Programming Languages: Theory and Practice

(WORKING DRAFT OF APRIL 5, 2006.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2006

[Draft of April 5, 2006.]

Preface

This is a collection of lecture notes for Computer Science 15–312 *Programming Languages*. I am grateful to Andrew Appel and Frank Pfenning for their advice, suggestions, and numerous examples. I am also grateful to our students at Carnegie Mellon, Princeton, and University of Massachusetts whose enthusiasm (and patience!) was instrumental in helping to create the course and this text.

What follows is a working draft of a planned book that seeks to strike a careful balance between developing the theoretical foundations of programming languages and explaining the pragmatic issues involved in their design and implementation. Many considerations come into play in the design of a programming language. I seek here to demonstrate the central role of type theory and operational semantics in helping to define a language and to understand its properties.

Comments and suggestions are most welcome. Enjoy!

Contents

Part I

Preliminaries

Chapter 1

Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. In this chapter we will develop the basic framework of inductive definitions, and give some examples of their use.

1.1 Judgements

We start with the notion of a *judgement*, or *assertion*, about one or more objects of study. In general a judgement is a statement of *knowledge*, including assertions such as "it is raining outside", "every natural number may be written as a product of primes", or "the sum of 2 and 2 is 4." In the study of programming languages we shall make use of many forms of judgement, including the following (informal) examples:

a ast	a is an abstract syntax tree
τ type	τ is a type
$e : \tau$	expression e has type τ
$e \Downarrow v$	expression e has value v

Each of these forms of judgement makes an assertion about one or more objects.[1]

The notation for writing judgements varies according to the situation, but when discussing the general case we use *postfix* notation, writing $x_1, \ldots, x_n\ J$,

[1]For the time being we do not specify what sort of objects may be the subject of a judgement, but see Section 1.7 for more on this.

or, briefly, $\vec{x}\ J$, to assert that J holds of the objects $\vec{x} = x_1, \ldots, x_n$. The judgement $\vec{x}\ J$ is sometimes called an *instance* of the *judgement form J*.

1.2 Rules and Derivations

In the study of programming languages we make frequent use of *inductive definitions* of judgement forms. An inductive definition of an n-ary judgement form, J, consists of a set of *inference rules* of the form

$$\frac{\vec{x}_1\ J \quad \cdots \quad \vec{x}_k\ J}{\vec{x}\ J}$$

where $\vec{x}$ and each $\vec{x}_1 \ldots, \vec{x}_k$ are n-tuples of objects, and J is the judgement being defined. The judgements above the horizontal line are called the *premises* of the rule, and the judgement below is called the *conclusion* of the rule. If a rule has no premises (*i.e.*, $n = 0$), the rule is called an *axiom*; otherwise it is a *proper rule*.

An inference rule may be read as an implication stating that *if* $\vec{x}_1\ J$ and …and $\vec{x}_k\ J$ are all valid, *then* $\vec{x}\ J$ is valid as well. Thus each inference rule specifies that the premises are a *sufficient* condition for the conclusion: to show $\vec{x}\ J$, it suffices to show that $\vec{x}_1\ J, \ldots, \vec{x}_k\ J$ hold. Notice that when $k = 0$ (*i.e.*, when the rule is an axiom), nothing is required for the conclusion to hold — it must hold under all circumstances.

A set of rules may be regarded as an *inductive definition* of the judgement form, J, by insisting that the rules are *necessary*, as well as sufficient, conditions for the validity of instances of J. This is to say that $\vec{x}\ J$ is valid *only if* there is some rule with that conclusion all of whose premises are also valid. The judgement, J, *inductively defined* by the given set of rules is the *strongest*, or *most restrictive*, judgements form satisfying the given rules.

An example will help clarify these ideas. The following set, $\mathcal{N}$, of rules constitute an inductive definition of the judgement form nat.

$$\frac{}{\texttt{zero nat}} \qquad \frac{x\ \texttt{nat}}{\texttt{succ}(x)\ \texttt{nat}}$$

According to these rules the judgement x nat is defined to hold exactly when either x is zero, or x is $\texttt{succ}(y)$ for some y such that y nat. In other words, x nat holds iff x is (the unary form of) a natural number.

Similarly, the following set, $\mathcal{T}$, of rules constitute an inductive definition of the judgement form tree:

$$\frac{}{\texttt{empty}\ \mathsf{tree}} \qquad \frac{x\ \mathsf{tree} \quad y\ \mathsf{tree}}{\texttt{node}(x,y)\ \mathsf{tree}}$$

According to these rules the judgement x tree holds exactly when x is a binary tree, either the empty tree, empty, or a node, $\texttt{node}(x_1, x_2)$, with children x_1 and x_2 such that x_1 tree and x_2 tree.

It is worth noting that the inductive definitions of natural numbers and trees each have *infinitely many* rules, not just two. In each case the variables x and y range over the universe of objects over which the definition takes place. What we have specified is a *rule scheme* with *metavariables* x and y. A rule scheme stands for infinitely many rules, one for each choice of object for each of the metavariables involved. This raises the question of what is the universe of objects over which we are working — just which objects are permissible in an inductive definition? For now we rely on informal intuition, but see Section 1.7 for further discussion.

To show that an instance of an inductively defined judgement form is valid, it is enough to exhibit a *derivation (tree)* of it consisting of a composition of rules, starting from axioms and ending with that judgement. Derivations have a natural tree structure arising from stacking rules on top of the another. If

$$\frac{\vec{x}_1\ J \quad \cdots \quad \vec{x}_k\ J}{\vec{x}\ J}$$

is an inference rule and $\mathcal{D}_1, \ldots, \mathcal{D}_k$ are derivations of its premises, then

$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_k}{\vec{x}\ J}$$

is a derivation of its conclusion.

For example, here is a derivation of $\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))$ nat according to the set of rules, $\mathcal{N}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero}\ \mathsf{nat}}}{\texttt{succ}(\texttt{zero})\ \mathsf{nat}}}{\texttt{succ}(\texttt{succ}(\texttt{zero}))\ \mathsf{nat}}}{\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))\ \mathsf{nat}}$$

Similarly, here is a derivation that $\texttt{node(node(empty,empty),empty)}$ $\texttt{tree}$ according to the set of rules $\mathcal{T}$:

$$\dfrac{\dfrac{\dfrac{}{\texttt{empty tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(empty,empty) tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(node(empty,empty),empty) tree}}$$

In general, to show that a judgement is derivable we need only find a derivation for it. There are two main methods for finding a derivation, called *forward chaining*, or *bottom-up construction*, and *backward chaining*, or *top-down construction*. Forward chaining starts with the axioms and works forward towards the desired judgement, whereas backward chaining starts with the desired judgement and works backwards towards the axioms.

More precisely, forward chaining search maintains a set of derivable judgements, and continually extends this set by adding to it the conclusion of any rule all of whose premises are in that set. Initially, the set is empty; the process terminates when the desired judgement occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgement, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgement is not derivable. We may go on and on adding more judgements to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgement is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a set of current goals, judgements whose derivations are to be sought. Initially, this set consists solely of the judgement we wish to derive. At each stage, we remove a judgement from the goal set, and consider all rules whose conclusion is that judgement. For each such rule, we add to the goal set the premises of that rule. The process terminates when the goal set is empty, all goals having been achieved. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgement, but there is no algorithmic method for determining in general whether the current goal is derivable. Thus we may futilely add more and more judgements to the goal set, never reaching a point at which all goals have been satisfied.

1.3 Rule Induction

Suppose that the judgement form, J, is inductively defined by a rule set $\mathcal{S}$. Since J is, by definition, the strongest (most restrictive) judgement for which the rules in $\mathcal{S}$ are valid, we may employ the important principle of *rule induction* to derive properties of those objects $\vec{x}$ such that $\vec{x}\ J$ is valid. Specifically, if we wish to show $P\ \vec{x}$, it is enough to show that the property P is *closed under*, or *respects*, the rules in $\mathcal{S}$. More precisely, for every rule in $\mathcal{S}$ of the form

$$\frac{\vec{x}_1\ J \quad \cdots \quad \vec{x}_k\ J}{\vec{x}\ J}$$

we must show that $P\ \vec{x}$ is valid, under the assumptions that $P\ \vec{x}_1, \ldots, P\ \vec{x}_k$ are all valid. These assumptions are called the *inductive assumptions* of the inference, and the conclusion establishes the *inductive step* corresponding to that rule. Remember that we must consider *all* rules in the definition of J in order to establish the desired conclusion!

Another way to justify the principle of rule induction is by analysis of the possible derivations of $\vec{x}\ J$ according to the rules. This amounts to a case-by-case analysis on the root of the derivation tree. For each rule that could occur at the root, we may inductively assume the result for each of the sub-derivations, and derive from these the result for the whole derivation. This process is sometimes known as *induction on derivations*; it is entirely equivalent to rule induction over the rules defining the judgement form under consideration.

The principle of rule induction associated with the rule set, $\mathcal{N}$, states that to show $P\ x$ whenever x nat, it is enough to show

1. P `zero`.
2. P `succ`(x), assuming $P\ x$.

This is just the familiar principle of *mathematical induction*. Similarly, the principle of rule induction associated with the rule set $\mathcal{T}$ states that to show that x tree implies $P\ x$, it is enough to show

1. P `empty`.

2. P `node`(x_1, x_2), assuming $P\ x_1$ and $P\ x_2$.

This is sometimes called the principle of *tree induction*.

As a simple example, let us show that every natural number is either zero, one, or two more than some other natural number. More precisely, the judgement $P\ x$ in this case states that either $x = \texttt{zero}$, or $x = \texttt{succ}(\texttt{zero})$, or $x = \texttt{succ}(\texttt{succ}(y))$ for some y nat. We prove by rule induction on the rule set $\mathcal{N}$ that if x nat, then $P\ x$, as follows:

1. Show that P `zero`. This is immediate.

2. Assume that $P\ x$, and show that $P\ \texttt{succ}(x)$. We have by the inductive assumption that $x = \texttt{zero}$, or $x = \texttt{succ}(\texttt{zero})$, or $x = \texttt{succ}(\texttt{succ}(y))$ for some y nat. We are to show that either $\texttt{succ}(x) = \texttt{zero}$, or $\texttt{succ}(x) = \texttt{succ}(\texttt{zero})$, or there exists z nat such that $\texttt{succ}(x) = \texttt{succ}(\texttt{succ}(z))$. In the first case, observe that $\texttt{succ}(x) = \texttt{succ}(\texttt{zero})$; in the second, observe that $\texttt{succ}(x) = \texttt{succ}(\texttt{succ}(\texttt{zero}))$, so we may take $z = \texttt{zero}$; in the third, observe that $\texttt{succ}(x) = \texttt{succ}(\texttt{succ}(\texttt{succ}(y)))$, so we may take $z = \texttt{succ}(y)$.

This completes the proof.

1.4 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. For example, the following set of rules, $\mathcal{L}$, defines the judgement form list, which expresses that an object is a list of natural numbers:

$$\frac{}{\mathsf{nil}\ \mathsf{list}} \qquad \frac{x\ \mathsf{nat} \quad y\ \mathsf{list}}{\mathsf{cons}(x,y)\ \mathsf{list}}$$

Notice that the second rule uses the judgement form nat defined earlier.

It is also common to give a *simultaneous* inductive definition of several judgements, $J_1, \ldots, J_n$, by a collection of rules, each of which may use as premises any of the n judgement forms being defined. Thus each rule has the form

$$\frac{\vec{x}_1\ J_{i_1} \quad \cdots \quad \vec{x}_m\ J_{i_m}}{\vec{x}\ J_i}$$

where $1 \leq i_j \leq n$ for each $1 \leq j \leq m$. As before, we may have $m = 0$ premises, in which case the rule is an axiom.

The difference between simultaneous and iterated inductive definitions is simply that in the iterated case we finish one inductive definition and then use it in a subsequent one, whereas in the simultaneous case two or more judgements are being defined in a mutually recursive manner, so that neither can be considered "finished" before the other.

The meaning of a simultaneous inductive definition is a generalization of that for a single inductive definition. It defines the strongest judgements $J_1, \ldots, J_n$ closed under the rules. This gives rise to the principle of rule induction for simultaneous inductive definitions, which permits us to prove properties about such inductively defined families of judgements. If we wish to show $P_1\ \vec{x}_1, \ldots, P_n\ \vec{x}_n$ whenever $\vec{x}_1\ J_1, \ldots, \vec{x}_n\ J_n$, it is enough to show for each rule

$$\frac{\vec{x}_1\ J_{i_1} \quad \cdots \quad \vec{x}_m\ J_{i_m}}{\vec{x}\ J_i},$$

that if $\vec{x}_1\ P_{i_1}, \ldots, \vec{x}_m\ P_{i_m}$, then $\vec{x}\ P_i$.

For example, consider the following rule set, which constitutes a simultaneous inductive definition of the judgement forms x even, stating that x is an even natural number, and x odd, stating that x is an odd natural number:

$$\frac{}{\texttt{zero}\ \mathsf{even}} \qquad \frac{x\ \mathsf{odd}}{\texttt{succ}(x)\ \mathsf{even}} \qquad \frac{x\ \mathsf{even}}{\texttt{succ}(x)\ \mathsf{odd}}$$

The associated principle of rule induction states that if we wish to show $P\ x$, whenever x even, and $Q\ x$, whenever x odd, it is enough to show

1. P zero;

2. if $Q\ x$, then P $\texttt{succ}(x)$;

3. if $P\ x$, then Q $\texttt{succ}(x)$.

These proof obligations are derived by considering the rules defining the even and odd judgement forms.

1.5 Admissible and Derivable Rules

Let $\mathcal{S}$ be an inductive definition of the judgement J. There are two senses in which a rule

$$\frac{\vec{x}_1\ J \quad \cdots \quad \vec{x}_k\ J}{\vec{x}\ J}$$

may be thought of as being "valid" for $\mathcal{S}$: it can be either *derivable* or *admissible*.

A rule is said to be *derivable* iff there is a derivation of its conclusion from its premises. This means that there is a composition of rules starting with the premises and ending with the conclusion. For example, the following rule is derivable in $\mathcal{N}$:

$$\frac{x\ \mathsf{nat}}{\mathtt{succ}(\mathtt{succ}(\mathtt{succ}(x)))\ \mathsf{nat}}$$

Its derivation is as follows:

$$\cfrac{\cfrac{\cfrac{x\ \mathsf{nat}}{\mathtt{succ}(x)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(\mathtt{succ}(x)))\ \mathsf{nat}}$$

A rule is said to be *admissible* iff its conclusion is derivable from no premises whenever its premises are derivable from no premises. For example, the following rule is *admissible* in $\mathcal{N}$:

$$\frac{\mathtt{succ}(x)\ \mathsf{nat}}{x\ \mathsf{nat}}$$

First, note that this rule is *not* derivable for any choice of x. For if x is `zero`, then the only rule that applies has no premises, and if x is $\mathtt{succ}(y)$ for some y, then the only rule that applies has as premise $y\ \mathsf{nat}$, rather than $x\ \mathsf{nat}$. However, this rule *is* admissible! We may prove this by induction on the derivation of the premise of the rule. For if $\mathtt{succ}(x)\ \mathsf{nat}$ is derivable from no premises, it can only be by second rule, which means that $x\ \mathsf{nat}$ is also derivable, as required. (This example shows that not every admissible rule is derivable.)

If a rule is derivable in a rule set $\mathcal{S}$, then it remains derivable in any rule set $\mathcal{S}' \supseteq \mathcal{S}$. This is because the derivation of that rule depends only on what rules are available, and is not sensitive to whether any other rules are also available. In contrast a rule can be admissible in $\mathcal{S}$, but inadmissible in some extension $\mathcal{S}' \supseteq \mathcal{S}$! For example, suppose that we add to $\mathcal{N}$ the rule

$$\frac{}{\texttt{succ}(\texttt{junk})\ \mathsf{nat}}$$

Now it is no longer the case that the rule

$$\frac{\texttt{succ}(x)\ \mathsf{nat}}{x\ \mathsf{nat}}$$

is admissible, for if the premise were derived using the additional rule, there is no derivation of `junk` nat, as would be required for this rule to be admissible.

Since admissibility is sensitive to which rules are *absent*, as well as to which are *present*, a proof of admissibility of a non-derivable rule must, at bottom, involve a use of rule induction. A proof by rule induction contains a case for each rule in the given set, and so it is immediately obvious that the argument is not stable under an expansion of this set with an additional rule. The proof must be reconsidered, taking account of the additional rule, and there is no guarantee that the proof can be extended to cover the new case (as the preceding example illustrates).

1.6 Defining Functions by Rules

A common use of inductive definitions is to define inductively its graph, a judgement, which we then prove is a function. For example, one way to define the addition function on natural numbers is to define inductively the judgement $A\ (m, n, p)$, with the intended meaning that p is the sum of m and n, as follows:

$$\frac{m\ \mathsf{nat}}{A\ (m, \texttt{zero}, m)} \quad \frac{A\ (m, n, p)}{A\ (m, \texttt{succ}(n), \texttt{succ}(p))}$$

We then must show that p is uniquely determined as a function of m and n. That is, we show that if m nat and n nat, then there exists a unique p such that $A\ (m, n, p)$ by rule induction on the rules defining the natural numbers.

1. From m nat and zero nat, show that there exists a unique p such that $A\ (m, n, p)$. Taking p to be m, it is easy to see that $A\ (m, n, p)$.

2. From m nat and succ(n) nat and the assumption that there exists a unique p such that $A\ (m, n, p)$, we are to show that there exists a unique q such that $A\ (m, \mathtt{succ}(n), q)$. Taking $q = \mathtt{succ}(p)$ does the job.

This fact may be summarized by saying that the *mode* of this judgement is $(\forall, \forall, \exists!)$, which means that the first two arguments may be thought of as *inputs*, and the third as a uniquely determined *output*. (It is implicit that the inputs and outputs are objects x such that x nat.) This mode specification states that $A\ (m, n, p)$ determines a *total function* in which each pair m nat and n nat determines a unique p nat such that $A\ (m, n, p)$. Other modes for this judgement are $(\forall, \forall, \exists)$, which merely asserts that to every pair of inputs there is a (not necessarily unique) output, and $(\forall, \forall, \exists^{\leq 1})$, which asserts that to every pair of inputs there is at most one output. The former states that the judgement is a *total relation*, the latter that it is a *partial function*. Of course the property of being a total function is stronger than either of these, but situations will arise in which only the weaker modes are available.

As another example, the following rules define the height of a binary tree, making use of an auxiliary "maximum" function on natural numbers that you may readily define yourself:

$$\frac{}{H\ (\mathtt{empty}, \mathtt{zero})} \qquad \frac{H\ (t_1, n_1) \quad H\ (t_2, n_2) \quad M\ (n_1, n_2, n)}{H\ (\mathtt{node}(t_1, t_2), \mathtt{succ}(n))}$$

One may readily show by tree induction that the mode of this judgement is $(\forall, \exists)$ over inputs and outputs x such that x tree.

Whenever we are defining a judgement that is intended to be a function (*i.e.*, one argument is determined as a function of the others), we often write the definition using equations. For example, we may re-state the inductive definition of addition above using equations as follows:

$$\frac{m\ \mathsf{nat}}{m + \mathtt{zero} = m\ \mathsf{nat}} \qquad \frac{m + n = p\ \mathsf{nat}}{m + \mathtt{succ}(n) = \mathtt{succ}(p)\ \mathsf{nat}}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgement is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse the notation by using the relation as function, writing just $m + n$ for the unique p such that $m + n = p$ `nat`.

1.7 Foundations

So far we have been vague about what sorts of "objects" may be the subjects of judgements. For example, the inductive definition of binary trees makes use of objects `empty` and `node`(x, y), where x and y are themselves objects, without saying precisely just what are these objects. More generally, we may ask, what sort of objects may we make judgements about? This is a delicate matter of foundations that we will only touch on briefly here.

One point of view is to simply take as given that the constructions we have mentioned so far are intuitively acceptable, and require no further justification or explanation. Roughly speaking, we admit as sensible any form of "finitary" construction in which finite entities are built up from other such finite entities by a finitely computable processes. Obviously this leaves quite a lot of room for interpretation, but in practice we never get into serious trouble and hence may safely adopt this rough-and-ready rule as a guide in the sequel.

If we're really worried about nailing down what sorts of objects are admissible, then we must work within some presumed well-understood framework, such as set theory, in which to carry out our work.[2] While this leads to an account that may be considered mathematically satisfactory, it ignores the very real question of whether and how our constructions can be justified on computational grounds. After all, the study of programming languages is all about things we can implement on a machine! Conventional set theory makes it difficult to discuss such matters, since it provides no computational interpretation of sets.

A more reasonable choice for our purposes is to work within the universe of *hereditarily finite sets*, which are finite sets whose elements are finite sets, whose elements are finite sets, and so on. Any construction that can be carried out in this universe may be taken as computationally and

[2]In effect we relegate all foundational questions to questions about the existence of appropriate sets.

foundationally meaningful. A more concrete, but technically awkward, approach is to admit only the natural numbers as objects — any other object of interest must be encoded as a natural number using the technique of *Gödel numbering*, which establishes a bijection between a set X of finitary objects and the set $\mathbb{N}$ of natural numbers.[3]

A natural universe of objects for programming purposes is provided by *well-founded, finitely branching trees*, or *algebraic terms*, which we will introduce in Chapter 5. These are quite convenient to use as a representation for a wide array of commonly occurring objects. Indeed, trees are easily represented in ML using data types and pattern matching.

1.8 Exercises

1. Let J be inductively defined by a rule set $\mathcal{S}$. Give an inductive definition of the judgement "$\mathcal{D}$ is a derivation of $\vec{x}\ J$" relative to the rule set $\mathcal{S}$.
2. Give an inductive definition of the forward-chaining and backward-chaining search strategies.
3. Introduce and discuss the internal and external consequence relations as two forms of hypothetical judgement ....

[3]One may then ask where the natural numbers come from. The answer is that they are taken as a primitive notion, rather than as being inductively defined. One has to start somewhere.

Chapter 2

Higher Order Judgement Forms

In Chapter 1 we introduced the concept of an inductively defined judgement expressing a relationship among a collection of objects. Such judgements are sometimes called *categorical* because they are unconditional assertions, such as the assertion that an object x is a binary tree. In this chapter we extend the framework of inductive definitions to permit two additional forms of judgement, the *hypothetical* and the *general*, which we combine to form the *hypothetico-general* judgement. These *higher-order judgements* play an important role in the theory of programming languages.

Note to the reader: Extending the framework of inductive definitions with hypothetical judgements requires no additional machinery beyond what was introduced in Chapter 1. However, the notion of a general judgement relies on material that we shall only present in Chapter 6. This chapter may be safely skipped, or lightly skimmed, on first reading; the concepts introduced here will not be needed until Chapter 9.

2.1 Hypothetical Judgements

The *hypothetical judgement* has the form

$$J_1, \ldots, J_n \vdash J,$$

where each J_i (for $1 \leq i \leq n$) and J are inductively defined categorical judgements. Informally, the hypothetical judgement expresses that J holds under the assumption that $J_1, \ldots, J_n$ all hold. The judgements $J_1, \ldots, J_n$ are called the *hypotheses* of the judgement, and J is called its *conclusion*. (The punctuation mark separating them is called a *turnstile*.) The hypothetical judgement form is also called an *entailment relation*, or a *consequence relation*.

For example, the hypothetical judgement

$$x \ \mathsf{nat} \vdash \mathtt{succ}(\mathtt{succ}(x)) \ \mathsf{nat}$$

expresses the conditional judgement that, for any object x, if x is a natural number, then so is its double successor. Intuitively, this is, of course, a valid judgement. Let us now make this intuition precise.

Recall from Chapter 1 that the evidence for an inductively defined categorical judgement is a derivation consisting of a composition of inference rules starting with axioms and ending that judgement. Evidence for a hypothetical judgement is defined similarly, except that the derivation of the conclusion may start with any or all of the hypotheses and end with the conclusion of the judgement. Put in other terms, evidence for a hypothetical judgement of the above form consists of evidence for J with respect to the extension of the rules defining the judgement form of J with $J_1, \ldots, J_n$ as new axioms (rules without premise).

The crucial point is that the evidence for a hypothetical judgement consists of a *uniform* way to transform presumed evidence for the hypotheses into evidence for the conclusion, without regard to what that presumed evidence may actually be. The derivation may be specialized by "plugging in" whatever evidence for $J_1, \ldots, J_n$ may arise, resulting in a derivation of J that does not make use of these new axioms. Of course, the evidence for the J_i's may itself be hypothetical, say in hypotheses $J'_1, \ldots, J'_m$, resulting in evidence for $J'_1, \ldots, J'_m \vdash J$.

This interpretation leads to the following *structural rules* for the hypothetical judgement. Let Γ stand for any sequence $J_1, \ldots, J_n$ of judgements under consideration. The structural rules governing the hypothetical judgement are as follows:

Reflexivity Every judgement is a consequence of itself: $J \vdash J$.

Weakening If $\Gamma \vdash J$, then $\Gamma, J' \vdash J$ for any judgement J'.

Permutation If $\Gamma, J_1, J_2, \Gamma' \vdash J$, then $\Gamma, J_2, J_1, \Gamma' \vdash J$.

Contraction If $\Gamma, J, J \vdash J'$, then $\Gamma, J \vdash J'$.

Transitivity If $\Gamma, J' \vdash J$ and $\Gamma' \vdash J'$, then $\Gamma, \Gamma' \vdash J$.

These properties follow directly from the meaning of the hypothetical judgement:

Reflexivity The additional "axiom" J counts as a derivation of J.

Weakening The derivation of J may make use of presumed derivations for the hypotheses, but need not do so.

Permutation The order of hypotheses does not matter in the given interpretation.

Contraction Since we may use the same hypothesis more than once, it does not matter if we repeat it.

Transitivity If we plug in actual evidence for a hypothesis, then the evidence for the conclusion may be specialized to use it, leaving as hypotheses those used as evidence for it.

There is a close connection between the notion of a hypothetical judgement and the notion of a derived rule. Specifically, the inference rule

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

is derivable iff the hypothetical judgement

$$J_1, \ldots, J_n \vdash J$$

is valid, for in both cases the meaning is that there exists a derivation of J from $J_1, \ldots, J_n$.

There is another form of hypothetical judgement corresponding to admissible rules, written

$$J_1, \ldots, J_n \models J.$$

This means that J is valid with respect to the *original* set of inference rules whenever $J_1, \ldots, J_n$ are also valid in the *original* set of rules. For example, with respect to the rule set defining natural numbers, given in Chapter 1, we have

$$\mathsf{succ}(x)\ \mathsf{nat} \models x\ \mathsf{nat}.$$

This may be proved by rule induction, for if $\texttt{succ}(x)$ nat, then this can only be by virtue of the rule

$$\frac{x \ \mathsf{nat}}{\texttt{succ}(x) \ \mathsf{nat}},$$

and hence the desired conclusion must hold (it is the premise of this inference). This is precisely the same as saying that the rule

$$\frac{\texttt{succ}(x) \ \mathsf{nat}}{x \ \mathsf{nat}}$$

is admissible.

Note that $\texttt{succ}(x) \ \mathsf{nat} \nvdash x \ \mathsf{nat}$ — there is no composition of rules that starts with the hypothesis and leads to the conclusion. Thus, the two forms of hypothetical judgement do not coincide, even though they satisfy the same structural properties. Here is a brief summary of why they are true for the second form:

Reflexivity If J is derivable from the original rules, then J is derivable from the original rules.

Weakening If J is derivable from the original rules assuming that $J_1, \ldots, J_n$ are, then so it must also be derivable from an additional assumption.

Permutation Obviously the order of our assumptions does not matter.

Contraction Assuming the same thing twice is the same as assuming it once.

Substitution The assumption of J' used, in addition to Γ, to derive J may be discharged by simply using the derivation of J' from Γ'. This means that J is valid with respect to the original rule set, under the assumptions Γ and Γ'.

When discussing the two forms of hypothetical judgement, we refer to the former as the *internal form*, and the latter as the *external form*. It should be immediately clear that the internal form is stronger than the external form (if $\Gamma \vdash J$, then $\Gamma \models J$), but, as we have just seen, the converse does not hold. This is just a re-statement of the observation that every derivable rule is admissible, but the converse does not, in general, hold. Both forms of hypothetical judgement arise in the study of programming languages, but the internal form is far and away the more important — it arises in the definition of nearly every language we shall study.

2.2 General Judgements

The *general judgement* expresses a "parameterized assertion," one that involves variables ranging over objects in the universe of discourse. The general judgement has the form

$$|_{x_1,\ldots,x_n}\ J,$$

where $x_1, \ldots, x_n$ are variables that may occur in J. Informally, the general judgement means that every instance of J obtained by choosing objects for the variables is valid.

But what is a variable? And what is an instance? For these concepts to make sense, we restrict attention to inductively defined judgements over the universe of abstract binding trees (abt's), which are introduced in Chapter 6. The reason for this restriction is that abt's provide a notion of variable and substitution, which we now use to explain the meaning of the general judgement.

The general judgement

$$|_{x_1,\ldots,x_n}\ J$$

is valid iff

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n]J$$

is valid for *every* choice of abt's a_i. Evidence for the validity of the general judgement $|_{x_1,\ldots,x_n}\ J$ consists of a *derivation scheme*, $\mathcal{D}$, a derivation with parameters, such that

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n]\mathcal{D}$$

is a derivation of

$$[x_1, \ldots, x_n \leftarrow a_1, \ldots, a_n]J.$$

Like the hypothetical judgement, the general judgement also obeys a collection of structural rules. For the sake of concision in stating these rules, let Δ range over finite sets of variables, $x_1, \ldots, x_n$.

Weakening If $|_{\Delta}\ J$, then $|_{\Delta,x}\ J$.

Permutation If $|_{\Delta,x_1,x_2,\Delta'}\ J$, then $|_{\Delta,x_2,x_1,\Delta'}\ J$.

Contraction If $|_{\Delta,x,x}\ J$, then $|_{\Delta,x}\ J$.

Instantiation If $|_{\Delta,x}\ J$, then $|_{\Delta}\ [x{\leftarrow}a]J$, provided that the free variables of a are among those in Δ.

Note the strong similarity to the structural rules governing the hypothetical judgement.

2.3 Hypothetico-General Judgements

The general judgement is most often used in conjunction with the hypothetical judgement. For example, the following general, hypothetical judgement is valid:

$$|_x\ (x\ \mathsf{nat} \vdash \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}).$$

The parentheses are written here for emphasis, but we usually omit them by treating the hypothetical judgement as binding more tightly than the general. The hypothesis x nat has the effect of constraining the parameter x to range over natural numbers, excluding "garbage" not of interest for the inference.

Note that this is a *single* judgement form expressing a *family* of hypothetical judgements

$$a\ \mathsf{nat} \vdash \mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat}.$$

In this way the general judgement permits us to capture the informal idea of a *rule scheme* as a single concept.

Since the general judgement occurs so often in conjunction with the hypothetical, we often combine the notation, writing $\Gamma \vdash_{\Delta} J$ for $|_{\Delta}\ \Gamma \vdash J$. This combined form is called a *hypothetico-general judgement* for obvious reasons. For example, the hypothetico-general judgement

$$x\ \mathsf{nat} \vdash_x \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}$$

is a short-hand for the general, hypothetical judgement given above.

Somewhat confusingly, the subscript on the turnstile is often omitted, writing just $\Gamma \vdash J$ for $\Gamma \vdash_{\Delta} J$, where Δ is the set of free variables occurring in Γ and J. For example, we may write just

$$x\ \mathsf{nat} \vdash \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}$$

for the preceding hypothetico-general judgement, it being understood that x is a parameter of the hypothetical judgement.

2.4 Inductive Definitions, Revisited

Hypothetical and general judgements permit a particularly elegant form of inductive definition that may be illustrated by the following example. The idea is to give an inductive definition of a function to compute the "depth" of a closed abt over some (unspecified) signature. Roughly speaking, the depth of a closed abt is defined to be the maximum nesting depth of abstractors within it. We will give a simultaneous inductive definition of two judgements, $d(a\ \mathsf{abt}) = n$, specifying that the abt a has depth n, and $d(\beta\ \mathsf{abs}) = n$, specifying that the abstractor β has depth n. Because abstractors bind variables, it is impossible to define the depth only for closed abt's, but instead we must consider open ones as well. The main "trick" is to use a hypothetico-general judgement in the rules to handle the variables that are introduced during the recursion. Here are the rules:

$$\frac{d(\beta_1\ \mathsf{abs}) = n_1 \quad \cdots \quad d(\beta_k\ \mathsf{abs}) = n_k}{d(o(\beta_1, \ldots, \beta_k)\ \mathsf{abt}) = \max(n_1, \cdots, n_k)}$$

$$\frac{d(a\ \mathsf{abt}) = n}{d(a\ \mathsf{abs}) = n} \qquad \frac{d(x\ \mathsf{abt}) = 0 \vdash_x d(\beta\ \mathsf{abs}) = n}{d(x\,.\,\beta\ \mathsf{abs}) = n + 1}$$

Observe that the premise of the third rule is a hypothetico-general judgement, which expresses the idea that the depth of an abstractor is one more than the depth of its body, *assuming* that the bound variable has depth 0. This assumption is used to provide a definition for the depth of a variable whenever they are encountered, using the reflexivity property of the hypothetical judgement.

This example illustrates an important convention, called the *freshness convention*, that we shall use tacitly throughout the book. Whenever a rule, such as the last one above, introduces a name using a hypothetico-general judgement, *it is assumed that this name is chosen so as not to otherwise occur in the parameter set*. This requirement may always be met by suitably renaming the bound variable of the abstractor before applying the rule. This is one important benefit of always working "up to α-conversion," which frees us from having to worry about the complications of name reuse within a given scope.

The use of (the internal form of) a hypothetical judgement in the premise of an inference rule goes beyond what we considered in Chapter 1, wherein only categorical judgements were permitted. This is a significant extension, and some justification is required to ensure that what we're doing is sensible. The key is to regard the rules as a simultaneous inductive definition of an *infinite* family of judgements of the form

$$d(x_1 \,\mathsf{abt}) = 0, \ldots, d(x_k \,\mathsf{abt}) = 0 \vdash_{x_1,\ldots,x_k} d(a \,\mathsf{abt}) = n$$

and

$$d(x_1 \,\mathsf{abt}) = 0, \ldots, d(x_k \,\mathsf{abt}) = 0 \vdash_{x_1,\ldots,x_k} d(\beta \,\mathsf{abs}) = n.$$

The idea is that for each choice of parameters set $x_1, \ldots, x_k$ and each set of corresponding assumptions $d(x_1 \,\mathsf{abt}) = 0$, ..., $d(x_k \,\mathsf{abt}) = 0$, we have two judgements defining the depth of an abt or abstractor with free variables among those parameters. The rule for abstractors augments the parameter and hypothesis sets, and thereby refers to another member of the same family of judgements in its definition.

To clarify the situation, let us re-write the inductive definition of the depth judgements in a form that makes the parameter and hypothesis sets explicit. Let Δ range over finite sets of variables $x_1, \ldots, x_k$, and, for each choice of parameter set Δ, let Γ range over sets of assumptions of the form $d(x_1 \,\mathsf{abt}) = 0, \ldots, d(x_k \,\mathsf{abt}) = 0$. Here are the same rules, written in a more explicit form:

$$\frac{\Gamma \vdash_\Delta d(\beta_1 \,\mathsf{abs}) = n_1 \quad \cdots \quad d(\beta_k \,\mathsf{abs}) = n_k}{\Gamma \vdash_\Delta d(o(\beta_1, \ldots, \beta_k) \,\mathsf{abt}) = \max(n_1, \cdots, n_k)}$$

$$\frac{\Gamma \vdash_\Delta d(a \,\mathsf{abt}) = n}{\Gamma \vdash_\Delta d(a \,\mathsf{abs}) = n} \qquad \frac{\Gamma, d(x \,\mathsf{abt}) = 0 \vdash_{\Delta,x} d(\beta \,\mathsf{abs}) = n}{\Gamma \vdash_\Delta d(x \,.\, \beta \,\mathsf{abs}) = n + 1}$$

This presentation makes clear the augmentation of the current parameter and hypothesis set in the inference. When we do so, we "switch" to another judgement in the same family, with the extended parameter and hypothesis sets.

What is the principle of rule induction associated with rules whose premises involve hypothetical and general judgements? An example will illustrate the general case. Suppose we wish to prove that the mode of the depth judgements is $(\forall, \exists!)$, expressing that they really do define functions.

Theorem 2.1

1. *If $d(x_1 \,\mathsf{abt}) = 0, \ldots, d(x_k \,\mathsf{abt}) = 0 \vdash_{x_1,\ldots,x_k} d(a \,\mathsf{abt}) = n$, then for every family $a_1, \ldots, a_k$ of abt's such that $d(a_1 \,\mathsf{abt}) = 0, \ldots, d(a_k \,\mathsf{abt}) = 0$, there exists a unique n such that $d([x_1, \ldots, x_k {\leftarrow} a_1, \ldots, a_k]a \,\mathsf{abt}) = n$.*

2. *If $d(x_1 \,\mathsf{abt}) = 0, \ldots, d(x_k \,\mathsf{abt}) = 0 \vdash_{x_1,\ldots,x_k} d(\beta \,\mathsf{abs}) = n$, then for every family $a_1, \ldots, a_k$ of abt's such that $d(a_1 \,\mathsf{abt}) = 0, \ldots, d(a_k \,\mathsf{abt}) = 0$, there exists a unique n such that $d([x_1, \ldots, x_k {\leftarrow} a_1, \ldots, a_k]\beta \,\mathsf{abt}) = n$.*

Proof: We may prove these facts simultaneously by rule induction. The most interesting case, of course, is the last. The inductive hypothesis states that if $d(b \,\mathsf{abt}) = 0$, then there exists a unique n such that

$$d([x_1, \ldots, x_k, x {\leftarrow} a_1, \ldots, a_k, b]a \,\mathsf{abt}) = n.$$

We are to show that there exists a unique n' such that

$$d(x \,.\, [x_1, \ldots, x_k {\leftarrow} a_1, \ldots, a_k]b \,\mathsf{abs}) = n'.$$

The result follows immediately by taking, in the inductive hypothesis, $b = x$, and taking, in the conclusion, $n' = n + 1$. ■

Finally, let us note that it is *not permissible* to use the external form of hypothetical judgement in an inference rule! The justification we have given for using the internal form in premises does not extend to the external form, precisely because its meaning is not defined by extending the rule set with new axioms. Indeed, admitting the external form in the premise of a rule destroys the mathematical underpinnings of the theory of inductive definitions, invalidating their use as a definitional tool.

2.5 Exercises

1. Investigate why the premise of an inference rule may *not* be taken to be the *external* form of hypothetical judgement. Show that there exists an "inductive definition" using the external form of hypothetical for which there is no strongest judgement closed under the given rules.

Chapter 3

Transition Systems

Transition systems are used to describe the execution behavior of programs by defining an abstract computing device with a set, S, of *states* that are related by a *transition judgement*, $\longmapsto$. The transition judgement describes how the state of the machine evolves during execution.

3.1 Transition Systems

A *transition system* is specified by the following judgements:

1. s state, asserting that s is a *state* of the transition system.

2. s final, asserting that s is a *final* state.

3. s init, asserting that s is an *initial* state.

4. $s \longmapsto s'$, where s state and s' state, asserting that state s may transition to state s'.

We require that if s final, then for no s' do we have $s \longmapsto s'$. In general, a state s for which there is no $s' \in S$ such that $s \longmapsto s'$ is said to be *stuck*. All final states are stuck, but not all stuck states need be final!

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that s_0 init, and $s_i \longmapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff $s_n \not\longmapsto$; it is *complete* iff it is maximal and, in addition, s_n final. Thus every complete transition sequence is maximal, but maximal sequences are not necessarily complete.

A transition system is *deterministic* iff for every state s there exists at most one state s' such that $s \longmapsto s'$. Most of the transition systems we will consider in this book are deterministic, the notable exceptions being those used to model concurrency.

The judgement $s \overset{*}{\longmapsto} s'$ is the *reflexive, transitive closure* of the transition judgement. It is inductively defined by the following rules:

$$\frac{}{s \overset{*}{\longmapsto} s} \qquad \frac{s \longmapsto s' \quad s' \overset{*}{\longmapsto} s''}{s \overset{*}{\longmapsto} s''}$$

It is easy to prove by rule induction that $\overset{*}{\longmapsto}$ is indeed reflexive and transitive.

Since the multistep transition is inductively defined, we may prove that $P(s, s')$ holds whenever $s \longmapsto^* s'$ by showing

1. $P(s, s)$.
2. if $s \longmapsto s'$ and $P(s', s'')$, then $P(s, s'')$.

The first requirement is to show that P is reflexive. The second is often described as showing that P is *closed under head expansion*, or *closed under reverse evaluation*.

The *n-times iterated* transition judgement, $s \overset{n}{\longmapsto} s'$, where n nat, is inductively defined by the following rules:

$$\frac{}{s \overset{0}{\longmapsto} s} \qquad \frac{s \longmapsto s' \quad s' \overset{n}{\longmapsto} s''}{s \overset{n+1}{\longmapsto} s''}$$

It is easy to show that $s \overset{*}{\longmapsto} s'$ iff $s \overset{n}{\longmapsto} s'$ for some n nat.

Finally, the *complete* transition judgement, $s \overset{!}{\longmapsto} s'$ is the restriction to $s \overset{*}{\longmapsto} s'$ so that s' final. That is, $s \overset{!}{\longmapsto} s'$ iff $s \overset{*}{\longmapsto} s'$ and s' final.

3.2 Exercises

1. Prove that $s \overset{*}{\longmapsto} s'$ iff there exists n nat such that $s \overset{n}{\longmapsto} s'$.

Part II

Levels of Syntax

Chapter 4

Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings, linear sequences of characters (or symbols) that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods (grounded in the theory of formal languages) for eliminating ambiguity, improving readability is, of course, a matter of taste about which reasonable people may disagree. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. Techniques for enhancing readability include the use of suggestive key words and phrases, and establishment of punctuation and layout conventions.

4.1 Strings

To begin with we must define what we mean by characters and strings. An *alphabet*, Σ, is a set of *characters*, or *symbols*. Often Σ is taken implicitly to be the set of ASCII or UniCode characters, but we shall need to make use of other character sets as well. The judgement form char is inductively defined by the following rules (one per choice of $c \in \Sigma$):

$$\frac{(c \in \Sigma)}{c\ \mathsf{char}}$$

The judgment form string_Σ states that s is a string of characters from Σ.

It is inductively defined by the following rules:

$$\frac{}{\varepsilon\ \mathsf{string}_\Sigma} \qquad \frac{c\ \mathsf{char} \quad s\ \mathsf{string}_\Sigma}{c \cdot s\ \mathsf{string}_\Sigma}$$

In most cases we omit explicit mention of the alphabet, Σ, and just write $s\ \mathsf{string}$ to indicate that s is a string over an implied choice of alphabet.

In practice strings are written in the usual manner, *abcd* instead of the more proper $a \cdot (b \cdot (c \cdot (d \cdot \varepsilon)))$. The function $s_1\,\hat{}\,s_2$ stands for string concatenation; it may be defined by induction on s_1. We usually just juxtapose two strings to indicate their concatentation, writing $s_1\,s_2$, rather than $s_1\,\hat{}\,s_2$.

4.2 Context-Free Grammars

The standard method for defining concrete syntax is by giving a *context-free grammar (CFG)* for the language. A grammar consists of three things:

1. An alphabet Σ of *terminals*.
2. A finite set $\mathcal{N}$ of *non-terminals* that stand for the syntactic categories.
3. A set $\mathcal{P}$ of *productions* of the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Whenever there is a set of productions

$$\begin{array}{rcl} A & ::= & \alpha_1 \\ & \vdots & \\ A & ::= & \alpha_n. \end{array}$$

all with the same left-hand side, we often abbreviate it as follows:

$$A ::= \alpha_1 \mid \cdots \mid \alpha_n.$$

A context-free grammar is essentially a simultaneous inductive definition of its syntactic categories. Specifically, we may associate a rule set R with a grammar according to the following procedure. First, we treat each non-terminal as a label of its syntactic category. Second, for each production

$$A ::= s_1\,A_1\,s_2\,\ldots\,s_n\,A_n\,s_{n+1}$$

of the grammar, where $A_1, \ldots, A_n$ are all of the non-terminals on the right-hand side of that production, and $s_1, \ldots, s_{n+1}$ are strings of terminals, add a rule

$$\frac{s'_1\ A_1 \quad \ldots \quad s'_n\ A_n}{s_1\, s'_1\, s_2 \ldots s_n\, s'_n\, s_{n+1}\ A}$$

to the rule set R. For each non-terminal A, we say that *s is a string of syntactic category A* iff $s\ A$ is derivable according to the rule set R so obtained.

An example will make these ideas clear. Let us give a grammar defining the syntax of a simple language of arithmetic expressions.

$$\begin{array}{lcll} \textit{Digits} & D & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ \textit{Numbers} & N & ::= & D \mid N\, D \\ \textit{Expressions} & E & ::= & N \mid E\texttt{+}E \mid E\texttt{*}E \end{array}$$

Here is this grammar presented as a simultaneous inductive definition:

$$\frac{}{0\ \mathsf{dig}} \quad \cdots \quad \frac{}{9\ \mathsf{dig}} \tag{4.1}$$

$$\frac{s\ \mathsf{dig}}{s\ \mathsf{num}} \quad \frac{s_1\ \mathsf{num} \quad s_2\ \mathsf{dig}}{s_1\, s_2\ \mathsf{num}} \tag{4.2}$$

$$\frac{s\ \mathsf{num}}{s\ \mathsf{exp}} \tag{4.3}$$

$$\frac{s_1\ \mathsf{exp} \qquad s_2\ \mathsf{exp}}{s_1\texttt{+}s_2\ \mathsf{exp}} \tag{4.4}$$

$$\frac{s_1\ \mathsf{exp} \qquad s_2\ \mathsf{exp}}{s_1\texttt{*}s_2\ \mathsf{exp}} \tag{4.5}$$

Each syntactic category of the grammar determines a judgement form. For example, the category of expressions corresponds to the judgement form exp, and so forth.

4.3 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to eliminate ambiguity. The grammar of arithmetic expressions given above is ambiguous in the sense that some strings may be thought of as arising in several different ways. For example, the string 1+2∗3 may be thought of as arising by applying the rule for multiplication first, then the rule for addition, or *vice versa*. The former interpretation corresponds to the expression (1+2)∗3; the latter corresponds to the expression 1+(2∗3).

The trouble is that we cannot tell from the generated string which reading is intended. This causes numerous problems, an example of which arises from an attempt to define the value, a natural number, of an arithmetic expression, e, represented as a string.

We will give an inductive definition of the following three forms of judgement:

$s\ \mathsf{exp} \Downarrow k\ \mathsf{nat}$	*expression s has value k*
$s\ \mathsf{num} \Downarrow k\ \mathsf{nat}$	*numeral s has value k*
$s\ \mathsf{dig} \Downarrow k\ \mathsf{nat}$	*digit s has value k*

The rules defining these judgements are as follows:

$$\frac{}{0\ \mathsf{dig} \Downarrow \mathtt{zero}\ \mathsf{nat}} \quad \frac{}{1\ \mathsf{dig} \Downarrow \mathtt{succ(zero)}\ \mathsf{nat}} \quad \cdots \tag{4.6}$$

$$\frac{s\ \mathsf{dig} \Downarrow k\ \mathsf{nat}}{s\ \mathsf{num} \Downarrow k\ \mathsf{nat}} \qquad \frac{s_1\ \mathsf{num} \Downarrow k_1\ \mathsf{nat} \quad s_2\ \mathsf{dig} \Downarrow k_2\ \mathsf{nat} \quad k = 10 \times k_1 + k_2}{s_1\, s_2\ \mathsf{num} \Downarrow k\ \mathsf{nat}} \tag{4.7}$$

$$\frac{s\ \mathsf{num} \Downarrow k\ \mathsf{nat}}{s\ \mathsf{exp} \Downarrow k\ \mathsf{nat}}$$

$$\frac{s_1\ \mathsf{exp} \Downarrow k_1\ \mathsf{nat} \quad s_2\ \mathsf{exp} \Downarrow k_2\ \mathsf{nat} \quad k = k_1 + k_2}{s_1\text{+}s_2\ \mathsf{exp} \Downarrow k\ \mathsf{nat}} \tag{4.8}$$

$$\frac{s_1\ \mathsf{exp} \Downarrow k_1\ \mathsf{nat} \quad s_2\ \mathsf{exp} \Downarrow k_2\ \mathsf{nat} \quad k = k_1 \times k_2}{s_1{*}s_2\ \mathsf{exp} \Downarrow k\ \mathsf{nat}}$$

(We have taken the liberty of assuming that rules for computing with natural numbers have already been defined.)

Given the intended interpretation of these judgements, it is natural to consider whether they have the mode $(\forall, \exists!)$, over the domain of expressions/numbers/digits (strings formed according to the grammar given earlier) as input and natural numbers as output. The all-important question is whether this is a valid mode for these judgements — do they determine a partial function from input to output, as might be expected?

Perhaps surprisingly, the answer is no! Informally, the reason is that a string such as 1+2∗3 arises in two different ways, using either the rule for addition expressions, thereby reading it as 1+(2∗3), or the rule for multiplication, thereby reading it as (1+2)∗3. Since these have different values, there does not exist a unique value for every string of the appropriate grammatical class.

It is instructive to see how an attempt to prove that the evaluation judgements have the specified mode breaks down. First, let us be precise about what we need to prove. We must show that for every s, if $s\ \mathsf{exp}$, there is a unique k such that $k\ \mathsf{nat}$ and $s\ \mathsf{exp} \Downarrow k\ \mathsf{nat}$, and similarly for the other two judgement forms. It is natural to proceed by rule induction on the rules defining the judgement $s\ \mathsf{exp}$. We consider each rule in turn. The crucial cases are when $s = s_1\text{+}s_2$ and when $s = s_1{*}s_2$, and we have by induction that $s_1\ \mathsf{exp} \Downarrow k_1\ \mathsf{nat}$ and $s_2\ \mathsf{exp} \Downarrow k_2\ \mathsf{nat}$ for some uniquely determined k_1 and k_2 such that $k_1\ \mathsf{nat}$ and $k_2\ \mathsf{nat}$. And in that case we may take k as the sum and product, respectively, of k_1 and k_2. Since the sum and product of k_1 and k_2 are uniquely determined, we seem to have completed the proof (the other cases being handled similarly).

