quantifier

This relationship has also be extended to many other types.

4 Term Assignment

Given a natural deduction proof, there is a corresponding well-typed System F expression. The transformation from a proof to an expression is often called *term assignment*. For example, given the following proof,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \phi \to \psi, \phi \vdash \phi \to \psi}\ \textsc{Axiom} \qquad \dfrac{}{\Gamma, \phi \to \psi, \phi \vdash \phi}\ \textsc{Axiom}}{\Gamma, \phi \to \psi, \phi \vdash \psi}\ \to\textsc{-elim}}{\Gamma, \phi \to \psi, \vdash \phi \to \psi}\ \to\textsc{-intro}}{\Gamma \vdash (\phi \to \psi) \to \phi \to \psi}\ \to\textsc{-intro}$$

we can build the following typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau}\ \textsc{Axiom} \qquad \dfrac{}{\Gamma, x : \sigma \to \tau, y : \sigma \vdash \sigma}\ \textsc{Axiom}}{\Gamma, x : \sigma \to \tau, y : \sigma \vdash x\ y : \tau}\ \to\text{-ELIM}}{\Gamma, x : \sigma \to \tau \vdash \lambda y{:}\sigma.\, x\ y : \sigma \to \tau}\ \to\text{-INTRO}}{\Gamma \vdash \lambda x{:}\sigma \to \tau.\, \lambda y{:}\sigma.\, x\ y : (\sigma \to \tau) \to \sigma \to \tau}\ \to\text{-INTRO}$$

Hence, the term

$$\lambda x{:}\sigma \to \tau.\, \lambda y{:}\sigma.\, x\ y$$

witnesses the proof of

$$(\phi \to \psi) \to \phi \to \psi$$

More generally, to prove a proposition ϕ, it suffices to find a well-typed expression e of type τ where ϕ and τ are related under propositions as types.

5 Negation and Continuations

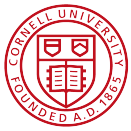
The problem of extending propositions-as-types to classical logic was an open question for many years. It was not at all obvious how to do this, as the usual constructive interpretation of proofs does not readily extend to rules such as excluded middle. In the late 1980s, Griffin showed that continuation-passing style corresponds to a way of embedding classical logic into constructive logic.

Given an expression e of type τ, we can think of the continuation-passing style transformation as converting the expression into one of type $(\tau \to \bot) \to \bot$. Intuitively, the $\tau \to \bot$ is the type of the continuation, which "never returns." Since $\neg\phi$ is just an abbreviation for $\phi \to \bot$, this corresponds to the following classical rule:

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\neg\phi}$$

This yields a way to prove any formula that is classically valid in constructive logic using the double-negation embedding.

CS 4110 – Programming Languages and Logics
Lecture #30: Featherweight Java



One way to model the features of an object-oriented languages is to encode it using standard type structure. This leads to so-called object encodings. A different (and arguably simpler) way is to model these features directly. This lecture considers a core calculus for Java developed by Igarashi, Pierce, and Wadler called *Featherweight Java*.

Featherweight Java is small by design. It reduces Java to its essential features including classes, inheritance, constructors, fields, methods, and casts, and omits everything else. In particular, the language does not include interfaces, assignment, concurrency, overloading, exceptions, or the `public`, `private`, and `protected` modifiers. Because the language is so simple, its proof of type soundness is short. It is also easy to extend—indeed, in the original paper on Featherweight Java, the authors present an extension with parametric polymorphism (i.e., generics).

1 Featherweight Java

The syntax of Featherweight Java is given by the following grammar.

$$
\begin{array}{rcll}
P & ::= & \overline{CL}\ e & \textit{programs} \\
CL & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{C\ f};\ K\ \overline{M}\} & \textit{classes} \\
K & ::= & C(\overline{C\ f})\ \{\texttt{super}(\overline{f}); \overline{\texttt{this}.f = f;}\} & \textit{constructors} \\
M & ::= & C\ m(\overline{C\ x})\{\texttt{return}\ e\} & \textit{methods} \\
e & ::= & x & \textit{expressions} \\
 & | & e.f & \\
 & | & e.m(\overline{e}) & \\
 & | & \texttt{new}\ C(\overline{e}) & \\
 & | & (C)\ e & \\
v & ::= & \texttt{new}\ C(\overline{v}) & \textit{values} \\
E & ::= & [\cdot] & \textit{evaluation contexts} \\
 & | & E.f & \\
 & | & E.m(\overline{e}) & \\
 & | & v.m(\overline{v}, E, \overline{e}) & \\
 & | & \texttt{new}\ C(\overline{v}, E, \overline{e}) & \\
 & | & (C)\ E &
\end{array}
$$