But have we? The problem is that a given string s can be *both* of the form $s_1 + s_2$ and $s_1 \times s_2$ at the same time, and we have no way to know which interpretation is intended! The preceding argument, which proceeds by rule induction on the rules defining $s\ \mathsf{exp}$, tells us that the value of s is uniquely determined *if we are given the rule used to form s* — that is, we are told how to interpret it (as a sum or as a product). If we are given s alone, with no information about how it was generated by the grammar, then the result is ambiguous — one string can have many values according to these rules.

4.4 Resolving Ambiguity

What do we do about ambiguity? The most common methods to eliminate this kind of ambiguity are these:

1. Introduce parenthesization into the grammar so that the person writing the expression can choose the intended intepretation.

2. Introduce precedence relationships that resolve ambiguities between distinct operations (*e.g.*, by stipulating that multiplication takes precedence over addition).

3. Introduce associativity conventions that determine how to resolve ambiguities between operators of the same precedence (*e.g.*, by stipulating that addition is right-associative).

Using these techniques, we arrive at the following revised grammar for arithmetic expressions.

Digits	D	$::=$	$0 \mid 1 \mid \cdots \mid 9$
Numbers	N	$::=$	$D \mid N\,D$
Factors	F	$::=$	$N \mid \texttt{(}E\texttt{)}$
Terms	T	$::=$	$F \mid F\texttt{*}T$
Expressions	E	$::=$	$T \mid T\texttt{+}E$

We have made two significant changes. The grammar has been "layered" to express the precedence of multiplication over addition and to express right-associativity of each, and an additional form of expression, parenthesization, has been introduced.

Re-writing this as an inductive definition, we obtain the following rules:

$$\frac{}{0\ \mathsf{dig}} \qquad \cdots \qquad \frac{}{9\ \mathsf{dig}} \tag{4.9}$$

$$\frac{s\ \mathsf{dig}}{s\ \mathsf{num}} \qquad \frac{s_1\ \mathsf{num} \quad s_2\ \mathsf{dig}}{s_1\,s_2\ \mathsf{num}} \tag{4.10}$$

$$\frac{s\ \mathsf{num}}{s\ \mathsf{fct}} \qquad \frac{s\ \mathsf{exp}}{\texttt{(}s\texttt{)}\ \mathsf{fct}} \tag{4.11}$$

$$\frac{s\ \mathsf{fct}}{s\ \mathsf{trm}} \qquad \frac{s_1\ \mathsf{fct} \quad s_2\ \mathsf{trm}}{s_1\texttt{*}s_2\ \mathsf{trm}} \tag{4.12}$$

$$\frac{s\ \mathsf{trm}}{s\ \mathsf{exp}} \qquad \frac{s_1\ \mathsf{trm} \quad s_2\ \mathsf{exp}}{s_1\texttt{+}s_2\ \mathsf{exp}} \tag{4.13}$$

Using these rules, it is then possible to prove that the evaluation judgements have mode $(\forall, \exists!)$ over grammatically correct strings as inputs and natural numbers as outputs. The crucial difference from our first attempt is that each string admits at most one decomposition consistent with the rules of the grammar. For example, the string $s = 1\texttt{+}2\ast3$ may be derived as $s\ \mathsf{exp}$ only by decomposing e as $s_1\texttt{+}s_2$, where $s_1 = 1$ and $s_2 = 2\ast3$; no other decomposition is possible (be sure you understand why). If we consider the input to be any strings at all, not just those that are grammatical according to the specified rules, then the best mode possible is $(\forall, \exists^{\leq 1})$, since an ungrammatical string has no value, but the value is uniquely determined for every grammatical string.

4.5 Exercises

Chapter 5

Abstract Syntax Trees

The concrete syntax of a language consists of a linear presentation of it as a set of strings — sequences of characters that reflect the conventional modes of reading and writing programs. The main job of concrete syntax design is to ensure the convenient readability and writability of the language, based on subjective criteria such as similarity to other languages, ease of editing, and so forth.

But languages are also the subject of analysis, for example to define what it means to evaluate an arithmetic expression we must analyze the structure of expressions. For this purpose the concrete syntax introduces a level of bureaucracy that we would like to avoid. For example, we must ensure that the syntax is presented in unambiguous form, and we are forced to deal with details of presentation that have no effect on its meaning, but rather are conveniences for reading and writing phrases in the language.

To avoid this clutter it is conventional to define an *abstract syntax* of a language that abstracts away from the concrete presentation of a phrase as a string, and instead focuses on the essential structure of the phrase in a manner amenable to analysis. *Parsing* is the process of translating the concrete to the abstract syntax; once parsed, we need never worry about the concrete presentation of a phrase again.

The abstract syntax of a language consists of an inductively-defined set of *abstract syntax trees*, or *ast's*. An ast is a tree structure whose nodes are labeled with *operators* of a specified *arity*, the number of children of a node labeled with that operator. The tree structure makes evident the overall form of a piece of abstract syntax, avoiding the need for any machinery to disambiguate.

5.1 Abstract Syntax Trees

Abstract syntax trees are constructed from other abstract syntax trees by combining them with a *constructor*, or *operator*, of a specified *arity*. The arity of an operator, o, is the number of arguments, or sub-trees, required by o to form an ast. A *signature* is a mapping assigning to each $o \in \mathrm{dom}(\Omega)$ its arity $\Omega(o)$. The judgement form ast_Ω is inductively defined by the following rules:

$$\frac{a_1\ \mathsf{ast}_\Omega \quad \cdots \quad a_n\ \mathsf{ast}_\Omega \quad (\Omega(o) = n)}{o(a_1, \ldots, a_n)\ \mathsf{ast}_\Omega}$$

Note that we need only one rule, since the arity of o might well be zero, in which case the above rule has no premises.

For example, the following signature, Ω_{expr}, specifies an abstract syntax for the language of arithmetic expressions:

Operator	*Arity*
`num[`n`]`	0
`plus`	2
`times`	2

Here n ranges over the natural numbers; the operator `num[`n`]` is the nth numeral, which takes no arguments. The operators `plus` and `times` take two arguments each, as might be expected. The abstract syntax of our language consists of those a such that $a\ \mathsf{ast}_{\Omega_{expr}}$.

Specializing the rules for abstract syntax trees to the signature Ω_{expr} (and suppressing explicit mention of it), we obtain the following inductive definition:

$$\frac{(n \in \mathbb{N})}{\mathtt{num}[n]\ \mathsf{ast}} \tag{5.1}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\mathtt{plus}(a_1, a_2)\ \mathsf{ast}} \tag{5.2}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\mathtt{times}(a_1, a_2)\ \mathsf{ast}} \tag{5.3}$$

In practice we do not explicitly declare the operators and their arities in advance of giving an inductive definition of the abstract syntax of a language. Instead we leave it to the reader to infer the set of operators and their arities required for the definition to make sense.

5.2 Structural Induction

The principal of rule induction for abstract syntax is called *structural induction*. We say that a proposition is proved "by induction on the structure of ..." or "by structural induction on ..." to indicate that we are applying the general principle of rule induction to the rules defining the abstract syntax. In the general case to show that $P\ a$ holds whenever $a\ \mathsf{ast}_\Omega$, it is enough to show that for each operator o such that $\Omega(o) = n$, if $P\ a_1$, ..., $P\ a_n$, then $P\ o(a_1,\ldots,a_n)$.

In the special case of arithmetic expressions the principal of structural induction state that to show $P\ a$ whenever $a\ \mathsf{ast}$, it is enough to show the following three facts:

1. $P\ \mathtt{num}[n]$ for every $n \in \mathbb{N}$.
2. if $P\ a_1$ and $P\ a_2$, then $P\ \mathtt{plus}(a_1,a_2)$.
3. if $P\ a_1$ and $P\ a_2$, then $P\ \mathtt{times}(a_1,a_2)$.

To illustrate the use of structural induction let us inductively define the evaluation judgement $a\ \mathsf{ast} \Downarrow k\ \mathsf{nat}$ by the following rules:

$$\frac{}{\mathtt{num}[n]\ \mathsf{ast} \Downarrow n\ \mathsf{nat}} \tag{5.4}$$

$$\frac{a_1\ \mathsf{ast} \Downarrow k_1\ \mathsf{nat} \quad a_2\ \mathsf{ast} \Downarrow k_2\ \mathsf{nat} \quad k = k_1 + k_2\ \mathsf{nat}}{\mathtt{plus}(a_1,a_2)\ \mathsf{ast} \Downarrow k\ \mathsf{nat}} \tag{5.5}$$

$$\frac{a_1\ \mathsf{ast} \Downarrow k_1\ \mathsf{nat} \quad a_2\ \mathsf{ast} \Downarrow k_2\ \mathsf{nat} \quad k = k_1 \times k_2\ \mathsf{nat}}{\mathtt{times}(a_1,a_2)\ \mathsf{ast} \Downarrow k\ \mathsf{nat}} \tag{5.6}$$

The evaluation judgement has mode $(\forall, \exists!)$, which is to say that for every a ast there exists a unique k nat such that a ast $\Downarrow k$ nat. This is easily proved by structural induction on a, showing that in each case there is a uniquely determined k such that a ast $\Downarrow k$ nat.

In the above presentation of the evaluation judgement we chose the output domain to be the natural numbers. But it would be equally easy and natural to choose the output domain to be the same as the input domain, so that the output is also an ast, albeit one in fully evaluated form. Here are the revised rules written in this style.

$$\frac{}{\texttt{num}[n]\ \mathsf{ast} \Downarrow \texttt{num}[n]\ \mathsf{ast}} \tag{5.7}$$

$$\frac{a_1\ \mathsf{ast} \Downarrow \texttt{num}[k_1]\ \mathsf{ast} \quad a_2\ \mathsf{ast} \Downarrow \texttt{num}[k_2]\ \mathsf{ast} \quad k = k_1 + k_2\ \mathsf{nat}}{\texttt{plus}(a_1, a_2)\ \mathsf{ast} \Downarrow \texttt{num}[k]\ \mathsf{ast}} \tag{5.8}$$

$$\frac{a_1\ \mathsf{ast} \Downarrow \texttt{num}[k_1]\ \mathsf{ast} \quad a_2\ \mathsf{ast} \Downarrow \texttt{num}[k_2]\ \mathsf{ast} \quad k = k_1 \times k_2\ \mathsf{nat}}{\texttt{times}(a_1, a_2)\ \mathsf{ast} \Downarrow \texttt{num}[k]\ \mathsf{ast}} \tag{5.9}$$

Note that this sort of presentation is quite impractical when working over the strings of the concrete syntax, for then the results would have to be formatted as strings on output and parsed on input to extract their meaning. In contrast this is very easily done using pattern matching over ast's.

5.3 Parsing

The process of translation from concrete to abstract syntax is called *parsing*. Typically the concrete syntax is specified by an inductive definition defining the grammatical strings of the language, and the abstract syntax is given by an inductive definition of the abstract syntax trees that constitute the language. In this case it is natural to formulate parsing as an inductively defined function mapping concrete the abstract syntax. Since parsing is to be a function, there is exactly one abstract syntax tree corresponding to a well-formed (grammatical) piece of concrete syntax. Strings that are not derivable according to the rules of the concrete syntax are not grammatical, and can be rejected as ill-formed.

As an example, consider the following inductive definition of several mutually recursive parsing judgements that relate the concrete to the abstract syntax.

$$\frac{}{0\ \mathsf{dig} \longleftrightarrow \mathtt{num[0]}\ \mathsf{ast}} \quad \cdots \quad \frac{}{9\ \mathsf{dig} \longleftrightarrow \mathtt{num[9]}\ \mathsf{ast}} \tag{5.10}$$

$$\frac{s\ \mathsf{dig} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}} \quad \frac{s_1\ \mathsf{num} \longleftrightarrow \mathtt{num}[k_1]\ \mathsf{ast} \quad s_2\ \mathsf{dig} \longleftrightarrow \mathtt{num}[k_2]\ \mathsf{ast}}{s_1\, s_2\ \mathsf{num} \longleftrightarrow \mathtt{num}[10 \times k_1 + k_2]\ \mathsf{ast}} \tag{5.11}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \quad \frac{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}}{(s)\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{5.12}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}} \quad \frac{s_1\ \mathsf{fct} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{trm} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\mathtt{*}s_2\ \mathsf{trm} \longleftrightarrow \mathtt{times}(a_1, a_2)\ \mathsf{ast}} \tag{5.13}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}} \quad \frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\mathtt{+}s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus}(a_1, a_2)\ \mathsf{ast}} \tag{5.14}$$

Observe, first of all, that a successful parse implies that the string must have been derived according to the unambiguous grammar and that the result is a valid ast.

Theorem 5.1

1. *If* $s\ \mathsf{dig} \longleftrightarrow a\ \mathsf{ast}$*, then* $s\ \mathsf{dig}$ *and* $a\ \mathsf{ast}$.
2. *If* $s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}$*, then* $s\ \mathsf{num}$ *and* $a\ \mathsf{ast}$.
3. *If* $s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}$*, then* $s\ \mathsf{fct}$ *and* $a\ \mathsf{ast}$.
4. *If* $s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}$*, then* $s\ \mathsf{trm}$ *and* $a\ \mathsf{ast}$.
5. *If* $s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}$*, then* $s\ \mathsf{exp}$ *and* $a\ \mathsf{ast}$.

These may be proved by induction on the rules defining the parser.

If a string is generated according to the rules of the grammar, then it has a parse as an ast.

Theorem 5.2

1. *If* s dig*, then there is a unique* a *such that* s dig $\longleftrightarrow a$ ast*.*

2. *If* s num*, then there is a unique* a *such that* s num $\longleftrightarrow a$ ast*.*

3. *If* s fct*, then there is a unique* a *such that* s fct $\longleftrightarrow a$ ast*.*

4. *If* s trm*, then there is a unique* a *such that* s trm $\longleftrightarrow a$ ast*.*

5. *If* s exp*, then there is a unique* a *such that* s exp $\longleftrightarrow a$ ast*.*

These are proved simultaneously by induction on the rules defining the unambiguous grammar.

5.4 Exercises

1. Give a right-recursive grammar for numbers, and show how to parse it. Discuss the relevance of this variation to writing a recursive descent parser.

2. Show that the parser may be "run backwards" to obtain an *unparser*, or *pretty printer*. Introduce judgements that characterize those ast's that unparse to a string of each grammatical class. Then show that the unparser also has mode $(\forall, \exists!)$ over appropriate domains.

Chapter 6

Abstract Binding Trees

Abstract syntax trees make explicit the hierarchical relationships among the components of a phrase by abstracting out from irrelevant surface details such as parenthesization. *Abstract binding trees*, or *abt*'s, go one step further and make explicit the binding and scope of identifiers in a phrase, abstracting from the "spelling" of bound names so as to focus attention on their fundamental role as designators.

6.1 Names

Names are widely used in programming languages: names of variables, names of fields in structures, names of types, names of communication channels, names of locations in the heap, and so forth. Names have no structure beyond their identity. In particular, the "spelling" of a name is of no intrinsic significance, but serves only to distinguish one name from another. Consequently, we shall treat names as atoms, and abstract away their internal structure. We shall assume that we have a judgement $x\ \mathsf{name}$ expressing that x is a name, and a judgement $x \# y\ \mathsf{name}$ stating that x and y are distinct names. We shall also assume that there are infinitely many x such that $x\ \mathsf{name}$. The judgement $[x{\leftrightarrow}y]z = z'\ \mathsf{name}$ is inductively defined by the following rules:

$$\frac{}{[x{\leftrightarrow}y]x = y\ \mathsf{name}} \qquad \frac{}{[x{\leftrightarrow}y]y = x\ \mathsf{name}} \qquad \frac{x \# z\ \mathsf{name} \quad y \# z\ \mathsf{name}}{[x{\leftrightarrow}y]z = z\ \mathsf{name}}$$

6.2 Abstract Syntax With Names

Suppose that we enrich the language of arithmetic expressions given in Chapter 5 with a means of binding the value of an arithmetic expression to an identifier for use within another arithmetic expression. To support this we extend the abstract syntax with two additional constructs:[1]

$$\frac{x\ \mathsf{name}}{\mathtt{id}(x)\ \mathsf{ast}_\Omega} \qquad \frac{x\ \mathsf{name} \quad a_1\ \mathsf{ast}_\Omega \quad a_2\ \mathsf{ast}_\Omega}{\mathtt{let}(x,a_1,a_2)\ \mathsf{ast}_\Omega}$$

The ast $\mathtt{id}(x)$ represents a use of a name, x, as a variable, and the ast $\mathtt{let}(x,a_1,a_2)$ introduces a name, x, that is to be bound to (the value of) a_1 for use within a_2.

The difficulty with abstract syntax trees is that they make no provision for specifying the binding and scope of names. For example, in the ast $\mathtt{let}(x,a_1,a_2)$, the name x is available for use within a_2, but not within a_1. That is, the name x is bound by the `let` construct for use within its scope, the sub-tree a_2. But there is nothing intrinsic to the ast that makes this clear. Rather, it is a condition imposed on the ast "from the outside", rather than an intrinsic property of the abstract syntax. Worse, the informal specification is vague in certain respects. For example, what does it mean if we nest bindings for the same identifier, as in the following example?

$$\mathtt{let}(x,a_1,\mathtt{let}(x,\mathtt{id}(x),\mathtt{id}(x)))$$

Which occurrences of x refer to which bindings, and why?

6.3 Abstract Binding Trees

Abstract binding trees are a generalization of abstract syntax trees that provide intrinsic support for binding and scope of names. Just as with ast's, *operators* may be used to combine several (possibly none) abt's to form another. In addition there are two other forms of abt: a *name*, and an *abstractor*. An abstractor binds a name within a specified abt. That name may be used within that abt to refer to the binding site represented by the

[1]One may also devise a concrete syntax, for example writing $\mathtt{let}\,x\,\mathtt{be}\,e_1\,\mathtt{in}\,e_2$ for the binding construct, and a parser to translate from the concrete to the abstract syntax.

abstractor. Since various operators may bind various names in various argument positions, we must generalize the arity of an operator to be a finite *sequence* of natural numbers specifying the *valence* of each constituent abt of the operator. The valence of an abt is simply the number of abstractors at the root of that abt, specifying how many variables are bound within it. This notion of arity generalizes that in Chapter 5 by taking the arity k in that chapter to mean the arity $(0, 0, \ldots, 0)$ of length k in this chapter.

This informal description can be made precise by giving an inductive definition of the judgement $a\ \mathsf{abt}^k_\Omega$ stating that a is a well-formed abt of valence k with respect to the signature Ω assigning an arity to each of a finite set of operators.

$$\frac{x\ \mathsf{name}}{x\ \mathsf{abt}^0_\Omega} \qquad \frac{a_1\ \mathsf{abt}^{n_1}_\Omega \quad \cdots \quad a_k\ \mathsf{abt}^{n_k}_\Omega}{o(a_1, \ldots, a_k)\ \mathsf{abt}^0_\Omega}\ (*) \qquad \frac{x\ \mathsf{name} \quad a\ \mathsf{abt}^n_\Omega}{x\,.\,a\ \mathsf{abt}^{n+1}_\Omega}$$

The condition marked $(*)$ states that $\Omega(o) = (n_1, \ldots, n_k)$.

An abt of valence n has the form $x_1\,.\,x_2\,.\ldots.\,x_n\,.\,a$, which we often write as $x_1, \ldots, x_n\,.\,a$. We tacitly assume that no name is repeated in such a sequence, since doing so serves no useful purpose. We usually omit explicit mention of the signature Ω when it is clear from context, and we often write just $a\ \mathsf{abt}$ to mean $a\ \mathsf{abt}^0$.

The language of arithmetic expressions consists of the abstract binding trees over the following signature.

Operator	*Arity*
`num[`n`]`	$()$
`plus`	$(0,0)$
`times`	$(0,0)$
`let`	$(0,1)$

The arity of the "let" operator indicates that no name is bound in the first position, but that one name is bound in the second.

This class of abt's over this signature may be explicitly defined by the following rules:

$$\frac{x\ \mathsf{name}}{x\ \mathsf{abt}} \qquad \frac{n\ \mathsf{nat}}{\mathtt{num}[n]\ \mathsf{abt}}$$

$$\frac{a_1\ \mathsf{abt} \quad a_2\ \mathsf{abt}}{\mathtt{plus}(a_1, a_2)\ \mathsf{abt}} \qquad \frac{a_1\ \mathsf{abt} \quad a_2\ \mathsf{abt}}{\mathtt{times}(a_1, a_2)\ \mathsf{abt}}$$

$$\frac{a_1\ \mathsf{abt} \quad x\ \mathsf{name} \quad a_2\ \mathsf{abt}}{\mathtt{let}(a_1, x\,.\,a_2)\ \mathsf{abt}}$$

By specializing the definition to a particular signature we avoid explicit mention of abt's of non-zero valence, these being only of auxiliary interest.

6.4 Renaming

A fundamental concept is the notion of a name, x, *lying apart from* an abt, a. This is expressed by the judgement $x \mathrel{\#} a\ \mathsf{abt}^n$, which is inductively defined by the following rules:[2]

$$\frac{x \mathrel{\#} y\ \mathsf{name}}{x \mathrel{\#} y\ \mathsf{abt}^0} \qquad \frac{x \mathrel{\#} a_1\ \mathsf{abt}^{n_1} \quad \cdots \quad x \mathrel{\#} a_k\ \mathsf{abt}^{n_k}}{x \mathrel{\#} o(a_1, \ldots, a_k)\ \mathsf{abt}^0}\ (*)$$

$$\frac{}{x \mathrel{\#} x\,.\,a\ \mathsf{abt}^{n+1}} \qquad \frac{x \mathrel{\#} y\ \mathsf{name} \quad x \mathrel{\#} a\ \mathsf{abt}^n}{x \mathrel{\#} y\,.\,a\ \mathsf{abt}^{n+1}}$$

We say that a name, x, *lies within*, or *is free in*, an abt, a, written $x \in a\ \mathsf{abt}$, iff it is *not* the case that $x \mathrel{\#} a\ \mathsf{abt}$. We leave as an exercise to give a direct inductive definition of this judgement.

The result, a', of *swapping* one name, x, for another, y, within an abt, a, written $[x{\leftrightarrow}y]a = a'\ \mathsf{abt}$ is inductively defined by the following rules:

$$\frac{[x{\leftrightarrow}y]z = z'\ \mathsf{name}}{[x{\leftrightarrow}y]z = z'\ \mathsf{abt}^0}$$

[2]Here and elsewhere in this chapter, the side condition marked $(*)$ on rules is as described in the preceding section.

$$\frac{[x\leftrightarrow y]a_1 = a_1' \ \mathsf{abt}^{n_1} \quad \cdots \quad [x\leftrightarrow y]a_k = a_k' \ \mathsf{abt}^{n_k}}{[x\leftrightarrow y]o(a_1,\ldots,a_k) = o(a_1',\ldots,a_k') \ \mathsf{abt}^0} \ (*)$$

$$\frac{[x\leftrightarrow y]z = z' \ \mathsf{name} \quad [x\leftrightarrow y]a = a' \ \mathsf{abt}^n}{[x\leftrightarrow y]z\,.\,a = z'\,.\,a' \ \mathsf{abt}^{n+1}}$$

It is easy to check that the swapping judgement has mode $(\forall,\forall,\forall,\exists!)$, and so we will henceforth use it as a function. Note that name-swapping is self-inverse in that applying it twice leaves the term invariant.

A chief characteristic of a binding operator is that the choice of bound names does not matter. This is captured by treating as equivalent any two abt's that differ only in the choice of bound names, but are otherwise identical. This relation is called, for historical reasons, α-equivalence. It is inductively defined by the following rules:

$$\frac{}{x =_\alpha x \ \mathsf{abt}^0} \qquad \frac{a_1 =_\alpha b_1 \ \mathsf{abt}^{n_1} \quad \cdots \quad a_k =_\alpha b_k \ \mathsf{abt}^{n_k}}{o(a_1,\ldots,a_k) =_\alpha o(b_1,\ldots,b_k) \ \mathsf{abt}^0}$$

$$\frac{a =_\alpha b \ \mathsf{abt}^n}{x\,.\,a =_\alpha x\,.\,b \ \mathsf{abt}^{n+1}} \qquad \frac{x \# y \ \mathsf{name} \quad y \# a \ \mathsf{abt} \quad [x\leftrightarrow y]a =_\alpha b \ \mathsf{abt}^n}{x\,.\,a =_\alpha y\,.\,b \ \mathsf{abt}^{n+1}}$$

In practice we abbreviate these relations to $a =_\alpha b$ and $\beta =_\alpha \gamma$, respectively.

As an exercise, check the following α-equivalences and inequivalences using the preceding definitions specialized to the signature given earlier.

$$\begin{array}{rcl}
\mathtt{let}(x,x\,.\,x) & =_\alpha & \mathtt{let}(x,y\,.\,y) \\
\mathtt{let}(y,x\,.\,x) & =_\alpha & \mathtt{let}(y,y\,.\,y) \\
\mathtt{let}(x,x\,.\,x) & \neq_\alpha & \mathtt{let}(y,y\,.\,y) \\
\mathtt{let}(x,x\,.\mathtt{plus}(x,y)) & =_\alpha & \mathtt{let}(x,z\,.\mathtt{plus}(z,y)) \\
\mathtt{let}(x,x\,.\mathtt{plus}(x,y)) & \neq_\alpha & \mathtt{let}(x,y\,.\mathtt{plus}(y,y))
\end{array}$$

The following rule of α-equivalence, which is often stated as a basic axiom, is derivable from the preceding rules:

$$\frac{x \# y \ \mathsf{name} \quad y \# a \ \mathsf{abt}^n}{x\,.\,a =_\alpha y\,.\,[x\leftrightarrow y]a \ \mathsf{abt}^{n+1}}$$

The following variation on the rule of α-equivalence for abstractors is also derivable, and, moreover, includes the rule we have given as a special case:

$$\frac{x \# y \,\mathsf{name} \quad z \# a \,\mathsf{abt} \quad z \# b \,\mathsf{abt} \quad [x{\leftrightarrow}z]a =_\alpha [y{\leftrightarrow}z]b \,\mathsf{abt}^n}{x\,.\,a =_\alpha y\,.\,b \,\mathsf{abt}^{n+1}}$$

Apartness respects α-equivalence:

Lemma 6.1
If $a =_\alpha b\,\mathsf{abt}$ and $x \# b\,\mathsf{abt}$, then $x \# a\,\mathsf{abt}$.

It may be shown by rule induction that α-equivalence is, in fact, an equivalence relation (*i.e.*, it is reflexive, symmetric, and transitive).

Theorem 6.2
The α-equivalence relation is reflexive, symmetric, and transitive.

From this point onwards we identify any two abt's a and b such that $a =_\alpha b\,\mathsf{abt}$. This means that an abt implicitly stands for its α-equivalence class, and that we tacitly assert that all operations and relations on abt's respect α-equivalence. Put the other way around, any operation or relation on abt's that fails to respect α-equivalence is illegitimate, and therefore ruled out of consideration. One consequence of this policy on abt's is that whenever we encounter an abstractor $x\,.\,a$, we may assume that x is *fresh* in the sense that it is distinct from any given finite set of names.

6.5 Structural Induction With Binding and Scope

The principle of structural induction for ast's generalizes to abt's, subject to freshness conditions that ensure bound names are not confused. To show simultaneously (for all $n \geq 0$) that $a\,\mathsf{abt}n$ implies $P_n\ a$ holds, it is enough to show the following:

1. For any name x, the judgement $P_0\ x$ holds.
2. For each operator, o, of arity $(m_1, \ldots, m_k)$, if $P_{m_1}\ a_1$ and ... and $P_{m_k}\ a_k$, then $P_0\ o(a_1, \ldots, a_k)$.
3. For some and any "fresh" name x, if $P_n\ a$, then $P_{n+1}\ x\,.\,a$.

In the last clause the choice of x is immaterial: *some* choice of fresh names is sufficient iff *all* choices of fresh names are sufficient.

Specializing this to arithmetic expressions as defined earlier, the principle of structural induction states that to show $P\ a$ for every $a\ \mathsf{abt}$, it is enough to show the following facts:

1. If $x\ \mathsf{name}$, then $P\ x$.
2. If $n\ \mathtt{nat}$, then $P\ \mathtt{num}[n]$.
3. If $P\ a_1$ and $P\ a_2$, then $P\ \mathtt{plus}(a_1, a_2)$.
4. If $P\ a_1$ and $P\ a_2$, then $P\ \mathtt{times}(a_1, a_2)$.
5. If $P\ a_1$ and, for some/every $x\ \mathsf{name}$, $P\ a_2$, then $P\ \mathtt{let}(a_1, x\,.\,a_2)$.

Here again the choice of bound variable name is irrelevant, provided that it is "fresh" in the sense of not clashing with any other name in P.

6.6 Substitution

Substitution is the process of replacing free occurrences of a name with a specified abt (of valence 0). The judgement $[x{\leftarrow}a]b = b'\ \mathsf{abt}$ states that b' is the result of substituting a for all free occurrences of x in b. It is inductively defined by the following rules:

$$\frac{}{[x{\leftarrow}a]x = a\ \mathsf{abt}^0} \qquad \frac{x \mathbin{\#} y\ \mathsf{name}}{[x{\leftarrow}a]y = y\ \mathsf{abt}^0}$$

$$\frac{[x{\leftarrow}a]b_1 = b_1'\ \mathsf{abt}^{n_1} \quad \cdots \quad [x{\leftarrow}a]b_k = b_k'\ \mathsf{abt}^{n_k}}{[x{\leftarrow}a]o(b_1, \ldots, b_k) = o(b_1', \ldots, b_k')\ \mathsf{abt}^0}\ (*)$$

$$\frac{x \mathbin{\#} y\ \mathsf{name} \quad y \mathbin{\#} a\ \mathsf{abt}^0 \quad [x{\leftarrow}a]b = c\ \mathsf{abt}^n}{[x{\leftarrow}a]y\,.\,b = y\,.\,c\ \mathsf{abt}^{n+1}}$$

The apartness conditions on the last rule imply no loss of generality, because they can always be satisfied by appropriate choice of bound variable name, y, in the target of the substitution.

Substitution defines a function up to α-equivalence.

Theorem 6.3

1. *If a abt^0, x name, and b abt^n, there exists b' abt^n such that $b =_\alpha b'$ abt^n and $[x{\leftarrow}a]b' = c$ abt^n for some c abt^n.*

2. *If a abt^0, x name, $b =_\alpha b'$ abt^n, $[x{\leftarrow}a]b = c$ abt^n and $[x{\leftarrow}a]b' = c'$ abt^n, then $c =_\alpha c'$ abt^n.*

6.7 Summary

Let us now take stock of what we have accomplished. The definition of α-equivalence for abt's makes clear the nature of bound names. Briefly, a bound name is merely a reference to a binding site; bound names have no intrinsic identity. This is enforced by treating abt's modulo α-equivalence, which is to say that we do not distinguish two abt's a and b such that $a =_\alpha b$. The significance of this identification may be briefly summarized in several equivalent ways.

1. A bound variable name may always be chosen to be different from any given finite set of names. This is because one representative, a, of an α-equivalence class is as good as any other, b.

2. It is illegitimate to rely upon the choice of a bound variable name, since it "changes under one's feet" without explicit mention. This is just to say that a property of an α-equivalence class is only well-defined if it respects α-equivalence — that is, if its meaning is independent of the choice of representative.

3. One choice of bound variable name is as good as any other; if we do not like the one we have, we may rename it at will without changing the abt in any material way. That is, we may always replace an abt by an α-equivalent one; they designate the same α-equivalence class.

4. Bound variable names "automatically" evade confusion with any other variable name in a given context. This is because we may always choose another representative in the case that one choice is inconvenient in a given context.

We will freely make use of these and similar conveniences afforded by α-equivalence throughout this book.

6.8 Exercises

1. Give a direct inductive definition of the judgements $x \in a\ \mathsf{abt}$.
2. Give a proof that substitution defines a function up to α-equivalence.
3. Give an inductive definition of *simultaneous substitution*,

$$[x_1, \ldots, x_k \leftarrow a_1, \ldots, a_k]b = c\ \mathsf{abt},$$

   which states that c is the result of replacing all free occurrences of x_i by a_i in b (for each $1 \leq i \leq k$).

Chapter 7

Specifying Syntax

Having defined the three levels of syntax, we may now summarize how we shall make use of them in the rest of this book. The focus of our work will be on the abstract syntax and binding structure of languages. We will not concern ourselves with the concrete syntax of the languages we consider. However, it will be necessary for us to write down examples, so we must establish conventions for defining the syntax of a language in a concise form. While the inference rule format always suffices, it is often more concise to use a modified form of grammar notation to define the abstract syntax and binding structure of a language.

This notation is best illustrated by example. Here is a presentation of the abstract syntax and binding structure of a language of expressions using grammar notation.

$$\begin{array}{lll l} \textit{Types} & \tau & ::= & \texttt{num} \mid \texttt{str} \\ \textit{Expr's} & e & ::= & x \mid \texttt{num[}n\texttt{]} \mid \texttt{str[}s\texttt{]} \mid \texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{cat(}e_1\texttt{,}e_2\texttt{)} \mid \\ & & & \texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \end{array}$$

The important point about this form of specification is that it specifies two categories of abstract binding trees, the *types* and the *terms*. The specification is given "by example", using meta-variables that range over the syntactic categories to illustrate the pattern. In this case the meta-variable τ ranges over the category of types, and the meta-variable e ranges over the category of expressions. In addition, the meta-variable x ranges over names of variables, and n ranges over natural numbers. (This convention is often used without explicit mention.)

The notation used in the grammar makes clear the intended binding and scope of variables. For example, it is clear from the notation used that `let` is an operator with arity $(0,1)$, specifying that it binds one variable in the second position. We take all such conventions to be tacitly understood without explicit mention, and treat all abt's module α-equivalence, which ensures that the names of bound variables may be chosen at will.

For the sake of writing examples, we will sometimes take liberties with the syntax for the sake of readability, relying on conventions of concrete syntax that are familiar to the reader from other contexts. For example, we may introduce parentheses to emphasize grouping, or use infix or other standard forms of notation that enhance the readability of the examples without being explicit about the intended meaning. For example, we might write e_1+e_2 for $\texttt{plus}(e_1,e_2)$, or $\texttt{let}\,x\,\texttt{be}\,e_1\,\texttt{in}\,e_2$ for $\texttt{let}(e_1,x.e_2)$, leaving it to the reader to interpret these in the evident manner.

Part III

Static and Dynamic Semantics

Chapter 8

The Phase Distinction

We will distinguish two *phases* of processing, the *static phase* and the *dynamic phase*. Roughly speaking, the static phase consists of ensuring that the program is well-formed, and the dynamic phase consists of executing well-formed programs. Depending on the level of detail one wishes to consider, one may make finer distinctions, considering, for example, parsing to be a separate phase from code generation, or linking to be separate from execution.

We will, in fact, find it useful to draw such fine distinctions later in the development, but before we can do that we must first answer these two fundamental questions:

1. Which are the well-formed programs?

2. What is the execution behavior of a well-formed programs?

The first question is answered by the *static semantics* of a language, and the second is answered by its *dynamic semantics*.

The central organizing principle for static semantics is the concept of a *type*. A type is characterized by two, closely related notions:

1. The primitive operations that *create*, or *introduce*, values of that type.

2. The primitive operations that *compute with*, or *eliminate*, values of that type.

The terminology of introduction and elimination is rooted in history, and over time the words have sometime acquired other meanings. In particular, please note that the "elimination" operations having nothing to do with storage reclamation!

As an illustrative example, the type of natural numbers may be characterized by the following primitive notions:

1. The primitives `zero` and `succ`$(-)$ for introducing natural numbers.

2. Operations such as addition and multiplication for computing with natural numbers, or, more generally, the ability to define a function by induction on the natural numbers.

Anything we wish to do with natural numbers can be accomplished using these operations alone. So, in particular, we never need to know what the natural numbers "really are" — they are an abstraction characterized by the introduction and elimination operations on them.

The dynamic semantics of a language determines how to execute the programs given by the static semantics. The key to the dynamic semantics is the *inverse principle*, which states that *the elimination operations are inverse to the introduction operations*. (Here again the terminology cannot be taken too seriously, but is only suggestive of the general idea.) The elimination operations compute with the values created by the introduction operations, taking apart what was introduced in order to obtain their result. Since the elimination operations take apart what the introduction operations put together, they may be seen as a kind of inverse relationship between them.

An example will help make this clear. The addition operation takes as input two natural numbers and computes their sum. Looking at, say, the first of the two numbers, it can have been created in only one of two ways. Either it is `zero`, in which case the sum is the second of the two numbers. Otherwise, it is `succ`(x), in which case the sum is the successor of the sum of x and the second number. Thus we see that addition "takes apart" what the introduction operations created in order to compute its result.

Interesting languages have more than one type. For example, we might also have a type `string` of strings of characters whose introduction operations include character strings enclosed in quotation marks, and whose elimination operations might include string concatenation and a length calculation. Once we have more than one type, we then have the potential for a *type error*, a combination of operations that violates the type structure. For example, it makes no sense to add a number to a string, or to concatenate two numbers. The role of a static semantics is to ensure that such erroneous programs are ruled out of consideration so that the dynamic semantics may concern itself only with well-formed combinations such as additions of two numbers or concatenations of two strings, and never have to be concerned with giving meaning to ill-typed phrases.

This leads to the central concept of *type safety*, which ensures that the static semantics and the dynamic semantics "fit together" properly. More precisely, we may think of the static semantics as imposing strictures that are tacitly assumed to hold by the dynamic semantics. But what if they don't? How do we know that the assumptions of the dynamic semantics match the strictures imposed by the static semantics? The answer is that we do not know this until we *prove* it. That is, we must prove a theorem, called the *type safety theorem*, that states that the static and dynamic semantics cohere. In practical terms this ensures that a whole class of run-time faults, which manifest themselves as "bus errors" or "core dumps" in familiar languages, cannot arise. One theorem about a *language* implies infinitely many theorems, one for *each program* written in it.

This establishes the pattern for the remainder of this book. Programming languages are organized as a *collection of types* — the "features" of the language emerge as the operations associated with a particular type. These types are given meaning by the static and dynamic semantics, and we ensure that the whole is well-defined by proving type safety. This simple methodology is surprisingly powerful, both as a tool for language design and as a tool for language implementation — for the theory and practice of programming languages.

Chapter 9

Static Semantics

In this chapter we will illustrate the specification of a static semantics for a simple language of expressions defined as follows:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{num} \mid \texttt{str} \\ \textit{Expr's} & e & ::= & x \mid \texttt{num[}n\texttt{]} \mid \texttt{str}[s] \mid \texttt{plus}(e_1,e_2) \mid \texttt{cat}(e_1,e_2) \mid \texttt{let(}e_1,x\texttt{.}e_2\texttt{)} \end{array}$$

The introduction forms for `num` are the numerals, $\texttt{num[}n\texttt{]}$, and the elimination form is $\texttt{plus}(e_1,e_2)$. The introduction forms for `str` are the string literals, $\texttt{str}[s]$, and the elimination form is $\texttt{cat}(e_1,e_2)$. Finally, we have a variable binding construct, $\texttt{let(}e_1,x\texttt{.}e_2\texttt{)}$.

9.1 Static Semantics of Expressions

The static semantics is defined by the typing judgement $e : \tau$, where τ is a type (either `num` or `str`), and e is an expression, which is an abstract binding tree over the preceding signature. The typing judgement is defined using higher-order judgement forms of the kind considered in Chapter 2, with an explicit representation of the typing hypotheses. We write $\Gamma \vdash e : \tau$, where Γ is of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$ with no two x_i's being equal, and the free names occurring in e are among the set $x_1, \ldots, x_n$.[1]

The static semantics of expressions is inductively defined by the fol-

[1]Officially, the turnstile is indexed by this set of variables, but we omit explicit mention of this in our notation for typing judgements.

lowing collection of *typing rules*:

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}}\ (s \in \Sigma^*) \qquad \frac{}{\Gamma \vdash \texttt{num[}n\texttt{]} : \texttt{num}}\ (n \in \mathbb{N})$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1\texttt{,}e_2) : \texttt{num}} \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1\texttt{,}e_2) : \texttt{str}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} : \tau_2}$$

The rule for variables is implicit in the meaning of the hypothetical judgement:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

We often state this rule explicitly for the sake of emphasis.

In the rule for `let`'s we tacitly assume that x is not otherwise declared in Γ; this may always be achieved by suitable renaming of bound variables. Also, if $x : \tau$ occurs anywhere in Γ, we may tacitly regard Γ as having the form $\Gamma', x : \tau$, in which the designated declaration occurs "last". This is justified by the admissibility of permutation for the hypothetical judgement form, as discussed in Chapter 2.

9.2 Properties of the Static Semantics

The structural rules governing the hypothetical judgement ensure that the typing judgement obeys the following weakening and substitution properties:

Theorem 9.1 (Structural Properties of Typing)

1. *If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$, provided that x is not already declared in Γ.*
2. *If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [x\leftarrow e]e' : \tau'$.*

Proof:

1. By induction on the derivation of $\Gamma \vdash e' : \tau'$.
2. By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$.

■

The typing rules are *syntax-directed* in the sense that there is exactly one rule for each form of expression. Consequently, we obtain the following *inversion principles* for typing.

Theorem 9.2 (Inversion for Typing)
If $\Gamma \vdash e : \tau$, *then*

1. *if* $e = x$ *for some variable* x, *then* $\Gamma = \Gamma', x : \tau$.
2. *if* $e = \texttt{plus}(e_1, e_2)$, *then* $\tau = \texttt{num}$, $e_1 : \texttt{num}$, *and* $e_2 : \texttt{num}$.
3. *if* $e = \texttt{cat}(e_1, e_2)$, *then* $\tau = \texttt{str}$, $e_1 : \texttt{str}$, *and* $e_2 : \texttt{str}$.
4. *if* $e = \texttt{let}(e_1, x.e_2)$, *then for some* τ_1, $e_1 : \tau_1$, *and* $\Gamma, x : \tau_1 \vdash e_2 : \tau$.

Proof: By induction on the derivation of $\Gamma \vdash e : \tau$. ■

A *value*, v, is either $\texttt{num}[n]$ for some n nat or $\texttt{str}[s]$ for some s string. We may characterize the closed values of a type as follows.

Theorem 9.3 (Canonical Forms)
If $v : \tau$, *where* v *is a value, then*

1. *If* $\tau = \texttt{num}$, *then* $v = \texttt{num}[n]$ *for some natural number* n.
2. *If* $\tau = \texttt{str}$, *then* $v = \texttt{str}[s]$ *for some string* s.

Proof: By induction on the derivation of $v : \tau$, taking account of the fact that v is a value. ■

9.3 Exercises

1. Show that the expression $e = \texttt{plus}(\texttt{num}[7], \texttt{str}[abc])$ is ill-typed in that there is no τ such that $e : \tau$.
2. Show that if $\Gamma \vdash e : \tau$ and $x \in e$, then $\Gamma = \Gamma', x : \tau$ for some τ.

Chapter 10

Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. There are two popular methods for specifying dynamic semantics. One method, called *structured operational semantics (SOS)*, or *transition semantics*, presents the dynamic semantics of a language as a transition system specifying the step-by-step execution of programs. Another, called *evaluation semantics*, or *ES*, presents the dynamic semantics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Each presentation has its uses, so we discuss both forms of dynamic semantics, as well as their relation to one another.

10.1 Structured Operational Semantics

A structured operational semantics for a language consists of an inductively defined transition system whose states are closed, well-formed expressions. Every state is an initial state, and the final states are the values, defined by the following rules:

$$\frac{}{\texttt{num}[n]\ \textsf{value}} \qquad \frac{}{\texttt{str}[s]\ \textsf{value}}$$

The transition judgement $e \longmapsto e'$ is inductively defined by the following rules:

$$\frac{(p = m + n)}{\texttt{plus(num[}m\texttt{],num[}n\texttt{])} \longmapsto \texttt{num[}p\texttt{]}} \qquad \frac{(u = s\,\hat{}\,t)}{\texttt{cat}(\texttt{str}[s],\texttt{str}[t]) \longmapsto \texttt{str}[u]}$$

$$\frac{e_1 \ \mathsf{value}}{\texttt{let}(e_1,x.e_2) \longmapsto [x{\leftarrow}e_1]e_2}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(e_1,e_2) \longmapsto \texttt{plus}(e_1',e_2)} \qquad \frac{e_1 \ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{plus}(e_1,e_2) \longmapsto \texttt{plus}(e_1,e_2')}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{cat}(e_1,e_2) \longmapsto \texttt{cat}(e_1',e_2)} \qquad \frac{e_1 \ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{cat}(e_1,e_2) \longmapsto \texttt{cat}(e_1,e_2')}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{let}(e_1,x.e_2) \longmapsto \texttt{let}(e_1',x.e_2)}$$

The first three rules defining the transition judgement are sometimes called *instructions*, since they correspond to the primitive execution steps of the machine. Addition is evaluated by adding and concatenation by appending; `let` bindings are evaluated by substituting the definition for the variable in the body. In all three cases the *principal arguments* of the constructor are required to be numbers. Both arguments of an addition or concatenation are principal, but only the binding of the variable in a *let* expression is principal. We say that these primitives are evaluated *by value*, because the instructions apply only when the principal arguments have been fully evaluated.

What if the principal arguments have not (yet) been fully evaluated? Then we must evaluate them! In the case of expressions we arbitrarily choose a left-to-right evaluation order. First we evaluate the first argument, then the second. Once both have been evaluated, the instruction rule applies. In the case of `let` expressions we first evaluate the binding, after which the instruction step applies. Note that evaluation of an argument can take multiple steps. The transition judgement is defined so that one step of evaluation is made at a time, reconstructing the entire expression as necessary.

For example, consider the following evaluation sequence.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{let(plus(num[1],num[2]),}x\texttt{.plus(plus(}x\texttt{,num[3]),num[4]))}} \\
\longmapsto & \texttt{let(num[3],}x\texttt{.plus(plus(}x\texttt{,num[3]),num[4]))} \\
\longmapsto & \texttt{plus(plus(num[3],num[3]),num[4])} \\
\longmapsto & \texttt{plus(num[6],num[4])} \\
\longmapsto & \texttt{num[10]}
\end{array}
$$

Each step is justified by a rule defining the transition judgement. Instruction rules are axioms, and hence have no premises, but all other rules are justified by a subsidiary deduction of another transition. For example, the first transition is justified by a subsidiary deduction of

$$\texttt{plus(num[1],num[2])} \longmapsto \texttt{num[3]},$$

which is justified by the first instruction rule definining the transition judgement. Each of the subsequent steps is justified similarly.

Observe that the expression $e = \texttt{cat(num[3],str}[abc]\texttt{)}$ is not a final state, but there is no e' such that $e \longmapsto e'$ — it is a stuck state. Fortunately it is also ill-typed! We shall prove in the next chapter that no well-typed expression is stuck.

Since the transition judgement in SOS is inductively defined, we may reason about it using rule induction. Specifically, to show that $P\ (e,e')$ holds whenever $e \longmapsto e'$, it is sufficient to show that P is closed under the rules defining the transition judgement. For example, it is a simple matter to show by rule induction that the transition judgement for evaluation of expressions is deterministic: if $e \longmapsto e'$ and $e \longmapsto e''$, then $e' = e''$. This may be proved by simultaneous rule induction over the rules defining the transition judgement.

10.2 Evaluation Semantics

Another method for defining the dynamic semantics of a language, called *evaluation semantics*, consists of an inductive definition of the evaluation judgement, $e \Downarrow v$, specifying the value, v, of a closed expression, e. This judgement is inductively defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow \texttt{num[}n\texttt{]}}$$

$$\frac{}{\texttt{str}[s] \Downarrow \texttt{str}[s]}$$

$$\frac{e_1 \Downarrow \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow \texttt{num[}n_2\texttt{]} \quad (n = n_1 + n_2)}{\texttt{plus}(e_1, e_2) \Downarrow \texttt{num[}n\texttt{]}}$$

$$\frac{e_1 \Downarrow \texttt{str}[s_1] \quad e_2 \Downarrow \texttt{str}[s_2] \quad (s = s_1 \hat{\ } s_2)}{\texttt{cat}(e_1, e_2) \Downarrow \texttt{str}[s]}$$

$$\frac{e_1 \Downarrow v_1 \quad [x \leftarrow v_1]e_2 \Downarrow v_2}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \Downarrow v_2}$$

The value of a `let` expression is determined by the value of its binding, and the value of the corresponding substitution instance of its body. Since the substitution instance is not a sub-expression of the `let`, the rules are not syntax-directed.

Since the evaluation judgement is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that the property $P\ (e, v)$ holds, it is enough to show that P is closed under the rules defining the evaluation judgement. Specifically, our proof obligations are:

1. Show that $P\ (\texttt{num[}n\texttt{]},\texttt{num[}n\texttt{]})$.

2. Show that $P\ (\texttt{str}[s], \texttt{str}[s])$.

3. Show that $P\ (\texttt{plus}(e_1, e_2), \texttt{num[}n\texttt{]})$, assuming $n = n_1 + n_2$, $P\ (e_1, \texttt{num[}n_1\texttt{]})$ and $P\ (e_2, \texttt{num[}n_2\texttt{]})$.

4. Show that $P\ (\texttt{cat}(e_1, e_2), \texttt{str}[s])$, assuming $s = s_1 \hat{\ } s_2$, $P\ (e_1, \texttt{str}[s_1])$ and $P\ (e_2, \texttt{str}[s_2])$.

5. Show that $P\ (\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)}, v_2)$, assuming $P\ (e_1, v_1)$ and $P\ ([x \leftarrow v_1]e_2, v_2)$.

This induction principle is *not* the same as structural induction on e, because the evaluation rules are not syntax-directed.

10.3 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for the same language. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The transition semantics describes a step-by-step process of execution, whereas the evaluation semantics suppresses the intermediate states, focussing attention on the initial and final states alone. This suggests that the appropriate correspondence is between *complete* execution sequences in the transition semantics and the evaluation judgement in the evaluation semantics.

Theorem 10.1
For all closed expressions e and natural numbers n, $e \overset{!}{\longmapsto}$ `num[n]` iff $e \Downarrow$ `num[n]`.

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

Lemma 10.2
If $e \Downarrow$ `num[n]`, then $e \overset{!}{\longmapsto}$ `num[n]`.

Proof: By induction on the definition of the evaluation judgement. For example, suppose that $\texttt{plus}(e_1,e_2) \Downarrow \texttt{num}[n]$ by the rule for evaluating additions. By induction we know that $e_1 \overset{!}{\longmapsto} \texttt{num}[n_1]$ and $e_2 \overset{!}{\longmapsto} \texttt{num}[n_2]$. We reason as follows:

$$\begin{array}{rcl}
\texttt{plus}(e_1,e_2) & \overset{*}{\longmapsto} & \texttt{plus}(\texttt{num}[n_1],e_2) \\
& \overset{*}{\longmapsto} & \texttt{plus}(\texttt{num}[n_1],\texttt{num}[n_2]) \\
& \longmapsto & \texttt{num}[n_1+n_2]
\end{array}$$

Therefore $\texttt{plus}(e_1,e_2) \overset{!}{\longmapsto} \texttt{num}[n_1+n_2]$, as required. The other cases are handled similarly. ■

For the converse, recall from Chapter 3 the definitions of multi-step evaluation and complete evaluation. Since $\texttt{num}[n] \Downarrow \texttt{num}[n]$, it suffices to show that evaluation is closed under head expansion.

Lemma 10.3
If $e \longmapsto e'$ and $e' \Downarrow$ `num[n]`, then $e \Downarrow$ `num[n]`.

Proof: By induction on the definition of the transition judgement. For example, suppose that $\texttt{plus}(e_1,e_2) \longmapsto \texttt{plus}(e_1',e_2)$, where $e_1 \longmapsto e_1'$. Suppose further that $\texttt{plus}(e_1',e_2) \Downarrow \texttt{num}[n]$, so that $e_1' \Downarrow \texttt{num}[n_1]$, and $e_2 \Downarrow \texttt{num}[n_2]$ and $n = n_1 + n_2$. By induction $e_1 \Downarrow \texttt{num}[n_1]$, and hence $\texttt{plus}(e_1,e_2) \Downarrow \texttt{num}[n]$, as required. ■

10.4 Environment Semantics

Both the transition semantics and the evaluation semantics given earlier rely on substitution to replace `let`-bound variables by their bindings during evaluation. This approach maintains the invariant that only closed expressions are ever considered, and, as we shall see in the next chapter, facilitates proving properties of the language. However, in practice, we do not perform substitution, but rather record the bindings of variables in some sort of data structure. In this section we show how this can be elegantly modeled using hypothetical judgements.

The basic idea is to consider hypotheses of the form $x \Downarrow v$, where x is a variable and v is a value, such that no two hypotheses govern the same variable. Let η range over finite sets of such hypotheses, which we call an *environment*. We will consider judgements of the form $\eta \vdash_X e \Downarrow v$, where X is the finite set of variables appearing on the left of a hypothesis in η. As usual, we will suppress explicit mention of the parameter set X, and simply write $\eta \vdash e \Downarrow v$. The rules defining this judgement are as follows:

$$\frac{}{\eta, x \Downarrow v \vdash x \Downarrow v}$$

$$\frac{\eta \vdash e_1 \Downarrow \texttt{num}[n_1] \quad \eta \vdash e_2 \Downarrow \texttt{num}[n_2]}{\eta \vdash \texttt{plus}(e_1,e_2) \Downarrow \texttt{num}[n_1 + n_2]} \qquad \frac{\eta \vdash e_1 \Downarrow \texttt{str}[s_1] \quad \eta \vdash e_2 \Downarrow \texttt{str}[s_2]}{\eta \vdash \texttt{cat}(e_1,e_2) \Downarrow \texttt{num}[s_1\texttt{\^{}}s_2]}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \texttt{let}(e_1,x.e_2) \Downarrow v_2}$$

The variable rule is an instance of the reflexivity rule for hypothetical judgements, and therefore need not be explicitly stated. We nevertheless include it here for clarity. The `let` rule augments the environment with a new assumption governing the bound variable (which, by α-conversion, may be chosen to be distinct from any other variable currently in η to preserve the invariant that no two assumptions govern the same variable).

The environment semantics is related to the evaluation semantics by the following theorem:

Theorem 10.4
$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ *iff* $[x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]e \Downarrow v$.

Proof: The left to right direction is proved by induction on the rules defining the evaluation semantics, making use of the definition of substitution and the definition of the evaluation semantics for closed expressions. The converse is proved by induction on the structure of e, again making use of the definition of substitution. Note that we must induct on e in order to detect occurrences of variables x_i in e, which are governed by a hypothesis in the environment semantics. ■

10.5 Exercises

1. Prove that if $e \longmapsto e_1$ and $e \longmapsto e_2$, then $e_1 \equiv e_2$.

2. Prove that if $e \Downarrow v$, then v value.

3. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 \equiv v_2$.

4. Complete the proof of equivalence of evaluation and transition semantics.

5. Is it possible to use environments in a structured operational semantics? What difficulties do you encounter?

Chapter 11

Type Safety

Many programming languages, including ML and Java, are said to be "safe" (or, "type safe", or "strongly typed"). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, it will never arise that an integer is to be applied to an argument, nor that two functions could be added to each other. What is remarkable is that we will be able to clarify the idea of type safety without making reference to an implementation. Consequently, the notion of type safety is extremely robust — it is shared by *all* correct implementations of the language.

Type safety states that the static and dynamic semantics of a language *cohere* in that the strictures of the type system ensure that execution is well-behaved. Simply put, the type system ensures that evaluation cannot "go off into the weeds" into an ill-defined state for which no definite result can be obtained. This is proved by showing that a transition from a well-defined state leads only to well-defined states, and that if a state is well-defined then it is either in a valid final state, or is capable of making a transition. The static semantics specifies what we mean by well-defined, and the dynamic semantics specifies what it means to make a transition. This leads to the following formal statements that, together, express the safety of the language:

1. **Preservation**: If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.

2. **Progress**: If $e : \tau$, then either e value, or there exists e' such that $e \longmapsto e'$.

The first says that the steps of evaluation preserve well-typedness (indeed, preserves typing), and the second says that well-typedness ensures that either we are done or we can make progress towards completion.

11.1 Preservation for Expressions

The preservation theorem for the language of expressions defined in Chapters 9 and 10 is proved by rule induction on the definition of the transition system for evaluating expressions (as given in Chapter 10).

Theorem 11.1 (Preservation)
If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.

Proof: Consider the rule

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(e_1,e_2) \longmapsto \texttt{plus}(e_1',e_2)}.$$

Assume that $\texttt{plus}(e_1,e_2) : \tau$. By inversion for typing, we have that $\tau = \texttt{num}$, $e_1 : \texttt{num}$, and $e_2 : \texttt{num}$. By induction we have that $e_1' : \texttt{num}$, and hence $\texttt{plus}(e_1',e_2) : \texttt{num}$. The case for concatenation is handled similarly.

Now consider the rule

$$\frac{}{\texttt{let}(e_1,x.e_2) \longmapsto [x \leftarrow e_1]e_2}.$$

Assume that $\texttt{let}(e_1,x.e_2) : \tau_2$. By inversion for typing, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By substitution $[x \leftarrow e_1]e_2 : \tau_2$, as desired.

We leave the remaining cases to the reader. ■

The proof of preservation must proceed by rule induction on the rules defining the transition judgement. It cannot, for example, proceed by induction on the structure of e, for in most cases there is more than one transition rule for each expression form. Nor can it be proved by induction on the typing rules, for in the case of the `let` rule, the context is enriched to consider an open term, to which no dynamic semantics is assigned.

11.2 Progress for Expressions

The progress theorem captures the idea that well-typed programs cannot "get stuck".

Theorem 11.2 (Progress)
If $e : \tau$, then either e value, or there exists e' such that $e \longmapsto e'$.

Proof: The proof is by induction on the typing derivation. The rule for variables cannot arise, because we are only considering closed typing judgements. Consider the typing rule

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \texttt{num}}$$

where Γ is empty. By induction we have that either e_1 value, or there exists e_1' such that $e_1 \longmapsto e_1'$. In the latter case it follows that $\texttt{plus}(e_1, e_2) \longmapsto \texttt{plus}(e_1', e_2)$, as required. In the former we also have by induction that either e_2 value, or there exists e_2' such that $e_2 \longmapsto e_2'$. In the latter case we have that $\texttt{plus}(e_1, e_2) \longmapsto \texttt{plus}(e_1, e_2')$, which is enough. In the former case we have, by canonical forms, that $e_1 = \texttt{num}[n_1]$ and $e_2 = \texttt{num}[n_2]$, and hence $\texttt{plus}(\texttt{num}[n_1], \texttt{num}[n_2]) \longmapsto \texttt{num}[n_1 + n_2]$.

The other cases are handled similarly, and are left to the reader. ■

Since the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of e, appealing to the inversion theorem at each step to characterize the types of the parts of e. But this approach breaks down when the typing rules are no longer syntax-directed, that is, when there may be more than one rule for a given expression form. In such cases it becomes clear that the most direct approach is to consider the typing rules one-by-one.

Summing up, the combination of preservation and progress together constitute a proof of safety. The progress theorem ensures that well-typed expressions do not "get stuck" in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the static and dynamic semantics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

11.3 Exercises

1. Complete the proof of preservation.

2. Complete the proof of progress.

3. Do something similar, in detail.

Part IV

Functions

Chapter 12

A Functional Language

The λ-calculus is a fundamental building block in the study of programming language concepts. In contrast to machine models, such as Turing machines or random-access machines, the λ-calculus is a *linguistic* foundation for computation that takes as primitive the notion of a *function*. In its barest form the entire language consists of nothing but functions — even data structures arise as functions in the λ-calculus!

Although elegant in its spartan simplicity, the λ-calculus is remarkably subtle and is best approached gradually as the culmination of the development of several key ideas. We will therefore begin our study with a simple functional language that, like the λ-calculus, takes the notion of function as a starting point, but which, unlike the λ-calculus, is not the only concept in the language. Later on we will see how the pure λ-calculus may be reconstructed on this foundation once we have developed a few more ideas.

12.1 Syntax

The syntax of MinML is given as follows:

$$\begin{array}{lcll}
\textit{Types} & \tau & ::= & \texttt{nat} \mid \texttt{arrow(}\tau_1\texttt{,}\tau_2\texttt{)} \\
\textit{Expr's} & e & ::= & x \mid \texttt{num[}n\texttt{]} \mid \texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{times(}e_1\texttt{,}e_2\texttt{)} \\
 & & & \texttt{ifz(}e\texttt{,}e_1\texttt{,}e_2\texttt{)} \mid \texttt{lambda(}\tau\texttt{,}x\texttt{.}e\texttt{)} \mid \texttt{app(}e_1\texttt{,}e_2\texttt{)} \\
 & & & \texttt{let(}\tau\texttt{,}e_1\texttt{,}x\texttt{.}e_2\texttt{)}
\end{array}$$

As discussed in Chapter 7, this grammar implicitly specifies a signature that determines the set of abt's described by the above grammar.

The constructs of the language may be classified by type. Associated with the type $\texttt{nat}$ are the numerals, $\texttt{num[}n\texttt{]}$, the arithmetic operations, $\texttt{plus(}e_1\texttt{,}e_2\texttt{)}$ and $\texttt{times(}e_1\texttt{,}e_2\texttt{)}$, and the zero-test, $\texttt{ifz(}e\texttt{,}e_1\texttt{,}e_2\texttt{)}$. Associated with the function type $\texttt{arrow(}\tau_1\texttt{,}\tau_2\texttt{)}$ are the *λ-abstractions*, $\texttt{lambda(}\tau\texttt{,}x\texttt{.}e\texttt{)}$, and the *applications*, $\texttt{app(}e_1\texttt{,}e_2\texttt{)}$. Finally, we have a generic construct for binding expressions to names, $\texttt{let(}\tau\texttt{,}e_1\texttt{,}x\texttt{.}e_2\texttt{)}$.

The following chart summarizes the concrete syntax corresponding to each form of abstract syntax in MinML:

Abstract Syntax	*Concrete Syntax*
$\texttt{num[}n\texttt{]}$	n
$\texttt{plus(}e_1\texttt{,}e_2\texttt{)}$	$e_1\texttt{+}e_2$
$\texttt{times(}e_1\texttt{,}e_2\texttt{)}$	$e_1\texttt{*}e_2$
$\texttt{ifz(}e_0\texttt{,}e_1\texttt{,}e_2\texttt{)}$	$\texttt{ifz}\,e_0\,\texttt{then}\,e_1\,\texttt{else}\,e_2$
$\texttt{lambda(}\tau\texttt{,}x\texttt{.}e\texttt{)}$	$\lambda\texttt{(}x\texttt{:}\tau\texttt{.}e\texttt{)}$
$\texttt{app(}e_1\texttt{,}e_2\texttt{)}$	$e_1\texttt{(}e_2\texttt{)}$
$\texttt{let(}\tau\texttt{,}e_1\texttt{,}x\texttt{.}e_2\texttt{)}$	$\texttt{let}\,x\texttt{:}\tau\,\texttt{be}\,e_1\,\texttt{in}\,e_2$

12.2 Static Semantics

The *typing judgement*, $e : \tau$, states that expression e has type τ. More generally, we will consider hypothetical judgements of the form

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau,$$

stating that e is of type τ under the assumptions that each variable x_i is of type τ_i. We let Γ stand for any such sequence of typing assumptions.

The typing judgement is inductively defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{12.1}$$

$$\frac{}{\Gamma \vdash \texttt{num[}n\texttt{]} : \texttt{nat}} \tag{12.2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{plus(}e_1\texttt{,}e_2\texttt{)} : \texttt{nat}} \tag{12.3}$$

$$\frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{times}(e_1, e_2) : \texttt{nat}} \tag{12.4}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{ifz}(e, e_1, e_2) : \tau} \tag{12.5}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lambda}(\tau_1, x.e) : \texttt{arrow}(\tau_1, \tau_2)} \tag{12.6}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{app}(e_1, e_2) : \tau} \tag{12.7}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(\tau_1, e_1, x.e_2) : \tau_2} \tag{12.8}$$

As usual, when introducing a new typing assumption we tacitly assume that the variable being introduced is not otherwise declared; this may always be achieved by α-conversion prior to application of the typing rule.

12.3 Basic Properties of the Static Semantics

A key observation about the typing rules is that there is exactly one rule for each form of expression — that is, there is one rule for the each of the numeric constants and numeric primitives, one rule for the conditional, and so forth. The typing rules are therefore said to be *syntax-directed* in that the form of the expression determines the only possible typing rule that may apply to it. This observation leads to a useful lemma, called the *inversion lemma*, which states that the typing rules are necessary, as well as sufficient.

Theorem 12.1 (Inversion)
Suppose that $\Gamma \vdash e : \tau$.

1. *If* $e = x$, *then* $\Gamma = \Gamma', x : \tau$.

2. *If* $e = \texttt{num[}n\texttt{]}$, *then* $\tau = \texttt{nat}$.

3. *If* $e = \texttt{plus}(e_1,e_2)$ *or* $e = \texttt{times}(e_1,e_2)$*, then* $\tau = \texttt{nat}$ *and* $\Gamma \vdash e_1 : \texttt{nat}$ *and* $\Gamma \vdash e_2 : \texttt{nat}$.

4. *If* $e = \texttt{ifz}(e_0,e_1,e_2)$*, then* $\Gamma \vdash e_0 : \texttt{nat}$*,* $\Gamma \vdash e_1 : \tau$ *and* $\Gamma \vdash e_2 : \tau$*.*

5. *If* $e = \texttt{lambda}(\tau_1,x.e)$*, then* $\tau = \texttt{arrow}(\tau_1,\tau_2)$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$*.*

6. *If* $e = \texttt{app}(e_1,e_2)$*, then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \texttt{arrow}(\tau_2,\tau)$ *and* $\Gamma \vdash e_2 : \tau_2$*.*

7. *If* $e = \texttt{let}(\tau_1,e_1,x.e_2)$*, then* $\Gamma \vdash e_1 : \tau_1$*,* $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$*, and* $\tau = \tau_2$*.*

Proof: The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies, and that the premises of the rule in question provide the required result. ■

The following substitution property for typing follows immediately from the meaning of the hypothetical judgement.

Lemma 12.2
1. *If* $\Gamma, x : \tau \vdash e' : \tau'$*, and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [x \leftarrow e]e' : \tau'$*.*

12.4 Dynamic Semantics

The dynamic semantics of MinML is given by a transition system whose states are closed expressions. All states are initial, and the final states are the *values*, inductively defined by the following axioms:

$$\frac{}{\texttt{num}[n]\ \mathsf{value}} \qquad \frac{}{\texttt{lambda}(\tau,x.e)\ \mathsf{value}}$$

We often use the meta-variable v in situations where we expect v to be an expression such that v value.

The transition judgement is inductively defined by the following rules.

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(e_1,e_2) \longmapsto \texttt{plus}(e_1',e_2)} \qquad \frac{v_1\ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{plus}(v_1,e_2) \longmapsto \texttt{plus}(v_1,e_2')}$$

$$\frac{(n = n_1 + n_2)}{\texttt{plus}(\texttt{num}[n_1],\texttt{num}[n_2]) \longmapsto \texttt{num}[n]}$$

$$\frac{e_0 \longmapsto e_0'}{\texttt{ifz}(e_0,e_1,e_2) \longmapsto \texttt{ifz}(e_0',e_1,e_2)}$$

$$\frac{}{\texttt{ifz(num[0]},e_1,e_2\texttt{)} \longmapsto e_1} \qquad \frac{(n \neq 0)}{\texttt{ifz(num[}n\texttt{]},e_1,e_2\texttt{)} \longmapsto e_2}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{app}(e_1,e_2) \longmapsto \texttt{app}(e_1',e_2)} \qquad \frac{v_1\ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{app}(v_1,e_2) \longmapsto \texttt{app}(v_1,e_2')}$$

$$\frac{v\ \mathsf{value}}{\texttt{app(lambda(}\tau\texttt{,}x\texttt{.}e\texttt{),}v\texttt{)} \longmapsto [x\leftarrow v]e}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{let}(\tau_1,e_1,x.e_2) \longmapsto \texttt{let}(\tau_1,e_1',x.e_2)}$$

$$\frac{v_1\ \mathsf{value}}{\texttt{let}(\tau_1,v_1,x.e_2) \longmapsto [x\leftarrow v_1]e_2}$$

(The rules for multiplication are very similar to those for addition, and are omitted here.)

Observe that the argument of a function must be simplified to a value before the application can occur. This is called the *call-by-value* evaluation strategy for function applications. The alternative, known as *call-by-name* for historical reasons, is to pass the argument to the function in unevaluated form, so that it is evaluated only if its value is actually necessary to compute the value of the call. This can be more efficient, because the argument is not evaluated unless it is needed, but it can also be less efficient, because the argument is repeatedly evaluated on each use.

12.5 Basic Properties of the Dynamic Semantics

Let us prove that evaluation is *deterministic*, which implies that the value of any expression, if it has one, is uniquely determined by the expression alone. In other words, the transition judgement has mode $(\forall, \exists^{\leq 1})$.

Lemma 12.3
For every closed expression e, there exists at most one e' such that $e \longmapsto e'$. In other words, the relation $\longmapsto$ is a partial function.

Proof: By induction on the structure of e. For example, if $e = \texttt{app(}e_1\texttt{,}e_2\texttt{)}$, then by induction applied to e_1, there is at most one e_1' such that $e_1 \longmapsto e_1'$. If such a transition is possible, then $e \longmapsto \texttt{app(}e_1'\texttt{,}e_2\texttt{)}$, and this is the only possible transition. Otherwise, if e_1 is not a value, there is no transition from e. If e_1 is a value, then there is at most one transition $e_2 \longmapsto e_2'$. If there is such a transition, then $e \longmapsto \texttt{app(}e_1\texttt{,}e_2'\texttt{)}$, because e_1 value. If not, then e_2 may or may not be a value. If not, there is no transition from e. If so, there is at most one transition, according to whether or not e_1 is a function. The other cases are handled similarly. ■

12.6 Iteration and Recursion

So far MinML is extremely weak because it lacks any form of iteration or recursion, which is necessary even to express the most rudimentary computations on natural numbers. For example, not even the factorial function is definable without some additional machinery!

One extension, which corresponds to "for" loops in familiar programming languages, is to replace the `ifz` construct by a more general iteration construct that allows us to perform an operation n times for any natural number n. This is achieved using an *iterator*, whose abstract syntax is

$$\texttt{rec(}\tau\texttt{,}e_0\texttt{,}e_1\texttt{,}x\texttt{.}e_2\texttt{)},$$

and whose concrete syntax is

$$\texttt{rec}\, e_0 \,\{0 \Rightarrow e_1 \mid x+1 \Rightarrow e_2\}.$$

The meaning of this construct is explained informally as follows. Let n_0 be the value of e_0, a natural number. If n_0 is 0, then the result is the value of e_1. Otherwise, $n_0 = n_0' + 1$, and we recursively evaluate the same iterator on n_0', then substitute this result for x in e_2 to obtain the final result. Put in other terms, if $n \geq 0$, we iterate e_2 for n times, starting with e_1.

To make this precise, here is the typing rule for the iterator:

$$\frac{\Gamma \vdash e_0 : \texttt{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \texttt{rec(}\tau\texttt{,}e_0\texttt{,}e_1\texttt{,}x\texttt{.}e_2\texttt{)}}.$$

As with the conditional, both "branches" must have the same type, τ.

The dynamic semantics of the iterator is given by the following rules:

$$\frac{e_0 \longmapsto e_0'}{\texttt{rec}(\tau, e_0, e_1, x.e_2) \longmapsto \texttt{rec}(\tau, e_0', e_1, x.e_2)}$$

$$\frac{}{\texttt{rec}(\tau, \texttt{num}[0], e_1, x.e_2) \longmapsto e_1}$$

$$\frac{}{\texttt{rec}(\tau, \texttt{num}[n+1], e_1, x.e_2) \longmapsto \texttt{let}(\tau, \texttt{rec}(\tau, \texttt{num}[n], e_1, x.e_2), x.e_2)}$$

The use of the `let` in the third rule ensures that the recursive call is evaluated before it is passed to e_2.

A significantly more powerful extension is to admit *general recursive functions* into the language. These are functions that may "call themselves" recursively, on any argument we wish to pass. Such functions are not guaranteed to terminate, but this is the price we pay for generality. In order for a function to call itself, it must have a name for itself. This can be arranged by generalizing the abstract syntax of a function to have a name for "itself":

$$\texttt{fun}(\tau_1, \tau_2, f.x.e).$$

The concrete syntax for this form is

$$\texttt{fun}\, f\, (x : \tau_1) : \tau_2 \;\texttt{is}\; e.$$

The variable f stands for the function itself, and the variable x stands for its argument. If the function e does not call itself, then the name f is superfluous, and may be omitted by writing "_" in place of f.

The static semantics for recursive functions is defined so that the function may call itself under the assumption that it has the type it will turn out to have, namely $\texttt{arrow}(\tau_1, \tau_2)$:

$$\frac{\Gamma, f : \texttt{arrow}(\tau_1, \tau_2), x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e) : \texttt{arrow}(\tau_1, \tau_2)}$$

Notice that the typing rule is seemingly "circular" in that the assumption governing f states that it has the type of the function itself, which is checked by asserting that the body has type τ_2 under this assumption and the additional assumption that the argument has type τ_1.

The dynamic semantics for function applications changes so as to replace the name, f, of the function itself by the function itself, thereby "tying the knot" in the recursion:

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value}}{\mathtt{app}(v_1, v_2) \longmapsto [f, x \leftarrow v_1, v_2]e}\ (v_1 = \mathtt{fun}(\tau_1, \tau_2, f\,.\,x\,.\,e))$$

The use of a variable to stand for the function itself is a common "trick" in programming languages. For example, in Java the identifier `this` stands for the object itself in exactly the same sense.

12.7 Exercises

1. Show that the mode of the typing judgement is $(\forall, \exists^{\leq 1})$ — for every (closed) expression there is at most one type for it. To prove this you must generalize the induction hypothesis to account for open expressions.

2. Formulate the *call-by-name* evaluation strategy for function applications in which arguments are passed unevaluated to functions.

3. Show that $\mathtt{let}(\tau_1, e_1, x\,.\,e_2)$ is *definable* in the sense that there is a translation of this construct in terms of the other constructs in the language such that its typing rule is derivable under this translation.

4. Define the *evaluation judgement* $e \Downarrow v$, where e is a closed expression and e value, and show that $e \Downarrow v$ iff $e \overset{!}{\longmapsto} v$.

5. Show that the conditional $\mathtt{ifz}(e_0, e_1, e_2)$ is definable in terms of the iterator.

6. Show that addition and multiplication are definable in terms of the iterator by giving a term $e : \mathtt{arrow(nat,arrow(nat,nat))}$ that implements these two arithmetic operations.

7. Show that the predecessor is definable in terms of the iterator, provided that we define the predecessor of 0 to be 0 and the predecessor of $n + 1$ to be n.

8. Investigate the trade-offs between the call-by-name and call-by-value evaluation strategies for function applications.

Chapter 13

Type Safety for MinML

We will use the methodology described in Chapter 11 to prove the type safety of MinML. As discussed there, type safety is the combination of two key relationships between the static and dynamic semantics of the language, *preservation* and *progress*.

13.1 Safety for MinML

Theorem 13.1 (Preservation)
If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.

Proof: Note that we are proving not only that e' is well-typed, but that it has the same type as e. The proof is by rule induction on the definition of one-step evaluation. We will consider each rule in turn.

Consider the rule

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(e_1, e_2) \longmapsto \texttt{plus}(e_1', e_2)}.$$

Assume that $\texttt{plus}(e_1, e_2) : \tau$. By inversion $\tau = \texttt{nat}$, $e_1 : \texttt{nat}$, and $e_2 : \texttt{nat}$. By induction $e_1' : \texttt{nat}$, and hence $\texttt{plus}(e_1', e_2) : \texttt{nat}$, as was to be shown.

Consider the rule

$$\frac{v_1 \ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{plus}(v_1, e_2) \longmapsto \texttt{plus}(v_1, e_2')}.$$

Suppose that $\texttt{plus}(v_1, e_2) : \tau$. Then, by inversion, $\tau = \texttt{nat}$, $v_1 : \texttt{nat}$, and $e_2 : \texttt{nat}$. By induction $e_2' : \texttt{nat}$, and hence $\texttt{plus}(v_1, e_2') : \texttt{nat}$, as required.

Consider the rule

$$\frac{(n = n_1 + n_2)}{\texttt{plus}(\texttt{num[}n_1\texttt{]},\texttt{num[}n_2\texttt{]}) \longmapsto \texttt{num[}n\texttt{]}.}$$

Assume that $\texttt{plus}(\texttt{num[}n_1\texttt{]},\texttt{num[}n_2\texttt{]}) : \tau$. Clearly, $\texttt{num[}n\texttt{]} : \texttt{nat}$, which is enough for the result.

The rules governing multiplication are handled similarly.

Consider the rule

$$\frac{e_0 \longmapsto e_0'}{\texttt{ifz(}e_0\texttt{,}e_1\texttt{,}e_2\texttt{)} \longmapsto \texttt{ifz(}e_0'\texttt{,}e_1\texttt{,}e_2\texttt{)}.}$$

Suppose that $\texttt{ifz(}e_0\texttt{,}e_1\texttt{,}e_2\texttt{)} : \tau$. By inversion, $e_0 : \texttt{nat}$, $e_1 : \tau$, and $e_2 : \tau$. By induction $e_0' : \texttt{nat}$, and hence $\texttt{ifz(}e_0'\texttt{,}e_1\texttt{,}e_2\texttt{)} : \tau$, as required.

The other rules governing the conditional test are handled similarly.

Consider the rule

$$\frac{v \ \mathsf{value}}{\texttt{app(lambda(}\tau\texttt{,}x\texttt{.}e\texttt{),}v\texttt{)} \longmapsto [x{\leftarrow}v]e.}$$

Suppose that $\texttt{app(lambda(}\tau\texttt{,}x\texttt{.}e\texttt{),}v\texttt{)} : \tau'$. By inversion $v : \tau$ and $x : \tau \vdash e : \tau'$. By substitution $[x{\leftarrow}v]e : \tau'$.

The other two rules governing application are handled similarly to the rules for the arithetic operations. The rules for the `let` construct are left to the reader. ■

A critical ingredient in the safety proof is the canonical forms lemma, which characterizes the form of values of a given type.

Lemma 13.2 (Canonical Forms)
Suppose that $v : \tau$ is a closed, well-formed value.

1. *If $\tau = \texttt{nat}$, then $v = \texttt{num[}n\texttt{]}$ for some n nat.*

2. *If $\tau = \texttt{arrow(}\tau_1\texttt{,}\tau_2\texttt{)}$, then $v = \texttt{lambda(}\tau_1\texttt{,}x\texttt{.}e\texttt{)}$ for some x and e such that $x : \tau_1 \vdash e : \tau_2$.*

Proof: By induction on the typing rules, using the assumption v value. ■

Theorem 13.3 (Progress)
If $e : \tau$, then either e is a value, or there exists e' such that $e \longmapsto e'$.

Proof: The proof is by rule induction on the definition of the typing judgement.

Consider the rule
$$\frac{}{\Gamma \vdash \mathtt{num}[n] : \mathtt{nat}}.$$

By definition $\mathtt{num}[n]$ value, which is sufficient for the conclusion.

Consider the rule
$$\frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{plus}(e_1, e_2) : \mathtt{nat}}.$$

By induction either e_1 value or there exists e_1' such that $e_1 \longmapsto e_1'$. In the latter case we have $\mathtt{plus}(e_1, e_2) \longmapsto \mathtt{plus}(e_1', e_2)$. In the former, we have by induction that either e_2 value or there exists e_2' such that $e_2 \longmapsto e_2'$. In the latter case we have $\mathtt{plus}(v_1, e_2) \longmapsto \mathtt{plus}(v_1, e_2')$. In the former we appeal to the canonical forms lemma (twice) to obtain that $v_1 = \mathtt{num}[n_1]$ for some n_1 nat and $v_2 = \mathtt{num}[n_2]$ for some n_2 nat. But then $\mathtt{plus}(v_1, v_2) \longmapsto \mathtt{num}[n]$, where $n = n_1 + n_2$ nat, as required.

Consider the rule
$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{ifz}(e, e_1, e_2) : \tau}.$$

By induction either e value or there exists e' such that $e \longmapsto e'$. In the latter case $\mathtt{ifz}(e, e_1, e_2) \longmapsto \mathtt{ifz}(e', e_1, e_2)$. In the former we have by the canonical forms lemma that $e = \mathtt{num}[n]$ for some n nat. If $n = \mathtt{zero}$, then $\mathtt{ifz}(e, e_1, e_2) \longmapsto e_1$, otherwise $\mathtt{ifz}(e, e_1, e_2) \longmapsto e_2$.

Consider the rule
$$\frac{\Gamma \vdash e_1 : \mathtt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{app}(e_1, e_2) : \tau}.$$

By induction either e_1 value or $e_1 \longmapsto e_1'$. In the latter case we have $\mathtt{app}(e_1, e_2) \longmapsto \mathtt{app}(e_1', e_2)$. Otherwise we have by induction either e_2 value or $e_2 \longmapsto e_2'$. In the latter case we have $\mathtt{app}(e_1, e_2) \longmapsto \mathtt{app}(e_1, e_2')$ (bearing in mind e_1 value). Otherwise, by the canonical forms lemma $e_1 = \mathtt{lambda}(\tau_2, x.e)$ for some x and e. But then $\mathtt{app}(e_1, e_2) \longmapsto [x \leftarrow e_2]e$, again bearing in mind that e_2 value.

The remaining cases are left to the reader. ■

13.2 Run-Time Errors and Safety

Type safety for MinML ensures that "stuck" states (those from which no transition is possible, yet are not values) are always ill-typed. But suppose that we wish to extend MinML with, say, a quotient operation that is undefined in some situations. For example, 3/0, being undefined, would be "stuck", yet is not a value and *is* well-typed — provided that we use the following typing rule for quotient.

$$\frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{div}(e_1,e_2) : \mathtt{nat}} .$$

What are we to make of this? Is the extension of MinML with quotient unsafe?

To ensure safety of MinML extended with quotients we have two options:

1. *Enhance the type system* so that no well-typed program can ever divide by zero.

2. *Modify the dynamic semantics* so that division by zero is not regarded as "stuck", but rather as a *checked error*.

Either option is, in principle, viable, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs. For now we consider the second option, which is widely used.

The general idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamic semantics of MinML need not check, when performing an addition, that its two arguments are, in fact, natural numbers, as opposed to, say, functions, because the type system ensures that this is the case. On the other hand the dynamic semantics for quotient *must* check for a zero denominator, because the type system does not rule out this possibility.

This may be achieved by adding to the language a new construct, `error`, which signals the occurrence of a checked error. The typing rule for a checked error permits it to be regarded as having any type at all:

$$\frac{}{\Gamma \vdash \texttt{error} : \tau} \tag{13.1}$$

To ensure that this is safe requires that we augment the dynamic semantics with rules that *propagate* errors — once an error arises, it aborts the entire computation.

Partially defined operations, such as quotient, give rise to errors:

$$\frac{}{\texttt{div(}v_1\texttt{,num[0])} \longmapsto \texttt{error}.} \tag{13.2}$$

Once an error arises, it propagates through all other constructs. For example, we add the following rules to the definition of the transition relation for MinML:

$$\frac{}{\texttt{app(error,}e_2\texttt{)} \longmapsto \texttt{error}} \tag{13.3}$$

$$\frac{}{\texttt{app(}v_1\texttt{,error)} \longmapsto \texttt{error}} \tag{13.4}$$

Similar rules propagate errors through the other constructs of the language.

The preservation theorem remains the same, and is proved similarly, bearing in mind that `error` has any type we like. The progress theorem must be modified as follows:

Theorem 13.4 (Progress With Error)
If $e : \tau$, then either $e = \texttt{error}$ or e value or there exists e' such that $e \longmapsto e'$.

Proof: The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof. ■

13.3 Exercises

1. Complete the proof of preservation and progress for MinML.

2. Complete the proof of progress for MinML extended with checked errors.

Chapter 14

Environments and Functions

In Chapter 10 we introduced the concept of an *environment semantics*, in which substitution is avoided in favor of maintaining a list of hypotheses specifying the bindings of the free variables of an expression. This corresponds more closely to practical implementations, which associate bindings to variables during execution, rather than perform substitution.

In this chapter we investigate the extensions of environment semantics to the functional language MinML. This extension is non-trivial, and is, in fact, the source of an infamous error in language design!

14.1 Environment Semantics for MinML

The environment semantics for expressions given in Chapter 10 is based on hypothetical judgements of the form

$$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$$

stating that the expression e evaluates to the value v, under the assumption that the variables x_i evaluate to v_i.

Let us naïvely extend this semantics to MinML using the following rules for functions and applications (the other rules are similar to those for expressions):

$$\frac{}{\eta \vdash \texttt{lambda}(\tau, x.e) \Downarrow \texttt{lambda}(\tau, x.e)}$$

$$\frac{\eta \vdash e_1 \Downarrow \texttt{lambda(}\tau\texttt{,}x\texttt{.}e\texttt{)} \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta, x \Downarrow v_2 \vdash e \Downarrow v}{\eta \vdash \texttt{app(}e_1\texttt{,}e_2\texttt{)} \Downarrow v}$$

The idea is that when applying a function to an argument, we bind the parameter of the function to the argument value, and proceed to evaluate the body of the function under the influence of that binding. The resulting value is the value of the application.

Superficially this formulation looks fine, but it is, in fact, incorrect! Consider the following example,

$$\texttt{let f be (}\lambda\,\texttt{x:nat. }\lambda\,\texttt{y:nat.x) 3 in let x be 5 in f x,}$$

which we have written using an informal concrete syntax for the sake of readability.

According to the environment semantics of MinML, evaluation of this expression proceeds by evaluating the binding of `f`, then binding this value to `f` for use within the body. The binding of `f` is an application, which is evaluated by binding `x` to `3`, and evaluating the body of the function. This is itself a function, λ `y:nat.x`, which is yielded as result. Then `f` is bound to this function, and evaluation proceeds with the inner `let`. The variable `x` is then bound to `5`, and the application `f x` is evaluated. After obtaining the bindings for `f` and `x`, we proceed by evaluating the body, `x`, in the extension of the current environment in which `y` is bound to `5`. The result, `5`, is the overall result of evaluation.

But now let us evaluate this same expression using the substitution semantics. First, we evaluate the binding of `f`. This is obtained by evaluating the application of λ `x:nat.`λ `y:nat.x` to the argument `3`, which obtained by substituting `3` for `x` in the body, obtaining λ `y:nat.3`. This function is substituted for `f` in the inner `let`, which is evaluated by substituting `5` for `x` in the application, obtaining `(`λ `y:nat.3) 5`, whose value is `3`.

What went wrong? The problem is that the environment semantics confuses the two distinct occurrences of the variable `x` in the program. When evaluating the binding for `f`, the environment semantics binds `x` to `3`, and evaluates λ `y:nat.x`, which is returned as a value — with the variable `x` occurring freely within it. Later on, the inner `let` binds the variable `x` to `5`, and this binding governs the evaluation of the body of `f`, inadvertently confusing the two variables.

Another way to see the problem is to consider the following α-equivalent expression in which we have renamed the inner `x` to `z`:

```
let f be (λx:nat. λy:nat.x)3 in let z be 5 in fz,
```

This should not change the meaning of the expression, yet when evaluated using the environment semantics, the evaluation process "gets stuck" because it lacks a binding for `x` at the point where the application `f z` is evaluated! The substitution semantics encounters no such difficulties, and properly assigns this expression the value `3`, as it should.

The confusion of bindings incurred by the evaluation semantics in this example is sometimes called *dynamic binding*. The idea is that the binding for `x` used during evaluation of the innermost application is the *dynamically most recent* binding for `x` in the environment. However, as we have just seen, this policy fails to respect α-equivalence (renamng of bound variables), and does not agree with the substitution semantics. It is therefore *wrong*, and must be corrected.[1]

The *correct* treatment of variables is, by contrast, called *static binding*. Static binding is simply the discipline of assigning binding sites to variables based on their textual occurrences, independently of any dynamic execution model, as we have detailed in Chapter 6. The problem, therefore, lies not with the concepts of binding and scope, but with the evaluation semantics itself.

14.2 Closures

To avoid these difficulties we must ensure that the free variables of a function are not detached from their environment. The main idea is to regard the environment as an *explicit substitution*, a data structure that records what is to be substituted for a variable without actually doing it. Only when the variable is encountered do we replace it by its binding in the environment, effectively *delaying* substitution as long as possible. To avoid the confusions described in the preceding section, we attach the environment to a λ-abstraction at the point where the abstraction is evaluated, resulting in a configuration of the form

$$\texttt{clos}(\eta,\texttt{lambda}(\tau,x.e)),$$

[1]Historically, this error was introduced in the very first implementation of Lisp, and was later diagnosed as a mistake by McCarthy. Coining the phrase *dynamic binding* is, to this author's mind, simply an attempt to turn the bug into a feature by giving it a name!

which is called a *closure*. The idea is that the environment "closes" the free variables of the λ-abstraction by providing bindings for them. These are the bindings that are used when the function body is evaluated, not those in the ambient environment at the point of application.

To give a proper environment semantics for MinML we introduce two new syntactic categories, *values* and *environments*.

$$\begin{array}{lll} \textit{Values} & v & ::= \texttt{num}[n] \mid \texttt{clos}(\eta,\texttt{lambda}(\tau,x.e)) \\ \textit{Env's} & \eta & ::= \bullet \mid \eta, x{=}v \end{array}$$

In this setting values are no longer forms of expression, but are instead a special syntactic category of their own.

The environment semantics for MinML is re-formulated to determine the value, in the sense of the preceding grammar, for each expression. The key changes are exemplified by the following two rules:

$$\frac{}{\eta \vdash \texttt{lambda}(\tau,x.e) \Downarrow \texttt{clos}(\eta,\texttt{lambda}(\tau,x.e))}$$

$$\frac{\eta \vdash e_1 \Downarrow \texttt{clos}(\eta',\texttt{lambda}(\tau,x.e)) \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta',x{=}v_2 \vdash e \Downarrow v}{\eta \vdash \texttt{app}(e_1,e_2) \Downarrow v}$$

Notice that we *switch environments* from η, the ambient environment, to η', the environment of the closure, when evaluating the function body. This ensures that the free variables of the body are governed by the environment in effect at the point where the function is created, not at the point where the function is applied.

To characterize the well-formed values we enrich the static semantics with rules for (closed) values. The typing rule for closures is as follows:

$$\frac{\eta : \Gamma' \quad \Gamma', x : \tau_1 \vdash e : \tau_2}{\texttt{clos}(\eta,\texttt{lambda}(\tau_1,x.e)) : \texttt{arrow}(\tau_1,\tau_2)}$$

Notice that the body of the function may have any number of free variables, which are governed by the hypotheses Γ'.

The typing rule for closures makes use of the following typing rules for environments:

$$\frac{}{\bullet : \emptyset} \qquad \frac{v : \tau \quad \eta : \Gamma'}{\eta, x{=}v : \Gamma', x : \tau}$$

The judgement $\eta : \Gamma'$ means that the environment η provides bindings for the variables in Γ'. The bindings are all closed values of appropriate type, as determined by Γ'.

The environment semantics may be proved equivalent to the substitution semantics using a technical device that "expands out" the delayed substitutions occurring in closures. This operation is inductively defined as follows:

$$\frac{}{\texttt{num}[n]^{*}\texttt{=}\texttt{num}[n]} \qquad \frac{[\eta]\,e\texttt{=}e'}{\texttt{clos}(\eta\texttt{,}\texttt{lambda}(\tau\texttt{,}x\texttt{.}e))^{*}\texttt{=}\texttt{lambda}(\tau\texttt{,}x\texttt{.}e')}$$

The $[\eta]\,e$ stands for the result of the simultaneous substitution aof η into e, as defined in Chapter 6.

The (corrected) environment and substitution semantics are equivalent.

Theorem 14.1 (Equivalence)
$\eta \vdash e \Downarrow v$ iff $[\eta]\,e \Downarrow v^{*}$.

14.3 Exercises

1. Complete the definition of the environment semantics for MinML.

2. Prove that if $\Gamma \vdash e : \tau$, $\emptyset \vdash \eta : \Gamma$, and $\eta \vdash e \Downarrow v$, then $\emptyset \vdash v : \tau$.

3. Prove the equivalence theorem.

4. Re-formulate the transition semantics for MinML in terms of environments. What difficulties do you encounter? How might they be overcome?

Part V

Products and Sums

Chapter 15

Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated eliminatory forms are *projections*, which select the first and second component of a pair. The *nullary product*, or *unit*, type consists solely of the unique "null tuple" of no values, and has no associated eliminatory form.

More generally, the *general*, or *n-ary*, *product* of $n \geq 0$ types consists of the *ordered n-tuples* of values, with the eliminatory forms being the ith projection, where $0 \leq i < n$.

The *labelled product*, or *record*, type consists of *labelled n-tuples* in which the components are *labelled* by names. The eliminatory forms access the field of a specified name.

15.1 Nullary and Binary Products

Let us extend the abstract syntax of MinML with the following constructs:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{unit} \mid \texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)} \\ \textit{Expr's} & e & ::= & \texttt{unit} \mid \texttt{pair(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{fst(}e\texttt{)} \mid \texttt{snd(}e\texttt{)} \end{array}$$

In examples we write $\tau_1 \times \tau_2$ for $\texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)}$, $\langle e_1\texttt{,}e_2\rangle$ for $\texttt{pair(}e_1\texttt{,}e_2\texttt{)}$ and $\langle\rangle$ for `unit`.

The type $\texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)}$ is sometimes called the *binary product* of the types τ_1 and τ_2, and the type `unit` is correspondingly called the *nullary product* (of no types). We sometimes speak loosely of *product types* in such as way as to cover both the binary and nullary cases.

The introductory form for the product type is called *pairing*, and its eliminatory forms are called *projections*. For the unit type the introductory form is called the *unit object*, or *null tuple*. There is no eliminatory form, there being nothing to extract from a null tuple!

The static semantics of product types is given by the following rules:

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair(}e_1\texttt{,}e_2\texttt{)} : \texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)}}$$

$$\frac{\Gamma \vdash e : \texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)}}{\Gamma \vdash \texttt{fst(}e\texttt{)} : \tau_1} \qquad \frac{\Gamma \vdash e : \texttt{prod(}\tau_1\texttt{,}\tau_2\texttt{)}}{\Gamma \vdash \texttt{snd(}e\texttt{)} : \tau_2}$$

There are two different dynamic semantics for product types, the *eager* semantics, and the *lazy* semantics. The eager semantics is specified by the following rules.

$$\frac{}{\texttt{unit}\ \mathsf{value}} \qquad \frac{e_1\ \mathsf{value} \quad e_2\ \mathsf{value}}{\texttt{pair(}e_1\texttt{,}e_2\texttt{)}\ \mathsf{value}}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{pair(}e_1\texttt{,}e_2\texttt{)} \longmapsto \texttt{pair(}e_1'\texttt{,}e_2\texttt{)}} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \longmapsto e_2'}{\texttt{pair(}e_1\texttt{,}e_2\texttt{)} \longmapsto \texttt{pair(}e_1\texttt{,}e_2'\texttt{)}}$$

$$\frac{e \longmapsto e'}{\texttt{fst(}e\texttt{)} \longmapsto \texttt{fst(}e'\texttt{)}} \qquad \frac{e \longmapsto e'}{\texttt{snd(}e\texttt{)} \longmapsto \texttt{snd(}e'\texttt{)}}$$

$$\frac{e_1\ \mathsf{value} \quad e_2\ \mathsf{value}}{\texttt{fst(pair(}e_1\texttt{,}e_2\texttt{))} \longmapsto e_1} \qquad \frac{e_1\ \mathsf{value} \quad e_2\ \mathsf{value}}{\texttt{snd(pair(}e_1\texttt{,}e_2\texttt{))} \longmapsto e_2}$$

According to these rules a pair $\texttt{pair(}e_1\texttt{,}e_2\texttt{)}$ is a value only if both e_1 and e_2 are values. Evaluation of a projection necessarily implies evaluation of its argument to determine what pair to project from.

The lazy semantics for tuples is specified by the following rules:

$$\frac{}{\texttt{unit}\ \mathsf{value}} \qquad \frac{}{\texttt{pair(}e_1\texttt{,}e_2\texttt{)}\ \mathsf{value}}$$

$$\frac{e \longmapsto e'}{\texttt{fst(}e\texttt{)} \longmapsto \texttt{fst(}e'\texttt{)}} \qquad \frac{e \longmapsto e'}{\texttt{snd(}e\texttt{)} \longmapsto \texttt{snd(}e'\texttt{)}}$$

$$\frac{}{\texttt{fst(pair(}e_1\texttt{,}e_2\texttt{))} \longmapsto e_1} \qquad \frac{}{\texttt{snd(pair(}e_1\texttt{,}e_2\texttt{))} \longmapsto e_2}$$

According to these rules any ordered pair is a value, regardless of whether its components are values. Therefore there are no "search" rules for evaluating the components of a projection.

Theorem 15.1 (Safety)
Under either the lazy or the eager semantics of pairing,

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.

2. *If* $e : \tau$ *then either* e value *or there exists* e' *such that* $e \longmapsto e'$.

Proof: The proof is left as an exercise to the reader. ■

15.2 General Products

The syntax of general product types is given by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{tpl}(\tau_0,\ldots,\tau_{n-1}) \\ \textit{Expr's} & e & ::= & \texttt{tpl}(e_0,\ldots,e_{n-1}) \mid \texttt{prj}(i,e) \end{array}$$

Formally, this grammar is indexed by the size, n, of the general product type under consideration. In addition the projections are indexed by a natural number constant, $0 \leq i < n$, indicating the position to select from the n-tuple. The re-use of the operator `tpl` for both a type constructor and a term constructor should cause no confusion, but formally there are two operators of arity n, one for forming types, the other for forming expressions.

We may either take these constructs as primitives, treating products as special cases, or define these constructs in terms of products, as follows:

$$\texttt{tpl}(\tau_0,\ldots,\tau_{n-1}) = \begin{cases} \texttt{unit} & \textit{if } n = 0 \\ \texttt{prod}(\tau_0,\texttt{tpl}(\tau_1,\ldots,\tau_{n-1})) & \textit{if } n > 0 \end{cases}$$

$$\texttt{tpl}(e_0,\ldots,e_{n-1}) = \begin{cases} \texttt{unit} & \textit{if } n = 0 \\ \texttt{pair}(e_0,\texttt{tpl}(e_1,\ldots,e_{n-1})) & \textit{if } n > 1 \end{cases}$$

$$\texttt{prj}(j,e) = \begin{cases} \texttt{fst}(e) & \textit{if } j = 0 \\ \texttt{prj}(j-1,\texttt{snd}(e)) & \textit{if } j > 0 \end{cases}$$

These definitions are a bit tricky. The definitions of the n-ary product type and the n-tuple expression are defined for $n > 0$ in terms of their definition for $n - 1$. Moreover, the projections are further parameterized by a constant $0 \leq i < n$ indicating the position to project; these are defined for $i > 0$ in terms of their definitions for $i - 1$.

We leave it to the reader to derive the static and dynamic semantics for general product types implied by these definitions.

15.3 Labelled Products

Labelled product, or *record*, types are a useful generalization of product types in which the components are accessed by name, rather than by position. The benefits of this should be clear: one cannot be expected to remember the intended meaning of the 7th component of a 13-tuple!