We will use the notation $\overline{e}$ to denote sequences of the form $e_1, \ldots, e_k$ (and $\overline{C\ f}$ for $C_1\ f_1, \ldots, C_k\ f_k$). By convention, metavariables B, C, and D range over class names, m ranges over method names, and f and g range over field names. As usual, x ranges over variables. Note that the syntax of Featherweight Java is a strict subset of Java. This means that every Featherweight Java program can be executed using a stock Java compiler and virtual machine.

At the top level, programs consist of a list of classes and a distinguished "main" expression. We will use the notation $P(C)$ to denote the definition of the class C in the program. A class has a name, a superclass, a list of fields (instance variables), a constructor, and a list of methods. A constructor takes a list of arguments, invokes the `super(...)` constructor, and then initializes its fields. A method takes a list of arguments and returns a single expression—a variable, field projection, method invocation, constructor call, or cast.

Although this language is simple, we can still write many useful programs (in fact, all useful programs—the language is Turing complete). Here is a simple example that illustrates how we can represent pairs in Featherweight Java:

```
class A extends Object {
  A() { super(); }
}

class B extends Object {
  B() { super(); }
}

class Pair extends Object  {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }

  Pair swap Object() {
    return new Pair(this.snd, this.fst);
  }
}
```

Using the small-step operational semantics described later in this lecture, it will be possible to evaluate the expression

```
new Pair(new A(), new B()).swap()
```

to the following:

```
new Pair(new B(), new A())
```

Note that because the language does not include assignment (except in constructors), Featherweight Java programs must be written in a functional style, constructing new objects instead of mutating old ones.

As another example, consider what happens when we evaluate the following expression:

```
(A) new B()
```

Because `B` is not declared to be a subtype of `A`, the cast fails. In the full Java language, the virtual machine would raise an exception. In Featherweight Java, we model this instead as a stuck term.

2 Subtype Relation

The subtype relation is the reflexive and transitive closure of the binary relation between classes and superclasses. Formally it is defined using the following axioms and inference rules:

$$\text{S-REFL}\ \frac{}{C \leq C} \qquad \text{S-TRANS}\ \frac{C \leq D \quad D \leq E}{C \leq E}$$

$$\text{S-CLASS}\ \frac{P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\}}{C \leq D}$$

Note that Featherweight Java subtyping is *nominal*, just like Java—the objects generated by a class are a subtype of the objects generated by its superclass.

3 Auxiliary Functions

Before we present the operational semantics for Featherweight Java, let us define a few auxiliary functions for looking up the methods and fields of classes.

Field Lookup The set of fields defined in a class is simply the list of all fields in the definition of the class in the program, as well as the fields of its superclass.

$$\text{F-OBJECT}\ \frac{}{\mathit{fields}(\texttt{Object}) = []}$$

$$\text{F-CLASS}\ \frac{P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \qquad \mathit{fields}(D) = \overline{D\ g}}{\mathit{fields}(C) = \overline{D\ g}\ @\ \overline{C\ f}}$$

Method Body Lookup Similarly, to lookup the body of a method we either read it off from the class definition, or take the method body of the superclass. Note that the structure returned includes both the arguments $\overline{x}$ and the body of the method e.

$$\text{MB-CLASS}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \in \overline{M}\end{array}}{\mathit{mbody}(m, C) = (\overline{x}, e)}$$

$$\text{MB-SUPER}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \notin \overline{M}\end{array}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

4 Operational Semantics

The operational semantics for Featherweight Java is defined in the usual way, using small-step operational semantics rules and evaluation contexts. It uses a call-by-value evaluation strategy.