The syntax for records is quite similar to that for n-tuples:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{rcd}(l_0,\tau_0,\ldots,l_{n-1},\tau_{n-1}) \\ \textit{Expr's} & e & ::= & \texttt{rcd}(l_0,e_0,\ldots,l_{n-1},e_{n-1}) \mid \texttt{prj}(l,e) \end{array}$$

We use the meta-variable l to range over *labels*, an infinite set of names disjoint from variable names. In concrete syntax one often writes

$$\{l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\}$$

for record types,

$$\{l_0 = e_0, \ldots, l_{n-1} = e_{n-1}\}$$

for record expressions, and $e \cdot l$ for field selection.

The components of a record are called *fields*. Each field has a label, called the *field name*, and a type. The components are accessed by field name. We tacitly assume that no two fields are given the same name, and that the order of fields in a record type is irrelevant. That is, we treat as equal any two record types that differ only in the ordering of their fields. That is, the order of fields is considered to be a matter of presentation, and not of substance. This makes sense because the fields are accessed by name, and not by position.

The static semantics of records is given by the following rules:

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \cdots \quad \Gamma \vdash e_{n-1} : \tau_{n-1}}{\Gamma \vdash \texttt{rcd}(l_0,e_0,\ldots,l_{n-1},e_{n-1}) : \texttt{rcd}(l_0,\tau_0,\ldots,l_{n-1},\tau_{n-1})}$$

$$\frac{\Gamma \vdash e : \texttt{rcd}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1})}{\Gamma \vdash \texttt{prj}(l_i, e) : \tau_i}$$

An eager dynamic semantics is specified by these rules:

$$\frac{e_0 \ \mathsf{value} \quad \cdots \quad e_{i-1} \ \mathsf{value} \quad e_i \longmapsto e_i'}{\texttt{rcd}(l_0, e_0, \ldots, l_i, e_i, \ldots, l_{n-1}, e_{n-1}) \longmapsto \texttt{rcd}(l_0, e_0, \ldots, l_i, e_i', \ldots, l_{n-1}, e_{n-1})}$$

$$\frac{e \longmapsto e'}{\texttt{prj}(l, e) \longmapsto \texttt{prj}(l, e')} \qquad \frac{e_0 \ \mathsf{value} \quad \cdots \quad e_{n-1} \ \mathsf{value}}{\texttt{prj}(l_i, \texttt{rcd}(l_0, e_0, \ldots, l_{n-1}, e_{n-1})) \longmapsto e_i}$$

Following the pattern for products, we may also formulate a lazy semantics for records.

Theorem 15.2 (Safety for Records)

1. *If $e : \tau$ and $e \longmapsto e'$, then $e : \tau'$.*

2. *If $e : \tau$, then either e value or $e \longmapsto e'$ for some e'.*

15.4 Exercises

1. State and prove the canonical forms lemma for unit and product types under the eager and under the lazy semantics.

2. Prove the safety theorem for unit and product types under either the eager or the lazy semantics.

3. State the static and dynamic semantics for general products implied by the definitions given in Section 15.1.

4. Functional update, concatenation, restriction, other record operations?

Chapter 16

Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by *sum types*, specifically the *binary sum*, which offers a choice of two things, and the *nullary sum*, which offers a choice of no things. These generalize to n-ary sums, a choice among n things, and to *labelled sums*, in which the selection is governed by a label.

16.1 Binary and Nullary Sums

Let us consider the extension of MinML with nullary and binary sums according to the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{void} \mid \texttt{sum}(\tau_1,\tau_2) \\ \textit{Expr's} & e & ::= & \texttt{abort}(\tau,e) \mid \texttt{inl}(\tau,e) \mid \texttt{inr}(\tau,e) \mid \texttt{case}(e,\tau_1,x_1.e_1,\tau_2,x_2.e_2) \end{array}$$

The concrete syntax for sum types is $\tau_1 + \tau_2$, and for case expressions is

$$\texttt{case}\, e \,\{\, \texttt{inl}(x_1{:}\tau_1) \Rightarrow e_1 \mid \texttt{inr}(x_2{:}\tau_2) \Rightarrow e_2 \,\}.$$

The type `void` is the *nullary sum* type, whose values are selected from a choice of zero alternatives — there are no values of this type, and so no introductory forms. The eliminatory form, $\texttt{abort}(\tau,e)$, aborts the computation in the event that e evaluates to a value, which it cannot. The type

$\tau = \texttt{sum}(\tau_1,\tau_2)$ is the *binary sum*. Its introductory forms have the form $\texttt{inl}(\tau,e)$ or $\texttt{inl}(\tau,e)$, indicating which of the two possible choices by *tagging* a value of the left or right summand as being a value of the sum type. The eliminatory form performs a case analysis on the tag of a value, decomposing it into its constituent parts.

The static semantics of sum types is given by the following rules:

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}(\tau,e) : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum}(\tau_1,\tau_2)}{\Gamma \vdash \texttt{inl}(\tau,e) : \tau} \qquad \frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum}(\tau_1,\tau_2)}{\Gamma \vdash \texttt{inr}(\tau,e) : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{sum}(\tau_1,\tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e,\tau_1,x_1\,.\,e_1,\tau_2,x_2\,.\,e_2) : \tau}$$

Just as for the conditional expression considered in Chapter 12, both branches of the case analysis must have the same type. Since the type expresses a static "prediction" on the form of the value of an expression, and since a value of sum type could evaluate to either form at run-time, we must insist that both branches yield the same type.

Just as with products, there are two forms of dynamic semantics, the *eager* form and the *lazy*. These differ according to whether the argument to an injection is evaluated at the point the injection is evaluated, or only when, if ever, that underlying value is used. We will give here the eager semantics, and leave the lazy semantics to the reader.

$$\frac{e \ \mathsf{value}}{\texttt{inl}(\tau,e) \ \mathsf{value}} \qquad \frac{e \ \mathsf{value}}{\texttt{inl}(\tau,e) \ \mathsf{value}}$$

$$\frac{e \longmapsto e'}{\texttt{inl}(\tau,e) \longmapsto \texttt{inl}(\tau,e')} \qquad \frac{e \longmapsto e'}{\texttt{inr}(\tau,e) \longmapsto \texttt{inr}(\tau,e')}$$

$$\frac{e \longmapsto e'}{\texttt{case}(e,\tau_1,x_1\,.\,e_1,\tau_2,x_2\,.\,e_2) \longmapsto \texttt{case}(e',\tau_1,x_1\,.\,e_1,\tau_2,x_2\,.\,e_2)}$$

$$\frac{e \ \mathsf{value}}{\texttt{case}(\texttt{inl}(\tau,e),\tau_1,x_1\,.\,e_1,\tau_2,x_2\,.\,e_2) \longmapsto [x_1 \leftarrow e]e_1}$$

$$\frac{e \ \mathsf{value}}{\texttt{case}(\texttt{inr}(\tau,e),\tau_1,x_1\,.\,e_1,\tau_2,x_2\,.\,e_2) \longmapsto [x_2 \leftarrow e]e_2}$$

The coherence of the static and dynamic semantics is stated and proved as usual.

Theorem 16.1 (Safety)

1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

2. *If $e : \tau$, then either e value or $e \longmapsto e'$ for some e'.*

One use of sum types is to define the *Boolean* type, which has the following syntax:

$$\begin{array}{lcll} \textit{Types} & ::= & \tau & \texttt{bool} \\ \textit{Expr's} & ::= & e & \texttt{tt} \mid \texttt{ff} \mid \texttt{if}(e,e_1,e_2) \end{array}$$

This type is definable in the presence of sums and nullary products according to the following equations:

$$\begin{array}{rcl} \texttt{bool} & = & \texttt{sum(unit,unit)} \\ \texttt{tt} & = & \texttt{inl(bool,unit)} \\ \texttt{ff} & = & \texttt{inr(bool,unit)} \\ \texttt{if}(e,e_1,e_2) & = & \texttt{case}(e,\texttt{unit},x_1\,.\,e_1,\texttt{unit},x_2\,.\,e_2) \end{array}$$

The variables x_1 and x_2 are dummies, since their type, `unit`, determines their value, `unit`, and, moreover, they do not occur freely in e_1 or e_2.

Another use of sums is to define the *option* types, which have the following syntax:

$$\begin{array}{llcl} \textit{Types} & \tau & ::= & \texttt{option}(\tau) \\ \textit{Expr's} & e & ::= & \texttt{nothing} \mid \texttt{just}(e) \mid \texttt{optcase}(\tau,e,e_1,x\,.\,e_2) \end{array}$$

The type $\texttt{option}(\tau)$ represents the type of "optional" values of type τ. The introductory forms are `nothing`, corresponding to "no value", and $\texttt{just}(e)$, corresponding to a specified value of type τ. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:

$$\begin{array}{rcl} \texttt{option}(\tau) & = & \texttt{sum(unit,}\tau\texttt{)} \\ \texttt{nothing} & = & \texttt{inl(option(}\tau\texttt{),unit)} \\ \texttt{just}(e) & = & \texttt{inr(option(}\tau\texttt{),}e\texttt{)} \\ \texttt{optcase}(\tau,e,e_1,x_2\,.\,e_2) & = & \texttt{case}(e,\texttt{unit},x_1\,.\,e_1,\tau,x_2\,.\,e_2) \end{array}$$

We leave it to the reader to examine the static and dynamic semantics implied by these definitions.

It is important to understand the difference between the types `unit` and `void`, which are often confused. The type `unit` has exactly one element, `unit`, whereas the type `void` has no elements at all. Consequently, if e : `unit`, then if e evaluates to a value, it must be `unit` — in other words, e has *no interesting value* (but it could diverge). On the other hand, if e : `void`, then *e must diverge*, because if it were to have a value, it would have to be a value of type `void`, of which there are none. This shows that the `void` type in Java and related languages is really the type `unit`, because it indicates that an expression of that type has no interesting result, not that it must diverge!

16.2 Labelled Sums

Binary and nullary sums are sufficient to define generalized n-ary sums, in a manner analogous to the definition of n-ary products from nullary and binary products in Chapter 15. We leave the details of this derivation to the reader, and concentrate instead on *labelled sums*, or *labelled variants*. Labelled sums are a form of n-ary sum in which the alternatives are labelled by names, rather than by positions.

The syntax of labelled sums is given by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \\ \textit{Expr's} & e & ::= & \texttt{inj}(\tau, l, e) \mid \texttt{case}(e, l_0, \tau_0, x_0 . e_0, \ldots, l_{n-1}, \tau_{n-1}, x_{n-1} . e_{n-1}) \end{array}$$

The syntax is a bit heavy compared to products, so it may help to see the concrete syntax as well. The concrete syntax of labelled sum types has the form

$$[l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}],$$

while that of injections has the form $[l = e]_\tau$, and that of case analyses have the form

$$\texttt{case}\, e \,\{\, [l_0 = x_0 : \tau_0] \Rightarrow e_0, \ldots, [l_{n-1} = x_{n-1} : \tau_{n-1}] \Rightarrow e_{n-1} \}.$$

It is an awkwardness of the syntax that injections must be marked with the sum type into which the injection is being made. This is to ensure that every expression has a unique type, since we cannot recover the entire sum type from the type of one of its variants. In Chapter 36 we will consider ways to relax this requirement by introducing subtyping.

The static semantics is given by the following rules:

$$\frac{\Gamma \vdash e : \tau_i \quad \tau = \texttt{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \quad 0 \leq i < n}{\Gamma \vdash \texttt{inj}(\tau, l_i, e) : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{sum}(l_0, \tau_0, \ldots, l_{n-1}, \tau_{n-1}) \\ \Gamma, x_0 : \tau_0 \vdash e_0 : \tau \quad \cdots \quad \Gamma, x_{n-1} : \tau_{n-1} \vdash e_{n-1} : \tau\end{array}}{\Gamma \vdash \texttt{case}(e, l_0, \tau_0, x_0 \,.\, e_0, \ldots, l_{n-1}, \tau_{n-1}, x_{n-1} \,.\, e_{n-1}) : \tau}$$

These rules are a straightforward generalization of those for binary sums to permit an arbitrary number of labelled variants.

We leave as an exercise to formulate the (eager or lazy) dynamic semantics of labelled sums and to prove this extension sound.

16.3 Exercises

1. Formulate general n-ary sums in terms of nullary and binary sums.

Part VI

Recursive Types

Chapter 17

Recursive Types

Many types can be characterized as the solution to a *type isomorphisms* equation. For example, the type of natural numbers can be characterized as a solution to the type isomorphism

$$\texttt{nat} \cong \texttt{unit} + \texttt{nat}.$$

This means a natural number is either zero ($\texttt{inl}(\langle\rangle)$) or the successor of another natural number ($\texttt{inr}(n)$). Similarly, the type of lists of natural numbers, `list`, is the solution to the type isomorphism

$$\texttt{list} \cong \texttt{unit} + (\texttt{nat} \times \texttt{list}),$$

and the type of "bare" binary trees, `tree`, is the solution to the type isomorphism

$$\texttt{tree} \cong \texttt{unit} + (\texttt{tree} \times \texttt{tree}).$$

These examples all fall within the class of inductively defined types, but other sorts of type isomorphisms are also of interest. For example, the type, `stream`, of infinite streams of natural numbers is the solution to the type isomorphism

$$\texttt{stream} \cong \texttt{nat} \times \texttt{stream},$$

stating that every stream consists of a natural number paired with another stream. Even more radically, we may consider the type isomorphism

$$\texttt{D} \cong \texttt{D} \rightarrow \texttt{D}$$

specifies a type that is isomorphic to the type of functions on itself!

The solution to an isomorphism equation is a *fixed point* of an associated operator on types, up to isomorphism. For example, the solution to the type isomorphism

$$\texttt{nat} \cong \texttt{unit} + \texttt{nat}$$

is a fixed point (up to isomorphism) of the type operator

$$T(t) = \texttt{unit} + t,$$

which is to say that we seek a type τ such that $\tau \cong T(\tau)$. A *recursive type* is the solution to such a fixed point equation, written $\mu(t\,.\,T(t))$. Thus, nat may be regarded as standing for the recursive type $\mu(t\,.\,\texttt{unit} + t)$, and list may be regarded as the recursive type $\mu(t\,.\,\texttt{unit} + (\texttt{nat} \times t))$, and so forth.

For the solution to be defined "up to isomorphism" means that there are operations, unroll and roll, that mediate between a recursive type, $\mu(t\,.\,\tau)$, and its *unrolling*, $[t\leftarrow\mu(t\,.\,\tau)]\tau$. For example, the operation unroll maps $\mu(t\,.\,\texttt{unit} + t)$ to $\texttt{unit} + \mu(t\,.\,\texttt{unit} + t)$, and roll maps in the opposite direction. In this sense unroll exposes the underlying structure of a natural number (it is either zero or a successor of another natural number), and roll is the inverse operation that treats zero, or the successor of another natural number, as a natural number. Thus this pair of operations, which constitute an isomorphism between a recursive type and its unrolling, capture the informal idea that something is a natural number iff it is either zero or the successor of another natural number — the isomorphism corresponds to the "if and only if" part of the informal characterization.

17.1 Recursive Types

The abstract syntax of recursive types is given by the following grammar:

$$\begin{array}{llcl} \textit{Types} & \tau & ::= & t \mid \texttt{rec}(t\,.\,\tau) \\ \textit{Expr's} & e & ::= & \texttt{roll}(\tau, e) \mid \texttt{unroll}(e) \end{array}$$

The meta-variable t ranges over a class of *type names*, which serve as names for types. In the recursive type $\texttt{rec}(t\,.\,\tau)$, the type name, t, refers to the recursive type itself, in a sense to be made clear in the static semantics given below. The one-step *unrolling* of $\texttt{rec}(t\,.\,\tau)$ is the type $[t\leftarrow\texttt{rec}(t\,.\,\tau)]\tau$ obtained by substituting the recursive type for t in τ. When this is the case, the recursive type is sometimes said to be the one-step *rolling* of the substituted type. Note, however, that the unrolling does not determine the rolling! That is, given $[t\leftarrow\texttt{rec}(t.\tau)]\tau$, one cannot recover $\texttt{rec}(t.\tau)$, because t may not occur in τ, or may occur multiply.

The introductory form, $\texttt{roll}(\tau,e)$, introduces a value of recursive type in terms of a value of its one-step unrolling, and the eliminatory form, $\texttt{unroll}(e)$, extracts from a value of recursive type a value of its unrolling. In implementation terms the operation $\texttt{roll}(\tau,e)$ may be thought of as an abstract "pointer" to a value of the unrolled type, and the operation $\texttt{unroll}(e)$ "chases" the pointer to obtain that value from a value of the corresponding rolled type.

The static semantics of this extension of MinML consists of *two* forms of judgement, τ type, and $e : \tau$. Whereas the latter is the familiar membership judgement stating that expression e is of type τ, the former is a *formation* judgement stating that τ is a well-formed type expression. This is required to rule out types that involve type names that do not refer to any recursive type. We may define this judgement by a set of rules that involve hypothetical judgements of the form

$$t_1\ \mathsf{type}, \ldots, t_n\ \mathsf{type} \vdash_{t_1,\ldots,t_n} \tau\ \mathsf{type},$$

where the assumptions govern the type names that may appear in τ. We write Δ for a finite set of assumptions of the above form, and drop the index from the turnstile as usual.

The rules for type formation in the presence of recursive types are as follows.

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \qquad \frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{rec}(t.\tau)\ \mathsf{type}}$$

In addition the other types must now have formation rules, since they may involve recursive type variables. For example, here are the formation rules for sums and products:

$$\frac{}{\Delta \vdash \texttt{unit}\ \mathsf{type}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{prod}(\tau_1,\tau_2)\ \mathsf{type}}$$

$$\frac{}{\Delta \vdash \texttt{void}\ \mathsf{type}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{sum}(\tau_1,\tau_2)\ \mathsf{type}}$$

The static semantics of recursive types is defined by the following rules:

$$\frac{\tau\ \mathsf{type} \quad \tau = \texttt{rec}(t.\tau') \quad \Gamma \vdash e : [t\leftarrow\tau]\tau'}{\Gamma \vdash \texttt{roll}(\tau,e) : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{rec}(t.\tau')}{\Gamma \vdash \texttt{unroll}(e) : [t{\leftarrow}\tau]\tau'}$$

These rules express an inverse relationship stating that a recursive type is *isomorphic* to its unrolling, with the operations `roll` and `unroll` being the witnesses to the isomorphism.

Operationally, this is expressed by the following dynamic semantics rules:

$$\frac{e \ \mathsf{value}}{\texttt{unroll}(\texttt{roll}(\tau,e)) \longmapsto e}$$

$$\frac{e \longmapsto e'}{\texttt{unroll}(e) \longmapsto \texttt{unroll}(e')} \qquad \frac{e \longmapsto e'}{\texttt{roll}(\tau,e) \longmapsto \texttt{roll}(\tau,e')}$$

These rules specify an eager semantics for rolling, but it would also be possible to consider a lazy semantics in which `roll`(τ,e) is a value, regardless of whether e is a value or not, and in which the last transition rule is correspondingly suppressed.

It is quite easy to establish the safety of this extension to MinML:

Theorem 17.1 (Safety)

1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

2. *If $e : \tau$, then either e value, or there exists e' such that $e \longmapsto e'$.*

17.2 Inductive Data Structures

Recursive types may be used to represent inductively defined types, such as lists or trees. For example, let us consider the representation of the type of lists of values of type τ, which is written `list`(τ). Lists have two introductory forms, the empty list, written `nil`, and the non-empty list, written `cons`(e_1,e_2), where e_1 is of type τ and e_2 is of type `list`(τ). The elimination form is a case analysis that distinguishes whether a list is empty or non-empty, and, if non-empty, extracts its head and tail elements. This is written

$$\texttt{listcase}(\tau,e,e_0,x,y.e_1),$$

where e is a list, e_0 is the value for the empty list, and e_1 is the value for the non-empty list with head x and tail y.

These constructs are all definable using recursive sum and product types according to the following equations:

$$\begin{array}{rcl}
\texttt{list}(\tau) & = & \texttt{rec}(t\texttt{.sum(unit,prod(}\tau\texttt{,}t\texttt{)))} \\
\texttt{nil} & = & \texttt{roll(list(}\tau\texttt{),unit)} \\
\texttt{cons(}e_1\texttt{,}e_2\texttt{)} & = & \texttt{roll(list(}\tau\texttt{),pair(}e_1\texttt{,}e_2\texttt{))} \\
\texttt{listcase(}\tau\texttt{,}e\texttt{,}e_0\texttt{,}x\texttt{,}y\texttt{.}e_1\texttt{)} & = & \texttt{case(unroll(}e\texttt{),unit,}u\texttt{.}e_0\texttt{,}\tau'\texttt{,}v\texttt{.}e_1'\texttt{)}
\end{array}$$

where $\tau' = \texttt{prod(}\tau\texttt{,list(}\tau\texttt{))}$, $u \# e_0$, $v \# e_1$, and $e_1' = [x, y \leftarrow \texttt{fst(}v\texttt{)}, \texttt{snd(}v\texttt{)}]e_1$.

Informally, the empty list, `nil`, is a "pointer" to a null-tuple tagged with `inl`, and the non-empty list, $\texttt{cons(}e_1\texttt{,}e_2\texttt{)}$, is a "pointer" to a pair tagged with `inr`, with first component e_1 and second component e_2. The `listcase` construct "chases" the pointer, then cases analyses on the tag, branching to the appopriate case, passing the appropriate data values.

Of course more efficient representations are possible, but this representation makes clear the pattern of lists are represented using recursive types. Using similar methods we may represent any inductively defined type such as the type of trees or the type of abstract syntax for a given language.

17.3 Recursive Functions

In Chapter 12 we considered recursive functions as primitive, generalizing $\lambda\texttt{(}x\texttt{:}\tau\texttt{.}\,e\texttt{)}$ to $\texttt{fun}\,f\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$, where f is a variable standing for "the function itself". What makes f stand for the function itself is that the dynamic semantics ensures this is so whenever such a function is applied:

$$\frac{(e_1 = \texttt{fun}\,f\texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e)}{e_1\texttt{(}e_2\texttt{)} \longmapsto [f, x \leftarrow e_1, e_2]e} \; .$$

In this section we show that recursive functions are *definable* in the presence of recursive types, and hence need not be taken as primitive.

To do so we will consider an extension of the core MinML language in which the function type contains only *non-recursive* λ-abstractions, but which, in compensation, has recursive types. Our strategy will be to show that we can define a special form of function type, $\tau_1 \rightarrow_{rec} \tau_2$, whose values are (possibly recursive) functions mapping arguments of type τ_1 to results of type τ_2. This means that we will show how to define recursive functions of this type, and show how to apply them to arguments.

The key to the definition is to use a technique called *self-application*, in which we arrange that a function of type $\tau_1 \rightarrow_{rec} \tau_2$ implicitly takes an "extra" argument, which will be the function itself. This means that the recursive function type must satisfy the isomorphism

$$\tau_1 \rightarrow_{rec} \tau_2 \cong (\tau_1 \rightarrow_{rec} \tau_2) \rightarrow \tau_1 \rightarrow \tau_2.$$

This may be achieved by defining

$$\tau_1 \rightarrow_{rec} \tau_2 = \mu(t\,.\,t \rightarrow \tau_1 \rightarrow \tau_2),$$

where t is chosen so that $t \;\#\; \tau_1$ and $t \;\#\; \tau_2$.

Application of recursive functions is then defined to ensure that the "self" argument really is the function itself:

$$\mathtt{ap}_{rec}\mathtt{(}e_1\mathtt{,}e_2\mathtt{)} := \mathtt{unroll(}e_1\mathtt{)(}e_1\mathtt{)(}e_2\mathtt{)},$$

Since the type of e_1 is $\tau_1 \rightarrow_{rec} \tau_2$, its unrolling has type $\mathtt{(}\tau_1 \rightarrow_{rec} \tau_2\mathtt{)} \rightarrow \mathtt{(}\tau_1 \rightarrow \tau_2\mathtt{)}$, so this is indeed type correct.

The recursive function $\mathtt{fun}_{rec}\, f\mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\;\mathtt{is}\;e$ is represented by the expression

$$\mathtt{roll(}\lambda\mathtt{(}f\mathtt{:}\tau_1 \rightarrow_{rec} \tau_2\mathtt{.}\,\lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e\mathtt{)))}.$$

The correctness of this interpretation follows by checking that the evaluation steps are properly simulated. Suppose that $e_1 = \mathtt{fun}_{rec}\, f\mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\;\mathtt{is}\;e$ and e_2 value.

$$\begin{array}{rcl}
\mathtt{ap}_{rec}\mathtt{(}e_1\mathtt{,}e_2\mathtt{)} & = & \mathtt{unroll(roll(}\lambda\mathtt{(}f\mathtt{:}\tau_1 \rightarrow_{rec} \tau_2\mathtt{.}\,\lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e\mathtt{))))(}e_1\mathtt{)(}e_2\mathtt{)} \\
& \longmapsto & \lambda\mathtt{(}f\mathtt{:}\tau_1 \rightarrow_{rec} \tau_2\mathtt{.}\,\lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e\mathtt{))(}e_1\mathtt{)(}e_2\mathtt{)} \\
& \longmapsto & [f,x \leftarrow e_1,e_2]e,
\end{array}$$

as required.

Let us examine the preceding development in a bit more detail. The application of a recursive function may be decomposed into two steps: first, convert the recursive function into an ordinary function by self-application, and, second, apply this function to the argument. Since the only thing we can do with a recursive function is to apply it, there is no loss of generality in assuming that the first step of this process is done eagerly, so that all uses of a recursive function $e : \tau_1 \rightarrow_{rec} \tau_2$ are of the form $\texttt{unroll}(e)(e)$. In particular, we may assume that the body of a recursive function has the form

$$\lambda(f : \tau_1 \rightarrow_{rec} \tau_2 . F(\texttt{unroll}(f)(f)))$$

for some function $F : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$. We then have

$$\texttt{unroll}(e)(e) \overset{*}{\longmapsto} F(\texttt{unroll}(e)(e)),$$

so that $\texttt{unroll}(e)(e)$ may be seen as a *fixed point* of the function F, and hence may be viewed as the recursive function of type $\tau_1 \rightarrow \tau_2$ determined by F.

17.4 Un(i)typed Languages

It is customary to distinguish between *typed* and *untyped* languages, as if they were alternatives to one another. While it is true that there are *ill-defined* languages that might be described as untyped, among the *well-defined*, or *safe*, languages the supposed distinction is fallacious. A popular form of the this misconception is to distinguish between *dynamically typed* and *statically typed* languages, often placing them in opposition to one another. Terminology notwithstanding, there is no fundamental distinction or opposition, but rather so-called dynamically typed languages are but a mode of use of static types!

The classical example is the so-called *untyped λ-calculus*, which is a very elegant language devised by Alonzo Church in the 1930's. It's chief characteristic is that the entire language consists of nothing but functions! Functions take functions as arguments and yield functions as results, and all data structures must be represented as functions. Surprisingly, this tiny language is sufficiently powerful to express any computable function![1]

The abstract syntax of the untyped λ-calculus is given by the following grammar:

$$\textit{λ-terms} \quad u \quad ::= \quad x \mid \lambda(x.u) \mid \texttt{ap}(u_1, u_2)$$

[1]Technically, the notion of a computable function is defined over the natural numbers. So in order for this statement to make sense, the natural numbers must somehow be encoded as functions. This is achieved using a device called the *Church numerals*, which anticipated by several decades the development of the so-called "object-oriented" view of data.

In concrete syntax these two forms are written $\lambda x.\, u$ and $u_1\, u_2$. The former is called a *λ-abstraction*, and the latter *application*. The entire language consists of these two constructs, plus variables that range over untyped λ-terms.

The basic form of execution in the untyped λ-calculus is defined by the following transition rules:

$$\frac{}{\texttt{ap}(\lambda(x\,.\,u_1),u_2) \longmapsto [x\leftarrow u_2]u} \qquad \frac{u_1 \longmapsto u_2}{\texttt{ap}(u_1,u_2) \longmapsto \texttt{ap}(u_1',u_2)}$$

In the λ-calculus literature this judgement is called *head reduction*. The first rule is called *β-reduction*; it defines the meaning of function application in terms of substitution. It is also possible to define a call-by-value variant of head reduction by insisting that the β-reduction step apply only when the argument is an explicit λ, and adding the following argument-evaluation rule:

$$\frac{u_2 \longmapsto u_2'}{\texttt{ap}(\lambda(x\,.\,u_1),u_2) \longmapsto \texttt{ap}(\lambda(x\,.\,u_1),u_2')}$$

The *untyped* λ-calculus may be *faithfully embedded* in the *typed* language MinML, enriched with recursive types. This means that every untyped λ-term has a representation as an expression in MinML in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. If the execution model of the λ-calculus is call-by-name, this correspondence holds for the call-by-name variant of MinML, and similarly for call-by-value.

It is important to understand that this form of embedding is *not* a matter of writing an *interpreter* for the λ-calculus in MinML, but rather a direct representation of λ-terms as certain expressions of MinML. It is for this reason that we say that untyped languages are just a special case of typed languages, *albeit* those with recursive types. Thus the supposed "opposition" between typed and untyped languages is nothing of the kind. Rather, the issue is simply that recursive types greatly increase the expressive power of typed languages, permitting styles of programming that are otherwise impossible.

The main idea is the observation that *untyped* really means *uni-typed*. The untyped λ-calculus does not have *zero* types, rather it has exactly *one* type! This type is the celebrated recursive type

$$D = \texttt{rec}(t\,.\,\texttt{arrow}(t,t)).$$

A value of type D is of the form $\texttt{roll}(D,e)$ where e is a value of type $\texttt{arrow}(D,D)$ — a function whose domain and range are both D. Any such function can be regarded as a value of type D by "rolling", and any value of type D can be turned into a function by "unrolling". Put in other terms, the recursive type D satisfies the *isomorphism*

$$D \cong \texttt{arrow}(D,D)$$

meaning that it is isomorphic to the function space on itself.

This leads to the following embedding, $u^\dagger$, of u into MinML:

$$\begin{array}{rcl} x^\dagger & = & x \\ \lambda(x.u)^\dagger & = & \texttt{roll}(D,\texttt{lambda}(D,x.u^\dagger)) \\ \texttt{ap}(u_1,u_2)^\dagger & = & \texttt{app}(\texttt{unroll}(u_1^\dagger),u_2^\dagger) \end{array}$$

Observe that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

$$\begin{array}{rcl} \texttt{ap}(\lambda(x.u_1),u_2)^\dagger & = & \texttt{app}(\texttt{unroll}(\texttt{roll}(D,\texttt{lambda}(D,x.u_1^\dagger))),u_2^\dagger) \\ & \longmapsto & \texttt{app}(\texttt{lambda}(D,x.u_1^\dagger),u_2^\dagger) \\ & \longmapsto & [x{\leftarrow}u_2^\dagger]u_1^\dagger \\ & = & ([x{\leftarrow}u_2]u_1)^\dagger. \end{array}$$

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of u_1. Thus β-reduction is faithfully implemented by evaluation of the embedded terms. It is also easy to show that if $u_1^\dagger \overset{*}{\longmapsto} v_1^\dagger$, then $\texttt{ap}(u_1,u_2)^\dagger \overset{*}{\longmapsto} \texttt{ap}(v_1,u_2)^\dagger$. Consequently, head reduction in the λ-calculus is faithfully implemented by evaluation in MinML enriched with recursive types.

Interest in the untyped λ-calculus stems from its surprising expressive power: it is a Turing-complete language in the sense that it has the same capability to expression computations on the natural numbers as does any other known programming language. The Church-Turing Thesis states that any conceivable notion of computable function on the natural numbers is equivalent to the λ-calculus. This is certainly true for all *known* means of defining computable functions on the natural numbers. The force of the Church-Turing Thesis is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. The Church-Turing Thesis is therefore a *scientific law* in precisely the same sense as, say, Newton's Law of Universal Gravitation makes a prediction about all future measurements of the acceleration due to the gravitational field of a massive object.

The key to understanding this is to consider how one may program with the natural numbers in the untyped λ-calculus. The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\begin{array}{rcl} \overline{0} & := & \lambda b.\, \lambda s.\, b \\ \overline{n+1} & := & \lambda b.\, \lambda s.\, s\, (\overline{n}\, b\, s) \end{array}$$

It follows that

$$\overline{n}\, u_1\, u_2 \overset{*}{\longmapsto} u_2\, (\cdots (u_2\, u_1)),$$

the n-fold application of u_2 to u_1. That is, $\overline{n}$ iterates its second argument (the induction step) n times, starting with its first argument (the basis).

Using this definition it is not difficult to define addition, multiplication, and a conditional test for zero function. Crucially, the predecessor function is also definable, using a "trick" involving the representation of ordered pairs. The idea is to compute, given n, the pair consisting of $n-1$ and n, starting with 0 and 0, so that the inductive step consists of passing from the pair $(n-1, n)$ to $(n, n+1)$, which is easily achieved. The predecessor is then the first projection of the result of this auxiliary function. This gives us all the apparatus of MinML, apart from recursion. This, too, is definable by considering the *Y-combinator*,

$$Y = \lambda F.\, (\lambda f.\, F\, (f\, f))\, (\lambda f.\, F\, (f\, f)).$$

Observe that

$$Y\, F \approx F\, (Y\, F),$$

where the equivalence means, informally, that both sides may be "symbolically evaluated" to obtained the same result. More precisely,

$$Y\, F \overset{*}{\longmapsto} F\, (u)\, (u),$$

where $Y\, F \longmapsto u$. For this reason, the term Y is called a *fixed point combinator*. As with the representation of recursive functions in MinML with recursive types, the key to the Y combinator is the possibility of self-application inherent in a un(i)typed language.

17.5 Exercises

1. Derive the static and dynamic semantics of lists induced by the definitions given in Section 17.2.

2. Give a representation of binary trees decorated with values of type τ at the leaves using recursive types.

3. Can MinML be faithfully embedded in the untyped λ-calculus?

4. Explore the compilation of Y, and relate this to the representation of recursive functions.

Chapter 18

Pattern Compilation

Chapter 19

Dynamic Typing

The formalization of type safety given in Chapter 11 states that a language is type safe iff it satisfies both *preservation* and *progress*. According to this account, "stuck" states — non-final states with no transition — must be rejected by the static type system as ill-typed. Although this requirement seems natural for relatively simple languages such as MinML, it is not immediately clear that our formalization of type safety scales to larger languages, nor is it entirely clear that the informal notion of safety is faithfully captured by the preservation and progress theorems.

One issue that we addressed in Chapter 11 was how to handle expressions such as `3 div 0`, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible ways to handle such a situation. One is to enrich the type system so that such an expression is ill-typed. However, this takes us considerably beyond the capabilities of current type systems for practical programming languages. The alternative is to ensure that such ill-defined states are not "stuck", but rather make a transition to a designated error state. To do so we introduced the notion of a checked error, which is explicitly detected and signalled during execution. Checked errors are constrasted with unchecked errors, which are ruled out by the static semantics.

In this chapter we will concern ourselves with question of why there should unchecked errors at all. Why aren't all errors, including type errors, checked at run-time? Then we can dispense with the static semantics entirely, and, in the process, execute more programs. Such a language is called *dynamically typed*, in contrast to MinML, which is *statically typed*.

One advantage of dynamic typing is that it supports a more flexible treatment of conditionals. For example, the expression

```
(if true then 7 else "7")+1
```

is statically ill-typed, yet it executes successfully without getting stuck or incurring a checked error. Why rule it out, simply because the type checker is unable to "prove" that the `else` branch cannot be taken? Instead we may shift the burden to the programmer, who is required to maintain invariants that ensure that no run-time type errors can occur, even though the program may contain conditionals such as this one.

Another advantage of dynamic typing is that it supports *heterogeneous data structures*, which may contain elements of many different types. For example, we may wish to form the "list"

```
[true, 1, 3.4, fn x=>x]
```

consisting of four values of distinct type. Languages such as ML preclude formation of such a list, insisting instead that all elements have the *same* type; these are called *homogenous* lists. The argument for heterogeneity is that there is nothing inherently "wrong" with such a list, particularly since its constructors are insensitive to the types of the components — they simply allocate a new node in the heap, and initialize it appropriately.

Note, however, that the additional flexibility afforded by dynamic typing comes at a cost. Since we cannot accurately predict the outcome of a conditional branch, nor the type of a value extracted from a heterogeneous data structure, we must program defensively to ensure that nothing bad happens, even in the case of a type error. This is achieved by turning type errors into checked errors, thereby ensuring progress and hence safety, even in the absence of a static type discipline. Thus dynamic typing catches type errors as late as possible in the development cycle, whereas static typing catches them as early as possible.

In this chapter we will investigate a dynamically typed variant of MinML in which type errors are treated as checked errors at execution time. Our analysis will reveal that, rather than being opposite viewpoints, dynamic typing is a *special case* of static typing! In this sense static typing is *more expressive* than dynamic typing, despite the superficial impression created by the examples given above. This viewpoint illustrates the *pay-as-you-go* principle of language design, which states that a program should only incur overhead for those language features that it actually uses. By viewing dynamic typing as a special case of static typing, we may avail ourselves of the benefits of dynamic typing whenever it is required, but avoid its costs whenever it is not.

19.1 Dynamic Typing

The fundamental idea of dynamic typing is to regard type clashes as *checked*, rather than *unchecked*, errors. Doing so puts type errors on a par with division by zero and other checked errors. This is achieved by augmenting the dynamic semantics with rules that explicitly check for stuck states. For example, the expression `true+7` is such an ill-typed, stuck state. By checking that the arguments of an addition are integers, we can ensure that progress may be made, namely by making a transition to `error`.

The idea is easily illustrated by example. Consider the rules for function application in MinML given in Chapter 12, which we repeat here for convenience:

$$\frac{v\ \mathsf{value}\quad v_1\ \mathsf{value}\quad (v = \mathtt{fun}\,f\ (x\,\mathtt{:}\,\tau_1)\,\mathtt{:}\,\tau_2\ \mathtt{is}\,e)}{\mathtt{apply}(v,v_1)\longmapsto [f,x{\leftarrow}v,v_1]e}$$

$$\frac{e_1\longmapsto e_1'}{\mathtt{apply}(e_1,e_2)\longmapsto \mathtt{apply}(e_1',e_2)}$$

$$\frac{v_1\ \mathsf{value}\quad e_2\longmapsto e_2'}{\mathtt{apply}(v_1,e_2)\longmapsto \mathtt{apply}(v_1,e_2')}$$

In addition to these rules, which govern the well-typed case, we add the following rules governing the ill-typed case:

$$\frac{v\ \mathsf{value}\quad v_1\ \mathsf{value}\quad (v \neq \mathtt{fun}\,f\ (x\,\mathtt{:}\,\tau_1)\,\mathtt{:}\,\tau_2\ \mathtt{is}\,e)}{\mathtt{apply}(v,v_1)\longmapsto \mathtt{error}}$$

$$\frac{}{\mathtt{apply}(\mathtt{error},e_2)\longmapsto \mathtt{error}}$$

$$\frac{v_1\ \mathsf{value}}{\mathtt{apply}(v_1,\mathtt{error})\longmapsto \mathtt{error}}$$

The first rule states that a run-time error arises from any attempt to apply a non-function to an argument. The other two define the propagation of such errors through other expressions — once an error occurs, it propagates throughout the entire program.

By entirely analogous means we may augment the rest of the semantics of MinML with rules to check for type errors at run time. Once we have done so, it is safe to eliminate the static semantics in its entirety.[1] Having done so, every expression is well-formed, and hence preservation holds vacuously. More importantly, the progress theorem also holds because we have augmented the dynamic semantics with transitions from every ill-typed expression to `error`, ensuring that there are no "stuck" states. Thus, the dynamically typed variant of MinML is safe in same sense as the statically typed variant. The meaning of safety does not change, only the means by which it is achieved.

19.2 Implementing Dynamic Typing

Since both the statically- and the dynamically typed variants of MinML are safe, it is natural to ask which is better. The main difference is in how early errors are detected — at compile time for static languages, at run time for dynamic languages. Is it better to catch errors early, but rule out some useful programs, or catch them late, but admit more programs? Rather than attempt to settle this question, we will sidestep it by showing that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a *mode of use* of static typing. From this point of view static and dynamic typing are matters of design for a particular program (which to use in a *given* situation), rather than a doctrinal debate about the design of a programming language (which to use in *all* situations).

To see how this is possible, let us consider what is involved in implementing a dynamically typed language. The dynamically typed variant of MinML sketched above includes rules for run-time type checking. For example, the dynamic semantics includes a rule that explicitly checks for an attempt to apply a non-function to an argument. How might such a check be implemented? The chief problem is that the natural representations of data values on a computer do not support such tests. For example,

[1]We may then simplify the language by omitting type declarations on variables and functions, since these are no longer of any use.

a function might be represented as a word representing a pointer to a region of memory containing a sequence of machine language instructions. An integer might be represented as a word interpreted as a two's complement integer. But given a word, you cannot tell, in general, whether it is an integer or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that it is possible to implement the required checks. We must be able to tell by looking at the value whether it is an integer, a boolean, or a function. Having done so, we must be able to recover the underlying value (integer, boolean, or function) for direct calculation. Whenever a value of a type is created, it must be marked with appropriate information to identify the sort of value it represents.

There are many schemes for doing this, but at a high level they all amount to attaching a *tag* to a "raw" value that identifies the value as an integer, boolean, or function. Dynamic typing then amounts to checking and stripping tags from data during computation, transitioning to `error` whenever data values are tagged inappropriately. From this point of view, we see that dynamic typing should *not* be described as "run-time type checking", because we are not checking *types* at run-time, but rather *tags*. The difference can be seen in the application rule given above: we check only that the first argument of an application is some function, not whether it is well-typed in the sense of the MinML static semantics.

To clarify these points, we will make explicit the manipulation of tags required to support dynamic typing. To begin with, we revise the grammar of MinML to make a distinction between *tagged* and *untagged* values, as follows:

$$\begin{array}{llcl}
\textit{Expressions} & e & ::= & x \mid v \mid o(e_1,\ldots,e_n) \mid \texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \mid \\
 & & & \texttt{apply}(e_1,e_2) \\
\textit{TaggedValues} & v & ::= & \texttt{Int}\,(n) \mid \texttt{Bool}\,(\texttt{true}) \mid \texttt{Bool}\,(\texttt{false}) \mid \\
 & & & \texttt{Fun}\,(\texttt{fun}\,x\,(y\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\,\texttt{is}\,e) \\
\textit{UntaggedValues} & u & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \texttt{fun}\,x\,(y\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\,\texttt{is}\,e
\end{array}$$

Note that *only* tagged values arise as expressions; untagged values are used strictly for "internal" purposes in the dynamic semantics. Moreover, we do not admit general tagged expressions such as `Int` (e), but only explicitly-tagged values.

Second, we introduce *tag checking* rules that determine whether or not a tagged value has a given tag, and, if so, extracts its underlying untagged value. In the case of functions these are given as rules for deriving judgements of the form $v\ \mathsf{is\_fun}\ u$, which checks that v has the form $\mathtt{Fun}\ (u)$, and extracts u from it if so, and for judgements of the form $v\ \mathsf{isnt\_fun}$, that checks that v does not have the form $\mathtt{Fun}\ (u)$ for any untagged value u.

$$\frac{}{\mathtt{Fun}\ (u)\ \mathsf{is\_fun}\ u}$$

$$\frac{}{\mathtt{Int}\ (\_)\ \mathsf{isnt\_fun}} \qquad \frac{}{\mathtt{Bool}\ (\_)\ \mathsf{isnt\_fun}}$$

Similar judgements and rules are used to identify integers and booleans, and to extract their underlying untagged values.

Finally, the dynamic semantics is re-formulated to make use of these judgement forms. For example, the rules for application are as follows:

$$\frac{v_1\ \mathsf{value} \quad v\ \mathsf{is\_fun}\ \mathtt{fun}\ f\ (x:\tau_1):\tau_2\ \mathtt{is}\ e}{\mathtt{apply}(v, v_1) \longmapsto [f, x \leftarrow v, v_1]e}$$

$$\frac{v\ \mathsf{value} \quad v\ \mathsf{isnt\_fun}}{\mathtt{apply}(v, v_1) \longmapsto \mathtt{error}}$$

Similar rules govern the arithmetic primitives and the conditional expression. For example, here are the rules for addition:

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value} \quad v_1\ \mathsf{is\_int}\ n_1 \quad v_2\ \mathsf{is\_int}\ n_2 \quad (n = n_1 + n_2)}{+(v_1, v_2) \longmapsto \mathtt{Int}\ (n)}$$

Note that we must explicitly check that the arguments are tagged as integers, and that we must apply the integer tag to the result of the addition.

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value} \quad v_1\ \mathsf{isnt\_int}}{+(v_1, v_2) \longmapsto \mathtt{error}}$$

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value} \quad v_1\ \mathsf{is\_int}\ n_1 \quad v_2\ \mathsf{isnt\_int}}{+(v_1, v_2) \longmapsto \mathtt{error}}$$

These rules explicitly check for non-integer arguments to addition.

19.3 Dynamic Typing as Static Typing

Once tag checking is made explicit, it is easier to see its hidden costs in both time and space — time to check tags, to apply them, and to extract the underlying untagged values, and space for the tags themselves. This is a significant overhead. Moreover, this overhead is imposed *whether or not* the original program is statically type correct. That is, even if we can prove that no run-time type error can occur, the dynamic semantics nevertheless dutifully performs tagging and untagging, just as if there were no type system at all.

This violates a basic principle of language design, called the *pay-as-you-go* principle. This principle states that a language should impose the cost of a feature only to the extent that it is actually used in a program. With dynamic typing we pay for the cost of tag checking, even if the program is statically well-typed! For example, if all of the lists in a program are homogeneous, we should not have to pay the overhead of supporting heterogeneous lists. The choice should be in the hands of the programmer, not the language designer.

It turns out that we can eat our cake and have it too! The key is a simple, but powerful, observation: dynamic typing is but a mode of use of static typing, provided that our static type system includes a type of *tagged data*! Dynamic typing emerges as a particular style of programming with tagged data.

The point is most easily illustrated using ML. The type of tagged data values for MinML may be introduced as follows:

```
(* The type of tagged values. *)
datatype tagged =
  Int of int |
  Bool of bool |
  Fun of tagged -> tagged
```

Values of type `tagged` are marked with a value constructor indicating their outermost form. Tags may be manipulated using pattern matching.

Second, we introduce operations on tagged data values, such as addition or function call, that explicitly check for run-time type errors.

```
exception TypeError
fun checked_add (m:tagged, n:tagged):tagged =
    case (m,n) of
      (Int a, Int b) => Int (a+b)
    | (_, _) => raise TypeError
fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
      Fun g => g a
    | _ => raise TypeError
```

Observe that these functions correspond precisely to the instrumented dynamic semantics given above.

Using these operations, we can then build heterogeneous lists as values of type `tagged list`.

```
val het_list : tagged list =
    [Int 1, Bool true, Fun (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Int 5)
```

The tags on the elements serve to identify what sort of element it is: an integer, a boolean, or a function.

It is enlightening to consider a dynamically typed version of the factorial function:

```
fun dyn_fact (n : tagged) =
    let fun loop (n, a) =
        case n
          of Int m =>
             (case m
                of 0 => a
                 | m => loop (Int (m-1),
                              checked_mult (m, a)))
           | _ => raise RuntimeTypeError
      in loop (n, Int 1)
    end
```

Notice that tags must be manipulated within the loop, even though we can prove (by static typing) that they are not necessary! Ideally, we would like to hoist these checks out of the loop:

```
fun opt_dyn_fact (n : tagged) =
    let fun loop (0, a) = a
          | loop (n, a) = loop (n-1, n*a)
     in case n
          of Int m => Int (loop (m, 1))
           | _ => raise RuntimeTypeError
    end
```

It is *very hard* for a compiler to do this hoisting reliably. But if you consider dynamic typing to be a special case of static typing, as we do here, there is no obstacle to doing this optimization yourself, as we have illustrated here.

Part VII

Polymorphism

Chapter 20

Polymorphism

MinML, and its extensions, are *explicitly typed* in the sense that every well-typed expression has a unique type. In particular, a function expression has specific domain and range types, which must be respected by all uses of that function. Thus it is impossible to write a fully general composition function that forms the composition of any two functions of appropriate type. Instead we must define a separate composition function for each choice of the three types involved:

$$\lambda(f:\tau_2\rightarrow\tau_3.\,\lambda(g:\tau_1\rightarrow\tau_2.\,\lambda(x:\tau_1.\,f(g(x)))))$$

There is one such function for each choice of types τ_1, τ_2, and τ_3, even though all such choices "compute the same way".

This limitation is rather irksome, and quickly gets out of hand. Historically, typed languages were strongly criticized for this reason, but the problem is not inherent in typed languages, rather just an annoying limitation of a particular type system. What is needed is some way to capture the generic pattern of a computation in a way that is *generic*, or *parametric*, in the types involved. This facility is called *polymorphism*.

20.1 Polymorphic λ-Calculus

The *polymorphic λ-calculus*, or Poly, is a minimal functional language that illustrates the core concepts of polymorphic typing, and permits us to examine its surprising expressive power in isolation from other language features. The abstract syntax of the polymorphic λ-calculus is given as follows:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & t \mid \texttt{arrow}(\tau_1,\tau_2) \mid \texttt{all}(t.\tau) \\ \textit{Expr's} & e & ::= & x \mid \texttt{lambda}(\tau,x.e) \mid \texttt{app}(e_1,e_2) \mid \texttt{Lambda}(t.e) \mid \texttt{App}(e,\tau) \end{array}$$

The meta-variable t ranges over a class of *type names* (also called *type variables*), and x ranges over a class of *expression names* (also called *expression variables*). The *type abstraction*, $\texttt{Lambda}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* t standing for an unpspecified type within e. The *type application*, or *instantiation*, $\texttt{App}(e,\tau)$, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type*, $\texttt{all}(t.\tau)$, that determines the type, τ, of the result as a function of the argument, t.