$$\text{E-CONTEXT}\ \frac{e \to e'}{E[e] \to E[e']}$$

$$\text{E-PROJ}\ \frac{\mathit{fields}(C) = \overline{C\ f}}{\texttt{new}\ C(\overline{v}).f_i \to v_i}$$

$$\text{E-INVK}\ \frac{\mathit{mbody}(m, C) = (\overline{x}, e)}{\texttt{new}\ C(\overline{v}).m(\overline{u}) \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C(\overline{v})]e}$$

$$\text{E-CAST}\ \frac{C \leq D}{(D)\ \texttt{new}\ C(\overline{v}) \to \texttt{new}\ C(\overline{v})}$$

Note that the rule for method invocation steps to the body of the method with the actual arguments substituted for the formal parameters and the object that the method is being invoked on, `new` C $(\overline{v})$, substituted for `this`. Also note that in the cast rule, the target type of the cast D must be a supertype of the object `new` $C(\overline{v})$ being cast—i.e., an upcast simply strips off the cast while downcasts and casts between unrelated classes get stuck.

5 Type System

The type system for Featherweight Java has three main pieces (and several auxiliary definitions). The first is the typing relation for expressions, which is a three-place relation $\Gamma \vdash e : C$ between a context Γ that maps variables to their types, an expression e, and a type C.

Method Type Lookup The typing relation for expressions relies on an auxiliary definition that calculates method types. To calculate the type of a method `m` in a class `C` we either look it up from the class definition, or take the type of the method in its superclass.

$$\text{MT-CLASS}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \in \overline{M}\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B}$$

$$\text{MT-SUPER}\ \frac{\begin{array}{c}P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \\ B\ m\ (\overline{B\ x})\ \{\texttt{return}\ e\} \notin \overline{M}\end{array}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

Expression Typing With this auxiliary definition in hand, we are ready to define the typing relation for expressions:

$$\text{T-VAR}\ \frac{\Gamma(x) = C}{\Gamma \vdash x : C}$$

$$\text{T-FIELD}\ \frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = \overline{C\ f}}{\Gamma \vdash e.f_i : C_i}$$

$$\text{T-INVK}\ \frac{\Gamma \vdash e : C \qquad \mathit{mtype}(m, C) = \overline{B} \to B \qquad \Gamma \vdash \overline{e} : \overline{A} \qquad \overline{A} \leq \overline{B}}{\Gamma \vdash e.m(\overline{e}) : B}$$

$$\text{T-NEW}\ \frac{\mathit{fields}(C) = \overline{C\ f} \qquad \Gamma \vdash \overline{e} : \overline{B} \qquad \overline{B} \leq \overline{C}}{\Gamma \vdash \texttt{new}\ C(\overline{e}) : C}$$

$$\text{T-UCAST}\ \frac{\Gamma \vdash e : D \qquad D \leq C}{\Gamma \vdash (C)\ e : C}$$

$$\text{T-DCAST}\ \frac{\Gamma \vdash e : D \qquad C \leq D \qquad C \neq D}{\Gamma \vdash (C)\ e : C}$$

$$\text{T-SCAST}\ \frac{\Gamma \vdash e : D \qquad C \not\leq D \qquad D \not\leq C \qquad \textit{stupid warning}}{\Gamma \vdash (C)\ e : C}$$

Note that it includes three typing rules for casts—one for upcasts, another for downcasts, and another for "stupid" casts between unrelated types. Stupid casts are not allowed by the standard Java typechecker but are needed to prove preservation.

Method Typing The next definition is a two place relation that checks that a method `m` is "okay" in a class `C`. It uses an auxiliary definition *override* that checks that a method validly overrides any methods with the same name defined by its superclass.

$$\text{OVERRIDE}\ \frac{\mathit{mtype}(m, D) = \overline{A} \to A \textit{ implies } \overline{A} = \overline{B} \text{ and } A = B}{\mathit{override}(m, D, \overline{B} \to B)}$$

$$\text{METHOD-OK}\ \frac{\overline{x : B}, \texttt{this} : C \vdash e : A \qquad A \leq B \qquad P(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\} \qquad \mathit{override}(m, D, \overline{B} \to B)}{B\ m(\overline{B\ x})\{\texttt{return}\ e\}\ \mathit{OK\ in}\ C}$$

Class Typing The final piece of the type system checks that a class is "okay".

$$\text{CLASS-OK}\ \frac{K = C(\overline{D\ g}, \overline{C\ f})\ \{\texttt{super}(\overline{g}); \overline{\texttt{this}.f = f;}\} \qquad \mathit{fields}(D) = \overline{D\ g} \qquad \overline{M}\ \mathit{OK\ in}\ C}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f};\ K\ \overline{M}\}\ \mathit{OK}}$$

To typecheck a whole program, we check that every class is okay, and that the main expression is well-typed under the empty context.

CS 4110 – Programming Languages and Logics
Lecture #31: Featherweight Java Properties and Object Encodings

In this lecture, we will develop a proof of type soundness for Featherweight Java in the usual way, as a corollary of progress and preservation. The details of these proofs will be a little different than the ones we have seen before, however, due to the presence of subtyping and casts. We will then develop a different way of formalizing object-oriented languages using object encodings.

1 Properties

1.1 Preservation

The proof of preservation relies on several supporting lemmas.

Lemma (Method Typing). *If* $mtype(m, C) = \overline{D} \to D$ *and* $mbody(m, C) = (\overline{x}, e)$ *then there exists types* C' *and* D' *such that* $\overline{x} : \overline{D}, \texttt{this} : C' \vdash e : D'$ *and* $D' \leq D$.

Lemma (Substitution). *If* $\Gamma, \overline{x} : \overline{B} \vdash e : C$ *and* $\Gamma \vdash \overline{u} : \overline{B'}$ *with* $\overline{B'} \leq \overline{B}$ *then there exists* C' *such that* $\Gamma \vdash [\overline{x} \mapsto \overline{u}]e : C'$ *and* $C' \leq C$.

Lemma (Weakening). *If* $\Gamma \vdash e : C$ *then* $\Gamma, x : B \vdash e : C$.

Lemma (Decomposition). *If* $\Gamma \vdash E[e] : C$ *then there exists a type* B *such that* $\Gamma \vdash e : B$

Lemma (Context). *If* $\Gamma \vdash E[e] : C$ *and* $\Gamma \vdash e : B$ *and* $\Gamma \vdash e' : B'$ *with* $B' \leq B$ *then there exists a type* C' *such that* $\Gamma \vdash E[e'] : C'$ *and* $C' \leq C$.

Lemma (Preservation). *If* $\Gamma \vdash e : C$ *and* $e \to e'$ *then there exists a type* C' *such that* $\Gamma \vdash e' : C'$ *and* $C' \leq C$.

Proof. By induction on $e \to e'$, with a case analysis of the last rule used in the derivation.

Case E-CONTEXT: $e = E[e_1]$ and $e_1 \to e_1'$ and $e' = E[e_1']$

By the decomposition lemma we have that there exists a type B such that $\Gamma \vdash e_1 : B$. By the induction hypothesis applied to e_1 we have that there exists a type B' such that $\Gamma \vdash e_1' : B'$ and $B' \leq B$. Then, by the context lemma we have that there exists a type C' such that $\Gamma \vdash E[e_1'] : C'$ and $C' \leq C$, as required.

Case E-PROJ: $e = \texttt{new}\ C_0(\overline{v}).f_i$ and $e' = v_i$ with *fields*$(C_0) = \overline{C\ f}$

As the typing rules for Featherweight Java are syntax-directed, the last rule used in the derivation of $\Gamma \vdash e : C$ must have been T-FIELD. Therefore we must also have a derivation $\Gamma \vdash \texttt{new}\ C_0(\overline{v}) : D_0$ with *fields*$(D_0) = \overline{D\ g}$ and $C = D_i$. By a similar argument, the last rule used in this derivation must have been T-NEW and so $D_0 = C_0$ and we have derivations $\Gamma \vdash \overline{v} : \overline{B}$ with $\overline{B} \leq \overline{D}$. From $D_0 = C_0$ (and as *fields* is a function) we have $\overline{C\ f} = \overline{D\ g}$, and hence $C = C_i$. Thus, $\Gamma \vdash v_i : B_i$ with $B_i \leq C_i$, as required.