In examples we use the following mathematical and concrete syntax for these constructs:

Abstract	*Concrete*
$\texttt{all}(t.\tau)$	$\forall(t.\tau)$
$\texttt{Lambda}(t.e)$	$\Lambda(t.e)$
$\texttt{App}(e,\tau)$	$e[\tau]$

The static semantics of Poly consists of two categorical judgement forms, τ type, stating that τ is a well-formed type, and $e : \tau$, stating that e is a well-formed expression of type τ. The definitions of these judgements make use of hypothetical judgements of the form

$$t_1 \ \mathsf{type}, \ldots, t_n \ \mathsf{type} \vdash_{t_1,\ldots,t_n} \tau \ \mathsf{type}$$

and

$$t_1 \ \mathsf{type}, \ldots, t_n \ \mathsf{type}; x_1 : \tau_1, \ldots, x_k : \tau_k \vdash_{t_1,\ldots,t_n,x_1,\ldots,x_k} e : \tau.$$

As usual we suppress the indices on the turnstile for the sake of clarity. As a notational convenience we abbreviate a sequence of type variable formation hypotheses by Δ and a sequence of expression variable typing hypotheses by Γ.

The rules for type formation are as follows:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{20.1}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{arrow}(\tau_1,\tau_2) \ \mathsf{type}} \tag{20.2}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{type}} \tag{20.3}$$

The rules for typing expressions are as follows:

$$\frac{\Delta, t \ \mathsf{type}; \Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{Lambda}(t.e) : \mathtt{all}(t.\tau)} \tag{20.4}$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \ \mathsf{type}}{\Delta;\Gamma \vdash \mathtt{App}(e,\tau) : [t{\leftarrow}\tau]\tau'} \tag{20.5}$$

As an example, the polymorphic composition function is written as follows:

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda(f{:}t_2{\to}t_3.\,\lambda(g{:}t_1{\to}t_2.\,\lambda(x{:}t_1.\,f(g(x)))))))).$$

It has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2{\to}t_3){\to}(t_1{\to}t_2){\to}(t_1{\to}t_3)))).$$

Typing is closed under substitution of types for type variables and terms for term variables.

Lemma 20.1 (Substitution)

1. *If* $\Delta, t \ \mathsf{type} \vdash \tau' \ \mathsf{type}$ *and* $\Delta \vdash \tau \ \mathsf{type}$*, then* $\Delta \vdash [t{\leftarrow}\tau]\tau' \ \mathsf{type}$.

2. *If* $\Delta, t \ \mathsf{type}; \Gamma \vdash e' : \tau'$ *and* $\Delta \vdash \tau \ \mathsf{type}$*, then* $\Delta; [t{\leftarrow}\tau]\Gamma \vdash [t{\leftarrow}\tau]e' : [t{\leftarrow}\tau]\tau'$.

3. *If* $\Delta; \Gamma, x : \tau \vdash e' : \tau'$ *and* $\Delta; \Gamma \vdash e : \tau$*, then* $\Delta; \Gamma \vdash [x{\leftarrow}e]e' : \tau'$.

Notice that the second part of the lemma requires substitution into the context, Γ, as well as into the term and its type, because the type variable t may occur freely in any of these positions.

Dynamic Semantics

The dynamic semantics of Poly . . . .

The dynamic semantics of Poly is a simple extension of that of MinML. We need only add the following two SOS rules:

$$\frac{}{\texttt{App}(\texttt{Lambda}(t.e),\tau) \longmapsto [t{\leftarrow}\tau]e} \tag{20.6}$$

$$\frac{e \longmapsto e'}{\texttt{App}(e,\tau) \longmapsto \texttt{App}(e',\tau)} \tag{20.7}$$

It is then a simple matter to prove safety for this language, using the by-now familiar methods.

Lemma 20.2 (Canonical Forms)
Suppose that $e : \tau$ and e value, then

1. *If $\tau = \texttt{arrow}(\tau_1,\tau_2)$, then $e = \texttt{lambda}(\tau_1,x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \texttt{all}(t.\tau')$, then $e = \texttt{Lambda}(t.e')$ with t type $\vdash e' : \tau'$.*

Theorem 20.3 (Preservation)
If $e : \sigma$ and $e \longmapsto e'$, then $e' : \sigma$.

Theorem 20.4 (Progress)
If $e : \sigma$, then either e value or there exists e' such that $e \longmapsto e'$.

20.2 Polymorphic Definability

Although we will not give a proof here, it is possible to show that every well-typed expression in Poly evaluates to a value — there is no possibility of writing an infinite loop. It might seem, at first glance, that this is obviously the case, because there is, apparently, no form of iteration or recursion available in the language. After all, the entire language consists solely of function types and polymorphic types, and nothing else, not even a base type!

Surprisingly, though, it *is* possible to define loops in Poly, albeit ones that always terminate. For example, it is possible to define within Poly a type of natural numbers whose elimination form is essentially the iterator described in Chapter 12. More generally, any inductively defined type may be represented in Poly in such a way that its associated iterator is definable as well!

Let us begin by showing that the type, nat, is definable in Poly. This means that we can fill in the following chart in such a way that the static and dynamic semantics are preserved:

$$\begin{aligned} \texttt{nat} &:= \ldots \\ \texttt{zero} &:= \ldots \\ \texttt{succ}(e) &:= \ldots \\ \texttt{rec}(\tau, e_0, e_1, x\,.\,e_2) &:= \ldots \end{aligned}$$

The key to understanding how this is achieved is to focus attention on the iterator.

Recall that the typing rule for the iterator is as follows:

$$\frac{e_0 : \texttt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\texttt{rec}(\tau, e_0, e_1, x\,.\,e_2) : \tau}\ .$$

Since the type τ is completely arbitrary, this means that if we have an iterator, then it can be used to define a polymorphic function of type

$$\texttt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument n, yields a polymorphic function that, for any result type, t, if given the initial result for zero, and if given a function transforming the result for x into the result for succ(x), then it returns the result of iterating the transformer n times starting with the initial result.

Since the *only* operation we can perform on a natural number is to iterate up to it in this manner, we may simply *identify* a natural number, n, with the polymorphic iterate-up-to-n function just described! This means that the chart sketched above may be completed as follows:

$$\begin{aligned} \texttt{nat} &:= \forall(t.t \to (t \to t) \to t) \\ \texttt{zero} &:= \Lambda(t.\lambda(z{:}t.\,\lambda(s{:}t \to t.\,z))) \\ \texttt{succ}(e) &:= \Lambda(t.\lambda(z{:}t.\,\lambda(s{:}t \to t.\,s(e[t](z)(s))))) \\ \texttt{rec}(\tau, e_0, e_1, x\,.\,e_2) &:= e_0[\tau](e_1)(\lambda(x{:}t.\,e_2)) \end{aligned}$$

It is a simple matter to check that the static semantics of these constructs is correctly derived from these definitions. We turn, then, to the dynamic semantics.

The number `zero` iterates a given s zero times starting from z, which means that it merely returns z. The successor, $\texttt{succ(}e\texttt{)}$, of e iterates s from z for e iterations, then iterates s one more time, as required. Letting $\overline{n}$ stand for the n-fold composition $\texttt{succ(}\cdots\texttt{succ(zero)}\cdots\texttt{)}$, and assuming a call-by-value semantics for function application, we may show by induction on n that

$$\overline{n}\texttt{[}\tau\texttt{]}e_1e_2 \overset{*}{\longmapsto} e_2\texttt{(}\cdots e_2\texttt{(}e_1\texttt{)}\cdots\texttt{)}.$$

That is, $\overline{n}$ is indeed the polymorphic iterator specialized to the number n!

Since we are identifying natural numbers with their associated iterators, it follows that we should define the iterator from e_0 to simply instantiate and apply e_0 to the result type, initial result, and result transformer associated with the iterator. Observe that

$$\texttt{zero[}\tau\texttt{](}e_1\texttt{)(}e_2\texttt{)} \overset{*}{\longmapsto} e_1$$

and that if

$$\overline{n}\texttt{[}\tau\texttt{](}e_1\texttt{)(}e_2\texttt{)} \overset{*}{\longmapsto} e,$$

then

$$\texttt{succ(}\overline{n}\texttt{)[}\tau\texttt{](}e_1\texttt{)(}e_2\texttt{)} \overset{*}{\longmapsto} e_2\texttt{(}e\texttt{)}.$$

Thus the dynamic semantics is correctly simulated by these definitions.

As an example, here is the addition function defined in terms of this representation of natural numbers:

$$\lambda\texttt{(}x\texttt{:nat.}\,\lambda\texttt{(}y\texttt{:nat.}\,y\texttt{[nat](}x\texttt{)(}\lambda\texttt{(}x\texttt{:nat.succ(}x\texttt{)))))}.$$

Given x and y of type `nat`, this function iterates the successor function (defined above) y times starting with x — that is, it computes the sum of x and y.

Following a similar pattern of reasoning, we may define product and sum types, and other, more general, recursive types. Here is a chart of the type definitions:

$$\begin{array}{rcl}
\texttt{unit} & := & \forall(t.t\to t) \\
\tau_1 \times \tau_2 & := & \forall(t.(\tau_1\to\tau_2\to t)\to t) \\
\texttt{void} & := & \forall(t.t) \\
\tau_1 + \tau_2 & := & \forall(t.(\tau_1\to t)\to(\tau_2\to t)\to t) \\
\tau\,\texttt{list} & := & \forall(t.t\to(\tau\to t\to t)\to t)
\end{array}$$

We leave it as an exercise to define the introduction and elimination forms for these types according to the same pattern as we did for natural numbers. Remember that the main idea is to represent each introduction form as the elimination form applied to that introduction form.

20.3 Restricted Forms of Polymorphism

The remarkable expressive power of the language Poly stems from the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let τ be the type $\forall(t.t\to t)$, and, assuming that $e : \tau$, we may apply e to its own type, obtaining the expression $e[\tau]$ of type $\tau\to\tau$. Written out in full, this is the type

$$(\forall(t.t\to t))\to(\forall(t.t\to t)),$$

which is obviously much "larger" than the type of e itself. In fact, this type is "large enough" that we can go ahead and apply $e[\tau]$ to e again, obtaining the expression $e[\tau](e)$, which is again of type τ — the very type of e!

Contrast this behavior with the situation in MinML, in which the type of an application of a function is evidently "smaller" than the type of the function itself. For if $e : \tau_1\to\tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of e. For this reason MinML is not powerful enough to permit types such as the natural numbers to be defined in terms of function spaces alone — such types have to be built in as primitives.

The source of the expressive power of Poly is that it permits polymorphic types to be instantiated with other polymorphic types, so that we may instantiate $\tau = \forall(t.t\to t)$ with itself to obtain a "larger" type as result. This property of Poly is called *impredicativity*[1], and we say that Poly permits *impredicative (type) quantification*.

The alternative, called *predicative*[2] *quantification*, is to restrict the quantifier to range only over *un-quantified* types. Under this restriction we may, for example, instantiate the type τ given above with the type $u\to u$ to obtain the type $(u\to u)\to(u\to u)$. This type is "larger" than τ in one sense (it

[1]pronounced *im-PRED-ic-a-tiv-it-y*

[2]pronounced *PRED-i-ca-tive*

has more symbols), but is "smaller" in another sense (it has fewer quantifiers). It turns out that for this reason the predicative fragment of the language is substantially less expressive than the impredicative part.

Prenex Fragment

An even more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is imposed in ML for the sake of *type inference*. Type inference permits the programmer to omit type information entirely from expressions in the knowledge that the compiler can always reconstruct the *most general*, or *principal*, type of an expression. We will not discuss type inference here, but we will give a formulation of the prenex fragment of Poly because it plays such an important role in the design of ML.

The prenex fragment of Poly is obtained by *stratifying* types into two classes, the *monotypes* and the *polytypes*. The monotypes are those that do not involve any quantification, and are thus eligible for instantiation of polymorphic quantifiers. The polytypes include the monotypes, and also permit quantification over monotypes to obtain another polytype.

$$\begin{array}{llcl} \textit{Monotypes} & \tau & ::= & t \mid \texttt{arrow}(\tau_1,\tau_2) \\ \textit{Polytypes} & \sigma & ::= & \tau \mid \texttt{all}(t\texttt{.}\sigma) \end{array}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes. The polytypes are always of the form

$$\forall(t_1.\cdots\forall(t_n.\tau)\cdots),$$

were τ is a monotype. We often abbreviate this to just $\forall(t_1,\ldots,t_n.\tau)$.

The static semantics of this fragment of Poly is given as follows. Expressions are, first of all, classified by monotypes. For example, we have the rule

$$\frac{\Delta;\Gamma,x:\tau_1 \vdash e_2:\tau_2}{\Delta;\Gamma \vdash \texttt{lambda}(\tau_1,x\texttt{.}e_2):\texttt{arrow}(\tau_1,\tau_2)}$$

and so on for any other monotypes in the language. Second, we may *generalize* with respect to any free type variable, and *instantiate* any quantified polytype as follows:

$$\frac{\Delta, t\ \mathsf{type}; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \mathtt{Lambda}(t.e) : \mathtt{all}(t.\sigma)} \qquad \frac{\Delta \vdash \tau\ \mathsf{type} \quad \Delta; \Gamma \vdash e : \mathtt{all}(t.\sigma)}{\Delta; \Gamma \vdash \mathtt{App}(e, \tau) : [t{\leftarrow}\tau]\sigma}$$

Note that since every monotype is also a polytype, these rules "get started" by assigning a monotype to an expression, then generalizing on its free type variables.

This type discipline may then be combined with the `let` construct to obtain the core of the ML type system:

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Gamma, x : \sigma_1 \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \mathtt{let}(\sigma_1, e_1, x.e_2) : \tau_2}.$$

Note that this rule requires that we consider hypotheses of the form $x : \sigma$, which include those of the form $x : \tau$ as a special case. This corresponds to the policy in ML that only variables can have polymorphic type — if you wish to use a function polymorphically, you must bind it to a variable so that it can be assigned a polytype. Each use of a variable must then be instantiated to obtain a monotype so that it can appear in another expression.

The following expression exemplifies the ML type discipline in action. The expression

$$\mathtt{let}\ I{:}\forall(t.t{\rightarrow}t)\ \mathtt{be}\ \Lambda(t.\lambda(x{:}t.\,x))\ \mathtt{in}\ I[u{\rightarrow}u](I[u])$$

has type $u{\rightarrow}u$, where u is a free type variable. The ML type inference mechanism permits us to suppress mention of types, writing only

$$\mathtt{let}\ I\ \mathtt{be}\ \lambda x.\,x\ \mathtt{in}\ I(I).$$

The type inference process fills in the missing type abstractions and type applications in the most general way possible, with the result being as just illustrated.

20.4 Exercises

Chapter 21

Data Abstraction

Data abstraction is perhaps the most fundamental technique for structuring programs. The fundamental idea of data abstraction is to separate a *client* from the *implementor* of an abstraction by an *interface*. The interface forms a "contract" between the client and implementor that specifies those properties of the abstraction on which the client may rely, and, correspondingly, those properties that the implementor must satisfy. This ensures that the client is insulated from the details of the implementation of an abstraction so that the implementation can be modified, without changing the client's behavior, provided only that the interface remains the same. This property is called *representation independence* for abstract types.

Data abstraction may be formalized by extending the language Poly with *existential types*. Interfaces are modelled as existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are modelled as packages, the introductory form for existentials, and clients are modelled as uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is modelled so naturally and directly by the logical concept of existential type quantification.

Existential types are closely connected with universal types, and hence are often treated together. The superficial reason is that both are forms of type quantification, and hence both require the machinery of type variables. The deeper reason is that existentials are *definable* from universals — surprisingly, data abstraction is actually just a form of polymorphism!

21.1 Existential Types

The extension of the polymorphic λ-calculus, Poly, with existential types is described by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{some}(t\texttt{.}\tau) \\ \textit{Expr's} & e & ::= & \texttt{pack}(\tau_1\texttt{,}\tau_2\texttt{,}e) \mid \texttt{open}(e_1\texttt{,}t\texttt{.}x\texttt{.}e_2) \end{array}$$

The following chart shows the correspondence between concrete and abstract syntax for these constructs.

Abstract	*Concrete*
$\texttt{some}(t\texttt{.}\tau)$	$\exists(t.\tau)$
$\texttt{pack}(\tau_1\texttt{,}\tau_2\texttt{,}e)$	$\texttt{pack}\,\tau_2\,\texttt{with}\,e\,\texttt{as}\,\tau_1$
$\texttt{open}(e_1\texttt{,}t\texttt{.}x\texttt{.}e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{in}\,e_2$

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\sigma$, where ρ is a type and e is an expression of type $[t{\leftarrow}\rho]\tau$. The type ρ is called the *representation type* of the package, and the expression e is called the *implementation* of the package. The eliminatory form for existentials is the expression $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{in}\,e_2$, which *opens* the package e_1 for use within the *client* e_2 by binding its representation type to t and its implementation to x for use within e_2. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the `open` construct specifies that the type variable, t, and the expression variable, x, are bound within the client. They may be renamed at will by α-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, t, may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to α-equivalence, and is not particularly tied to data abstraction.

21.1.1 Static Semantics

The static semantics of existential types is specified by rules defining when an existential is well-formed, and by giving typing rules for the associated introductory and eliminatory forms.

$$\frac{\Delta, t\ \mathsf{type} \vdash \sigma\ \mathsf{type}}{\Delta \vdash \mathtt{some}(t.\sigma)\ \mathsf{type}} \tag{21.1}$$

It is implicit that t is chosen so that it is not already declared in Δ.

$$\frac{\Delta \vdash \tau\ \mathsf{type} \quad \Delta \vdash \mathtt{some}(t.\sigma)\ \mathsf{type} \quad \Delta;\Gamma \vdash e : [t \leftarrow \tau]\sigma}{\Delta;\Gamma \vdash \mathtt{pack}(\mathtt{some}(t.\sigma),\rho,e) : \mathtt{some}(t.\sigma)} \tag{21.2}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathtt{some}(t.\sigma) \quad \Delta, t\ \mathsf{type};\Gamma, x : \sigma \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta;\Gamma \vdash \mathtt{open}(e_1,t.x.e_2) : \tau_2} \tag{21.3}$$

This is a complex rule, so study it carefully! Two things to note:

1. The type of the client, τ_2, must not involve the abstract type t. This prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.
2. The body of the client, e_2, is type checked without knowledge of the representation type, t. The client is, in effect, polymorphic in t.

21.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified by the following rules.

$$\frac{e \longmapsto e'}{\mathtt{pack}(\tau,\rho,e) \longmapsto \mathtt{pack}(\tau,\rho,e')} \tag{21.4}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{open}(e_1,t.x.e_2) \longmapsto \mathtt{open}(e_1',t.x.e_2)} \tag{21.5}$$

$$\frac{e\ \mathsf{value}}{\mathtt{open}(\mathtt{pack}(\mathtt{some}(t.\tau),\rho,e),t.x.e_2) \longmapsto [t, x \leftarrow \rho, e]e_2} \tag{21.6}$$

Observe that *there are no abstract types at run time*! The representation type is fully exposed to the client during evaluation. Data abstraction is a compile-time discipline that imposes no run-time overhead.

21.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for Poly to the new constructs.

Theorem 21.1 (Preservation)
If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.

Lemma 21.2 (Canonical Forms)
If $e : \texttt{some}(t.\tau)$ and e value, then $e = \texttt{pack}(\texttt{some}(t.\tau),\rho,e')$ for some type ρ and some e' value such that $e' : [t{\leftarrow}\rho]\tau$.

Theorem 21.3 (Progress)
If $e : \tau$ then either e value or there exists e' such that $e \longmapsto e'$.

21.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of (persistent) queues supporting three operations:

1. Formation of the empty queue.

2. Inserting an element at the tail of the queue.

3. Remove the head of the queue.

This is clearly a bare-bones interface, but is sufficient to illustrate the main ideas of data abstraction. Queue elements may be taken to be of any type, τ, of our choosing; we will not be specific about this choice, since nothing depends on it.

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. This is captured by the following existential type, σ, which serves as the interface of the queue abstraction:[1]

$$\exists(t.\{\texttt{emp}:t, \texttt{ins}:\tau \times t \to t, \texttt{rem}:t \to \tau \times t\}).$$

[1]For the sake of illustration, we assume that type constructors such as products, records, and lists are also available in the language.

The representation type, t, of queues is *abstract* — all that is specified about it is that it supports the operations `emp`, `ins`, and `rem`, with the specified types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations. Here is a very simple implementation, e_l, in which queues are represented as lists:

$$\mathtt{pack}\,\tau\,\mathtt{list}\,\mathtt{with}\,\{\mathtt{emp}=\mathtt{nil},\mathtt{ins}=e_i,\mathtt{rem}=e_r\}\,\mathtt{as}\,\sigma,$$

where

$$e_i : \tau \times \tau\,\mathtt{list}{\rightarrow}\tau\,\mathtt{list} = \lambda(x{:}\tau \times \tau\,\mathtt{list}.\,e_i'),$$

and

$$e_r : \tau\,\mathtt{list}{\rightarrow}\tau \times \tau\,\mathtt{list} = \lambda(x{:}\tau\,\mathtt{list}.\,e_r').$$

Here the expression e_i' conses the first component of x, the element, onto the second component of x, the queue. Correspondingly, the expression e_r' reverses its argument, and returns the head element paired with the reversal of the tail. These operations "know" that queues are represented as values of type $\tau\,\mathtt{list}$, and are programmed accordingly.

It is also possible to give another implementation, e_p, of the same interface, σ, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half". This representation avoids the need for reversals on each call, and, as a result, achieves amortized constant-time behavior:

$$\mathtt{pack}\,\tau\,\mathtt{list} \times \tau\,\mathtt{list}\,\mathtt{with}\,\{\mathtt{emp}=\langle\mathtt{nil},\mathtt{nil}\rangle,\mathtt{ins}=e_i,\mathtt{rem}=e_r\}\,\mathtt{as}\,\sigma.$$

In this case e_i has type

$$\tau \times (\tau\,\mathtt{list} \times \tau\,\mathtt{list}){\rightarrow}(\tau\,\mathtt{list} \times \tau\,\mathtt{list}),$$

and e_r has type

$$(\tau\,\mathtt{list} \times \tau\,\mathtt{list}){\rightarrow}\tau \times (\tau\,\mathtt{list} \times \tau\,\mathtt{list}).$$

These operations "know" that queues are represented as values of type

$$\tau\,\mathtt{list} \times \tau\,\mathtt{list},$$

and are implemented accordingly.

Clients of the queue abstraction are shielded from the implementation details by the `open` construct. If e is *any* implementation of σ, then a client of the abstraction has the form

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{in}\,e' : \tau',$$

where the type, τ', of e' does not involve the abstract type t. Within e' the variable x has type

$$\{\texttt{emp}:t, \texttt{ins}:\tau\times t\to t, \texttt{rem}:t\to\tau\times t\},$$

in which t is unspecified — or, as is often said, *held abstract*.

Observe that *only* the type information specified in σ is propagated to the client, e', and nothing more. Consequently, the `open` expression above type checks properly regardless of whether e is e_l (the implementation of σ in terms of lists) or e_p (the implementation in terms of pairs of lists), or, for that matter, any other implementation of the same interface. This property is called *representation independence*, because the client is guaranteed to be independent of the representation of the abstraction.

21.3 Definability of Existentials

Strictly speaking, it is not necessary to extend Poly with existential types in order to model data abstraction, because they are definable in terms of universals! Before giving the details, let us consider why this should be possible. The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{in}\,e' : \tau',$$

where $e : \exists(t.\tau)$, specifies that $e' : \tau'$ under the assumptions t `type` and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau\to\tau'),$$

where t may occur in τ (the type of the operations), but not in τ' (the type of the result).

This suggests the following encoding of existential types:

$$\begin{array}{rcl} \exists(t.\sigma) & := & \forall(t'.\forall(t.\sigma\to t')\to t') \\ \texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\sigma) & := & \Lambda(t'.\lambda(x{:}\forall(t.\sigma\to t').\,x\texttt{[}\tau\texttt{]}\texttt{(}e\texttt{)})) \\ \texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x\,\texttt{in}\,e' & := & e\texttt{[}\tau'\texttt{]}\texttt{(}\Lambda(t.\lambda(x{:}\sigma.\,e'))\texttt{)} \end{array}$$

An existential is encoded as a polymorphic function taking the overall result type, t', as argument, followed by a polymorphic function representing the client with result type t', and yielding a value of type t' as overall result. Consequently, the `open` construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, τ', and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, τ', of the `open` construct.) Finally, a package consisting of a representation type τ and an implementation e is a polymorphic function that, when given the result type, t', and the client, x, instantiates x with τ and passes to it the implementation e.

It is then a straightforward exercise to show that this translation correctly reflects the static and dynamic semantics of existential types.

21.4 Exercises

Chapter 22

Dot Notation for Abstract Types

The elimination form for existential types introduced in Chapter 21 is known as a *closed-scope* abstraction mechanism, because the elimination form holds the representation type abstract for use within a particular expression. This means that all clients of an abstraction must lie within the same scope, because each open of an existential results in a "fresh" abstract type, different from all others. While this is not an inordinate convenience, it does not integrate smoothly with other forms of binding in the language, such as λ-abstraction or `let`-binding, which apply equally well to values of any type, and are not tied to the use of existential types. In particular, if we wish to model linking of separately compiled units as a form of `let`-binding, then it is natural to treat `let` as primitive, and avoid the introduction of a separate binding mechanism for existentials.

This can be achieved through the introduction of *dot notation*, which permits direct acccess to the representation type- and operations components of a package of existential type. (The terminology stems from the concrete syntax, in which we write e`.rpn` for the representation type of e and e`.ops` for its associated operations.) The dot notation is the basis for an *open-scope* abstraction mechanism, in which the abstractness of abstract types is managed by the concept of *determinacy*.

22.1 Dot Notation

A first step towards a more flexible abstraction mechanism is to decompose the machinery of data abstraction into simpler parts. At a high level this consists of the following decompositions:

1. Existential types are replaced by *signatures*, which describe the association of a type with operations on it.

2. Packages are separated into two constructs: (1) a *structure* consisting of a representation type together with its associated operations, and (2) *sealing* a structure with a signature.

3. The `open` construct is broken into two constructs: (1) *binding* a structure to an identifier for use within a scope, and (2) *projecting* the type and value components from a structure.

The effect of these modifications is to separate abstraction from binding, so that abstract types are intrinsically abstract, rather than held abstract within a specified scope. This is called *open-scope* abstraction, in contrast to the *closed-scope* abstraction provided by existential types.

We will work with an extension of Poly with the following abstract syntax:

$$\begin{array}{llcl} \textit{Types} & \tau & ::= & \texttt{sig}(t.\tau) \mid \texttt{rpn}(e) \\ \textit{Expr's} & e & ::= & \texttt{str}(\tau,e) \mid \texttt{seal}(e,\texttt{sig}(t.\tau)) \mid \texttt{ops}(e) \mid \texttt{let}(e_1,x.e_2) \end{array}$$

The corresponding concrete syntax is given by the following chart:

Abstract	*Concrete*
$\texttt{rpn}(e)$	$e\texttt{.rpn}$
$\texttt{ops}(e)$	$e\texttt{.ops}$
$\texttt{str}(\tau,e)$	$\texttt{str}(\tau,e)$
$\texttt{seal}(e,\texttt{sig}(t.\tau))$	$e :> \texttt{sig}(t.\tau)$

A type of the form $\texttt{sig}(t.\tau)$ is called a *signature*. The introductory form for a signature is a *structure* of the form $\texttt{str}(\rho,e)$. The type ρ is called the *representation type* of the structure, and e is called its *operations*. A structure is therefore just a "bare" package consisting of a representation type and associated operations, but without a specified signature. To impose a signature on an expression we use *sealing*, written $\texttt{seal}(e,\texttt{sig}(t.\tau))$, which ascribes the signature $\texttt{sig}(t.\tau)$ to the expression e, thereby hiding the representation type of e. The eliminatory forms for signature types are $\texttt{rpn}(e)$, which extracts the representation type of the structure e, and $\texttt{ops}(e)$, which extracts its operations.

22.1.1 Determinacy

The most important thing to notice about this language is that *types of the form `rpn(e)` involve expressions*. This would appear to violate the phase distinction (discussed in Chapter 8) by intermixing the dynamic and static aspects of the language (expressions and types). In particular, to determine whether two types are equal would seem now to involve determining whether two expressions are equal, for `rpn(`e`)` should be equal to `rpn(`e'`)` whenever e and e' are equal. But when are two expressions equal? At the very least this is a complex issue that is not easily resolved, particularly for a full-featured language in which expressions might not terminate, or might raise exceptions, or might perform input or output.

Fortunately, the situation is not as dire as it would appear at first, for we can impose restrictions on the use of expressions in types that avoids violating the phase distinction. The key to these restrictions is to consider carefully what is meant by the type expression `rpn(`e`)` when e is a general expression of signature type. For a type expression of the form `rpn(`e`)` to be sensible, it must be the case that e actually has a single, well-determined representation type during the static phase of processing. But this need not be the case in general! For example, if e is an infinitely looping computation, it has no value, and hence has no representation type, so what is `rpn(`e`)` supposed to mean? Worse, in a language with state, the expression e might change its value each time it is executed (*e.g.*, by flipping a coin), so that once again it makes no sense to refer to "the" representation type of e.

But even if e has a *dynamically* well-determined type component, this is not enough to ensure that e has a *statically* well-determined representation type. The point of sealing a structure with a signature is to obscure the representation type of the structure from all clients of that structure. That is, no use of that structure can rely on knowing its representation type, because that may change over time. Therefore "the" representation type of a sealed structure is not statically well-determined, even if it is dynamically well-determined.

This suggests that we limit formation of `rpn(`e`)` to expressions e whose representation type is statically well-determined. We will call such expressions *determinate*, for short. The class of determinate expressions should *include* at least structure values, which give their representation types explicitly, and variables, which are bound to structure values (in a call-by-value regimen, which we assume here). Since signatures can be nested, it also makes sense to consider as determinate the projection $\texttt{ops}(e)$ whenever e is itself determinate. On the other hand, the class of determinate structures should clearly *exclude* sealed structures, and any structures whose representation types may not be dynamically well-determined.

This leads to the following inductive definition of the judgement $e\ \mathsf{det}$ stating that e is a determinate expression.

$$\frac{}{x\ \mathsf{det}} \qquad \frac{e\ \mathsf{det}}{\texttt{ops}(e)\ \mathsf{det}} \qquad \frac{e\ \mathsf{value}}{\texttt{str}(\rho,e)\ \mathsf{det}}$$

This requires defining the judgement $e\ \mathsf{value}$ for *open* expressions e:

$$\frac{}{x\ \mathsf{value}} \qquad \frac{}{\texttt{lambda}(\tau,x.e)\ \mathsf{value}}$$

$$\frac{}{\texttt{Lambda}(t.e)\ \mathsf{value}} \qquad \frac{e\ \mathsf{value}}{\texttt{str}(t,e)\ \mathsf{value}}$$

Notice that if $e\ \mathsf{value}$, then $e\ \mathsf{det}$.

We will define the static semantics so that $\texttt{rpn}(e)$ is limited to determinate structures. In fact, we will go one step further an insist that e be a *path* according to the following rules:

$$\frac{}{x\ \mathsf{path}} \qquad \frac{e\ \mathsf{path}}{\texttt{ops}(e)\ \mathsf{path}}$$

Paths are determinate, but exclude structure values. The restriction of $\texttt{rpn}(p)$ to paths ensures that type equality never involves expression equality, except for variables. In this manner we avoid violating the phase distinction, since type equality remains syntactic identity.

22.1.2 Static Semantics

The judgements of the static semantics are now of the form $\Delta;\Gamma \vdash \tau\ \mathsf{type}$ and $\Delta;\Gamma \vdash e : \tau$, in which we have both type formation and typing assumptions in both cases.

The rules for type formation, which are mutually recursive with the rules for typing, are as follows:[1]

$$\frac{\Delta, t\ \mathsf{type};\Gamma \vdash \tau\ \mathsf{type}}{\Delta;\Gamma \vdash \texttt{sig}(t.\tau)\ \mathsf{type}} \qquad \frac{p\ \mathsf{path} \quad \Delta;\Gamma \vdash p : \texttt{sig}(t.\tau)}{\Delta;\Gamma \vdash \texttt{rpn}(p)\ \mathsf{type}}$$

[1]We omit repeating the rules for Poly from Chapter 20.

The rules for typing, which are mutually recursive with the preceding rules, are as follows:

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta;\Gamma \vdash e : [t{\leftarrow}\rho]\tau}{\Delta;\Gamma \vdash \mathtt{str}(\rho,e) : \mathtt{sig}(t.\tau)} \qquad \frac{\Delta;\Gamma \vdash e : \mathtt{sig}(t.\tau)}{\Delta;\Gamma \vdash \mathtt{seal}(e,\mathtt{sig}(t.\tau)) : \mathtt{sig}(t.\tau)}$$

$$\frac{p\ \mathsf{path} \quad \Delta;\Gamma \vdash p : \mathtt{sig}(t.\tau)}{\Delta;\Gamma \vdash \mathtt{ops}(p) : [t{\leftarrow}\mathtt{rpn}(p)]\tau} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma,x : \tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \mathtt{let}(e_1,x.e_2) : \tau_2}$$

Structure values do not have unique types, since the only requirement on the implementation, e, is that it have type $[t{\leftarrow}\rho]\tau$, from which we cannot uniquely determine the signature type $\mathtt{sig}(t.\tau)$ in the conclusion. Both elimination forms are restricted to paths. This ensures that type equality does not involve comparison of expressions other than variables.

22.1.3 Substitution

The dynamic semantics will involve substitution of structure values for variables. But the class of paths is not closed under such substitutions, because it can turn the path $\mathtt{ops}(x)$ into $\mathtt{ops}(\mathtt{str}(\rho,e))$, which is determinate, but not a path. This would turn the valid type, $\mathtt{rpn}(\mathtt{ops}(x))$ into the invalid type $\mathtt{rpn}(\mathtt{ops}(\mathtt{str}(\rho,e)))$, ruining type preservation. The simplest way around this is to define a notion of substitution that avoids creation of such illegal type expressions.

First, we define the operation of substitution of a determinate expression, e, for a variable, x, in a path, p, written $\{e{\leftarrow}x\}p$, as follows:

$$\begin{array}{rcl} \{e{\leftarrow}x\}x & = & e \\ \{e{\leftarrow}x\}\mathtt{ops}(p) & = & \begin{cases} e' & \textit{if } \{e{\leftarrow}x\}p = \mathtt{str}(\rho,e') \\ \mathtt{ops}(p') & \textit{if } \{e{\leftarrow}x\}p = p' \end{cases} \end{array}$$

This form of substitution is defined by induction on the structure of paths, and is well-defined whenever e is a determinate expression.

Second, we define substitution of a determinate expression, e, for a variable, x, in a type, τ, as follows:

$$\begin{array}{rcl} \{e{\leftarrow}x\}t & = & t \\ \{e{\leftarrow}x\}\mathtt{rpn}(p) & = & \begin{cases} \rho & \textit{if } \{e{\leftarrow}x\}p = \mathtt{str}(\rho,e') \\ \mathtt{rpn}(p') & \textit{if } \{e{\leftarrow}x\}p = p' \end{cases} \end{array}$$

This definition is justified by induction on the structure of types, and is well-defined whenever e is determinate.

The other cases of substitution are defined as usual. We shall only have need of substitution when the substituting expression is determinate.

22.1.4 Dynamic Semantics

The dynamic semantics of structures is defined by the following transition rules on closed expressions:

$$\frac{e \longmapsto e'}{\texttt{ops(}e\texttt{)} \longmapsto \texttt{ops(}e'\texttt{)}} \qquad \frac{e\ \mathsf{value}}{\texttt{ops(str(}\rho\texttt{,}e\texttt{))} \longmapsto e}$$

$$\frac{e \longmapsto e'}{\texttt{str(}\rho\texttt{,}e\texttt{)} \longmapsto \texttt{str(}\rho\texttt{,}e'\texttt{)}}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \longmapsto \texttt{let(}e_1'\texttt{,}x\texttt{.}e_2\texttt{)}} \qquad \frac{e_1\ \mathsf{value}}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \longmapsto \{e_1 \leftarrow x\}e_2}$$

Type formation is preserved by substitution of determinate expressions for free variables.

Lemma 22.1
If $\Delta;\Gamma,x:\tau \vdash \sigma\ \mathsf{type}$, $\Delta;\Gamma \vdash e:\tau$, *and* $e\ \mathsf{det}$, *then* $\Delta;\Gamma \vdash \{e\leftarrow x\}\sigma\ \mathsf{type}$.

It is straightforward to prove safety of this language.

Theorem 22.2 (Safety)
1. *If* $e:\tau$ *and* $e \longmapsto e'$, *then* $e':\tau$.

2. *If* $e:\tau$, *then either* $e\ \mathsf{value}$ *or there exists* e' *such that* $e \longmapsto e'$.

22.2 Relating Existentials to Signatures

Existential types are definable from signatures and structures:

$$\begin{array}{rcl}
\texttt{some(}t\texttt{.}\tau\texttt{)} & = & \texttt{sig(}t\texttt{.}\tau\texttt{)} \\
\texttt{pack(some(}t\texttt{.}\tau\texttt{),}\rho\texttt{,}e\texttt{)} & = & \texttt{seal(str(}\rho\texttt{,}e\texttt{),sig(}t\texttt{.}\tau\texttt{))} \\
\texttt{open(}e_1\texttt{,}t\texttt{.}x\texttt{.}e_2\texttt{)} & = & \texttt{let(}e_1\texttt{,}y\texttt{.}[t,x\leftarrow\texttt{rpn(}y\texttt{),ops(}y\texttt{)}]e_2\texttt{)} \quad (y \# e_2)
\end{array}$$

It is straightforward to verify that the static and dynamic semantics are preserved by these definitions.

It is also possible to prove a converse result, stating that the dot notation is no stronger than the original existential formalism. Since type and operation projections are limited to paths, which are rooted at variables, the binding site for variables of existential type can be used as the site at which to open the associated package. Within the scope of that binding, we may replace type- and operation projections from paths rooted at that variable with a variable introduced by the open. The only complication is that we must account for nested existentials by a corresponding nesting of open's. Thus, whenever we encounter a `let`- or λ-bound variable, x, of existential type of the form $\texttt{some(}t_1\texttt{.some(}t_2\texttt{....some(}t_n\texttt{.}\tau\texttt{)))}$ (where τ is not an existential type) and whose scope is the expression e, we replace e with the following code:

$$\texttt{open}\,x\,\texttt{as}\,t_1\,\texttt{with}\,x_1\,\texttt{in}\cdots\texttt{open}\,x_n\,\texttt{as}\,t_n\,\texttt{with}\,x_{n+1}\,\texttt{in}\,e',$$

where, for all paths p_i in e of the form $\texttt{ops(ops(}\cdots\texttt{ops(}x\texttt{)}\cdots\texttt{))}$ consisting of i operation projections from x,

$$e' = [t_i{\leftarrow}\texttt{rpn(}x_i\texttt{)}][p_i{\leftarrow}x_i]e.$$

In essence we "pre-compute" all paths rooted at x used within e by opening the existential package, and then replace uses of the ith projection from x by the variable x_i in e.

22.3 Exercises

Part VIII

Control Flow

Chapter 23

Abstract Machines

The technique of specifying the dynamic semantics as a transition system is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of "search rules" requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record "where we are" in the expression so that we may "resume" from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a *control stack*, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit the transition rules avoid the need for any premises — every rule is an axiom! This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it.

By making the control stack explicit we move closer to an actual implementation of the language. As we expose more and more of the mechanisms required in an implementation we get closer and closer to the physical machine. At each step along the way, starting with the SOS description and continuing down towards an assembly-level description, we are working with a particular *abstract*, or *virtual*, *machine*. The closer we get to the physical machine, the less "abstract" and the more "concrete" it becomes. But there is no clear dividing line between the levels, rather it is a matter of progressive exposure of implementation details. After all, even machine instructions are implemented using gates, and gates are implemented using transistors, and so on down to the level of fundamental physics.

Nevertheless, some abstract machines are more concrete than others, and recently there has been a resurgence of interest in using them to provide a hardware-independent computing platform. The idea is to define a low-enough level abstract machine such that (a) it is easily implementable on typical hardware platforms, and (b) higher-level languages can be translated (compiled) to it. In this way it is hoped that most software can be freed of dependence on specific hardware platforms.[1] It is of paramount importance that the abstract machine be precisely defined, for otherwise it is not clear how to translate to it, nor is it clear how to implement it on a given platform.

In this chapter we introduce the *C machine*, an abstract machine that makes control flow explicit. Using the tools we have developed in this book, we give a precise definition of the C machine, and show how to prove its correctness relative to the semantics of MinML.

23.1 The C Machine

A state, s, of the C machine consists of a *control stack*, k, and a closed expression, e. States may take one of two forms:

1. An *evaluation* state of the form $k > e$ corresponds to the evaluation of a closed expression, e, relative to a control stack, k.

2. A *return* state of the form $k < e$, where e value, corresponds to the evaluation of a stack, k, relative to a closed value, e.

As an aid to memory, note that the separator "points to" the focal entity of the state, the expression in an evaluation state and the stack in a return state.

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is to be returned. Formally, a control stack is a list of *frames*:

$$\frac{}{\varepsilon \ \mathsf{stack}} \qquad \frac{f \ \mathsf{frame} \quad k \ \mathsf{stack}}{f ; k \ \mathsf{stack}} \tag{23.1}$$

[1]This is much easier said than done; it remains an active area of research and development.

The definition of frame depends on the language we are evaluating. For MinML the frames are inductively defined by the following rules:[2]

$$\frac{e_2 \ \mathsf{exp}}{\mathtt{plus}(-,e_2) \ \mathsf{frame}} \qquad \frac{v_1 \ \mathsf{value}}{\mathtt{plus}(v_1,-) \ \mathsf{frame}}$$

$$\frac{e_1 \ \mathsf{exp} \quad e_2 \ \mathsf{exp}}{\mathtt{ifz}(-,e_1,e_2) \ \mathsf{frame}} \tag{23.2}$$

$$\frac{e_2 \ \mathsf{exp}}{\mathtt{app}(-,e_2) \ \mathsf{frame}} \qquad \frac{v_1 \ \mathsf{value}}{\mathtt{app}(v_1,-) \ \mathsf{frame}}$$

A frame corresponds to an elimination form in which one argument position is currently under evaluation.

The transition judgement between states of the C is inductively defined by a set of inference rules. We begin with the rules for numbers and arithmetic.

$$\frac{}{k > \mathtt{num}[n] \longmapsto k < \mathtt{num}[n]}$$

$$\frac{}{k > \mathtt{plus}(e_1,e_2) \longmapsto \mathtt{plus}(-,e_2);k > e_1}$$

$$\frac{e_1 \ \mathsf{value}}{\mathtt{plus}(-,e_2);k < e_1 \longmapsto \mathtt{plus}(e_1,-);k > e_2} \tag{23.3}$$

$$\frac{n_1 + n_2 = n \ \mathsf{nat}}{\mathtt{plus}(\mathtt{num}[n_1],-);k < \mathtt{num}[n_2] \longmapsto k < \mathtt{num}[n]}$$

To evaluate a number, we simply return it to the stack. To evaluate an addition, we push a frame onto the stack recording that we are currently working on its first argument, and continue evaluating that argument. When a value is returned to a stack whose top frame records that we are evaluating the first argument of an addition, we swap that frame with a frame recording that we have completed that evaluation, and continue by evaluating the second argument. Finally, when a value is returned to such a frame, we perform the addition and return the result to the stack.

[2] We give only the frames for the primitive operation of addition; those for multiplication and any other primitive operations are defined analogously.

Next, we consider the rules for conditionals.

$$\frac{}{k > \texttt{ifz}(e,e_1,e_2) \longmapsto \texttt{ifz}(-,e_1,e_2)\texttt{;}k > e}$$

$$\frac{}{\texttt{ifz}(-,e_1,e_2)\texttt{;}k < \texttt{num[0]} \longmapsto k > e_1} \tag{23.4}$$

$$\frac{(n \neq 0)}{\texttt{ifz}(-,e_1,e_2)\texttt{;}k < \texttt{num}[n] \longmapsto k > e_2}$$

These rules follow a similar pattern. First, the test expression is evaluated, recording the pending conditional branch on the stack. Once the value of the test has been determined, we branch to the appropriate arm of the conditional.

Finally, we consider the rules for functions.

$$\frac{}{k > \texttt{fun}(\tau_1,\tau_2,f\,.\,x\,.\,e) \longmapsto k < \texttt{fun}(\tau_1,\tau_2,f\,.\,x\,.\,e)}$$

$$\frac{}{k > \texttt{app}(e_1,e_2) \longmapsto \texttt{app}(-,e_2)\texttt{;}k > e_1}$$

$$\frac{e_1 \ \mathsf{value}}{\texttt{app}(-,e_2)\texttt{;}k < e_1 \longmapsto \texttt{app}(e_1,-)\texttt{;}k > e_2} \tag{23.5}$$

$$\frac{e_2 \ \mathsf{value} \quad e_1 = \texttt{fun}(\tau_1,\tau_2,f\,.\,x\,.\,e)}{\texttt{app}(e_1,-)\texttt{;}k < e_2 \longmapsto k > [f,x{\leftarrow}e_1,e_2]e}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated.

The initial and final states of the C are defined by the following rules:

$$\frac{}{\varepsilon > e \ \mathsf{init}} \qquad \frac{e \ \mathsf{value}}{\varepsilon < e \ \mathsf{final}} \tag{23.6}$$

The type safety of the C machine may be proved by defining a judgement s ok stating that state s is well-formed, and proving progress and preservation for this judgement. For a state to be well-formed means that its control stack is well-formed and is prepared to accept a value of type τ, and that its expression is of type τ.

$$\frac{k : \texttt{stack}(\tau) \quad e : \tau}{k > e \ \mathsf{ok}} \qquad \frac{k : \texttt{stack}(\tau) \quad e : \tau \quad e \ \mathsf{value}}{k < e \ \mathsf{ok}} \tag{23.7}$$

For a stack to be well-formed means that it is properly composed from well-formed frames. Since each frame has a single "hole" in it, the stack may be seen as accepting a value of type appropriate to that hole. Each frame fills the hole in the preceding frame, until we reach the end of the stack. This raises the question of what is the type of the empty stack? We will fix a type τ_{ans} of *answers*, the ultimate result of the evaluation of a complete program, and use this as the type of the empty stack.

$$\frac{}{\varepsilon : \mathtt{stack}(\tau_{ans})} \qquad \frac{k : \mathtt{stack}(\tau') \quad f : \mathtt{frame}(\tau,\tau')}{f;k : \mathtt{stack}(\tau)} \tag{23.8}$$

Finally, the type $\mathtt{frame}(\tau,\tau')$ is the type of frames that accept a value of type τ (for the hole) and yield a value of type τ' (once the hole is filled and the frame step is executed).

$$\frac{e_2 : \mathtt{nat}}{\mathtt{plus}(-,e_2) : \mathtt{frame}(\mathtt{nat},\mathtt{nat})} \qquad \frac{e_1 : \mathtt{nat} \quad e_1 \ \mathsf{value}}{\mathtt{plus}(e_1,-) : \mathtt{frame}(\mathtt{nat},\mathtt{nat})}$$

$$\frac{e_1 : \mathtt{nat} \quad e_2 : \mathtt{nat}}{\mathtt{ifz}(-,e_1,e_2) : \mathtt{frame}(\mathtt{nat},\mathtt{nat})}$$

$$\frac{e_2 : \tau_2}{\mathtt{app}(-,e_2) : \mathtt{frame}(\mathtt{arrow}(\tau_2,\tau),\tau)} \qquad \frac{e_1 : \mathtt{arrow}(\tau_2,\tau) \quad e_1 \ \mathsf{value}}{\mathtt{app}(e_1,-) : \mathtt{frame}(\tau_2,\tau)} \tag{23.9}$$

With these definitions in hand we may state the safety of the C in the usual manner.

Theorem 23.1 (Safety)

1. *If* s ok *and* $s \longmapsto s'$*, then* s' ok.
2. *If* s ok*, then either* s *final or there exists* s' *such that* $s \longmapsto s'$.

23.2 Correctness of the C Machine

The structured operational semantics for MinML given in Chapter 12 can be construed as a high-level abstract machine, called the M machine, whose states are closed expressions and whose transitions are as defined there. It is natural to ask whether the C machine correctly implements the semantics of MinML: if we evaluate a given expression, e, using the C machine, do we get the same result as would be given by the M machine, and *vice versa*?

Answering this question decomposes into two propositions relating the C and M machines.

> **Completeness** If $e \overset{*}{\longmapsto} v$, where v value, then $\varepsilon > e \overset{*}{\longmapsto} \varepsilon < v$.
>
> **Soundness** If $\varepsilon > e \overset{*}{\longmapsto} \varepsilon < v$, then $e \overset{*}{\longmapsto} v$.

Let us consider, in turn, what is involved in the proof of each part.

For completeness, applying rule induction to the definition of multistep transition from Chapter 3, we must show two things:

1. If v value, then $\varepsilon > v \overset{*}{\longmapsto} \varepsilon < v$.
2. If $e \longmapsto e'$, then, for every v value, if $\varepsilon > e' \overset{*}{\longmapsto} \varepsilon < v$, then $\varepsilon > e \overset{*}{\longmapsto} \varepsilon < v$.

The first follows immediately from the C machine transition rules. The second, closure under head expansion, requires some work. The obvious strategy is to proceed by induction on the SOS rules defining the M machine transition relation. The chief difficulty in the proof is that we cannot maintain an empty stack during the induction, but we must, instead, consider a general stack, k, in order to complete the argument.

Lemma 23.2
If $e \longmapsto e'$, then for every v value and every k stack, if $k > e' \overset{*}{\longmapsto} \varepsilon < v$, then $k > e \overset{*}{\longmapsto} \varepsilon < v$.

For soundness, observe that it is awkward to reason inductively about the multistep transition from $\varepsilon > e \overset{*}{\longmapsto} \varepsilon < v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each C machine state as encoding an expression, and show that C machine transitions are simulated by M machine transitions under this encoding.