Case E-INVK: $e = (\texttt{new}\ C_0(\overline{v})).m(\overline{u})$ and $e' = [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C_0(\overline{v})]e$ with *mbody*$(m, C_0) = (\overline{x}, e)$

By similar reasoning as in the previous case, the last two rules in the derivation of $\Gamma \vdash e : C$ must have been T-INVK and T-NEW with $\Gamma \vdash \texttt{new}\ C_0(\overline{v}) : C_0$ and $\Gamma \vdash \overline{u} : \overline{B}$ and *mtype*$(m, C_0) = \overline{C} \to C$ with $\overline{B} \leq \overline{C}$. By the method typing lemma, there exist types C_0' and C' such that $\overline{x} : \overline{C}, \texttt{this} : C_0' \vdash e : C'$. By the substitution lemma we have $\vdash [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C_0(\overline{v})]e : C''$ with $C'' \leq C'$. By weakening we have $\Gamma \vdash [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C_0(\overline{v})]e : C''$. The required result follows as $C'' \leq C$ by S-TRANS.

Case E-CAST: $e = (C)\ (\texttt{new}\ C_0(\overline{v}))$ and $e' = \texttt{new}\ C_0(\overline{v})$ with $C_0 \leq C$

By similar reasoning as the previous cases, the last two rules in the derivation of $\Gamma \vdash e : C$ must have been T-UCAST and T-NEW with $\Gamma \vdash \texttt{new}\ C_0(\overline{v}) : C_0$. The result is immediate as $C_0 \leq C$.

□

1.2 Progress

The proof of progress also relies on a few supporting lemmas.

Lemma (Canonical Forms)**.** *If* $\vdash v : C$ *then* $v = \texttt{new}\ C(\overline{v})$.

Lemma (Inversion)**.**

1. *If* $\vdash (\texttt{new}\ C(\overline{v})).f_i : C_i$ *then* $\texttt{fields}(C) = \overline{C\ f}$ *and* $f_i \in \overline{f}$.
2. *If* $\vdash (\texttt{new}\ C(\overline{v})).m(\overline{u}) : C$ *then* $\texttt{mbody}(m, C) = (\overline{x}, e)$ *and* $|\overline{u}| = |\overline{e}|$.

Lemma (Progress)**.** *Let* e *be an expression such that* $\vdash e : C$*. Then either:*

1. e *is a value,*
2. *there exists an expression* e' *such that* $e \to e'$*, or*
3. $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ *with* $A \not\leq B$.

Proof. By induction on $\vdash e : C$, with a case analysis on the last rule used in the derivation.

Case T-VAR: $e = x$ with $\emptyset(x) = C$

Can't happen, as $\emptyset(x)$ is undefined.

Case T-FIELD: $e = e_0.f$ with $\vdash e_0 : C_0$ and *fields*$(C_0) = \overline{C\ f}$ and $C = C_i$

By the induction hypothesis applied to e_0 we have that either e_0 is a value, there exists e_0' such that $e_0 \to e_0'$, or there exists E such that $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$. We analyze each of these subcases:

1. If e_0 is a value then by the canonical forms lemma, $e_0 = \texttt{new}\ C_0(\overline{v})$ and by the inversion lemma $f \in \overline{f}$. By E-PROJ we have $e \to v_i$.
2. Alternatively, if there exists an expression such that $e_0 \to e_0'$ then by E-CONTEXT we have $e = E[e_0] \to E[e_0']$ where $E = [\cdot].f$.
3. Otherwise, if $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$ then we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ where $E = [\cdot].f$, which finishes the case.

Case T-INVK: $e = e_0.m(\overline{e})$ with $\vdash e_0 : C_0$ and *mtype*$(m, C_0) = \overline{B} \to C$ and $\vdash \overline{e} : \overline{A}$ and $\overline{A} \leq \overline{B}$

By the induction hypothesis applied to e_0 we have that either e_0 is a value, there exists e_0' such that $e_0 \to e_0'$, or there exists E such that $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$. We analyze each of these subcases:

1. If e_0 is a value then by the canonical forms lemma, $e_0 = \texttt{new}\ C_0(\overline{v})$. If $\overline{e}$ is a list of values $\overline{u}$, then by the inversion lemma we have $|\overline{u}| = |\overline{x}|$ where *mbody*$(m, C_0) = (\overline{x}, e_0')$. By E-INVK we have $e \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C_0(\overline{v})]e_0'$. Otherwise, let i be the least index of an expression in $\overline{e}$ that is not a value. By the induction hypothesis applied to e_i we have that e_i is a value, or there exists e_i' such that $e_i \to e_i'$ or there exists E_i such that $e_i = E_i[(B)\ (\texttt{new}\ A(\overline{v}))]$ and $A \not\leq B$. In the first subsubcase, then we have a contradiction to our assumption that i is the index of the least expression in $\overline{e}$ that is not a value. Otherwise let $E = (\texttt{new}\ C_0(\overline{v})).m(e_1, \ldots, e_{i-1}, E_i, e_{i+1}, \ldots |\overline{e}|)$. In the second subcase, we have $e = E[e_i] \to E[e_i']$ by E-CONTEXT. In the third subcase, we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$.
2. Alternatively, if there exists an expression such that $e_0 \to e_0'$ then by E-CONTEXT we have $E[e_0] \to E[e_0']$ where $E = [\cdot].m(\overline{e})$.
3. Otherwise, if $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$ then we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ where $E = [\cdot].m(\overline{e})$, which finishes the case.

Case T-NEW: $e = \texttt{new}\ C(\overline{e})$ and *fields*$(C) = \overline{C\ f}$ and $\vdash \overline{e} : \overline{B}$ and $\overline{B} \leq \overline{C}$

If $\overline{e}$ is a list of values $\overline{u}$, then e is a value. Otherwise, let i be the least index of an expression in $\overline{e}$ that is not a value. By the induction hypothesis applied to e_i we have that e_i is a value, or there exists e_i' such that $e_i \to e_i'$ or there exists E_i such that $e_i = E_i[(B)\ (\texttt{new}\ A(\overline{v}))]$ and $A \not\leq B$. We analyze each of these subcases:

1. If e_i is a value then we have a contradiction to our assumption that i is the index of the least expression in $\overline{e}$ that is not a value.

2. If there exists e_i' such that $e_i \to e_i'$ then let $E = (\texttt{new}\ C(e_1, \ldots, e_{i-1}, E_i, e_{i+1}, \ldots, |\overline{e}|)$. By E-CONTEXT we have $e = E[e_i] \to E[e_i']$.
3. Otherwise, if there exists E_i with $e_i = E_i[(B)\ (\texttt{new}\ A(\overline{v}))]$ and $A \not\leq B$ then let $E = (\texttt{new}\ C(e_1, \ldots, e_{i-1}, E_i, e_{i+1}, \ldots, |\overline{e}|)$. By construction we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$, which finishes the case.

Case T-UCAST: $e = (C)\ e$ with $\vdash e_0 : D$ and $D \leq C$

By the induction hypothesis applied to e_0 we have that either e_0 is a value, there exists e_0' such that $e_0 \to e_0'$, or there exists E such that $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$. We analyze each of these subcases:

1. If e_0 is a value then by the canonical forms lemma, $e_0 = \texttt{new}\ D(\overline{v})$. By E-CAST we have $e \to \texttt{new}\ D(\overline{v})$.
2. Alternatively, if there exists an expression such that $e_0 \to e_0'$ then by E-CONTEXT we have $e = E[e_0] \to E[e_0']$ where $E = (C)\ [\cdot]$.
3. Otherwise, if $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$ then we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ where $E = (C)\ [\cdot]$, which finishes the case.

Case T-DCAST: $e = (C)\ e$ with $\vdash e_0 : D$ and $C \leq D$ and $C \neq D$

By the induction hypothesis applied to e_0 we have that either e_0 is a value, there exists e_0' such that $e_0 \to e_0'$, or there exists E such that $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$. We analyze each of these subcases:

1. If e_0 is a value then by the canonical forms lemma we have that $e = \texttt{new}\ D(\overline{v})$. Let $E = [\cdot]$. We immediately $e = E[(C)\ \texttt{new}\ C(\overline{v})]$ with $D \not\leq C$.
2. Alternatively, if there exists an expression such that $e_0 \to e_0'$ then by E-CONTEXT we have $e = E[e_0] \to E[e_0']$ where $E = (C)\ [\cdot]$.
3. Otherwise, if $e_0 = E_0[(B)\ (\texttt{new}\ A(\overline{v}))]$ with $A \not\leq B$ then we have $e = E[(B)\ (\texttt{new}\ A(\overline{v}))]$ where $E = (C)\ [\cdot]$, which finishes the case.

Case T-SCAST: similar to the previous case.

□

2 Object Encodings

Another way to formalize the semantics of object-oriented languages is to define translations that map them into λ-calculus. In fact, with records, fixpoints, subtyping, and recursive/existential types, we have all of the tools needed to do this. We begin by briefly reviewing the main features of object-oriented languages.

Dynamic dispatch Dynamic dispatch allows the code executed when a message is sent to an object—e.g., $o.m(x)$—to be determined by run-time values and not (just) by compile-time information such as types. As a result, different objects may respond to the same message in different ways. For example, consider the following Java program:

```
interface Shape { ... void draw() { ... } }
class Circle extends Shape { ... void draw() { ... } }
class Square extends Shape { ... void draw() { ... } }
...
Shape s = ...; //could be a circle a square, or something else.
s.draw();
```

Invoking `s.draw()` could run the code for any of the methods shown in the program (or for any other class that extends `Shape`).

In Java, all methods (except for static methods) are dispatched dynamically. In C++, only virtual members are dispatched dynamically. Note that dynamic dispatch is not the same as overloading, which is usually resolved using the static types of the arguments to the function being called.

Encapsulation Encapsulation allows an object to hide the representations of certain internal data structures. For example, Java programmers often keep instance variables private and write public methods for accessing and modifying the data stored in those variables.

```
class Circle extends Shape {
  private Point center;
  private int radius;
  ...
  public Point getX() { return center.x }
  public Point getY() { return center.y }
}
```

the coordinates of the center of the circle can only be obtained by invoking the `getX` and `getY` methods. The result is that all interactions with the object must be performed by invoking the methods exposed in its public interface and not by directly manipulating its instance variables.

Subtyping Another characteristic feature of object-oriented languages is subtyping. Subtyping fits naturally with object-oriented languages because (ignoring languages such as Java that allow certain objects to manipulate instance variables directly) the only way to interact with an object is to invoke a method. As a result, an object that supports the same methods as another object can be used wherever the second is expected. For example, if we write a method that takes an object of type `Shape` above as a parameter, it is safe to pass `Circle`, `Square`, or any other subtype of `Shape`, because they each support the methods listed in the `Shape` interface.

Inheritance To avoid writing the same code twice, it is often useful to be able to reuse the definition of one kind of object to define another kind of object. In class-based languages, inheritance is often supported through subclassing. For example, in the following Java program,

```
class A {
  public int f(...) { ... g(...) ... }
  public bool g(...) { ... }
}
```

```
class B extends A {
  public bool g(...) { ... }
}
...
new B.f(...)
```

`B` inherits the `f` method of its superclass `A`.

One way to implement inheritance is by duplicating code but this wastes space. Most languages introduce a level of indirection instead so that the code compiled for the object being inherited from can be used directly by the object doing the inheriting.

Note that inheritance is different than subtyping: subtyping is a relation on types while inheritance is a relation on implementations. These two notions are conflated in some languages like Java but kept separate in languages like C++ (which allows a "private base class") as well as in languages that are not based on classes.

Open recursion Finally, many object-oriented languages allow objects to invoke their own methods using the special keyword `this` (or `self`). Implementing `this` in the presence of inheritance requires deferring the binding of `this` until the object is actually created. We will see an example of this in the next section.

2.1 Simple Record Encoding

Let us start with a simple example, developing a representation for two-dimensional point objects using records and references. Records provide both dynamic lookup and subtyping: given a value v of some record type τ, the expression $v.f$ evaluates to a value that is determined by v not by τ—i.e., dynamic dispatch! Moreover, because the subtyping relation on record types allows extension, code that expects an object to have type τ can be used with a value of any subtype of τ.

Here is a simple example showing how we can encode records using objects. For concreteness, we use OCaml syntax rather than λ-calculus. The notation `(fun x -⟩ e)` denotes a λ-abstraction.

```
type pointRep = { x:int ref;
                  y:int ref }

type point = { movex:int -> unit;
               movey:int -> unit }

let pointClass : pointRep -> point =
  (fun (r:pointRep) ->
    { movex = (fun d -> r.x := !(r.x) + d);
      movey = (fun d -> r.y := !(r.x) + d) })

let newPoint : int -> int -> point =
  (fun (x:int) ->
    (fun (y:int) ->
      pointClass { x=ref x; y = ref y }))
```

The `pointRep` type defines the representation for the object's instance variables—a record with a mutable reference for each field. The `pointClass` function takes a record with this type and builds an object—a record with functions `movex` and `movey`, which translate the point horizontally and vertically. The constructor `newPoint` takes two integers, `x` and `y`, and uses `pointClass` to build an object whose fields are initialized to those coordinates.

2.2 Inheritance

Just as in standard object-oriented languages, we can extend our two-dimensional point with an extra coordinate by defining a subclass that inherits the methods of its superclass.

```
type point3D = { movex:int -> unit;
                 movey:int -> unit;
                 movez:int -> unit }

let point3DClass : point3DRep -> point3D =
  (fun (r:point3DRep) ->
    let super = pointClass r in
    { movex = super.movex;
      movey = super.movey;
      movez = (fun d -> r.z := !(r.x) + d) } )

let newPoint3D : int -> int -> int -> point3D =
  (fun (x:int) ->
    (fun (y:int) ->
      (fun (z:int) ->
        point3DClass { x=ref x; y = ref y; z = ref z }))))
```

The most interesting part of this program is the `point3DClass` function. It takes an argument of type `point3DRep` and uses `pointClass` to build a `point` object `super`. It fills in the `movex` and `movey` methods for the object being constructed with the corresponding fields from `super`—i.e., it inherits those methods from the superclass—and defines the new method `movez` directly. Note that we can pass a record of type `point3DRep` to `pointClass` because `point3DRep` is a subtype of `pointRep`.

2.3 Self

Adding support for `self` is a bit trickier because we need `self` to be bound late. Here is an example that illustrates one possible implementation technique:

```
type altPointRep = { x:int ref;
                     y:int ref }

type altPoint = { movex:int -> unit;
                  movey:int -> unit;
                  move: int -> int -> unit }
```

```
let altPointClass : altPointRep -> altPoint ref -> altPoint =
  (fun (r:altPointRep) ->
    (fun (self:altPoint ref) ->
      { movex = (fun d -> r.x := !(r.x) + d);
        movey = (fun d -> r.y := !(r.y) + d);
        move  = (fun dx dy -> (!self.movex) dx; (!self.movey) dy) }))

let dummyAltPoint : altPoint =
  { movex = (fun d -> ());
    movey = (fun d -> ());
    move  = (fun dx dy -> ()) }

let newAltPoint : int -> int -> altPoint =
  (fun (x:int) ->
    (fun (y:int) ->
      let r = { x=ref x; y = ref y } in
      let cref = ref dummyAltPoint in
      cref := altPointClass r cref;
      !cref ))
```

For the sake of the example, we have added a method `move` that takes two integers and translates the point both horizontally and vertically. The implementation of `move` invokes the `movex` and `movey` methods from the current object—i.e., `self`.

To make `self` work as expected, we use a trick similar to the one we used to implement recursive definitions in our λ-calculus interpreter. Compared to our previous object encodings there are two key changes. First, the `newAltPointClass` now takes the `self` reference as an explicit parameter. This parameter is filled in with the actual object when it is constructed. Second, the `newAltPoint` constructor "ties the recursive knot" by allocating a reference cell for the object—filled in initially with a dummy value—and then "back-patching" the reference with the actual object returned by the class.

There is a small problem with this encoding of self: the `self` parameter to `altPointClass` has type `altPoint ref` and references have an *invariant* subtyping rule. As a result, the type system will not allow us to pass a reference to an object generated by a subclass. However, as we do not assign to `self`, it would be safe to use a *covariant* subtyping rule. See Pierce, Chapter 18 for details on how this issue can be resolved.

2.4 Encapsulation

The simple object encoding we have developed in this section already gives us basic encapsulation. After we build an object, the instance variables are totally hidden—we can only manipulate them using object's methods. More complicated forms of abstraction and information hiding can be obtained using existential types. For the details of how records and existential types can be combined to encode objects: see "Comparing Object Encodings", by Bruce, Cardelli, and Pierce, Information and Computation 155(1/2):108–133, 1999.