Specifically, we define a judgement, $s \rightsquigarrow e$, stating that state s "unravels to" expression e. It will turn out that for initial states we have $\varepsilon > e \rightsquigarrow e$

and for final states $\varepsilon < v \rightsquigarrow v$. Then we show that if $s \overset{*}{\longmapsto} s'$, where s' final, $s \rightsquigarrow e$, and $s' \rightsquigarrow e'$, then $e \overset{*}{\longmapsto} e'$. Applying rule induction to the definition of multistep transition, it is enough to show the following two facts:

1. If $s \rightsquigarrow e$, then $e \overset{*}{\longmapsto} e$.
2. If $s \longmapsto s'' \overset{*}{\longmapsto} s'$ with s' final, then if $s \rightsquigarrow e$, $s'' \rightsquigarrow e''$, $s' \rightsquigarrow e'$, and $e'' \overset{*}{\longmapsto} e'$, then $e \overset{*}{\longmapsto} e'$.

The first is trivial; for the second, it is enough to show the following lemma.

Lemma 23.3
If $s \longmapsto s'$, $s \rightsquigarrow e$, and $s' \rightsquigarrow e'$, then $e \overset{*}{\longmapsto} e'$.

The remainder of this section is devoted to the proofs of these lemmas.

23.2.1 Proof of Completeness

Proof: [of Lemma 23.2] The proof is by induction on the transition rules defining the M machine. We will consider some representative cases here, leaving the rest for the reader.

Suppose that $e = \texttt{plus}(e_1, e_2)$, $e' = \texttt{plus}(e_1', e_2)$, and $e_1 \longmapsto e_1'$. Suppose further that $k > e' \overset{*}{\longmapsto} \varepsilon < v$ for some v value. Given the form of e', the initial transition must be of the form

$$k > e' \longmapsto \texttt{plus}(-, e_2); k > e_1' \overset{*}{\longmapsto} \varepsilon < v.$$

By induction we have $\texttt{plus}(-, e_2); k > e_1 \overset{*}{\longmapsto} \varepsilon < v$, and hence

$$k > \texttt{plus}(e_1, e_2) \longmapsto \texttt{plus}(-, e_2); k > e_1 \overset{*}{\longmapsto} \varepsilon < v.$$

Note that it is important here that the induction hypothesis be universally quantified with respect to the stack portion of the state!

Suppose that $e = \texttt{plus}(\texttt{num[}n_1\texttt{]}, \texttt{num[}n_2\texttt{]})$ and $e' = \texttt{num[}n\texttt{]}$, where $n_1 + n_2 = n$ nat. Suppose further that $k > \texttt{num[}n\texttt{]} \overset{*}{\longmapsto} \varepsilon < v$ for some v value. This must have the form $k > \texttt{num[}n\texttt{]} \longmapsto k < \texttt{num[}n\texttt{]} \overset{*}{\longmapsto} \varepsilon < v$.

Then we have

$$\begin{array}{rcl} k > \texttt{plus}(\texttt{num}[n_1],\texttt{num}[n_2]) & \longmapsto & \texttt{plus}(-,\texttt{num}[n_2]);k > \texttt{num}[n_1] \\ & \longmapsto & \texttt{plus}(-,\texttt{num}[n_2]);k < \texttt{num}[n_1] \\ & \longmapsto & \texttt{plus}(\texttt{num}[n_1],-);k > \texttt{num}[n_2] \\ & \longmapsto & \texttt{plus}(\texttt{num}[n_1],-);k < \texttt{num}[n_2] \\ & \longmapsto & k < \texttt{num}[n] \\ & \overset{*}{\longmapsto} & \varepsilon < v \end{array}$$

The other cases follow a similar pattern. ■

23.2.2 Proof of Soundness

We first define the unravelling translation, $k \mathbin{@} e \rightsquigarrow e'$, by the following rules:

$$\frac{}{\varepsilon \mathbin{@} e \rightsquigarrow e}$$

$$\frac{k \mathbin{@} \texttt{plus}(e_1,e_2) \rightsquigarrow e}{\texttt{plus}(-,e_2);k \mathbin{@} e_1 \rightsquigarrow e} \qquad \frac{k \mathbin{@} \texttt{plus}(e_1,e_2) \rightsquigarrow e}{\texttt{plus}(e_1,-);k \mathbin{@} e_2 \rightsquigarrow e}$$

$$\frac{k \mathbin{@} \texttt{ifz}(e_1,e_2,e_3) \rightsquigarrow e}{\texttt{ifz}(-,e_2,e_3);k \mathbin{@} e_1 \rightsquigarrow e}$$

$$\frac{k \mathbin{@} \texttt{app}(e_1,e_2) \rightsquigarrow e}{\texttt{app}(-,e_2);k \mathbin{@} e_1 \rightsquigarrow e} \qquad \frac{k \mathbin{@} \texttt{app}(e_1,e_2) \rightsquigarrow e}{\texttt{app}(e_1,-);k \mathbin{@} e_2 \rightsquigarrow e}$$

The notation $k \mathbin{@} e$ is stands ambiguously for either form of state, since the distinction does not affect the unravelling translation.

Observe that if $e \longmapsto e'$, $k \mathbin{@} e \rightsquigarrow d$, $k \mathbin{@} e' \rightsquigarrow d'$, then $d \longmapsto d'$. In other words unravelling the stack around a transition does not affect the transition.

We are now in a position to complete the proof of soundness.

Proof: [of Lemma 23.3]

The proof is by case analysis on the transitions of the C machine. In each case after unravelling the transition will correspond to zero or one transitions of the M machine.

Suppose that $s = k > \texttt{plus}(e_1,e_2)$ and $s' = \texttt{plus}(-,e_2) > e_1$. Note that $k \mathbin{@} \texttt{plus}(e_1,e_2) \rightsquigarrow e$ iff $\texttt{plus}(-,e_2);k \mathbin{@} e_1 \rightsquigarrow e$, from which the result is immediate.

Suppose that $s = \texttt{plus}(\texttt{num[}n_1\texttt{]},-);k < \texttt{num[}n_2\texttt{]}$, $s' = k < \texttt{num[}n\texttt{]}$, where $n_1 + n_2 = n$ nat. Let e be such that $s \leadsto e$, and note that

$$k \mathbin{@} \texttt{plus}(\texttt{num[}n_1\texttt{]},\texttt{num[}n_2\texttt{]}) \leadsto e$$

as well. Let e' be such that $s' \leadsto e'$, and so $e \longmapsto e'$, as required.

■

23.3 The E Machine

The C machine is still quite "high level" in that function application is performed by substitution of the function itself and its argument into the body of the function, a rather complex operation. This is unrealistic for two reasons. Substitution is a complicated process, not one that we would ordinarily think of as occurring as a single step of execution of a computer. More importantly, the use of substitution means that the program itself, and not just the data it acts upon, changes during evaluation. This is a departure from more familiar models of computation, which maintain a separation between programs and data.

In this section we will present another abstract machine, the E machine, which avoids substitution by maintaining an environment that records the bindings of variables. This introduces complications to do with confusion of variables similar to those discussed in Chapter 14, and we use a similar solution, namely closures, to avoid them.

A significant difference compared to the C machine is that values are no longer forms of expression, but are rather drawn from a new class of *E machine values*, V. Corresponding, *E machine environments*, η, bind machine values to variables. Finally, E machine stacks, K, and frames, F, may involve environments, and hence are not purely syntactic either.

The states of the E have one of two forms:

1. $K > e\ [\eta]$, corresponding to evaluating the expression e on the stack K relative to the environment η.

2. $K < V$, corresponding to returning the value, V, to the stack, K.

These states are similar to those for the C machine.

The judgements η mvalue, K mstack, F mframe, and V menv are simultaneously inductively defined by the following rules. First, an E machine value is either a number or a closure.

$$\frac{n \text{ nat}}{\texttt{num[}n\texttt{]} \text{ mvalue}} \qquad \frac{\texttt{fun(}\tau_1,\tau_2,f\,.\,x\,.\,e\texttt{)} \text{ exp} \quad \eta \text{ menv}}{\texttt{fun(}\tau_1,\tau_2,f\,.\,x\,.\,e\texttt{)}[\eta] \text{ mvalue}} \tag{23.10}$$

An E machine stack is a sequence of E machine frames.

$$\frac{}{\varepsilon \text{ mstack}} \qquad \frac{F \text{ mframe} \quad K \text{ mstack}}{F;K \text{ mstack}} \tag{23.11}$$

An E machine frame is similar to a C frame, except for the attachment of environments to frames that contain expressions.

$$\frac{e_2 \text{ exp}}{\texttt{plus}(-,e_2)[\eta] \text{ mframe}} \qquad \frac{V_1 \text{ mvalue}}{\texttt{plus}(V_1,-) \text{ mframe}}$$

$$\frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{\texttt{ifz}(-,e_1,e_2)[\eta] \text{ mframe}} \tag{23.12}$$

$$\frac{e_2 \text{ exp}}{\texttt{app}(-,e_2)[\eta] \text{ mframe}} \qquad \frac{V_1 \text{ mvalue}}{\texttt{app}(V_1,-) \text{ mframe}}$$

An E environment is a sequence of bindings of variables to E values such that no variable is bound more than once.

$$\frac{}{\varepsilon \text{ menv}} \qquad \frac{\eta \text{ menv} \quad x \# \eta \quad V \text{ mvalue}}{\eta, x{=}V \text{ menv}} \tag{23.13}$$

The transition rules for the E machine are given as follows.

To evaluate a variable x, we look up its binding and pass the associated value to the top frame of the control stack.

$$\frac{\eta(x) = V}{K > x\ [\eta] \longmapsto K < V} \tag{23.14}$$

Arithmetic is handled similarly to the C, except that we must be careful to close expressions that may have free variables in them.

$$\frac{}{K > \texttt{num[}n\texttt{]}\ [\eta] \longmapsto K < \texttt{num[}n\texttt{]}} \tag{23.15}$$

$$\frac{}{K > \texttt{plus}(e_1,e_2)\ [\eta] \longmapsto \texttt{plus}(-,e_2)[\eta];K > e_1\ [\eta]} \tag{23.16}$$

$$\frac{}{\texttt{plus}(-,e_2)[\eta];K < V_1 \longmapsto \texttt{plus}(V_1,-);K > e_2\ [\eta]} \tag{23.17}$$

$$\frac{n_1 + n_2 = n \ \mathsf{nat}}{\texttt{plus}(\texttt{num}[n_1],-);K < \texttt{num}[n_2] \longmapsto K < \texttt{num}[n]} \tag{23.18}$$

To evaluate a conditional, we evaluate the test expression, pushing a frame on the control stack to record the two pending branches, once again closed with respect to the current environment.

$$\frac{}{K > \texttt{ifz}(e,e_1,e_2)\ [\eta] \longmapsto \texttt{ifz}(-,e_1,e_2)[\eta];K > e\ [\eta]} \tag{23.19}$$

$$\frac{}{\texttt{ifz}(-,e_1,e_2)[\eta];K < \texttt{num}[0] \longmapsto K > e_1\ [\eta]} \tag{23.20}$$

$$\frac{(n \neq 0)}{\texttt{ifz}(-,e_1,e_2)[\eta];K < \texttt{num}[n] \longmapsto K > e_2\ [\eta]} \tag{23.21}$$

To evaluate a function expression, we close it with respect to the current environment to ensure that its free variables are not inadvertently captured, and pass the resulting closure to the control stack.

$$\frac{}{K > \texttt{fun}(\tau_1,\tau_2,f.x.e)\ [\eta] \longmapsto K < \texttt{fun}(\tau_1,\tau_2,f.x.e)[\eta]} \tag{23.22}$$

The notation here may be a bit deceptive. On the left-hand side the environment, η, is part of the machine state, whereas on the right-hand side it is attached to the function expression to form a closure.

Finally, function applications are evaluated similarly to the C, except that care must be taken with the environment.

$$\frac{}{K > \texttt{app}(e_1,e_2)\ [\eta] \longmapsto \texttt{app}(-,e_2)[\eta];K > e_1\ [\eta]} \tag{23.23}$$

$$\frac{}{\texttt{app}(-,e_2)[\eta];K < V \longmapsto \texttt{app}(V,-);K > e_2\ [\eta]} \tag{23.24}$$

$$\frac{V_1 = \texttt{fun}(\tau_1, \tau_2, f\,.\,x\,.\,e)[\eta]}{\texttt{app}(V_1, -); K < V_2 \longmapsto K > e\ [\eta, f{=}V_1, x{=}V_2]} \qquad (23.25)$$

Notice that the environment of the closure is installed as the environment of execution for the body, augmented with bindings for the function itself and its argument.

Initial states have the form $\varepsilon > e\ [\varepsilon]$, with the stack and environment initially empty. The final states of the E machine have the form $\varepsilon < V$, where V mvalue, and the stack is again empty.

23.4 Exercises

1. Prove type safety for the C machine.
2. Finish the proofs of the soundness and completeness lemmas.
3. State and prove the correctness of the E relative to the C.

Chapter 24

Exceptions

Exceptions effects a non-local transfer of control from the point at which the exception is *raised* to a dynamically enclosing *handler* for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition, or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, one could use explicit conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly since the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

24.1 Failures

To begin with let us consider a simple control mechanism, called *failures*, which permits the evaluation of an expression to "fail" by passing control to the nearest enclosing handler, which is said to "catch" the failure. Failures are a simplified form of exception in which no value is associated with the failure. This allows us to concentrate on the control flow aspects, and to treat the associated value separately.

The following grammar describes an extension to MinML to include failures:

$$\textit{Expr's} \quad e \quad ::= \quad \texttt{fail} \mid \texttt{catch}(e_1, e_2)$$

The expression `fail` aborts the current evaluation. The expression $\texttt{catch}(e_1, e_2)$

evaluates e_1. If it terminates normally, its value is returned; if it fails, its value is the value of e_2.

The static semantics of exceptions is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{24.1}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch(}e_1\texttt{,}e_2\texttt{)} : \tau} \tag{24.2}$$

Observe that a failure can have any type, because it never returns to the site of the failure. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

Stack Unwinding

One way to give a dynamic semantics to failures is to use a technique called *stack unwinding*. Evaluation of a `catch` installs a handler on the control stack. Evaluation of a `fail` unwinds the control stack by popping frames until it reaches the nearest enclosing handler, to which control is passed. The handler is evaluated in the context of the surrounding control stack, so that failures within it propagate further up the stack.

This behavior is naturally specified usng the C machine (defined in Chapter 23), because it makes the control stack explicit. The set of frames is extended with the following additional rules:

$$\frac{e_2 \ \mathsf{expr}}{\texttt{catch(}-\texttt{,}e_2\texttt{)}\ \mathsf{frame}} \tag{24.3}$$

The transition rules of the C machine are extended with the following additional rules:

$$\frac{}{k > \texttt{catch(}e_1\texttt{,}e_2\texttt{)} \longmapsto \texttt{catch(}-\texttt{,}e_2\texttt{);}k > e_1} \tag{24.4}$$

$$\frac{}{\texttt{catch(}-\texttt{,}e_2\texttt{);}k < v \longmapsto k < v} \tag{24.5}$$

$$\frac{}{\texttt{catch(}-\texttt{,}e_2\texttt{);}k > \texttt{fail} \longmapsto k > e_2} \tag{24.6}$$

$$\frac{(f \neq \texttt{catch}(-,e_2))}{f;k > \texttt{fail} \longmapsto k > \texttt{fail}} \tag{24.7}$$

To evaluate $\texttt{catch}(e_1,e_2)$ we begin by evaluating e_1. If it achieves a value, we pop the pending handler and yield that value. If, however, it fails, we continue by evaluating the nearest enclosing handler. We explicitly pop non-handler frames while processing a failure; this is sometimes called *unwinding* the control stack.

The definition of initial state remains the same as for the C machine, but we change the definition of final state to include these two forms:

$$\frac{e \ \mathsf{value}}{\varepsilon < e \ \mathsf{final}} \qquad \frac{}{\varepsilon > \texttt{fail} \ \mathsf{final}} \tag{24.8}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 23 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different! A well-typed program does not "get stuck" ... but it may well raise an uncaught exception.

Theorem 24.1 (Safety)

1. *If s ok and $s \longmapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \longmapsto s'$.*

Handler Stacks

Most implementations of exceptions do not unwind the stack frame-by-frame, but rather implement a direct transfer of control to the appropriate handler. But it is not just a matter of an indirect jump; the control stack must also be reset to what it was at the point that handler was established. This can be modelled by augmenting the abstract machine state with a stack of stacks, called the *handler stack*, which records the information required for a non-local control transfer.

Let us call the augmented machine the *H machine*. The states of the H machine have one of two forms:

1. $h \mid k > e$ corresponding to evaluation of expression e on control stack k and handler stack h;

2. $h \mid k < e$, corresponding to return of a value e to control stack k and handler stack h.

A handler stack is just a stack of control stacks:

$$\frac{}{\varepsilon\ \mathsf{hstack}} \qquad \frac{k\ \mathsf{stack} \quad h\ \mathsf{hstack}}{k;h\ \mathsf{hstack}} \tag{24.9}$$

A crucial invariant of the H machine execution is that each control stack on the handler stack is an extension of the preceding one.

The key transition rules for the H machine are as follows. Evaluating a handler installs a new handler by pushing an extension of the current control stack onto the handler stack:

$$\frac{}{h \mid k > \texttt{catch(}e_1\texttt{,}e_2\texttt{)} \longmapsto (\texttt{catch(}-\texttt{,}e_2\texttt{);}k)\texttt{;}h \mid \texttt{catch(}-\texttt{,}e_2\texttt{);}k > e_1} \tag{24.10}$$

Notice that the same frame is pushed onto k to form the top of the handler stack and the current control stack.

When a value is returned to a handler context, both the control stack and the handler stack must be popped:

$$\frac{e_1\ \mathsf{value}}{(\texttt{catch(}-\texttt{,}e_2\texttt{);}k)\texttt{;}h \mid \texttt{catch(}-\texttt{,}e_2\texttt{);}k < e_1 \longmapsto h \mid k < e_1} \tag{24.11}$$

On normal return from an expression guarded by a handler, the handler is removed and control passes up the control stack.

When a failure occurs, the current control stack is disregarded, and is replaced by the top of the handler stack, which is itself popped:

$$\frac{}{(\texttt{catch(}-\texttt{,}e_2\texttt{);}k)\texttt{;}h \mid k' > \texttt{fail} \longmapsto h \mid k > e_2} \tag{24.12}$$

Control is passed to the handler, running on the control stack in effect at the time the handler was installed. If the handler stack is empty, the failure is uncaught, and we stop the machine:

$$\frac{}{\varepsilon \mid k > \texttt{fail} \longmapsto \varepsilon \mid \varepsilon > \texttt{fail}} \tag{24.13}$$

Two invariants of the H machine are crucial to its implementation:

1. Each control stack on the handler stack is an extension of its predecessor.

2. Each control stack on the handler stack is a (not necessarily proper) prefix of the current control stack.

This means that we can implement the handler stack as a stack of "pointers" into the control stack recording the spots at which handlers are placed. On failure we simply pop the control stack to the point indicated by the top of the handler stack, pop that handler stack, and resume execution.

The prefix property may be taken as a formal justification of an implementation based on the `setjmp` and and `longjmp` constructs of the C language. Unlike `setjmp` and `longjmp`, the exception mechanism is completely safe — it is impossible to return past the "finger" yet later attempt to "pop" the control stack to that point. In C the fingers are kept as addresses (pointers) in memory, and there is no discipline for ensuring that the set point makes any sense when invoked later in a computation.

Failure-Passing Style

Another approach to giving the semantics of exception is to eliminate exceptions by translating into a language with sum types. An expression of type τ is translated into one of type $\texttt{unit}+\tau$, where the left summand represents the occurrence of an error, and the right summand represents the normal return of a value of type τ. Each language construct is translated so as to case analyze on the occurrence of an error, propagating failures upward, except for the `catch` construct, which re-directs errors to the handler.

This interpretation is best described as a *type-directed translation* between the *source* language, the extension of MinML with exceptions, to the *target* language, the extension of MinML with sum types. A type-directed translation is specified by two judgement forms:

1. A type translation, $\tau \rightsquigarrow \tau'$, which states that source type τ is translated to target type τ'.

2. An expression translation, $e : \tau \rightsquigarrow e' : \texttt{unit}+\tau'$, which states that the source expression $e : \tau$ translates to the target expression $e' : \texttt{unit}+\tau'$, where $\tau \rightsquigarrow \tau'$.

To account for free variables in the expression e we must consider hypothetical translation judgements of the form

$$\Theta \vdash e : \tau \leadsto e' : \texttt{unit} + \tau'.$$

The hypothesis list Θ consists of hypotheses of the form

$$x_i : \tau_i \leadsto \texttt{inr}\,(x_i') : \texttt{unit} + \tau_i',$$

where $\tau_i \leadsto \tau_i'$. Each such hypothesis implicitly declares the source language variable $x_i : \tau_i$ and the target language variable $x_i' : \tau_i'$, and specifies the translation of the one in terms of the other.

The rules for the type translation are as follows:

$$\frac{}{\texttt{nat} \leadsto \texttt{nat}} \qquad \frac{\tau_1 \leadsto \tau_1' \quad \tau_2 \leadsto \tau_2'}{\tau_1 \to \tau_2 \leadsto \tau_1' \to (\texttt{unit} + \tau_2')}$$

And here are the corresponding rules for the type translation:

$$\frac{}{\Theta \vdash n : \texttt{nat} \leadsto \texttt{inr}\,(n) : \texttt{unit} + \texttt{nat}}$$

$$\frac{\tau_1 \leadsto \tau_1' \quad \Theta, x : \tau_1 \leadsto \texttt{inr}\,(x') : \texttt{unit} + \tau_1' \vdash e : \tau_2 \leadsto e' : \texttt{unit} + \tau_2'}{\Theta \vdash \lambda\,(x{:}\tau_1.\,e) : \tau_1 \to \tau_2 \leadsto \texttt{inr}\,(\lambda\,(x'{:}\tau_1'.\,e')) : \texttt{unit} + (\tau_1' \to (\texttt{unit} + \tau_2'))}$$

$$\frac{\Theta \vdash e_1 : \tau_2 \to \tau \leadsto e_1' : \texttt{unit} + (\tau_2' \to (\texttt{unit} + \tau')) \quad \Theta \vdash e_2 : \tau_2 \leadsto e_2' : \texttt{unit} + \tau_2'}{\Theta \vdash e_1\,(e_2) : \tau \leadsto e' : \texttt{unit} + \tau'}$$

$$\textit{where} \left\{ \begin{array}{lcl} e' & = & \texttt{let}\,x_1'\,\texttt{be}\,e_1'\,\texttt{in}\,\texttt{let}\,x_2'\,\texttt{be}\,e_2'\,\texttt{in}\,e'' \\ e'' & = & \texttt{case}\,x_1'\,\{\,\texttt{inl}(\_{:}\texttt{unit}) \Rightarrow \texttt{inl}\,(\langle\rangle) \mid \texttt{inr}(y_1'{:}\tau_2' \to (\texttt{unit} + \tau')) \Rightarrow e'''\,\} \\ e''' & = & \texttt{case}\,x_2'\,\{\,\texttt{inl}(\_{:}\texttt{unit}) \Rightarrow \texttt{inl}\,(\langle\rangle) \mid \texttt{inr}(y_2'{:}\tau_2') \Rightarrow y_1'\,(y_2')\,\} \end{array} \right.$$

$$\frac{\tau \leadsto \tau'}{\Theta \vdash \texttt{fail} : \tau \leadsto \texttt{inl}\,(\langle\rangle) : \texttt{unit} + \tau'}$$

$$\frac{\Theta \vdash e_1 : \tau \leadsto e_1' : \texttt{unit} + \tau' \quad \Theta \vdash e_2 : \tau \leadsto e_2' : \texttt{unit} + \tau'}{\Theta \vdash \texttt{try}\,e_1\,\texttt{ow}\,e_2 : \tau \leadsto e' : \texttt{unit} + \tau'}$$

$$\textit{where} \quad e' \;=\; \texttt{case}\,e_1'\,\{\,\texttt{inl}(\_{:}\texttt{unit}) \Rightarrow e_2' \mid \texttt{inr}(x_1'{:}\tau') \Rightarrow \texttt{inr}\,(x_1')\,\}$$

24.2 Exceptions

Let us now consider enhancing the simple failures mechanism of the preceding section with an exception mechanism that permits a value to be associated with the failure, which is then passed to the handler as part of the control transfer. The syntax of exceptions is given by the following grammar:

$$\textit{Expr's} \quad e \quad ::= \quad \texttt{raise}(e) \mid \texttt{handle}(e_1, x.e_2)$$

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\texttt{handle}(e_1, x.e_2)$ binds a variable, x, in the handler, e_2, to which the associated value of the exception is bound, should an exception be raised during the execution of e_1.

The static semantics of exceptions is a straightforward generalization of that of failures.

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}(e) : \tau} \tag{24.14}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1, x.e_2) : \tau} \tag{24.15}$$

These rules are parameterized by the type of values associated with exceptions, τ_{exn}. But what should be the type τ_{exn}?

The first thing to observe is that *all* exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by *any* raise expression occurring during the execution of the expression that it guards. If different exceptions could have different associated values, the handler could not predict (statically) what type of value to expect, and hence could not dispatch on it without violating type safety.

Since the data associated with an exception is intended to indicate the reason for the failure, it may seem reasonable to choose τ_{exn} to be a string that describes the reason for the failure. For example, one might write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault, or

```
raise "File not found."
```

to indicate failure to open a specified file. While this might be reasonable for exceptions that are not intended to be caught, it is quite unreasonable for those that may wish to be caught by an exception handler — it would have to parse the associated string, according to some conventions, to determine what happened and how to respond! Similar criticisms apply to choosing τ_{exn} to be, say, `nat`, associating an "error number" with each form of failure, and requiring the handler to dispatch on the number. This, too, is obviously rather primitive and error-prone, and would not permit a natural means of associating other, exception-specific data with the failure.

A much more reasonable choice would be to distinguish a labelled sum type of the form

$$\tau_{exn} = [\texttt{div}:\texttt{unit},\texttt{fnf}:\texttt{string},\ldots].$$

Each variant of the sum specifies the type of data associated with that variant. The handler may perform a case analysis on the tag of the variant, thereby recovering the underlying data value of the appropriate type. For example,

```
try $e_1$ ow x.case x {
  div $\langle\rangle$ ⇒ $e_{div}$
| fnf $s$ ⇒ $e_{fnf}$
| ··· }
```

This code closely resembles the exception mechanisms found in many well-known languages.

The only difficulty with this choice of τ_{exn} is that we must specify it *globally* for the entire program. This precludes associating types with exceptions that only make sense within a region of code, and precludes introducing new exceptions in a modular fashion. For this reason a natural choice is to introduce an *extensible sum* type, which is similar to the labelled sum type, except that new cases can be added to it *dynamically*. This is a very useful notion beyond its role as providing a convenient type of exception values; we will discuss extensible sums in more detail in Chapter 31.

24.3 Exercises

1. Hand-simulate the evaluation of a few simple expressions with exceptions and handlers to get a feeling for how it works.

2. State and prove the safety of the formulation of exceptions using a handler stack.

3. Prove that the H machine indeed maintains the invariants stated above.

Chapter 25

Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. First-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are *persistent* data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a "new" version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar *ephemeral* data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

Returning to the case in point, the standard implementation of a control stack is as an ephemeral data structure, a pointer to a region of mutable storage that is overwritten whenever we push a frame. This makes it impossible to maintain an "old" and a "new" copy of the control stack at the same time, making time travel impossible. If, however, we represent the control stack as a persistent data structure, then we can easily reify a control stack by simply binding it to a variable, and continue working. If we wish we can easily return to that control stack by referring to the variable that is bound to it. This is achieved in practice by representing the control stack as a list of frames in the heap so that the persistence of lists can be extended to control stacks. While we will not be specific about implementation strategies in this note, it should be born in mind when considering the semantics outlined below.

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor. In Section 26 we will show how to build a threads package for concurrent programming using continuations.

25.1 Informal Overview

We will extend MinML with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. The introductory form for $\texttt{cont}(\tau)$ is $\texttt{letcc}(\tau,x.e)$, which binds the *current continuation* (*i.e.*, the current control stack) to the variable x, and evaluates the expression e. The eliminatory form is $\texttt{throw}(e_1,e_2)$, which restores the value of e_1 to the control stack that is the value of e_2.[1] This description makes clear the need for a persistent representation of control stacks so that they may be bound to variables and restored a value is thrown to one.

[1]Close relatives of these primitives are available in SML/NJ in the following forms: for $\texttt{letcc}(\tau,x.e)$, write $\texttt{SMLofNJ.Cont.callcc (fn } x \texttt{ => } e\texttt{)}$, and for $\texttt{throw}(e_1,e_2)$, write $\texttt{SMLofNJ.Cont.throw } e_2\ e_1$.

Here is a simple example, written in an informal concrete syntax. The idea is to multiply the elements of a list, short-circuiting the computation in case zero is encountered. Here's the code:

```
fun mult_list (l:int list):int =
    letcc ret:int cont in
      let fun mult nil = 1
            | mult (0::_) = throw 0 to ret
            | mult (n::l) = n * mult l
      in  mult l end
```

Ignoring the `letcc` for the moment, the body of `mult_list` is a `let` expression that defines a recursive procedure `mult`, and applies it to the argument of `mult_list`. The job of `mult` is to return the product of the elements of the list. Ignoring the second line of `mult`, it should be clear why and how this code works.

Now let's consider the second line of `mult`, and the outer use of `letcc`. Intuitively, the purpose of the second line of `mult` is to short circuit the multiplication, returning `0` immediately in the case that a `0` occurs in the list. This is achieved by throwing the value `0` (the final answer) to the continuation bound to the variable `ret`. This variable is bound by `letcc` surrounding the body of `mult_list`. What continuation is it? It's the continuation that runs upon completion of the body of `mult_list`. This continuation would be executed in the case that no `0` is encountered and evaluation proceeds normally. In the unusual case of encountering a `0` in the list, we branch directly to the return point, passing the value `0`, effecting an early return from the procedure with result value `0`.

Here's another formulation of the same function:

```
fun mult_list l =
    let fun mult nil ret = 1
          | mult (0::_) ret = throw 0 to ret
          | mult (n::l) ret = n * mult l ret
     in letcc ret:int cont in (mult l) ret end
```

Here the inner loop is parameterized by the return continuation for early exit. The multiplication loop is obtained by calling `mult` with the current continuation at the exit point of `mult_list` so that throws to `ret` effect an early return from `mult_list`, as desired.

Let's look at another example: given a continuation k of type τ `cont` and a function f of type $\tau' \rightarrow \tau$, return a continuation k' of type τ' `cont` with the following behavior: throwing a value v' of type τ' to k' throws the value $f(v')$ to k. This is called *composition of a function with a continuation*. We wish to fill in the following template:

```
fun compose(f:τ′→τ,k:τ cont):τ′ cont = ....
```

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. This is the continuation that, when given a value v', applies f to it, and throws the result to k. We can seize this continuation using `letcc`, writing

```
throw f(letcc x:τ′ cont in ...) to k
```

At the point of the ellipsis the variable x is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code:

```
fun compose (f:τ′→τ, k:τ cont):τ′ cont =
    letcc ret:τ′ cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

Notice that the type of `ret` is that of a continuation-expecting continuation!

We can do without first-class continuations by creating our own during execution. The idea is that we can perform (by hand or automatically) a systematic program transformation in which a "copy" of the control stack is maintained as a function, called a continuation. Every function takes as an argument the control stack to which it is to pass its result by applying given stack (represented as a function) to the result value. Functions never return in the usual sense; they pass their result to the given continuation. Programs written in this form are said to be in *continuation-passing style*, or *CPS* for short.

Here's the code to multiply the elements of a list, without short-circuiting, in continuation-passing style:

```
    fun cps_mult nil k = k 1
      | cps_mult (n::l) k = cps_mult l (fn r => k (n * r))
    fun mult l = cps_mult l (fn r => r)
```

The short-circuiting version is just as simple:

```
fun cps_mult_list l k =
    let fun cps_mult nil k0 k = k 1
      | fun cps_mult (0::_) k0 k = k0 0
      | fun cps_mult (n::l) k0 k = cps_mult k0 l (fn p => k (n*p))
    in cps_mult l k k end
```

The continuation `k0` never changes; it is always the return continuation for `cps_mult_list`. The argument continuation to `cps_mult_list` is duplicated on the call to `cps_mult`.

25.2 Semantics of Continuations

We extend the language of MinML expressions with these additional forms:

$$\begin{array}{lll} \textit{Types} & \tau & ::= \ \texttt{cont}(\tau) \\ \textit{Expr's} & e & ::= \ \texttt{letcc}(\tau, x.e) \mid \texttt{throw}(e_1, e_2) \mid \texttt{cont}(k) \end{array}$$

The expression $\texttt{cont}(k)$ is a reified control stack; they arise during evaluation, but are not available as expressions to the programmer.

The static semantics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}(\tau, x.e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}(e_1, e_2) : \tau'} \tag{25.1}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The static semantics of continuation values is given by the following rule:

$$\frac{k : \texttt{stack}(\tau)}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{25.2}$$

A continuation value $\texttt{cont}(k)$ has type $\texttt{cont}(\tau)$ exactly if it is a stack accepting values of type τ.

To define the dynamic semantics, we extend the C machine stacks with two new forms of frame:

$$\frac{e_2 \text{ exp}}{\texttt{throw(}-\texttt{,}e_2\texttt{)} \text{ frame}} \qquad \frac{e_1 \text{ value}}{\texttt{throw(}e_1\texttt{,}-\texttt{)} \text{ frame}} \tag{25.3}$$

Every reified control stack is a value:

$$\frac{k \text{ stack}}{\texttt{cont(}k\texttt{)} \text{ value}} \tag{25.4}$$

The transition rules for the continuation constructs are as follows:

$$\frac{}{k > \texttt{letcc(}\tau\texttt{,}x\texttt{.}e\texttt{)} \longmapsto k > [x \leftarrow \texttt{cont(}k\texttt{)}]e} \tag{25.5}$$

$$\frac{}{\texttt{throw(}v\texttt{,}-\texttt{);}k < \texttt{cont(}k'\texttt{)} \longmapsto k' < v} \tag{25.6}$$

$$\frac{}{k > \texttt{throw(}e_1\texttt{,}e_2\texttt{)} \longmapsto \texttt{throw(}-\texttt{,}e_2\texttt{);}k > e_1} \tag{25.7}$$

$$\frac{e_1 \text{ value}}{\texttt{throw(}-\texttt{,}e_2\texttt{);}k < e_1 \longmapsto \texttt{throw(}e_1\texttt{,}-\texttt{);}k > e_2} \tag{25.8}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of MinML may be established by a simple extension to the safety proof for the C machine given in Chapter 23.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \texttt{cont(}\tau\texttt{)}}{\texttt{throw(}-\texttt{,}e_2\texttt{)} : \texttt{frame(}\tau\texttt{,}\tau'\texttt{)}} \qquad \frac{e_1 : \tau \quad e_1 \text{ value}}{\texttt{throw(}e_1\texttt{,}-\texttt{)} : \texttt{frame(cont(}\tau\texttt{),}\tau'\texttt{)}} \tag{25.9}$$

The rest of the definitions remain as in Chapter 23.

Lemma 25.1 (Canonical Forms)
If e : $\texttt{cont(}\tau\texttt{)}$ and e value, then $v = \texttt{cont(}k\texttt{)}$ for some k such that k : $\texttt{stack(}\tau\texttt{)}$.

Theorem 25.2 (Safety)
1. *If s ok and $s \longmapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \longmapsto s'$.*

25.3 Failures from Continuations

The dynamic semantics of failures may be specified by a translation into a language with continuations. The general idea is to translate expressions to functions that take as argument a continuation of type $\tau_{exn}\ \texttt{cont}$, representing the current exception handler. To fail, simply throw the unit value to the current handler. To establish a new handler we must create a new continuation to serve as the handler for the expression to be evaluated, while ensuring that the handling expression is itself evaluated in the context of the outer handler.

We will translate from the *source langage*, the extension of MinML with exceptions described in Chapter 24 to the *target language*, the extension of MinML with continuations described in the present chapter. The translation consists of two parts:

1. A type translation, $\tau \rightsquigarrow \tau'$, from source language types to target language types.

2. An expression translation, $e : \tau \rightsquigarrow e' : \texttt{unit cont}{\to}\tau'$, where $\tau \rightsquigarrow \tau'$, from source language expressions to target language expressions.

Notice that the type of the translation makes the failure continuation parameter explicit.

The type translation is defined by the following rules:

$$\frac{}{\texttt{nat} \rightsquigarrow \texttt{nat}} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \quad \tau_2 \rightsquigarrow \tau_2'}{\tau_1{\to}\tau_2 \rightsquigarrow \tau_1'{\to}(\texttt{unit cont}{\to}\tau_2')}$$

The translation augments functions with an additional argument representing the exception handler to use while executing the body of the function.

The term translation must be defined for open, as well as closed, terms. To do so we consider hypothetical translation judgements of the form

$$\Theta \vdash e : \tau \rightsquigarrow e' : \texttt{unit cont}{\to}\tau',$$

where $\tau_i \rightsquigarrow \tau_i'$, in which Θ consists of hypotheses of the form

$$x_i : \tau_i \rightsquigarrow \lambda(k{:}\texttt{unit cont}.\ x_i') : \texttt{unit cont}{\to}\tau_i'.$$

Each such hypothesis is to be understood as implicitly declaring fresh variables $x_i : \tau_i$ for the source language and $x_i' : \tau_i'$ for the target language. It specifies that the variable x_i, when encountered in e, is to be translated to the constant function $\lambda(k{:}\texttt{unit cont}.\, x_i')$ of type $\texttt{unit cont}{\to}\tau'$. This choice reflects the call-by-value interpretation of variables in the source language.

The expression translation is given by the following rules, wherein, for the sake of concision, we abbreviate `unit cont` by τ_h.

$$\frac{}{\Theta \vdash n : \texttt{nat} \leadsto \lambda(k{:}\tau_h.\, n) : \tau_h{\to}\texttt{nat}}$$

$$\frac{\tau_1 \leadsto \tau_1' \quad \Theta, x : \tau_1 \leadsto \lambda(k{:}\tau_h.\, x') : \tau_h{\to}\tau_1' \vdash e : \tau_2 \leadsto e' : \tau_h{\to}\tau_2'}{\Theta \vdash \lambda(x{:}\tau_1.\, e) : \tau_1{\to}\tau_2 \leadsto \lambda(k{:}\tau_h.\, \lambda(x'{:}\tau_1'.\, \lambda(k'{:}\tau_h.\, e'(k')))) : \tau_h{\to}(\tau_1'{\to}\tau_h{\to}\tau_2')}$$

$$\frac{\Theta \vdash e_1 : \tau_2{\to}\tau \leadsto e_1' : \tau_h{\to}(\tau_2'{\to}\tau_h{\to}\tau') \quad \Theta \vdash e_2 : \tau_2 \leadsto e_2' : \tau_h{\to}\tau_2'}{\Theta \vdash e_1(e_2) : \tau \leadsto \lambda(k{:}\tau_h.\, e_1'(k)(e_2'(k))) : \tau_h{\to}\tau'}$$

$$\frac{\tau \leadsto \tau'}{\Theta \vdash \texttt{fail} : \tau \leadsto \lambda(k{:}\tau_h.\, \texttt{throw}\,\langle\rangle\,\texttt{to}\,k) : \tau_h{\to}\tau'}$$

$$\frac{\Theta \vdash e_1 : \tau \leadsto e_1' : \tau_h{\to}\tau' \quad \Theta \vdash e_2 : \tau \leadsto e_2' : \tau_h{\to}\tau'}{\Theta \vdash \texttt{try}\,e_1\,\texttt{ow}\,e_2 : \tau \leadsto \lambda(k{:}\tau_h.\, \texttt{let}\,k'\,\texttt{be}\,e_{k,k'}'\,\texttt{in}\,e_1'(k')) : \tau_h{\to}\tau'}$$

In the translation for handlers the expression $e_{k,k'}'$ is as follows:

$$\texttt{letcc}\,r\,\texttt{in}\,\texttt{let}\,x\,\texttt{be}\,(\texttt{letcc}\,k'\,\texttt{in}\,\texttt{throw}\,k'\,\texttt{to}\,r)\,\texttt{in}\,e_2'(k).$$

This ensures that the handler, e_2, runs in the context of the *outer* exception handler, while building an *inner* exception handler for use by e_1. This handler simply passes the unit value to e_2, the handler for the exception.

25.4 Exercises

1. Study the short-circuit multiplication example carefully to be sure you understand why it works!

2. Attempt to solve the problem of composing a continuation with a function yourself, before reading the solution.

3. Simulate the evaluation of `compose` (f, k) on the empty stack. Observe that the control stack substituted for x is

$$\mathtt{app}(f,-);\mathtt{throw}(-,k);\varepsilon \tag{25.10}$$

This stack is returned from `compose`. Next, simulate the behavior of throwing a value v' to this continuation. Observe that the above stack is reinstated and that v' is passed to it.

4. Verify the type preservation theorem for the exception translation.

Chapter 26

Coroutines

A *subroutine* is a standard pattern of control flow in a program in which one routine passes control to another by passing to it a data value, the *argument*, to the subroutine, and a *return point* at which to resume control when finished. This arrangement is asymmetric in the sense that there is a clear separation between the roles of the caller and the callee. A *coroutine* is a symmetric pattern of control flow in which two routines are each subroutines of the other, each transferring control to the other by passing a data value and a return point for that call. Execution consists of an interleaving of the execution steps of the two routines determined by their mutual transfers of control to each other.

While it is relatively easy to visualize and implement coroutines involving only two partners, it is more complex, and less useful, to consider a similar pattern of control among $n \geq 2$ participants. In such cases it is more common to structure the interaction as a collection of n routines each coroutining with a central *scheduler*. A routine that transfers control to the scheduler is said to *yield* to another routine determined by the scheduler according to some (usually unspecified) other routine. This pattern of control flow is called *cooperative multi-threading*. Each routine is called a *thread of control*, or just *thread* for short, whose execution is managed by a central scheduler. Transfers of control only occur when one thread yields to another, and otherwise continues without interruption or preemption.

26.1 Coroutines from Continuations

Coroutines may be naturally and elegantly implemented using a combination of continuations and recursive types. Consider two symmetric routines that transfer control back and forth between each other by passing a data value and a continuation. The data value plays the role of the "argument" of the partner routine, and the continuation specifies the point at which execution is to be resumed by the partner. Thus, the *state* of the coroutine is described by a type satisfying the isomorphism

$$\texttt{state} \cong (\tau \times \texttt{state})\,\texttt{cont},$$

where τ is the type of the data values exchanged by the routines. The solution to such an isomorphism is, of course, the recursive type

$$\texttt{state} = \mu(t\,.\,(\tau \times t)\,\texttt{cont}).$$

Thus a state, s, encapsulates a pair consisting of a value of type τ together with another state. The value represents the "current" data value being exchanged by the routines, and the state represents the point of resumption.

Control is transferred from one routine to another by applying the function `resume` to a data value and a state. The function `resume` is defined as follows:

$$\lambda(\langle x,s\rangle\texttt{:nat} \times \texttt{state.}\,\texttt{letcc}\,k\,\texttt{in}\,\texttt{throw}\,\langle x,\texttt{roll(}k\texttt{)}\rangle\,\texttt{to}\,\texttt{unroll(}s\texttt{))}$$

The application $\texttt{resume(}\langle x,s\rangle\texttt{)}$ captures the continuation at the application site, and passes x along with this continuation to the continuation represented by s.

Both of the routines have the general form

$$\texttt{fun}\,\texttt{loop(}\langle x,s\rangle\texttt{:}\tau \times \texttt{state):}\tau \times \texttt{state}\,\texttt{is}\,\texttt{loop(resume(}\langle f(x)\texttt{,}s\rangle\texttt{)),}$$

where $f : \tau\to\tau$ transforms the "current" data value into the "next" data value. Different instances of the routine are obtained by specifying different functions f.

To obtain a classic producer/consumer pair, let `producer` and `consumer` be two routines of the above form. The initialization routine, called `run`, for the producer/consumer coroutines starts the producer with an initial data value, i, of type τ, and arranges that it resumes the consumer with the specified data value and resumption state.

$$\lambda(\langle\rangle\texttt{:unit.}\,\texttt{consumer(letcc}\,k\,\texttt{in}\,\texttt{producer(}\langle i\texttt{,roll(}k\texttt{)}\rangle\texttt{)))}.$$

Notice that execution starts with the producer, and passes control to the consumer when the producer resumes the state passed to it as argument. This establishes the interaction loop between the producer and consumer, which thereafter alternate execution in lockstep order.

26.2 Excercises

Part IX

Propositions and Types

Chapter 27

Curry-Howard Isomorphism

The *Curry-Howard Isomorphism* is a central organzing principle of type theory. Roughly speaking, the Curry-Howard Isomorphism states that there is a correspondence between *propositions* and *types* such that *proofs* correspond to *programs*. To each proposition, ϕ, there is an associated type, τ, such that to each proof p of ϕ, there is a corresponding expression e of type τ. Among other things, this correspondence tells us that *proofs have computational content* and that *programs are a form of proof*. It also suggests that programming language features may be expected to give rise to concepts of logic, and conversely that concepts from logic give rise to programming language features. It is a remarkable fact that this correspondence, which began as a rather modest observation about types and logics, has developed into a central principle of language design whose implications are still being explored.

This informal discussion leaves open what we mean by proposition and proof. The original isomorphism observed by Curry and Howard pertains to a particular branch of logic called *constructive logic*, of which we will have more to say in the next section. However, the observation has since been extended to an impressive array of logics, all of which are, by virtue of the correspondence, "constructive", but which extend the interpretation to richer notions of proposition and proof. Thus one might say that there are *many* Curry-Howard Isomorphisms, of which the original is but one!

We will focus our attention on constructive propositional logic, which involves a minimum of technical machinery to motivate and explain. We will concentrate on the "big picture", and make only glancing reference to the considerable technical details involved in fully working out the correspondence between propositions and types.

27.1 Constructive Logic

27.1.1 Constructive Semantics

Constructive logic is concerned with two judgement forms, ϕ prop, stating that ϕ expresses a proposition, and ϕ true, stating that ϕ is a true proposition. In constructive logic a proposition is a *specification* describing a *problem to be solved*. The *solution* to the problem posed by a proposition is a *proof*. If a proposition has a proof (*i.e.*, it specifies a soluble problem), then the proposition is said to be *true*. The characteristic feature of constructive logic is that *there is no other criterion of truth than the existence of a proof.*

In a contructive setting the notion of falsity of a proposition is not primitive. Rather, to say that a proposition is false is simply to say that the assumption that it is true (*i.e.*, that it has a proof) is contradictory. In other words, for a proposition to be false, constructively, means that there is a *refutation* of it, which consists of a *proof* that assuming it to be true is contradictory. In this sense constructive logic is a logic of *positive*, or *affirmative*, *information* — we must have explicit evidence in the form of a proof in order to affirm the truth or falsity of a proposition.

One consequence is that a given proposition need not be either true or false! While at first this might seem absurd (what else could it be, green?), a moment's reflection on the semantics of propositions reveals that this consequence is quite natural. There are, and always will be, unsolved problems that can be posed as propositions. For a problem to be unsolved means that we are not in possession of a proof of it, nor do we have a refutation of it. Therefore, in an affirmative sense, we cannot say that the proposition is either true or false! As an example, the famous $P \overset{?}{=} NP$ problem is a proposition that is, constructively, neither true nor false at the time of this writing.

A proposition, ϕ, for which we possess either a proof or a refutation of it is said to be *decidable*. Any proposition for which we have either a proof or a refutation is, of course, decidable, because we have already "decided" it by virtue of having that information! But we can also make general statements about decidability of propositions. For example, if ϕ expresses an inequality between natural numbers, then ϕ is decidable, because we can always work out, for given natural numbers m and n, whether $m \leq n$ or $m \not\leq n$ — we can either prove or refute the given inequality. Once we step outside the realm of such immediately checkable conditions, it is not clear whether a given proposition has a proof or a refutation. It's a matter of rolling up one's sleeves and doing some work! And there's no guarantee of success! Life's hard, but we muddle through somehow.

The judgements ϕ prop and ϕ true are basic, or *categorical*, judgements. These are the building blocks of reason, but they are rarely of interest by themselves. Rather, we are interested in the more general case of the *hypothetical judgement*, or *consequence relation*, of the form

$$\phi_1 \ \mathsf{true}, \ldots, \phi_n \ \mathsf{true} \vdash \phi \ \mathsf{true}.$$

This judgement expresses that the proposition ϕ is true (*i.e.*, has a proof), *under the assumptions* that each of $\phi_1, \ldots, \phi_n$ are also true (*i.e.*, have proofs). Of course, when $n = 0$ this is just the same as the categorical judgement ϕ true. We let Γ range over finite sets of assumptions.

The hypothetical judgement satisfies the following *structural properties*, which characterize what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \ \mathsf{true} \vdash \phi \ \mathsf{true}} \tag{27.1}$$

$$\frac{\Gamma \vdash \phi \ \mathsf{true} \quad \Gamma, \phi \ \mathsf{true} \vdash \psi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{27.2}$$

$$\frac{\Gamma \vdash \psi \ \mathsf{true}}{\Gamma, \phi \ \mathsf{true} \vdash \psi \ \mathsf{true}} \tag{27.3}$$

$$\frac{\Gamma, \phi \ \mathsf{true}, \phi \ \mathsf{true} \vdash \theta \ \mathsf{true}}{\Gamma, \phi \ \mathsf{true} \vdash \theta \ \mathsf{true}} \tag{27.4}$$

$$\frac{\Gamma, \psi \ \mathsf{true}, \phi \ \mathsf{true}, \Gamma' \vdash \theta \ \mathsf{true}}{\Gamma, \phi \ \mathsf{true}, \psi \ \mathsf{true}, \Gamma' \vdash \theta \ \mathsf{true}} \tag{27.5}$$

The last two rules are implicit in that we regard Γ as a *set* of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

27.1.2 Propositional Logic

The connectives of propositional logic (truth, falsehood, conjunction, disjunction, implication, and negation) are given meaning by rules that determine (a) what constitutes a "direct" proof of a proposition formed from a given connective, and (b) how to exploit the existence of such a proof in an "indirect" proof of another proposition. These are called the *introduction* and *elimination* rules for the connective. The principle of *conservation of proof* states that these rules are inverse to one another — the elimination rule cannot extract more information (in the form of a proof) than was put into it by the introduction rule, and the introduction rules can be used to reconstruct a proof from the information extracted from it by the elimination rules.

The abstract syntax of propositional logic is given by the following rules for deriving judgements of the form ϕ prop.

$$\frac{}{\texttt{true}\ \mathsf{prop}} \tag{27.6}$$

$$\frac{}{\texttt{false}\ \mathsf{prop}} \tag{27.7}$$

$$\frac{\phi\ \mathsf{prop} \quad \psi\ \mathsf{prop}}{\texttt{and}(\phi,\psi)\ \mathsf{prop}} \tag{27.8}$$

$$\frac{\phi\ \mathsf{prop} \quad \psi\ \mathsf{prop}}{\texttt{imp}(\phi,\psi)\ \mathsf{prop}} \tag{27.9}$$

$$\frac{\phi\ \mathsf{prop} \quad \psi\ \mathsf{prop}}{\texttt{or}(\phi,\psi)\ \mathsf{prop}} \tag{27.10}$$

The following table summarizes the concrete syntax of propositions:

Abstract	*Concrete*
`true`	$\top$
`false`	$\bot$
$\texttt{and}(\phi,\psi)$	$\phi \wedge \psi$
$\texttt{imp}(\phi,\psi)$	$\phi \supset \psi$
$\texttt{or}(\phi,\psi)$	$\phi \vee \psi$

Truth Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it.

$$\frac{}{\Gamma \vdash \top \ \mathsf{true}} \tag{27.11}$$

$$\textit{(no elimination rule)} \tag{27.12}$$

Conjunction Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi \ \mathsf{true} \quad \Gamma \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}} \tag{27.13}$$

$$\frac{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{27.14}$$

$$\frac{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{27.15}$$

Implication Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi \ \mathsf{true} \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \supset \psi \ \mathsf{true}} \tag{27.16}$$

$$\frac{\Gamma \vdash \phi \supset \psi \ \mathsf{true} \quad \Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{27.17}$$

Falsehood Falsehood expresses the trivially false (refutable) proposition.

$$\textit{(no introduction rule)} \tag{27.18}$$

$$\frac{\Gamma \vdash \bot \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{27.19}$$

Disjunction Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{27.20}$$

$$\frac{\Gamma \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{27.21}$$

$$\frac{\Gamma \vdash \phi \vee \psi \ \mathsf{true} \quad \Gamma, \phi \ \mathsf{true} \vdash \theta \ \mathsf{true} \quad \Gamma, \psi \ \mathsf{true} \vdash \theta \ \mathsf{true}}{\Gamma \vdash \theta \ \mathsf{true}} \tag{27.22}$$

27.1.3 Explicit Proofs

The key to the Curry-Howard Isomorphism is to make explict the forms of proof. The categorical judgement ϕ true, which states that ϕ has a proof, is replaced by the judgement $p : \phi$, stating that p is a proof of ϕ. The hypothetical judgement is modified correspondingly, with variables standing for the presumed, but unknown, proofs:

$$x_1 : \phi_1, \ldots, x_n : \phi_n \vdash p : \phi.$$

We again let Γ range over such hypothesis lists, subject to the restriction that no variable occurs more than once.

The rules of constructive propositional logic may be restated using proof terms as follows.

$$\frac{}{\Gamma \vdash \texttt{true-i} : \top} \tag{27.23}$$

$$\frac{\Gamma \vdash p : \phi \quad \Gamma \vdash q : \psi}{\Gamma \vdash \texttt{and-i}(p,q) : \phi \wedge \psi} \tag{27.24}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{and-e-l}(p) : \phi} \tag{27.25}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{and-e-r}(p) : \psi} \tag{27.26}$$

$$\frac{\Gamma, x:\phi \vdash p:\psi}{\Gamma \vdash \texttt{imp-i}(\phi, x.p):\phi \supset \psi} \tag{27.27}$$

$$\frac{\Gamma \vdash p:\phi \supset \psi \quad \Gamma \vdash q:\phi}{\Gamma \vdash \texttt{imp-e}(p,q):\psi} \tag{27.28}$$

$$\frac{\Gamma \vdash p:\bot}{\Gamma \vdash \texttt{false-e}(\phi, p):\phi} \tag{27.29}$$

$$\frac{\Gamma \vdash p:\phi}{\Gamma \vdash \texttt{or-i-l}(\psi, p):\phi \vee \psi} \tag{27.30}$$

$$\frac{\Gamma \vdash p:\psi}{\Gamma \vdash \texttt{or-i-r}(\phi, p):\phi \vee \psi} \tag{27.31}$$

$$\frac{\Gamma \vdash p:\phi \vee \psi \quad \Gamma, x:\phi \vdash q:\theta \quad \Gamma, y:\psi \vdash r:\theta}{\Gamma \vdash \texttt{or-e}(p, \phi, x.q, \psi, y.r):\theta} \tag{27.32}$$

27.2 Propositions as Types

The Curry-Howard Isomorphism amounts to the observation that there is a close correspondence between propositions and their proofs, on the one hand, and types and their elements, on the other. The following chart summarizes the correspondence between propositions, ϕ, and types, ϕ^*:

Proposition	*Type*
$\top$	`unit`
$\bot$	`void`
$\phi \wedge \psi$	$\phi^* \times \psi^*$
$\phi \supset \psi$	$\phi^* \rightarrow \psi^*$
$\phi \vee \psi$	$\phi^* + \psi^*$

The correspondence extends to proofs and programs as well:

Proof	*Program*
`true-i`	$\langle\rangle$
`false-e`(ϕ, p)	`abort`(p^*)
`and-i`(p, q)	$\langle p^*, q^*\rangle$
`and-e-l`(p)	`fst`(p^*)
`and-e-r`(p)	`snd`(p^*)
`imp-i`$(\phi, x\,.\,p)$	$\lambda(x:\phi^*.\,p^*)$
`imp-e`(p, q)	$p^*(q^*)$
`or-i-l`(ψ, p)	$\mathtt{inl}_{\psi^*}(p^*)$
`or-i-r`(ϕ, p)	$\mathtt{inr}_{\phi^*}(p^*)$
`or-e`$(p, \phi, x\,.\,q, \psi, y\,.\,r)$	...

`case` p^* { `inl`$(x{:}\phi^*) \Rightarrow q^*$ | `inr`$(y{:}\psi^*) \Rightarrow r^*$ }

The translations above preserve and reflect formation and membership when viewed as a translation into a typed language with unit, product, void, sum, and function types.

Theorem 27.1 (Curry-Howard Isomorphism)

1. *If* ϕ prop*, then* ϕ^* type

2. *If* $\Gamma \vdash p : \phi$*, then* $\Gamma^* \vdash p^* : \phi^*$.

The preceding theorem establishes a *static* correspondence between propositions and types and their associated proofs and programs. It also extends to a *dynamic* correspondence, in which we see that the execution behavior of programs arises from the cancellation of elimination and introduction rules in the following manner:

$$\begin{array}{rcl}
\mathtt{and\text{-}e\text{-}l}(\mathtt{and\text{-}i}(p,q)) & \longmapsto & p \\
\mathtt{and\text{-}e\text{-}r}(\mathtt{and\text{-}i}(p,q)) & \longmapsto & q \\
\mathtt{imp\text{-}e}(\mathtt{imp\text{-}i}(\phi, x\,.\,q), p) & \longmapsto & [x\leftarrow p]q \\
\mathtt{or\text{-}e}(\mathtt{or\text{-}i\text{-}l}(\psi, p), \phi, x\,.\,q, \psi, y\,.\,r) & \longmapsto & [x\leftarrow p]q \\
\mathtt{or\text{-}e}(\mathtt{or\text{-}i\text{-}r}(\phi, p), \phi, x\,.\,q, \psi, y\,.\,r) & \longmapsto & [y\leftarrow p]r
\end{array}$$

These are precisely the primitive instructions associated with the programs corresponding to these proofs! Indeed, these rules may be understood as the codification of the *computational content* of proofs — the precise sense in which proofs in propositional logic correspond, both statically and dynamically, to programs.

It is worth stressing that the correspondence does *not* include general recursive functions, which would correspond to "recursive proofs!" Indeed, if we permit general recursive proofs, then the result is logically inconsistent, since we can then find a proof of *every* proposition — namely, the non-terminating recursive "proof"!

27.3 Exercises

Chapter 28

Classical Logic

In Chapter 27 we saw that constructive logic is a logic of positive information in that the meaning of the judgement ϕ true is that there exists a proof of ϕ. A refutation of a proposition ϕ consists of a proof of the hypothetical judgement ϕ true $\vdash \perp$ true, asserting that the assumption of ϕ leads to a proof of logical falsehood (*i.e.*, a contradiction). Since there are propositions, ϕ, for which we possess neither a proof nor a refutation, we cannot assert, in general, $\phi \vee \neg\phi$ true.

By contrast classical logic (the one we all learned in school) maintains a complete symmetry between truth and falsehood — that which is not true is false and that which is not false is true. Obviously such an interpretation conflicts with the constructive interpretation, for lack of a proof of a proposition is not a refutation, nor is lack of a refutation a proof.[1] In this sense classical logic is a logic of perfect information, in which all mathematical problems have been resolved, and for each one it is clear whether it is true or false. One might consider this "god's view" of mathematics, in constrast to the "mortal's view" we are stuck with.

Despite this absolutism, classical logic nevertheless has computational content, *albeit* in a somewhat attenuated form compared to constructive logic. Whereas in constructive logic truth is identified with the existence of certain positive information, in classical logic it is identified with the absence of a refutation, a much weaker criterion. Dually, falsehood is identified with the absence of a proof, which is also much weaker than pos-

[1]Or, in the words of the brilliant military strategist Donald von Rumsfeld, the absence of evidence is not evidence of absence.

session of a refutation. This weaker interpretation is responsible for the pleasing symmetries of classical logic. The drawback is that in classical logic propositions means much less than they do in constructive logic. For example, in classical logic the proposition $\phi \vee \neg\phi$ does not state that we have either a proof of ϕ or a refutation of it, rather just that it is impossible that we have both a proof of it and a refutation of it.

28.1 Classical Propositional Logic

Classical logic is concerned with three categorical judgement forms:

1. ϕ true, stating that proposition ϕ is true;
2. ϕ false, stating that proposition ϕ is false;
3. #, stating that a contradiction has been derived.

Hypothetical judgements have the form

$$\phi_1\ \mathsf{false}, \ldots, \phi_m\ \mathsf{false}; \psi_1\ \mathsf{true}, \ldots, \psi_n\ \mathsf{true} \vdash J,$$

where J is any of the three categorical judgement forms.

Rather than give the rules in this form, it is expedient to consider directly the corresponding judgements with explicit proof terms.

1. $p : \phi$, stating that p is a proof of ϕ;
2. $k \div \phi$, stating that k is a refutation of ϕ;
3. $k \,\#\, p$, stating that k and p are contradictory.

Hypothetical judgements have the form

$$\underbrace{u_1 \div \phi_1, \ldots, u_m \div \phi_m}_{\Delta}; \underbrace{x_1 : \psi_1, \ldots, x_n : \psi_n}_{\Gamma} \vdash J,$$

where J is any of the preceding three categorical judgements. The structural rules discussed in Chapter 27 apply to both the truth and the falsehood contexts of the hypothetical judgement.

Statics

A contradiction arises from the conflict between a proof and a refutation:

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash k \mathbin{\#} p}$$

The reflexivity rules capture the meaning of hypotheses:

$$\frac{}{\Delta, u \div \phi; \Gamma \vdash u \div \phi} \qquad \frac{}{\Delta;\Gamma, x : \psi \vdash x : \phi}$$

Truth and falsity are complementary:

$$\frac{\Delta, u \div \phi; \Gamma \vdash k \mathbin{\#} p}{\Delta;\Gamma \vdash \mathtt{ccr}(u \div \phi\,.\,k \mathbin{\#} p) : \phi} \qquad \frac{\Delta;\Gamma, x : \phi \vdash k \mathbin{\#} p}{\Delta;\Gamma \vdash \mathtt{ccp}(x : \phi\,.\,k \mathbin{\#} p) \div \phi}$$

In both of these rules the entire contradiction, $k \mathbin{\#} p$, lies within the scope of the abstractor!

The rules for the connectives are organized as introductory rules for truth and for falsity, the latter playing the role of eliminatory rules in constructive logic.

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \top} \qquad \frac{}{\Delta;\Gamma \vdash \mathtt{abort} \div \bot}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash q : \psi}{\Delta;\Gamma \vdash \langle p, q\rangle : \phi \wedge \psi}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \wedge \psi}{\Delta;\Gamma \vdash \mathtt{fst}; k \div \phi} \qquad \frac{\Delta;\Gamma \vdash k \div \phi \wedge \psi}{\Delta;\Gamma \vdash \mathtt{snd}; k \div \psi}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash p : \psi}{\Delta;\Gamma \vdash \lambda(x{:}\phi.\, p) : \phi \supset \psi} \qquad \frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \mathtt{ap}(p); k \div \phi \supset \psi}$$

$$\frac{\Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash \mathtt{inl}_{\psi}(p) : \phi \vee \psi} \qquad \frac{\Delta;\Gamma \vdash p : \psi}{\Delta;\Gamma \vdash \mathtt{inr}_{\phi}(p) : \phi \vee \psi}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash l \div \psi}{\Delta;\Gamma \vdash \mathtt{case}(k, l) \div \phi \vee \psi}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi}{\Delta;\Gamma \vdash \mathtt{not}(k) : \neg\phi} \qquad \frac{\Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash \mathtt{not}(p) \div \neg\phi}$$

28.1.1 Dynamics

The dynamic semantics of classical logic may be described as a process of *conflict resolution*. The state of the abstract machine is a contradiction, $k \# p$, between a refutation, k, and a proof, p, of the same proposition. Execution consists of "simplifying" the conflict based on the form of k and p. This process is formalized by an inductive definition of a transition relation between contradictory states.

Here are the rules for each of the logical connectives, which all have the form of resolving a conflict between a proof and a refutation of a proposition formed with that connective.

$$\begin{array}{rcl}
\texttt{fst};k \# \langle p,q\rangle & \longmapsto & k \# p \\
\texttt{snd};k \# \langle p,q\rangle & \longmapsto & k \# q \\
\texttt{case}(k,l) \# \texttt{inl}_{\psi}(p) & \longmapsto & k \# p \\
\texttt{case}(k,l) \# \texttt{inr}_{\phi}(q) & \longmapsto & l \# q \\
\texttt{ap}(p);k \# \lambda(x{:}\phi.\,q) & \longmapsto & k \# [x{\leftarrow}p]q \\
\texttt{not}(p) \# \texttt{not}(k) & \longmapsto & k \# p
\end{array}$$

The symmetry of the transition rule for negation is particularly elegant.

Finally, here are the rules for the generic primitives relating truth and falsity.

$$\begin{array}{rcl}
\texttt{ccp}(x : \phi\,.\,k \# p) \# q & \longmapsto & [x{\leftarrow}q]k \# [x{\leftarrow}q]p \\
k \# \texttt{ccr}(u \div \phi\,.\,l \# p) & \longmapsto & [u{\leftarrow}k]l \# [u{\leftarrow}k]p
\end{array}$$

These rules explain the terminology: "ccp" means "call with current proof", and "ccr" means "call with current refutation". The former is a refutation that binds a variable to the current proof and installs the corresponding instance of its constituent state as the current state. The latter is a proof that binds a variable to the current refutation and installs the corresponding instance of its constituent state as the current state.

It is important to observe that the last two rules overlap in the sense that there are two possible transitions for a state of the form

$$\texttt{ccp}(x : \phi\,.\,k \# p) \# \texttt{ccr}(u \div \phi\,.\,l \# q).$$

This state may transition either to the state

$$[x{\leftarrow}r]k \# [x{\leftarrow}r]p,$$

where r is $\texttt{ccr}(u \div \phi\,.\,l\ \#\ q)$, or to the state

$$[u{\leftarrow}m]l\ \#\ [u{\leftarrow}m]q,$$

where m is $\texttt{ccp}(x : \phi\,.\,k\ \#\ p)$, and these are not equivalent.

There are two possible attitudes about this. One is to simply accept that classical logic has a non-deterministic dynamic semantics, and leave it at that. But this means that it is difficult to predict the outcome of a computation, since it could be radically different in the case of the overlapping state just described. The alternative is to impose an arbitrary priority ordering among the two cases, either preferring the first transition to the second, or *vice versa*. Preferring the first corresponds, very roughly, to a "lazy" semantics for proofs, because we pass the unevaluated proof, r, to the refutation on the left, which is thereby activated. Preferring the second corresponds to an "eager" semantics for proofs, in which we pass the unevaluated refutation, m, to the proof, which is thereby activated. Dually, these choices correspond to an "eager" semantics for refutations in the first case, and a "lazy" one for the second. Take your pick.

The final issue is the initial state: how is computation to be started? Or, equivalently, when is it finished? The difficulty is that we need both a proof and a refutation of the same proposition! While this can easily come up in the "middle" of a proof, it would be impossible to have a finished proof and a finished refutation of the same proposition! The solution for an eager interpretation of proofs (and, correspondingly, a lazy interpretation of refutations) is simply to postulate an initial (or final, depending on your point of view) refutation, `halt`, and to deem a state of the form $\texttt{halt}\ \#\ p$ to be initial, and also final, provided that p is not a "ccr" instruction. The solution for a lazy interpretation of proofs (and an eager interpretation of refutations) is dual, taking $k\ \#\ \texttt{halt}$ as initial, and also final, provided that k is not a "ccp" instruction.

Part X

State

Chapter 29

Storage Effects

MinML is said to be a *pure* language because the execution model consists entirely of evaluating an expression for its value. ML is an *impure* language because its execution model also includes *effects*, specifically, *control effects* and *store effects*. Control effects are non-local transfers of control; these were studied in Chapters 25 and 24. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

29.1 References

The MinML type language is extended with *reference types* $\texttt{ref}(\tau)$ whose elements are to be thought of as mutable storage cells. We correspondingly extend the expression language with these primitive operations:

$$e \ ::= \ l \mid \texttt{new}(e) \mid \texttt{get}(e) \mid \texttt{set}(e_1,e_2)$$

As in Standard ML, $\texttt{new}(e)$ allocates a "new" reference cell, $\texttt{get}(e)$ retrieves the contents of the cell e, and $\texttt{set}(e_1,e_2)$ sets the contents of the cell e_1 to the value e_2. The variable l ranges over a set of *locations*, an infinite set of names disjoint from variables. These are needed for the dynamic semantics, but are not expected to be notated directly by the programmer. The set of *values* is extended to include locations.

The typing judgement, $e : \tau$, for the extension of MinML with references must be considered in the context of two forms of assumptions:

1. *Variable assumptions* of the form $x_i : \tau_i$, introducing a variable x_i with type τ_i.

2. *Location assumptions* of the form $l_i : \tau_i$, introducing a location l_i whose contents is of type τ_i.

The hypothetical typing judgement has the form

$$\Lambda; \Gamma \vdash e : \tau,$$

where Γ stands for a finite set of variable assumptions, and Λ stands for a finite set of location assumptions, such that no variable and no location is the subject of more than one assumption.

The typing rules are those of MinML (extended to carry a location typing), plus the following rules governing the new constructs of the language:

$$\frac{(\Lambda(l) = \tau)}{\Lambda; \Gamma \vdash l : \texttt{ref}(\tau)} \tag{29.1}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \texttt{new}(e) : \texttt{ref}(\tau)} \tag{29.2}$$

$$\frac{\Lambda; \Gamma \vdash e : \texttt{ref}(\tau)}{\Lambda; \Gamma \vdash \texttt{get}(e) : \tau} \tag{29.3}$$

$$\frac{\Lambda; \Gamma \vdash e_1 : \texttt{ref}(\tau_2) \qquad \Lambda; \Gamma \vdash e_2 : \tau_2}{\Lambda; \Gamma \vdash \texttt{set}(e_1, e_2) : \tau_2} \tag{29.4}$$

Notice that the location typing is not extended during type checking! Locations arise only during execution, and are not part of complete programs, which must not have any free locations in them. The role of the location typing will become apparent in the proof of type safety for MinML extended with references.

A *memory* is a finite function mapping locations to closed values (but possibly involving locations). The dynamic semantics of MinML with references is given by an abstract machine. The states of this machine have the form (M, e), where M is a memory and e is an expression possibly involving free locations in the domain of M. The locations in $\text{dom}(M)$ are bound simultaneously in (M, e); the names of locations may be changed at will without changing the identity of the state.

The transitions for this machine are similar to those of the M machine, but with these additional steps:

$$\frac{(M, e) \longmapsto (M', e')}{(M, \texttt{new(}e\texttt{)}) \longmapsto (M', \texttt{new(}e'\texttt{)})} \tag{29.5}$$

$$\frac{(l \notin \text{dom}(M))}{(M, \texttt{new(}v\texttt{)}) \longmapsto (M[l{=}v], l)} \tag{29.6}$$

$$\frac{(M, e) \longmapsto (M', e')}{(M, \texttt{get(}e\texttt{)}) \longmapsto (M', \texttt{get(}e'\texttt{)})} \tag{29.7}$$

$$\frac{(l \in \text{dom}(M))}{(M, \texttt{get(}l\texttt{)}) \longmapsto (M, M(l))} \tag{29.8}$$

$$\frac{(M, e_1) \longmapsto (M', e_1')}{(M, \texttt{set(}e_1, e_2\texttt{)}) \longmapsto (M', \texttt{set(}e_1', e_2\texttt{)})} \tag{29.9}$$

$$\frac{(M, e_2) \longmapsto (M', e_2')}{(M, \texttt{set(}v_1, e_2\texttt{)}) \longmapsto (M', \texttt{set(}v_1, e_2'\texttt{)})} \tag{29.10}$$

$$\frac{(l \in \text{dom}(M))}{(M, \texttt{set(}l, v\texttt{)}) \longmapsto (M[l{=}v], v)} \tag{29.11}$$

A state (M, e) is *final* iff e is a value (possibly a location).

To prove type safety for this extension we will make use of some auxiliary relations. Most importantly, the typing relation between memories and location typings, written $M : \Lambda$, is inductively defined by the following rule:

$$\frac{\text{dom}(M) = \text{dom}(\Lambda) \quad \forall l \in \text{dom}(\Lambda)\ \Lambda; \bullet \vdash M(l) : \Lambda(l)}{M : \Lambda} \tag{29.12}$$

It is very important to study this rule carefully! First, we require that Λ and M govern the same set of locations. Second, for each location l in their common domain, we require that the value at location l, namely $M(l)$, have the type assigned to l, namely $\Lambda(l)$, relative to the *entire* location typing Λ. This means, in particular, that memories may be "circular" in the sense that the value at location l may contain an occurrence of l, for example if that value is a function.

The typing rule for memories is reminiscent of the typing rule for recursive functions — we are allowed to assume the typing that we are trying to prove while trying to prove it. This similarity is no accident, as the following example shows.

```
let diverge:nat->nat be fun d(x:nat):nat = d x in
let fcell:(nat->nat) ref be new(diverge) in
let fpre:nat->nat be
    λ (n:nat.ifz(n; 1; n * (get(fcell))(n-1))) in
let _:nat = set(fcell,f) in
let n = f 5
```

This technique is called *backpatching*. It is used in some compilers to implement recursive functions (and other forms of looping construct).

Exercise 29.1

1. *Sketch the contents of the memory after each step in the above example. Observe that after the assignment to* `fc` *the memory is "circular" in the sense that some location contains a reference to itself.*

2. *Prove that every cycle in well-formed memory must "pass through" a function. Suppose that $M(l_1) = l_2$, $M(l_2) = l_3, \ldots, M(l_n) = l_1$ for some sequence $l_1, \ldots, l_n$ of locations. Show that there is no location typing Λ such that $M : \Lambda$.*

The well-formedness of a machine state is inductively defined by the following rule:

$$\frac{M : \Lambda \qquad \Lambda; \bullet \vdash e : \tau}{(M, e)\ \mathsf{ok}} \tag{29.13}$$

That is, (M, e) is well-formed iff there is a location typing for M relative to which e is well-typed.

Theorem 29.2 (Preservation)
If (M, e) ok and $(M, e) \longmapsto (M', e')$, then (M', e') ok.

Proof: The trick is to prove a stronger result by induction on evaluation: if $(M, e) \longmapsto (M', e')$, $\vdash M : \Lambda$, and $\Lambda; \bullet \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $M' : \Lambda'$ and $\Lambda'; \bullet \vdash e' : \tau$. ■

Exercise 29.3
Prove Theorem 29.2. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation — the type of a location never changes once it is established at the point of allocation. This is crucial for the induction.

Theorem 29.4 (Progress)
If (M, e) ok then either (M, e) is a final state or there exists (M', e') such that $(M, e) \longmapsto (M', e')$.

Proof: The proof is by induction on typing: if $M : \Lambda$ and $\Lambda; \bullet \vdash e : \tau$, then either e is a value or there exists $M' \supseteq M$ and e' such that $(M, e) \longmapsto (M', e')$. ■

29.2 Exercises

Exercise 29.5
Prove Theorem 29.4 by induction on typing of machine states.

Chapter 30

Monadic Storage Effects

As we saw in Chapter 29 one way to combine functional and imperative programming is to add a type of reference cells to MinML. This approach works well for call-by-value languages, because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

Enriching ML with a type of references has an additional consequence that one can no longer determine from the type alone whether an expression mutates storage. For example, a function of type `arrow(int,int)` must taken an integer as argument and yield an integer as result, but may or may not allocate new reference cells or mutate existing reference cells. The expressive power of the type system is thereby weakened, because we cannot distinguish *pure* (effect-free) expressions from *impure* (effect-ful) expressions.

Another approach to introducing effects in a purely functional language is to make the possibility of effects explicit in the type system. This is achieved by introducing a *modality*, called a *monad*, that segregates the effect-free fragment of the language from the effect-ful fragment. These two sub-languages are related by two principles: (a) every effect-free expression may be regarded as (vacuously) effectful, and (b) an effectful expression may be suspended and packaged as an effect-free expression, and, correspondingly, unpackaging and activating such an expression is an effect-ful operation. A packaged effectful expression is a value of *monadic type*, τ `comp`, which signals that it classifies an impure computation yielding a value of type τ.

In this setting the type `nat`$\rightarrow$`nat` consists only of pure, possibly non-terminating, functions on the natural numbers. Applying such a function can have no effect on the store. However, the type `nat`$\rightarrow$`nat comp` consists of functions that, when applied to a natural number, yield an effectful computation that, when activated, yields a natural number and may also modify the store. Thus, the type distinguishes the possibility of there being an effect.

30.1 A Monadic Language

The syntax of a monadic re-formulation of the extension of the language MinML with references is given by the following grammar:

$$\begin{array}{llcl}
\textit{Types} & \tau & ::= & \mathtt{nat} \mid \mathtt{arrow}(\tau_1,\tau_2) \mid \mathtt{ref}(\tau) \mid \mathtt{comp}(\tau) \\
\textit{Pure} & e & ::= & x \mid l \mid \mathtt{num}[n] \mid \mathtt{plus}(e_1,e_2) \mid \ldots \mid \mathtt{ifz}(e_0,e_1,e_2) \mid \\
 & & & \mathtt{fun}(\tau_1,\tau_2,f\,.\,x\,.\,e) \mid \mathtt{app}(e_1,e_2) \mid \mathtt{comp}(m) \\
\textit{Impure} & m & ::= & \mathtt{return}(e) \mid \mathtt{letcomp}(e,x\,.\,m) \mid \\
 & & & \mathtt{new}(e) \mid \mathtt{get}(e) \mid \mathtt{set}(e_1,e_2)
\end{array}$$

The type system is extended to include a new type, `comp`(τ), of suspended computations of type τ. The introductory form, which is pure, for this type is `comp`(m), and the corresponding eliminatory form, which is impure, is `letcomp`($e,x\,.\,m$). The inclusion of pure into impure expressions is written `return`(e). The other constructs are familiar from MinML and its extension with references described in Chapter 29. Note that the operations for allocating, accessing, and modifying reference cells are all regarded as impure.

The concrete syntax corresponding to the new forms of abstract syntax is given by the following chart:

Abstract	*Concrete*
$\mathtt{comp}(\tau)$	$\tau\ \mathtt{comp}$
$\mathtt{return}(e)$	$\mathtt{return}\, e$
$\mathtt{letcomp}(e,x\,.\,m)$	$\mathtt{let\ comp}(x)\ \mathtt{be}\ e\ \mathtt{in}\ m$
$\mathtt{comp}(m)$	$\mathtt{comp}(m)$

The static semantics of this language consists of two forms of typing judgement, $e : \tau$, stating that pure expression e has type τ, and $m \sim \tau$, stating that the impure expression m has type τ. Both of these judgement forms are considered with respect to hypotheses of the form $x_i : \tau_i$, which introduces a variable x_i with type τ_i, and of the form $l_i : \tau_i$, which introduces a location l_i whose contents is to be of type τ_i. We will not have need of hypotheses of the form $u_i \sim \tau_i$, because variables are only ever bound to values, which are always pure (since they are fully evaluated). As in Chapter 29, we will write Γ for a finite set of variable assumptions, and Λ for a finite set of location assumptions. We will segregate these assumptions from one another when writing hypothetical judgements.

The typing rules for this extension are an extension of those for MinML, with the following additional rules.

$$\frac{\Lambda;\Gamma \vdash m \sim \tau}{\Lambda;\Gamma \vdash \texttt{comp}(m) : \texttt{comp}(\tau)}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \texttt{return}(e) \sim \tau}$$

$$\frac{\Lambda;\Gamma \vdash e : \texttt{comp}(\tau) \quad \Lambda;\Gamma, x : \tau \vdash m \sim \tau'}{\Lambda;\Gamma \vdash \texttt{letcomp}(e, x.m) \sim \tau'}$$

$$\frac{}{\Lambda, l : \tau;\Gamma \vdash l : \texttt{ref}(\tau)}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \texttt{new}(e) \sim \texttt{ref}(\tau)}$$

$$\frac{\Lambda;\Gamma \vdash e : \texttt{ref}(\tau)}{\Lambda;\Gamma \vdash \texttt{get}(e) \sim \tau}$$

$$\frac{\Lambda;\Gamma \vdash e_1 : \texttt{ref}(\tau) \quad \Lambda;\Gamma \vdash e_2 : \tau}{\Lambda;\Gamma \vdash \texttt{set}(e_1, e_2) \sim \texttt{unit}}$$

The dynamic semantics of the monadic formulation of MinML with references is structured into two parts:

1. A transition relation $e \longmapsto e'$ for pure expressions.

2. A transition relation $(M, m) \longmapsto (M', m')$ for impure expressions.

The former relation is defined just as it is for MinML, amended to account for the new pure expression forms. The latter is defined similarly to Chapter 29, again amended to account for the extensions with monadic primitives.

There are two additional forms of value at the pure expression level:

$$\frac{}{\texttt{comp}(m)\ \textsf{value}} \qquad \frac{}{l\ \textsf{value}}$$

That is, both suspended computations and locations are values.

The rules governing the monadic primitives are as follows.

$$\frac{e \longmapsto e'}{(M, \texttt{return}(e)) \longmapsto (M, \texttt{return}(e'))}$$

$$\frac{e \longmapsto e'}{(M, \texttt{letcomp}(e, x\,.\,m)) \longmapsto (M, \texttt{letcomp}(e', x\,.\,m))}$$

$$\frac{(M, m_1) \longmapsto (M', m_1')}{(M, \texttt{letcomp}(\texttt{comp}(m_1), x\,.\,m_2)) \longmapsto (M', \texttt{letcomp}(\texttt{comp}(m_1'), x\,.\,m_2))}$$

$$\frac{e\ \textsf{value}}{(M, \texttt{letcomp}(\texttt{comp}(\texttt{return}(e)), x\,.\,m)) \longmapsto (M, [x{\leftarrow}e]m)}$$

The evaluation rules for the reference primitives are as follows:

$$\frac{e \longmapsto e'}{(M, \texttt{new}(e)) \longmapsto (M, \texttt{new}(e'))} \qquad \frac{e\ \textsf{value} \quad l \,\#\, M}{(M, \texttt{new}(e)) \longmapsto (M[l \mapsto e], \texttt{return}(l))}$$

$$\frac{e \longmapsto e'}{(M, \texttt{get}(e)) \longmapsto (M, \texttt{get}(e'))} \qquad \frac{e\ \textsf{value} \quad l \,\#\, M}{(M[l \mapsto e], \texttt{get}(l)) \longmapsto (M[l \mapsto e], \texttt{return}(e))}$$

$$\frac{e_1 \longmapsto e_1'}{(M, \texttt{set}(e_1, e_2)) \longmapsto (M, \texttt{set}(e_1', e_2))} \qquad \frac{e_1\ \textsf{value} \quad e_2 \longmapsto e_2'}{(M, \texttt{set}(e_1, e_2)) \longmapsto (M, \texttt{set}(e_1, e_2'))}$$

$$\frac{e\ \textsf{value} \quad l \,\#\, M}{(M[l \mapsto e'], \texttt{set}(l, e)) \longmapsto (M[l \mapsto e], \texttt{return}(e))}$$

The transition rules for the monadic elimination form is somewhat unusual. First, the expression e is evaluated to obtain an encapsulated impure computation. Once such a computation has been obtained, execution continues by evaluating it in the current memory, updating that memory as appropriate during its execution. This process ends once the encapsulated computation is a `return` statement, in which case this value is passed to the body of the `letcomp`.

30.2 Exercises

1. Consider other forms of effect such as I/O.
2. State and prove type safety for the monadic formulation of storage effects.

Chapter 31

Extensible Sums

Part XI

Lazy Evaluation

Chapter 32

Laziness

Lazy evaluation is a general term applied to a variety of concepts that have in common the idea of avoiding "unnecessary" computations during evaluation. The basic example of laziness is the *call-by-name* evaluation strategy for function applications, which passes the argument to a function in unevaluated form so that if it is never needed to determine the result, it is never evaluated. For example, if e is a very lengthy computation of a natural number, then under the call-by-name strategy the application $\lambda(x\mathtt{:nat.}3)(e)$ takes one step to complete, and the entire computation of e is avoided. On the other hand, call-by-name does not always result in less work. For example, consider the application $\lambda(x\mathtt{:nat.}\,x\mathtt{+}x)(e)$. Under call-by-name the expression e is evaluated twice, even though the outcome must be the same in both cases.

Eager evaluation is a term that generally connotes the opposite point of view, in which computations are performed "early" so as to avoid their repetition later should the result be needed more than once. Thus, the second example above would, under call-by-value, evaluate e exactly once, before the application is performed, so that the work of doing so need never be repeated. On the other hand in the first example call-by-value performs needless work computing e since it is never needed to determine the result.

From this point of view we can see that both call-by-name and call-by-value may be seen as efforts to save work, yet neither of them succeeds in minimizing the work done in every case. One way to get the best of both worlds is to use a hybrid strategy, known as *call-by-need*, that behaves like call-by-name in avoiding evaluation of the argument to a function application at the call site, but which avoids replication of effort by employing *memoization* to save the result of an evaluation for future reference, ensuring that a computation is performed at most once. In Section 1 we will present a formal semantics of call-by-need to help clarify this important idea.

All of these laziness concepts focus on function application. But there are other important aspects of laziness as well. One aspect is that laziness permits a general form of recursive self-reference. In MinML self-reference is restricted to functions, which are given a name so that they may "call themselves" recursively. But it is also possible to exploit laziness to give a general form of recursive self-reference that is not tied to functions. In Section 2 we describe the semantics of recursion in a call-by-need setting. The semantics of call-by-need suggests the concept of *speculative execution*, in which delayed bindings are evaluated "in parallel" with the "main" thread of execution. We discuss speculative execution in Section 3.

Another aspect of laziness is *lazy data structures*. Should pairing evaluate its arguments before forming the pair? Should injections into a sum type evaluation their argument before performing the injection? Should recursive rolling evaluate its argument? In all these cases of data constructors the *eager* approach is to do so, and the *lazy* approach is to not. While in principle each of these decisions can be made independently of one another, it is natural to consider all three as eager or lazy together. However, it is interesting to observe that the treatment of data constructors is independent of the evaluation strategy for function calls, for the simple reason that formation of a pair, for example, is not a matter of calling a pairing function, but rather of employing the primitive operation of pair formation, and similarly for the other data constructors.

A *lazy language* is one that employs lazy evaluation for data constructors and for function applications; an *eager language* is one that imposes an eager evaluation strategy on both data constructors and function applications. Many lazy languages also permit consideration of eager data constructors, but there is little benefit in doing so compared to the alternative of supporting laziness in an eager language. This is achieved by introducing a type of *recursive suspensions* that (a) suspend evaluation, (b) memoize its computation, and (c) permit recursive self-reference. This permits us to mix-and-match any combination of eagerness and laziness on a per-program basis, rather than imposing the choice on all programs as a matter of language design. Note that it is not possible to support eagerness in a lazy language in a fully general way, because the laziness of function applications is unaffected by type information. Moreover, lazy languages typically permit recursive self-reference for any expression, which cannot be "defeated" through the use of types. Thus, eager languages are strictly more powerful than lazy languages, because the former can simulate the latter, but the converse fails. In Section 4 we detail the semantics of recursive suspensions. A speculative variant of suspensions, called *futures*, are also discussed.

32.1 Call-By-Need

The distinguishing feature of call-by-need, as compared to call-by-name, is that it records in memory the bindings of all variables so that when the binding of a variable is first needed, it is evaluated and the result is re-bound to that variable. Subsequent demands for the binding simply retrieve the stored value without having to repeat the computation. Of course, if the binding is never needed, it is never evaluated, consistently with the call-by-name semantics.

We will give the dynamic semantics of call-by-need using a transition system with states of the form (M, e), where M is a memory (a finite function mapping variables to open expressions) and e is an open expression. States satisfy the invariant that a free variable of e or any binding in M must lie within the domain of M. We write $x \# M$ to indicate that x is a variable that does not lie in the (finite) domain of M, and hence does not occur free in the configuration (M, e). If $x \# M$, then $M[x \mapsto e]$ denotes the memory M' such that $M'(y) = M(y)$ whenever y is in the domain of M, and $M'(x) = e$.

An initial state of the transition system has the form $(\emptyset, e)$, where e is a closed expression. A final state has the form (M, e), where e is an open value — but note well that variables themselves are not values! The transition judgement is inductively defined by the following rules:

$$\frac{e\ \mathsf{value}}{(M[x \mapsto e], x) \longmapsto (M[x \mapsto e], e)}$$

$$\frac{(M[x \mapsto \bullet], e) \longmapsto (M'[x \mapsto \bullet], e')}{(M[x \mapsto e], x) \longmapsto (M'[x \mapsto e'], x)}$$

$$\frac{(M, e_1) \longmapsto (M', e_1')}{(M, e_1(e_2)) \longmapsto (M', e_1'(e_2))}$$

$$\frac{x \# M}{(M, \lambda(x{:}\tau.\, e)(e_2)) \longmapsto (M[x \mapsto e_2], e)}$$

We omit here the presentation of the rules for the other constructs, which follow a similar pattern.

The crucial rules are those for variables, since application merely binds the unevaluated argument to the parameter of the function before evaluating its body. If the binding of a variable is a value, then that value is returned immediately. Otherwise, the binding must be evaluated to determine its value, which replaces the binding for future reference. This is accomplished in the second rule by performing a transition on the binding, e, of the variable, x, then replacing the binding with the result, e'. During evaluation of e the binding of x is replaced by a special construct, called a *black hole*, which ensures that evaluation would be "stuck" should the binding of x ever be required during evaluation of e. (Observe that since the black hole is not a value, and admits no transitions, a state of the form $(M[x \mapsto \bullet], x)$ is "stuck".) The main reason to replace the binding of x with a black hole is primarily to do with recursion, which will be discussed in the next section. It *is* important, however, that there be a binding for x in the memory while evaluating its contents, so as to ensure that any "fresh" variables that are added to the memory during this evaluation are different from x so as to avoid confusion.

Type safety for the call-by-need interpretation is proved by methods similar to those used to prove safety for mutable references (see Chapter 29). However, unlike the situation with reference cells, no cyclic dependencies are possible. Moreover, we wish to show that stuck states such as $(M[x \mapsto \bullet], x)$ do not arise during evaluation. We define the judgement $M : \Gamma$ iff $M(x) = e$ implies that $x : \tau$ occurs in Γ for some τ such that

$\Gamma \vdash e : \tau$. We then define the judgement (M,e) ok iff the following three conditions are met:

1. There exists Γ and τ such that $M : \Gamma$ and $\Gamma \vdash e : \tau$.
2. If $M(x) = e$, then $x \notin \mathrm{FV}(e)$.
3. If $y \in \mathrm{FV}(e)$ or $y \in \mathrm{FV}(M(x))$ for some x, then $M(y) \neq \bullet$.

The first condition captures the typing invariants, the second rules out self-reference, and the third suffices to ensure that evaluation does not get stuck due to black holes.

Theorem 32.1

1. *If (M,e) ok and $(M,e) \longmapsto (M',e')$, then (M',e') ok.*
2. *If (M,e) ok, then either (M,e) final, or there exists M' and e' such that $(M,e) \longmapsto (M',e')$.*

The first part is proved by rule induction on the definition of the transition judgement. The second part is proved by induction on the definition of (M,e) ok, treating the derivations of typing for e and the bindings in M as sub-derivations.

32.2 General Recursion

In MinML we introduced recursive functions as a special construct so as to permit functions that "call themselves". This approach is well-suited to a call-by-value interpretation in which variables range only over values. But in a call-by-name setting a much more general form of recursion is also available that permits self-reference for arbitrary expressions. This is possible under call-by-name, because variables range over general expressions, and not just values. The abstract syntax for general recursion has the form $\texttt{rec}(\tau, x.e)$, which we write in concrete syntax as $\texttt{rec}(x{:}\tau.e)$. The static semantics of general recursion is given by the following typing rule:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}(\tau, x.e) : \tau}$$

The dynamic semantics for recursive expressions in a call-by-name language is given by the following transition rule:

$$\frac{}{\texttt{rec}(\tau, x.e) \longmapsto [x \leftarrow \texttt{rec}(\tau, x.e)]e}$$

That is, recursive expressions are "unrolled" when they are evaluated. It is easy to check type safety for this variant of MinML.

Given general recursive expressions, recursive functions may be seen as derived forms by defining $\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e$ to stand for the expression $\texttt{rec}(f{:}\tau_1 \to \tau_2 . \lambda(x{:}\tau_1 . e))$. It is easy to show that we obtain the following derived transition:

$$\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e(e_1) \overset{*}{\longmapsto} [f, x \leftarrow \texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e, e_1]e,$$

as would be expected for recursive functions under a call-by-name semantics for function applications.

One approach to defining the dynamic semantics of general recursion is simply to mimic the call-by-name interpretation:

$$\frac{}{(M, \texttt{rec}(x{:}\tau.e)) \longmapsto (M, [x \leftarrow \texttt{rec}(x{:}\tau.e)]e)}.$$

But this rule does not share the evaluation of $\texttt{rec}(x{:}\tau.e)$ among the various occurrences of x in e. Instead, we adopt the following rule, which shares this evaluation across all occurrences of x in e.

$$\frac{x \,\#\, M}{(M, \texttt{rec}(x{:}\tau.e)) \longmapsto (M[x \mapsto e], x)}.$$

Observe that x may occur freely in e, in which case the binding is self-referential. In particular, if $x \in \mathrm{FV}(e)$, then evaluation of e may well require the binding of x, in which case evaluation gets stuck with a state of the form $(M[x \mapsto \bullet], x)$. For example, $\texttt{rec}(x{:}\tau.x)$ gets stuck in precisely this manner. Such stuck states correspond to infinite loops under the call-by-name semantics. We may regard this as a "checked error" for certain forms of non-termination, namely those that result from an infinite regress of self-reference.

Obviously we may prove only a weakened form of type safety for the call-by-need semantics of the language with general recursion. In particular, the second and third conditions on well-formedness for states given above cannot be maintained as an invariant, and, as a result, evaluation may get stuck at a black hole. It is best to regard this as a form of "checked error", and state the progress theorem accordingly.

32.3 Speculative Execution

An interesting variant of the call-by-need semantics is obtained by relaxing the restriction that the bindings of variables be evaluated only once they are needed. Instead, we may permit a step of execution of the binding of any variable to occur at any time. Specifically, we replace the second variable rule given in Section 1 by the following general rule:

$$\frac{(M[y \mapsto \bullet], e) \longmapsto (M[y \mapsto \bullet], e')}{(M[y \mapsto e], e_0) \longmapsto (M[y \mapsto e'], e_0)}$$

This rule permits any variable binding to be chosen at any time as the focus of attention for the next evaluation step. The first variable rule remains as-is, so that, as before, a variable may be evaluated only after the value of its binding has been determined.

This formulation is said to be *speculative*, because there is no reason to believe that the binding of the variable y will ever be needed to complete the computation — we are speculating based on no evidence that it might be and are willing to risk performing computation that is not strictly necessary. The steps of such computation may be arbitrarily threaded among the other steps of computation. This corresponds to a form of parallel execution in which we interleave the individual steps of a parallel computation to form a sequential trace of their execution.

While this may seem contrary to the principle of call-by-need evaluation, it is interesting to consider it because it leads to a particular model of parallel computation, called *speculative parallelism*. The idea is that in the context of a massively parallel computer, we may wish to employ otherwise idle processors by using them to speculatively evaluate unevaluated bindings of variables. To be sure, the effort may be wasted, but if the processors would otherwise be idle, there is no real loss in doing the extra work. Of course, one may wonder whether we might instead make better use of idle processors; this topic is examined more closely in Chapter 34.

32.4 Suspension Types

Call-by-need evaluation addresses only one aspect of laziness, namely to defer evaluation of function arguments until they are needed, and to share the result among all other demands for it. Another aspect of laziness is the evaluation of data constructors. When is a pair $\langle e_1, e_2 \rangle$ a value? When is an injection $\texttt{inr}(e)$ or $\texttt{inl}(e)$ a value? And when is $\texttt{roll}(e)$ a value? According to the *eager* interpretation, these expressions are values only if their constituent expressions are values — the components are "eagerly" evaluated when the composite is created. According to the *lazy* interpretation, these are all values regardless of whether their constituent expressions are values — they are evaluated only if they are required for computation to proceed. In the case of pairing one could also consider a "half lazy" interpretation, in which one component is evaluated eagerly, the other lazily, but this is rarely considered in practice.

Which interpretation to take for the data constructors is entirely separable from the decision on whether to evaluate function applications by-value or by-need. Moreover, the decisions governing the various forms of constructor are independent of one another — one could consider eager injection and lazy pairing, for example. Once we see the degree of generality, it becomes clear that decisions about evaluation order ought not be made at the level of the *language*, but rather at the level of the individual *program*. Any fixed language-level policy is sure to be inconvenient (or worse) for certain programs. It is preferable to put these decisions in the the hands of the programmer.

This may be achieved by consolidating all aspects of laziness into a single *type*, the type of memoized, suspended, self-referential computations, which we will call *suspensions* for short. Values of type $\texttt{susp}(\tau)$ are suspended computations of type τ. The expression forms associated with this type are given by the following grammar:

$$e \ ::= \ \texttt{delay}(e) \mid \texttt{force}(e) \mid \texttt{rec}(x\texttt{:susp}(\tau)\texttt{.}e)$$

Note that general recursion is now limited to suspension types!

The static semantics of these constructs is given by the following typing rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{delay}(e) : \texttt{susp}(\tau)}$$

$$\frac{\Gamma \vdash e : \texttt{susp}(\tau)}{\Gamma \vdash \texttt{force}(e) : \tau}$$

$$\frac{\Gamma, x : \texttt{susp}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{rec}(x\texttt{:susp}(\tau)\texttt{.}e) : \texttt{susp}(\tau)}$$

Note that the body of a recursive suspension has type τ — it is implicitly delayed when created.

The dynamic semantics of suspensions is given by a transition system that is reminiscent of that used for call-by-need. Indeed, the same principles of memoization and self-reference apply, *albeit* restricted to a particular type. The "ambient" evaluation strategy for applications and data constructors is *eager* — laziness is expresssed using suspension types. Thus a call-by-name function of type $\tau_1 \rightarrow \tau_2$ is represented by a call-by-value function of type $\tau_1\,\texttt{susp} \rightarrow \tau_2$, which takes as argument a suspended computation of type τ_1, which is a value of type $\tau_1\,\texttt{susp}$. Similarly, the type of fully lazy pairs is the type $\tau_1\,\texttt{susp} \times \tau_2\,\texttt{susp}$ consisting of pairs of suspended comptuations of type τ_1 and τ_2, respectively. In this way we can recover the benefits of laziness within the context of an eager language.

In the dynamic semantics a memory is now a finite mapping from *locations* to suspensions of the form `delay(`e`)`. Evaluation of a `delay` allocates a suspension in memory, and returns its location. Subsequent forces of that location signal that evaluation is to be performed on the associated suspension. Recursion is handled as in call-by-need, except that we must take care to distinguish locations from variables, and we implicitly delay the body.

$$\frac{}{(M,\texttt{delay}(e)) \longmapsto (M[l \mapsto \texttt{delay}(e)], l)}$$

$$\frac{(M,e) \longmapsto (M',e')}{(M,\texttt{force}(e)) \longmapsto (M',\texttt{force}(e'))}$$

$$\frac{e \text{ value}}{(M[l = \texttt{delay}(e)],\texttt{force}(l)) \longmapsto (M[l = \texttt{delay}(e)], e)}$$

$$\frac{(M[l = \bullet], e) \longmapsto (M[l = \bullet], e')}{(M[l = \texttt{delay}(e)],\texttt{force}(l)) \longmapsto (M'[l = \texttt{delay}(e')],\texttt{force}(l))}$$

$$\frac{}{(M,\texttt{rec}(x{:}\texttt{susp}(\tau).e)) \longmapsto (M[l = \texttt{delay}([x{\leftarrow}l]e)], l)}$$

We leave it as an exercise to formulate and prove type safety for the language with explicit suspensions.

Just as with call-by-need, there is also a speculative version of suspensions, which are called *futures*. Conceptually, a delayed computation in memory is evaluated speculatively "in parallel" (modeled by non-determinism in the rules) while computation along the main thread proceeds. When a suspension is forced, evalation of the main thread is blocked until the suspension has been evaluated, at which point the value is propagated to the main thread and execution proceeds. We leave to the reader as an exercise to give a precise formulation of the semantics of futures.

32.5 Excercises

1. Formulate an evaluation semantics of call-by-need, with and without general recursion, and show its equivalence with the transition semantics.

2. State and prove safety for suspensions.

3. Formulate the semantics of futures.

Part XII

Cost Semantics and Parallelism

Chapter 33

Cost Semantics

The dynamic semantics of MinML is given by a transition relation $e \longmapsto e'$ defined using Plotkin's method of Structured Operational Semantics (SOS). One benefit of a transition semantics is that it provides a natural measure of the time complexity of an expression, namely the number of steps required to reach a value.

An evaluation semantics, on the other hand, has an appealing simplicity, since it defines directly the value of an expression, suppressing the details of the process of execution. However, by doing so, we no longer obtain a direct account of the cost of evaluation as we do in the transition semantics.

The purpose of a *cost semantics* is to enrich evaluation semantics to record not only the value of each expression, but also the cost of evaluating it. One natural notion of cost is the number of instructions required to evaluate the expression to a value. The assignment of costs in the cost semantics can be justified by relating it to the transition semantics.

33.1 Evaluation Semantics

The evaluation relation, $e \Downarrow v$, for MinML is inductively defined by the following inference rules.

$$\frac{}{n \Downarrow n} \tag{33.1}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{+(e_1, e_2) \Downarrow n_1 + n_2} \tag{33.2}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}} \tag{33.3}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow v} \tag{33.4}$$

$$\frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow v} \tag{33.5}$$

$$\frac{}{\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e \Downarrow \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e} \tag{33.6}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [f, x \leftarrow v_1, v_2]e \Downarrow v}{\texttt{apply(}e_1\texttt{,}e_2\texttt{)} \Downarrow v} \tag{33.7}$$

(where $v_1 = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e$.)

This concludes the definition of the evaluation semantics of MinML. As you can see, the specification is quite small and is very intuitively appealing.

33.2 Relating Evaluation Semantics to Transition Semantics

The precise relationship between SOS and ES is given by the following theorem.

Theorem 33.1

1. *If $e \Downarrow v$, then $e \longmapsto^* v$.*
2. *If $e \longmapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. Consequently, if $e \longmapsto^* v$, then $e \Downarrow v$.*

Proof:

1. By induction on the rules defining the evaluation relation. The result is clearly true for values, since trivially $v \longmapsto^* v$. Suppose that $e = \texttt{apply(}e_1\texttt{,}e_2\texttt{)}$ and assume that $e \Downarrow v$. Then $e_1 \Downarrow v_1$, where $v_1 =$

$\texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e$, $e_2 \Downarrow v_2$, and $[f, x \leftarrow v_1, v_2]e \Downarrow v$. By induction we have that $e_1 \longmapsto^* v_1$, $e_2 \longmapsto^* v_2$ and $[f, x \leftarrow v_1, v_2]e \longmapsto^* v$. It follows that $\texttt{apply(}e_1\texttt{,}e_2\texttt{)} \longmapsto^* \texttt{apply(}v_1\texttt{,}e_2\texttt{)} \longmapsto^* \texttt{apply(}v_1\texttt{,}v_2\texttt{)} \longmapsto [f, x \leftarrow v_1, v_2]e \longmapsto^* v$, as required. The other cases are handled similarly.

2. By induction on the rules defining single-step transition. Suppose that $e = \texttt{apply(}v_1\texttt{,}v_2\texttt{)}$, where $v_1 = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e$, and $e' = [f, x \leftarrow v_1, v_2]e$. Suppose further that $e' \Downarrow v$; we are to show that $e \Downarrow v$. Since $v_1 \Downarrow v_1$ and $v_2 \Downarrow v_2$, the result follows immediately from the assumption that $e' \Downarrow v$. Now suppose that $e = \texttt{apply(}e_1\texttt{,}e_2\texttt{)}$ and $e' = \texttt{apply(}e_1'\texttt{,}e_2\texttt{)}$, where $e_1 \longmapsto e_1'$. Assume that $e' \Downarrow v$; we are to show that $e \Downarrow v$. It follows that $e_1' \Downarrow v_1$, $e_2 \Downarrow v_2$, and $[f, x \leftarrow v_1, v_2]e \Downarrow v$. By induction $e_1 \Downarrow v_1$, and hence $e \Downarrow v$. The remaining cases are handled similarly. It follows by induction on the rules defining multi-step evaluation that if $e \longmapsto^* v$, then $e \Downarrow v$. The base case, $v \longmapsto^* v$, follows from the fact that $v \Downarrow v$. Now suppose that $e \longmapsto e' \longmapsto^* v$. By induction $e' \Downarrow v$, and hence $e \Downarrow v$ by what we have just proved.

■

33.3 Cost Semantics

In this section we will give a cost semantics for MinML that reflects the number of steps required to complete evaluation according to the structured operational semantics given in Chapter 12.

Evaluation judgements have the form $e \Downarrow^n v$, with the informal meaning that e evaluates to v in n steps. The rules for deriving these judgements are easily defined.

$$\frac{}{n \Downarrow^0 n} \tag{33.8}$$

$$\frac{e_1 \Downarrow^{k_1} n_1 \quad e_2 \Downarrow^{k_2} n_2}{+(e_1,e_2) \Downarrow^{k_1+k_2+1} n_1 + n_2} \tag{33.9}$$

(and similarly for the other primitive operations).

$$\frac{}{\texttt{true} \Downarrow^0 \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow^0 \texttt{false}} \tag{33.10}$$

$$\frac{e \Downarrow^{k} \texttt{true} \quad e_1 \Downarrow^{k_1} v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow^{k+k_1+1} v} \tag{33.11}$$

$$\frac{e \Downarrow^{k} \texttt{false} \quad e_2 \Downarrow^{k_2} v}{\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow^{k+k_2+1} v} \tag{33.12}$$

$$\frac{}{\texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e \Downarrow^{0} \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e} \tag{33.13}$$

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad e_2 \Downarrow^{k_2} v_2 \quad [f, x \leftarrow v_1, v_2]e \Downarrow^{k} v}{\texttt{apply(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{k_1+k_2+k+1} v} \tag{33.14}$$

(where $v_1 = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e$.)

This completes the definition of the cost semantics for MinML.

33.4 Relating Cost Semantics to Transition Semantics

What is it that makes the cost semantics given above "correct"? Informally, we expect that if $e \Downarrow^{k} v$, then e should evaluate to v in k steps. Moreover, we also expect the converse to hold — the cost semantics should be completely faithful to the underlying execution model. This is captured by the following theorem.

To state the theorem we need one additional bit of notation. Define $e \overset{k}{\longmapsto} e'$ by induction on k as follows. For the basis, we define $e \overset{0}{\longmapsto} e'$ iff $e = e'$; if $k = k' + 1$, we define $e \overset{k}{\longmapsto} e'$ to hold iff $e \longmapsto e'' \overset{k'}{\longmapsto} e'$.

Theorem 33.2
For any closed expression e and closed value v of the same type, $e \Downarrow^{k} v$ iff $e \overset{k}{\longmapsto} v$.

Proof: From left to right we proceed by induction on the definition of the cost semantics. For example, consider the rule for function application. We have $e = \texttt{apply(}e_1\texttt{,}e_2\texttt{)}$ and $k = k_1 + k_2 + k + 1$, where

1. $e_1 \Downarrow^{k_1} v_1$,

2. $e_2 \Downarrow^{k_2} v_2$,
3. $v_1 = \texttt{fun}\,f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$,
4. $[f, x{\leftarrow}v_1, v_2]e \Downarrow^{k} v$.

By induction we have

1. $e_1 \xrightarrow{k_1} v_1$,
2. $e_2 \xrightarrow{k_2} v_2$,
3. $[f, x{\leftarrow}v_1, v_2]e \xrightarrow{k} v$,

and hence

$$\begin{array}{rcl} e_1\texttt{(}e_2\texttt{)} & \overset{k_1}{\longmapsto} & v_1\texttt{(}e_2\texttt{)} \\ & \overset{k_2}{\longmapsto} & v_1\texttt{(}v_2\texttt{)} \\ & \longmapsto & [f, x{\leftarrow}v_1, v_2]e \\ & \overset{k}{\longmapsto} & v \end{array}$$

which is enough for the result.

From right to left we proceed by induction on k. For $k = 0$, we must have $e = v$. By inspection of the cost evaluation rules we may check that $v \Downarrow^0 v$ for every value v. For $k = k' + 1$, we must show that if $e \longmapsto e'$ and $e' \Downarrow^{k'} v$, then $e \Downarrow^k v$. This is proved by a subsidiary induction on the transition rules. For example, suppose that $e = e_1\texttt{(}e_2\texttt{)} \longmapsto e_1'\texttt{(}e_2\texttt{)} = e'$, with $e_1 \longmapsto e_1'$. By hypothesis $e_1'\texttt{(}e_2\texttt{)} \Downarrow^k v$, so $k = k_1 + k_2 + k_3 + 1$, where

1. $e_1' \Downarrow^{k_1} v_1$,
2. $e_2 \Downarrow^{k_2} v_2$,
3. $v_1 = \texttt{fun}\,f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$,
4. $[f, x{\leftarrow}v_1, v_2]e \Downarrow^{k_3} v$.

By induction $e_1 \Downarrow^{k_1+1} v_1$, hence $e \Downarrow^{k+1} v$, as required. ■

33.5 Exercises

Chapter 34

Implicit Parallelism

In this chapter we study the extension of MinML with *implicit data parallelism*, a means of speeding up computations by allowing expressions to be evaluated simultaneously. By "implicit" we mean that the use of parallelism is invisible to the programmer as far as the ultimate results of computation are concerned. By "data parallel" we mean that the parallelism in a program arises from the simultaneous evaluation of the components of a data structure.

Implicit parallelism is very natural in an effect-free language such as MinML. The reason is that in such a language it is not possible to determine the order in which the components of an aggregate data structure are evaluated. They might be evaluated in an arbitrary sequential order, or might even be evaluated simultaneously, without affecting the outcome of the computation. This is in sharp contrast to effect-ful languages, for then the order of evaluation, or the use of parallelism, is visible to the programmer. Indeed, dependence on the evaluation order must be carefully guarded against to ensure that the outcome is determinate.

34.1 Tuple Parallelism

We begin by considering a parallel semantics for tuples according to which all components of a tuple are evaluated simultaneously. For simplicity we consider only pairs, but the ideas generalize in a straightforward manner to tuples of any size. Since the "widths" of tuples are specified statically as part of their type, the amount of parallelism that can be induced in any one step is bounded by a static constant. In Section 34.3 we will extend this to permit a statically unbounded degree of parallelism.

To facilitate comparison, we will consider two operational semantics for this extension of MinML, the *sequential* and the *parallel*. The sequential semantics is as in Chapter 15. However, we now write $e \mapsto_{seq} e'$ for the transition relation to stress that this is the sequential semantics. The sequential evaluation rules for pairs are as follows:

$$\frac{e_1 \mapsto_{seq} e_1'}{(e_1,e_2) \mapsto_{seq} (e_1',e_2)} \tag{34.1}$$

$$\frac{v_1\ \mathsf{value} \quad e_2 \mapsto_{seq} e_2'}{(v_1,e_2) \mapsto_{seq} (v_1,e_2')} \tag{34.2}$$

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value}}{\texttt{split}\ (v_1,v_2)\ \texttt{as}\ (x,y)\ \texttt{in}\ e \mapsto_{seq} [x,y\leftarrow v_1,v_2]e} \tag{34.3}$$

$$\frac{e_1 \mapsto_{seq} e_1'}{\texttt{split}\ e_1\ \texttt{as}\ (x,y)\ \texttt{in}\ e_2 \mapsto_{seq} \texttt{split}\ e_1'\ \texttt{as}\ (x,y)\ \texttt{in}\ e_2} \tag{34.4}$$

The parallel semantics is similar, except that we evaluate *both* components of a pair *simultaneously* whenever this is possible. This leads to the following rules:[1]

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{(e_1,e_2) \mapsto_{par} (e_1',e_2')} \tag{34.5}$$

$$\frac{e_1 \mapsto_{par} e_1' \quad v_2\ \mathsf{value}}{(e_1,v_2) \mapsto_{par} (e_1',v_2)} \tag{34.6}$$

$$\frac{v_1\ \mathsf{value} \quad e_2 \mapsto_{par} e_2'}{(v_1,e_2) \mapsto_{par} (v_1,e_2')} \tag{34.7}$$

Three rules are required to account for the possibility that evaluation of one component may complete before the other.

[1]It might be preferable to admit progress on either e_1 or e_2 alone, without requiring the other to be a value.

When presented two semantics for the same language, it is natural to ask whether they are equivalent. They are, in the sense that both semantics deliver the same value for any expression. This is the precise statement of what we mean by "implicit parallelism".

Theorem 34.1
For every closed, well-typed expression e, $e \mapsto^*_{seq} v$ *iff* $e \mapsto^*_{par} v$.

Proof: For the implication from left to right, it suffices to show that if $e \mapsto_{seq} e' \mapsto^*_{par} v$, then $e \mapsto^*_{par} v$. This is proved by induction on the sequential evaluation relation. For example, suppose that

$$(e_1, e_2) \mapsto_{seq} (e_1', e_2) \mapsto^*_{par} (v_1, v_2),$$

where $e_1 \mapsto_{seq} e_1'$. By inversion of the parallel evaluation sequence, we have $e_1' \mapsto^*_{par} v_1$ and $e_2 \mapsto^*_{par} v_2$. Hence, by induction, $e_1 \mapsto^*_{par} v_1$, from which it follows immediately that $(e_1, e_2) \mapsto^*_{par} (v_1, v_2)$. The other case of sequential evaluation for pairs is handled similarly. All other cases are immediate since the sequential and parallel semantics agree on all other constructs.

For the other direction, it suffices to show that if $e \mapsto_{par} e' \mapsto^*_{seq} v$, then $e \mapsto^*_{seq} v$. We proceed by induction on the definition of the parallel evaluation relation. For example, suppose that we have

$$(e_1, e_2) \mapsto_{par} (e_1', e_2') \mapsto^*_{seq} (v_1, v_2)$$

with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. We are to show that $(e_1, e_2) \mapsto^*_{seq} (v_1, v_2)$. Since $(e_1', e_2') \mapsto^*_{seq} (v_1, v_2)$, it follows that $e_1' \mapsto^*_{seq} v_1$ and $e_2' \mapsto^*_{seq} v_2$. By induction $e_1 \mapsto^*_{seq} v_1$ and $e_2 \mapsto^*_{seq} v_2$, which is enough for the result. The other cases of evaluation for pairs are handled similarly.

■

One important consequence of this theorem is that parallelism is *semantically invisible*: whether we use parallel or sequential evaluation of pairs, the result is the same. Consequently, parallelism may safely be left *implicit*, at least as far as *correctness* is concerned. However, as one might expect, parallelism effects the *efficiency* of programs.

34.2 Work and Depth

An operational semantics for a language induces a measure of time complexity for expressions, namely the number of steps required to evaluate that expression to a value. The *sequential complexity* of an expression is its time complexity relative to the sequential semantics; the *parallel complexity* is its time complexity relative to the paralle semantics. These can, in general, be quite different. Consider, for example, the following naïve implementation of the Fibonacci sequence in MinML with products:

```
fun fib (n:int):int is
    if n=0 then 1
    else if n=1 then 1
    else plus(fib(n-1),fib(n-2)) fi fi
```

where `plus` is the following function on ordered pairs:

```
fun plus (p:int*int):int is
    split p as (m:int,n:int) in m+n
```

The sequential complexity of `fib` n is $O(2^n)$, whereas the parallel complexity of the same expression is $O(n)$. The reason is that each recursive call spawns two further recursive calls which, if evaluated sequentially, lead to an exponential number of steps to complete. However, if the two recursive calls are evaluated in parallel, then the number of parallel steps to completion is bounded by n, since n is decreased by 1 or 2 on each call. Note that the same number of arithmetic operations is performed in each case! The difference is only in whether they are performed simultaneously.

This leads naturally to the concepts of *work* and *depth*. The *work* of an expression is the total number of primitive instruction steps required to complete evaluation. Since the sequential semantics has the property that each rule has at most one premise, each step of the sequential semantics amounts to the execution of exactly one instruction. Therefore the sequential complexity coincides with the work required. (Indeed, work and sequential complexity are often taken to be synonymous.) The work required to evaluate `fib` n is $O(2^n)$.

On the other hand the *depth* of an expression is the length of the longest chain of sequential dependencies in a complete evaluation of that expression. A sequential dependency is induced whenever the value of one expression depends on the value of another, forcing a sequential evaluation ordering between them. In the Fibonacci example the two recursive calls have no sequential dependency among them, but the function itself sequentially depends on both recursive calls — it cannot return until both calls have returned. Since the parallel semantics evaluates both components of an ordered pair simultaneously, it exactly captures the independence of the two calls from each, but the dependence of the result on both. Thus the parallel complexity coincides with the depth of the computation. (Indeed, they are often taken to be synonymous.) The depth of the expression `fib` n is $O(n)$.

With this in mind, the cost semantics introduced in Chapter 33 may be extended to account for parallelism by specifying both the work and the depth of evaluation. The judgements of the parallel cost semantics have the form $e \Downarrow^{w,d} v$, where w is the work and d the depth. For all cases but evaluation of pairs the work and the depth track one another. The rule for pairs is as follows:

$$\frac{e_1 \Downarrow^{w_1,d_1} v_1 \quad e_2 \Downarrow^{w_2,d_2} v_2}{(e_1,e_2) \Downarrow^{w_1+w_2,\max(d_1,d_2)} (v_1,v_2)} \tag{34.8}$$

The remaining rules are easily derived from the sequential cost semantics, with both work and depth being additively combined at each step.[2]

The correctness of the cost semantics states that the work and depth costs are consistent with the sequential and parallel complexity, respectively, of the expression.

Theorem 34.2
For any closed, well-typed expression e, $e \Downarrow^{w,d} v$ iff $e \mapsto^{w}_{seq} v$ and $e \mapsto^{d}_{par} v$.

Proof: From left to right, we proceed by induction on the cost semantics. For example, we must show that if $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_2}_{par} v_2$, then

$$(e_1,e_2) \mapsto^{d}_{par} (v_1,v_2),$$

where $d = \max(d_1, d_2)$. Suppose that $d = d_2$, and let $d' = d - d_1$ (the case $d = d_1$ is handled similarly). We have $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_1}_{par} e_2' \mapsto^{d'}_{par} v_2$.

[2]If we choose, we might evaluate arguments of primop's in parallel, in which case the depth complexity would be calculated as one more than the maximum of the depths of its arguments. We will not do this here since it would only complicate the development.

It follows that

$$\begin{array}{rll} (e_1,e_2) & \mapsto_{par}^{d_1} & (v_1,e_2') \\ & \mapsto_{par}^{d'} & (v_1,v_2). \end{array}$$

For the converse, we proceed by considering work and depth costs separately. For work, we proceed as in Chapter 33. For depth, it suffices to show that if $e \mapsto_{par} e'$ and $e' \Downarrow^{d} v$, then $e \Downarrow^{d+1} v$.[3] For example, suppose that $(e_1,e_2) \mapsto_{par} (e_1',e_2')$, with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. Since $(e_1',e_2') \Downarrow^{d} v$, we must have $v = (v_1,v_2)$, $d = \max(d_1,d_2)$ with $e_1' \Downarrow^{d_1} v_1$ and $e_2' \Downarrow^{d_2} v_2$. By induction $e_1 \Downarrow^{d_1+1} v_1$ and $e_2 \Downarrow^{d_2+1} v_2$ and hence $(e_1,e_2) \Downarrow^{d+1} (v_1,v_2)$, as desired. ■

34.3 Vector Parallelism

To support *vector parallelism* we will extend MinML with a type of *vectors*, which are finite sequences of values of a given type whose length is not determined until execution time. The primitive operations on vectors are chosen so that they may be executed in parallel on a *shared memory multiprocessor*, or *SMP*, in constant depth for an arbitrary vector.

The following primitives are added to MinML to support vectors:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \tau\,\texttt{vector} \\ \textit{Expr's} & e & ::= & \texttt{[}e_0\texttt{,}\ldots\texttt{,}e_{n-1}\texttt{]} \mid \texttt{elt(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{size(}e\texttt{)} \mid \texttt{index(}e\texttt{)} \mid \\ & & & \texttt{map(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{update(}e_1\texttt{,}e_2\texttt{)} \\ \textit{Values} & v & ::= & \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]} \end{array}$$

These expressions may be informally described as follows. The expression $\texttt{[}e_0\texttt{,}\ldots\texttt{,}e_{n-1}\texttt{]}$ evaluates to an n-vector whose elements are given by the expressions e_i, $0 \leq i < n$. The operation $\texttt{elt(}e_1\texttt{,}e_2\texttt{)}$ retrieves the element of the vector given by e_1 at the index given by e_2. The operation $\texttt{size(}e\texttt{)}$ returns the number of elements in the vector given by e. The operation $\texttt{index(}e\texttt{)}$ creates a vector of length n (given by e) whose elements are $0,\ldots,n-1$. The operation $\texttt{map(}e_1\texttt{,}e_2\texttt{)}$ applies the function given by e_1 to every element of e_2 in parallel. Finally, the operation $\texttt{update(}e_1\texttt{,}e_2\texttt{)}$ yields a new vector of the same size, n, as the vector v given by e_1, but

[3]The work component of the cost is suppressed here for the sake of clarity.

whose elements are updated according to the vector v' given by e_2. The elements of e_2 are triples of the form (b, i, x), where b is a boolean flag, i is a non-negative integer less than or equal to n, and x is a value, specifying that the ith element of v should be replaced by x, provided that $b = \texttt{true}$.

The static semantics of these primitives is given by the following typing rules:

$$\frac{\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{[}e_0\texttt{,} \ldots \texttt{,}e_{n-1}\texttt{]} : \tau\,\texttt{vector}} \tag{34.9}$$

$$\frac{\Gamma \vdash e_1 : \tau\,\texttt{vector} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{elt(}e_1\texttt{,}e_2\texttt{)} : \tau} \tag{34.10}$$

$$\frac{\Gamma \vdash e : \tau\,\texttt{vector}}{\Gamma \vdash \texttt{size(}e\texttt{)} : \texttt{int}} \tag{34.11}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{index(}e\texttt{)} : \texttt{int}\,\texttt{vector}} \tag{34.12}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arrow}(\tau, \tau') \quad \Gamma \vdash e_2 : \tau\,\texttt{vector}}{\Gamma \vdash \texttt{map(}e_1\texttt{,}e_2\texttt{)} : \tau'\,\texttt{vector}} \tag{34.13}$$

$$\frac{\Gamma \vdash e_1 : \tau\,\texttt{vector} \quad \Gamma \vdash e_2 : (\texttt{bool*int*}\tau)\,\texttt{vector}}{\Gamma \vdash \texttt{update(}e_1\texttt{,}e_2\texttt{)} : \tau\,\texttt{vector}} \tag{34.14}$$

The parallel dynamic semantics is given by the following rules. The most important is the parallel evaluation rule for vector expressions, since this is the sole source of parallelism:

$$\frac{\forall i \in I\ (e_i \mapsto_{par} e_i') \quad \forall i \notin I\ (e_i' = e_i \ \&\ e_i \ \textsf{value})}{\texttt{[}e_0\texttt{,} \ldots \texttt{,}e_{n-1}\texttt{]} \mapsto_{par} \texttt{[}e_0'\texttt{,} \ldots \texttt{,}e_{n-1}'\texttt{]}} \tag{34.15}$$

where $\emptyset \neq I \subseteq \{\, 0, \ldots, n-1 \,\}$. This allows for the parallel evaluation of all components of the vector that have not yet been evaluated.

For each of the primitive operations of the language there is a rule specifying that its arguments are evaluated in left-to-right order. We omit these rules here for the sake of brevity. The primitive instructions are as follows:

$$\frac{}{\texttt{elt([}v_0\texttt{,} \ldots \texttt{,}v_{n-1}\texttt{],}i\texttt{)} \mapsto_{par} v_i} \tag{34.16}$$

$$\frac{}{\texttt{size([}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{])} \mapsto_{par} n} \tag{34.17}$$

$$\frac{}{\texttt{index(}n\texttt{)} \mapsto_{par} \texttt{[0,}\ldots\texttt{,}n-1\texttt{]}} \tag{34.18}$$

$$\frac{}{\texttt{map(}v\texttt{,[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{])} \mapsto_{par} \texttt{[apply(}v\texttt{,}v_0\texttt{),}\ldots\texttt{,apply(}v\texttt{,}v_{n-1}\texttt{)]}} \tag{34.19}$$

$$\frac{}{\begin{array}{c}\texttt{update([}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{],[(}b_0\texttt{,}i_0\texttt{,}x_0\texttt{),}\ldots\texttt{,(}b_{k-1}\texttt{,}i_{k-1}\texttt{,}x_{k-1}\texttt{)])}\\ \mapsto_{par}\\ \texttt{[}v'_0\texttt{,}\ldots\texttt{,}v'_{n-1}\texttt{]}\end{array}} \tag{34.20}$$

where for each $i \in \{ i_0,\ldots,i_{k-1} \}$, if b_i is `true`, then $v'_i = x_i$, and otherwise $v'_i = v_i$. If an index i appears more than once, the rightmost occurrence takes precedence over the others.

The sequential dynamic semantics of vectors is defined similarly to the parallel semantics. The only difference is that vector expressions are evaluated in left-to-right order, rather than in parallel. This is expressed by the following rule:

$$\frac{e_i \mapsto_{seq} e'_i}{\texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{i-1}\texttt{,}e_i\texttt{,}e_{i+1}\texttt{,}\ldots\texttt{,}e_{n-1}\texttt{]} \longmapsto \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{i-1}\texttt{,}e'_i\texttt{,}e_{i+1}\texttt{,}\ldots\texttt{,}e_{n-1}\texttt{]}} \tag{34.21}$$

We write $e \mapsto_{seq} e'$ to indicate that e steps to e' under the sequential semantics.

With these two basic semantics in mind, we may also derive a cost semantics for MinML with vectors, where the work corresponds to the number of steps required in the sequential semantics, and the depth corresponds to the number of steps required in the parallel semantics. The rules are as follows.

Vector expressions are evaluated in parallel.

$$\frac{\forall\, 0 \leq i < n\ (e_i \Downarrow^{w_i,d_i} v_i)}{\texttt{[}e_0\texttt{,}\ldots\texttt{,}e_{n-1}\texttt{]} \Downarrow^{w,d} \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]}} \tag{34.22}$$

where $w = \sum_{i=0}^{n-1} w_i$ and $d = \max_{i=0}^{n-1} d_i$.

Retrieving an element of a vector takes constant work and depth.

$$\frac{e_1 \Downarrow^{w_1,d_1} \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]} \quad e_2 \Downarrow^{w_2,d_2} i \quad (0 \leq i < n)}{\texttt{elt(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{w_1+w_2+1,d_1+d_2+1} v_i} \tag{34.23}$$

Retrieving the size of a vector takes constant work and depth.

$$\frac{e \Downarrow^{w,d} \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]}}{\texttt{size(}e\texttt{)} \Downarrow^{w+1,d+1} n} \tag{34.24}$$

Creating an index vector takes linear work and constant depth.

$$\frac{e \Downarrow^{w,d} n}{\texttt{index(}e\texttt{)} \Downarrow^{w+n,d+1} \texttt{[}0\texttt{,}\ldots\texttt{,}n-1\texttt{]}} \tag{34.25}$$

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

$$\frac{\begin{array}{c} e_1 \Downarrow^{w_1,d_1} v \quad e_2 \Downarrow^{w_2,d_2} \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]} \\ \texttt{[apply(}v\texttt{,}v_0\texttt{),}\ldots\texttt{,apply(}v\texttt{,}v_{n-1}\texttt{)]} \Downarrow^{w,d} \texttt{[}v'_0\texttt{,}\ldots\texttt{,}v'_{n-1}\texttt{]} \end{array}}{\texttt{map(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1} \texttt{[}v'_0\texttt{,}\ldots\texttt{,}v'_{n-1}\texttt{]}} \tag{34.26}$$

Updating a vector takes linear work and constant depth.

$$\frac{e_1 \Downarrow^{w_1,d_1} \texttt{[}v_0\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{]} \quad e_2 \Downarrow^{w_2,d_2} \texttt{[(}b_1\texttt{,}i_1\texttt{,}x_1\texttt{),}\ldots\texttt{,(}b_k\texttt{,}i_k\texttt{,}x_k\texttt{)]}}{\texttt{update(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{w_1+w_2+k+n,d_1+d_2+1} \texttt{[}v'_0\texttt{,}\ldots\texttt{,}v'_{n-1}\texttt{]}} \tag{34.27}$$

where for each $i \in \{ i_1, \ldots, i_k \}$, if b_i is `true`, then $v'_i = x_i$, and otherwise $v'_i = v_i$. If an index i appears more than once, the rightmost occurrence takes precedence over the others.

Theorem 34.3
For the extension of MinML with vectors, $e \Downarrow^{w,d} v$ iff $e \mapsto^{d}_{par} v$ and $e \mapsto^{w}_{seq} v$.

Chapter 35

A Parallel Abstract Machine

The parallel operational semantics described in Chapter 34 abstracts away some important aspects of the implementation of parallelism. For example, the parallel evaluation rule for ordered pairs

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\texttt{(}e_1\texttt{,}e_2\texttt{)} \mapsto_{par} \texttt{(}e_1'\texttt{,}e_2'\texttt{)}}$$

does not account for the overhead of allocating e_1 and e_2 to two (physical or virtual) processors, or for synchronizing with those two processors to obtain their results. In this chapter we will discuss a more realistic operational semantics that accounts for this overhead.

35.1 A Simple Parallel Language

Rather than specify which primitives, such as pairing, are to be evaluated in parallel, we instead introduce a "parallel let" construct that allows the programmer to specify the simultaneous evaluation of two expressions. Moreover, we restrict the language so that the arguments to all primitive operations must be values. This forces the programmer to decide for herself which constructs are to be evaluated in parallel, and which are to be evaluated sequentially.

$$\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \tau_1\texttt{*}\tau_2 \mid \texttt{arrow}(\tau_1,\tau_2) \\
\textit{Expressions} & e & ::= & v \mid \texttt{let}\, x_1\,\texttt{:}\,\tau_1\,\texttt{be}\, e_1\,\texttt{and}\, x_2\,\texttt{:}\,\tau_2\,\texttt{be}\, e_2\,\texttt{in}\, e\,\texttt{end} \mid \\
 & & & o(v_1,\ldots,v_n) \mid \texttt{if}\,\tau\,\texttt{then}\, v\,\texttt{else}\, e_1 e_2 \mid \\
 & & & \texttt{apply(}v_1\texttt{,}v_2\texttt{)} \mid \texttt{split}\, v\,\texttt{as}\,\texttt{(}x_1\texttt{,}x_2\texttt{)}\,\texttt{in}\, e \\
\textit{Values} & v & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{(}v_1\texttt{,}v_2\texttt{)} \mid \\
 & & & \texttt{fun}\, x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\, e
\end{array}$$

The binding conventions are as for MinML with product types, with the additional specification that the variables x_1 and x_2 are bound within the body of a `let` expression. Note that variables are regarded as values only for the purpose of defining the syntax of the language; evaluation is, as ever, defined only on closed terms.

As will become apparent when we specify the dynamic semantics, the "sequential let" is definable from the "parallel let":

$$\texttt{let}\,\tau_1\,\texttt{:}\,x_1\,\texttt{be}\,e_1\,\texttt{in}\,e_2 := \texttt{let}\,x_1\,\texttt{:}\,\tau_1\,\texttt{be}\,e_1\,\texttt{and}\,x\,\texttt{:unit}\,\texttt{be}\,\texttt{()}\,\texttt{in}\,e_2\,\texttt{end}$$

where x does not occur free in e_2. Using these, the "parallel pair" is definable by the equation

$$\texttt{(}e_1\texttt{,}e_2\texttt{)}_{par} := \texttt{let}\,x_1\,\texttt{:}\,\tau_1\,\texttt{be}\,e_1\,\texttt{and}\,x_2\,\texttt{:}\,\tau_2\,\texttt{be}\,e_2\,\texttt{in}\,\texttt{(}x_1\texttt{,}x_2\texttt{)}\,\texttt{end}$$

whereas the "(left-to-right) sequential pair" is definable by the equation

$$\texttt{(}e_1\texttt{,}e_2\texttt{)}_{seq} := \texttt{let}\,\tau_1\,\texttt{:}\,x_1\,\texttt{be}\,e_1\,\texttt{in}\,\texttt{let}\,\tau_2\,\texttt{:}\,x_2\,\texttt{be}\,e_2\,\texttt{in}\,\texttt{(}x_1\texttt{,}x_2\texttt{)}.$$

The static semantics of this language is essentially that of MinML with product types, with the addition of the following typing rule for the parallel let construct:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{let}\,x_1\,\texttt{:}\,\tau_1\,\texttt{be}\,e_1\,\texttt{and}\,x_2\,\texttt{:}\,\tau_2\,\texttt{be}\,e_2\,\texttt{in}\,e\,\texttt{end} : \tau} \tag{35.1}$$

It is a simple exercise to give a parallel structured operational semantics to this language in the style of Chapter 34. In particular, it would employ the following rules for the parallel `let` construct.

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\begin{array}{c}\texttt{let}\,x_1\,\texttt{:}\,\tau_1\,\texttt{be}\,e_1\,\texttt{and}\,x_2\,\texttt{:}\,\tau_2\,\texttt{be}\,e_2\,\texttt{in}\,e\,\texttt{end} \\ \mapsto_{par} \\ \texttt{let}\,x_1\,\texttt{:}\,\tau_1\,\texttt{be}\,e_1'\,\texttt{and}\,x_2\,\texttt{:}\,\tau_2\,\texttt{be}\,e_2'\,\texttt{in}\,e\,\texttt{end}\end{array}} \tag{35.2}$$

$$\frac{e_1 \mapsto_{par} e_1'}{\begin{array}{c}\texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ v_2\ \texttt{in}\ e\ \texttt{end} \\ \mapsto_{par} \\ \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ e_1'\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ v_2\ \texttt{in}\ e\ \texttt{end}\end{array}} \quad (35.3)$$

$$\frac{e_2 \mapsto_{par} e_2'}{\begin{array}{c}\texttt{let}\ x_1 : \tau_1\ \texttt{be}\ v_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2\ \texttt{in}\ e\ \texttt{end} \\ \mapsto_{par} \\ \texttt{let}\ x_1 : \tau_1\ \texttt{be}\ v_1\ \texttt{and}\ x_2 : \tau_2\ \texttt{be}\ e_2'\ \texttt{in}\ e\ \texttt{end}\end{array}} \quad (35.4)$$

However, these rules ignore the overhead associated with allocating the sub-expression to processors. In the next section we will consider an abstract machine that accounts for this overhead.

Exercise 35.1
Prove preservation and progress for the static and dynamic semantics just given.

35.2 A Parallel Abstract Machine

The essence of parallelism is the simultaneous execution of several programs. Each execution is called a *thread of control*, or *thread*, for short. The problem of devising a parallel abstract machine is how to represent multiple threads of control, in particular how to represent the creation of new threads and synchronization between threads. The P-machine is designed to represent a parallel computer with an unbounded number of processors in a simple and elegant manner.

The main idea of the P-machine is represent the state of a parallel computer by a nested composition of parallel `let` statements representing the *active* threads in a program. Each step of the machine consists of executing *all* of the active instructions in the program, resulting in a new P-state.

In order to account for the activation of threads and the synchronization of their results we make explicit the process of *activating* an expression, which corresponds to assigning it to a processor for execution. Execution of a parallel `let` instruction whose constituent expressions have not yet been activated consists of the activation of these expressions. Execution of a parallel `let` whose constituents are completely evaluated consists of substituting the values of these expressions into the body of the `let`, which is itself then activated. Execution of all other instructions is exactly as before, with the result being made active in each case.

This can be formalized using *parallelism contexts*, which capture the tree structure of nested parallel computations. Let l and variants range over a countable set of *labels*. These will serve to identify the abstract processors assigned to the execution of an active expression. The set of parallelism contexts $\mathcal{L}$ is defined by the following grammar:

$$\mathcal{L} \; ::= \; l{:}- \mid l{:}\texttt{let}\,x_1{:}\tau_1\,\texttt{be}\,\mathcal{L}_1\,\texttt{and}\,x_2{:}\tau_2\,\texttt{be}\,\mathcal{L}_2\,\texttt{in}\,e\,\texttt{en}$$

$$l{:}\texttt{let}\,x_1{:}\tau_1\,\texttt{be}\,\mathcal{L}_1\,\texttt{and}\,x_2{:}\tau_2\,\texttt{be}\,v_2\,\texttt{in}\,e\,\texttt{end} \mid$$
$$l{:}\texttt{let}\,x_1{:}\tau_1\,\texttt{be}\,v_1\,\texttt{and}\,x_2{:}\tau_2\,\texttt{be}\,\mathcal{L}_2\,\texttt{in}\,e\,\texttt{end}$$

A parallelism context is *well-formed* only if all labels occurring within it are distinct; hereafter we will consider only well-formed parallelism contexts.

A labelled "hole" in a parallelism context represents an active computation site; a labelled `let` expression represents a pending computation that is awaiting completion of its child threads. We have arranged things so that all active sites are children of pending sites, reflecting the intuition that an active site must have been spawned by some (now pending) site.

The *arity* of a context is defined to be the number of "holes" occurring within it. The arity is therefore the number of active threads within the context. If $\mathcal{L}$ is a context with arity n, then the expression $\mathcal{L}[l = e]_{i=1}^{n}$ represents the result of "filling" the hole labelled l_i with the expression e_i, for each $1 \leq i \leq n$. Thus the e_i's represent the active expressions within the context; the label l_i represents the "name" of the processor assigned to execute e_i.

Each step of the P-machine consists of executing *all* of the active instructions in the current state. This is captured by the following evaluation rule:

$$\frac{e_1 \longrightarrow e_1' \quad \cdots \quad e_n \longrightarrow e_n'}{\mathcal{L}[l = e]_{i=1}^{n} \mapsto_{\mathsf{P}} \mathcal{L}[l = e']_{i=1}^{n}}$$

The relation $e \longrightarrow e'$ defines the atomic instruction steps of the P-machine. These are defined by a set of axioms. The first is the *fork* axiom, which initiates execution of a parallel `let` statement:

$$\frac{}{\begin{array}{c}\texttt{let}\,x_1{:}\tau_1\,\texttt{be}\,e_1\,\texttt{and}\,x_2{:}\tau_2\,\texttt{be}\,e_2\,\texttt{in}\,e\,\texttt{end}\\ \longrightarrow \\ \texttt{let}\,x_1{:}\tau_1\,\texttt{be}\,l_1{:}e_1\,\texttt{and}\,x_2{:}\tau_2\,\texttt{be}\,l_2{:}e_2\,\texttt{in}\,e\,\texttt{end}\end{array}} \tag{35.5}$$

Here l_1 and l_2 are "new" labels that do not otherwise occur in the computation. They serve as the labels of the processors assigned to execute e_1 and e_2, respectively.

The second instruction is the *join* axiom, which completes execution of a parallel `let`:

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value}}{\texttt{let}\, x_1 : \tau_1 \,\texttt{be}\, l_1 : v_1 \,\texttt{and}\, x_2 : \tau_2 \,\texttt{be}\, l_2 : v_2 \,\texttt{in}\, e \,\texttt{end} \longrightarrow [x_1, x_2 \leftarrow v_1, v_2]e} \tag{35.6}$$

The other instructions are inherited from the M-machine. For example, function application is defined by the following instruction:

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \quad (v_1 = \texttt{fun}\, f\ (x : \tau_1) : \tau_2 \,\texttt{is}\, e)}{\texttt{apply}(v_1, v_2) \longrightarrow [f, x \leftarrow v_1, v_2]e} \tag{35.7}$$

This completes the definition of the P-machine.

Exercise 35.2
State and prove preservation and progress relative to the P*-machine.*

35.3 Cost Semantics, Revisited

A primary motivation for introducing the P-machine was to achieve a proper accounting for the cost of creating and synchronizing threads. In the simplified model of Chapter 34 we ignored these costs, but here we seek to take them into account. This is accomplished by taking the following rule for the cost semantics of the parallel `let` construct:

$$\frac{e_1 \Downarrow^{w_1, d_1} v_1 \quad e_2 \Downarrow^{w_2, d_2} v_2 \quad [x_1, x_2 \leftarrow v_1, v_2]e \Downarrow^{w, d} v}{\texttt{let}\, x_1 : \tau_1 \,\texttt{be}\, e_1 \,\texttt{and}\, x_2 : \tau_2 \,\texttt{be}\, e_2 \,\texttt{in}\, e \,\texttt{end} \Downarrow^{w', d'} v} \tag{35.8}$$

where $w' = w_1 + w_2 + w + 2$ and $d' = \max(d_1, d_2) + d + 2$. Since the remaining expression forms are all limited to values, they have unit cost for both work and depth.

The calculation of work and depth for the parallel `let` construct is justified by relating the cost semantics to the P-machine. The work performed in an evaluation sequence $e \mapsto^{*}_{\mathsf{P}} v$ is the total number of primitive instruction steps performed in the sequence; it is the sequential cost of executing the expression e.

Theorem 35.3
If $e \Downarrow^{w,d} v$, then $l : e \mapsto^{d}_{\mathsf{P}} l : v$ with work w.

Proof: The proof from left to right proceeds by induction on the cost semantics. For example, consider the cost semantics of the parallel `let` construct. By induction we have

1. $l_1 : e_1 \mapsto^{d_1}_{\mathsf{P}} l_1 : v_1$ with work w_1;
2. $l_2 : e_2 \mapsto^{d_2}_{\mathsf{P}} l_2 : v_2$ with work w_2;
3. $l : [x_1, x_2 \leftarrow v_1, v_2]e \mapsto^{d}_{\mathsf{P}} l : v$ with work w.

We therefore have the following P-machine evaluation sequence:

$$\begin{array}{ll}
l : \mathtt{let}\, x_1 : \tau_1\, \mathtt{be}\, e_1\, \mathtt{and}\, x_2 : \tau_2\, \mathtt{be}\, e_2\, \mathtt{in}\, e\, \mathtt{end} & \mapsto_{\mathsf{P}} \\
l : \mathtt{let}\, x_1 : \tau_1\, \mathtt{be}\, l_1 : e_1\, \mathtt{and}\, x_2 : \tau_2\, \mathtt{be}\, l_2 : e_2\, \mathtt{in}\, e\, \mathtt{end} & \mapsto^{\max(d_1,d_2)}_{\mathsf{P}} \\
l : \mathtt{let}\, x_1 : \tau_1\, \mathtt{be}\, l_1 : v_1\, \mathtt{and}\, x_2 : \tau_2\, \mathtt{be}\, l_2 : v_2\, \mathtt{in}\, e\, \mathtt{end} & \mapsto_{\mathsf{P}} \\
l : [x_1, x_2 \leftarrow v_1, v_2]e & \mapsto^{d}_{\mathsf{P}} \\
l : v &
\end{array}$$

The total length of the evaluation sequence is $\max(d_1, d_2) + d + 2$, as required by the depth cost, and the total work is $w_1 + w_2 + w + 2$, as required by the work cost. ■

35.4 Provable Implementations (Summary)

The semantics of parallelism given above is based on an idealized parallel computer with an unlimited number of processors. In practice this idealization must be simulated using some fixed number, p, of physical processors. In practice p is on the order of 10's of processors, but may even rise (at the time of this writing) into the 100's. In any case p does not vary with input size, but is rather a fixed parameter of the implementation platform. The important question is how efficiently can one simulate unbounded parallelism using only p processors? That is, how realistic are the costs assigned to the language by our semantics? Can we make accurate predictions about the running time of a program on a real parallel computer based on the idealized cost assigned to it by our semantics?

The answer is *yes*, through the notion of a *provably efficient implementation*. While a full treatment of these ideas is beyond the scope of this book, it is worthwhile to summarize the main ideas.

Theorem 35.4 (Blelloch and Greiner)
If $e \Downarrow^{w,d} v$, then e can be evaluated on an SMP with p-processors in time $O(w/p + d \lg p)$.

For our purposes, an SMP is any of a wide range of parallel computers, including a CRCW PRAM, a hypercube, or a butterfly network. Observe that for $p = 1$, the stated bound simplifies to $O(w)$, as would be expected.

To understand the significance of this theorem, observe that the definition of work and depth yields a lower bound of $\Omega(\max(w/p, d))$ on the execution time on p processors. We can never complete execution in fewer than d steps, and can, at best, divide the total work evenly among the p processors. The theorem tells us that we can come within a constant factor of this lower bound. The constant factor, $\lg p$, represents the overhead of scheduling parallel computations on p processors.

The goal of parallel programming is to maximize the use of parallelism so as to minimize the execution time. By the theorem this will occur if the term w/p dominates, which occurs if the ratio w/d of work to depth is at least $p \lg p$. This ratio is sometimes called the *parallelizability* of the program. For highly sequential programs, d is directly proportional to w, yielding a low parallelizability — increasing the number of processors will not speed up the computation. For highly parallel programs, d might be constant or proportional to $\lg w$, resulting in a large parallelizability, and good utilization of the available computing resources. It is important to keep in mind that *it is not known* whether there are inherently sequential problems (for which no parallelizable solution is possible), or whether, instead, all problems can benefit from parallelism. The best that we can say at the time of this writing is that there are problems for which no parallelizable solution is known.

To get a sense of what is involved in the proof of Blelloch and Greiner's theorem, let us consider the assumption that the index operation on vectors (given in Chapter 34) has constant depth. The theorem implies that index is implementable on an SMP in time $O(n/p + \lg p)$. We will briefly sketch a proof for this one case. The main idea is that we may assume that every processor is assigned a unique number from 0 to $p - 1$. To implement index, we simply allocate, but do not initialize, a region of memory of the appropriate size, and ask each processor to simultaneously store its identifying number i into the ith element of the allocated array. This works directly if the size of the vector is no more than the number of processors. Otherwise, we may divide the problem in half, and recursively build two index vectors of half the size, one starting with zero, the other with $n/2$. This process need proceed at most $\lg p$ times before the vectors are small enough, leaving n/p sub-problems of size at most p to be solved. Thus the total time required is $O(n/p + \lg p)$, as required by the theorem.

The other primitive operations are handled by similar arguments, justifying the cost assignments made to them in the operational semantics. To complete the proof of Blelloch and Greiner's theorem, we need only argue that the total work w can indeed be allocated to p processors with a cost of only $\lg p$ for the overhead. This is a consequence of Brent's Theorem, which states that a total workload w divided into d parallel steps may be implemented on p processors in $O(n/p + d \lg p)$ time. The argument relies on certain assumptions about the SMP, including the ability to perform a parallel fetch-and-add operation in constant time.

Part XIII

Subtyping

Chapter 36

Subtyping

A *subtype* relation is a pre-order[1] on types that validates the *subsumption principle*: if σ is a subtype of τ, then a value of type σ may be provided whenever a value of type τ is required. This means that a value of the subtype should "act like" a value of the supertype when used in supertype contexts.

36.1 Subsumption

We will consider two extensions of MinML with subtyping. The first, *MinML with implicit subtyping*, is obtained by adding the following rule of *implicit subsumption* to the typing rules of MinML:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

With implicit subtyping the typing relation is no longer syntax-directed, since the subsumption rule may be applied to any expression e, without regard to its form.

The second, called *MinML with explicit subtyping*, is obtained by adding to the syntax by adding an explicit *cast* expression, $(\tau)\, e$, with the following typing rule:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e : \tau}$$

[1] A pre-order is a reflexive and transitive binary relation.

The typing rules remain syntax-directed, but all uses of subtyping must be explicitly indicated.

We will refer to either variation as $\mathsf{MinML}_{<:}$ when the distinction does not matter. When it does, the implicit version is designated $\mathsf{MinML}^{i}_{<:}$, the implicit $\mathsf{MinML}^{e}_{<:}$.

To obtain a complete instance of $\mathsf{MinML}_{<:}$ we must specify the subtype relation. This is achieved by giving a set of *subtyping axioms*, which determine the primitive subtype relationships, and a set of *variance rules*, which determine how type constructors interact with subtyping. To ensure that the subtype relation is a pre-order, we tacitly include the following rules of reflexivity and transitivity:

$$\frac{}{\tau <: \tau} \qquad \frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau}$$

Note that pure MinML is obtained as an instance of $\mathsf{MinML}^{i}_{<:}$ by giving no subtyping rules beyond these two, so that $\sigma <: \tau$ iff $\sigma = \tau$.

The dynamic semantics of an instance of $\mathsf{MinML}_{<:}$ must be careful to take account of subtyping. In the case of implicit subsumption the dynamic semantics must be defined so that the primitive operations of a supertype apply equally well to a value of any subtype. In the case of explicit subsumption we need only ensure that there be a means of *casting* a value of the subtype into a corresponding value of the supertype.

The type safety of $\mathsf{MinML}_{<:}$, in either formulation, is assured, provided that the following *subtyping safety conditions* are met:

- For $\mathsf{MinML}^{e}_{<:}$, if $\sigma <: \tau$, then casting a value of the subtype σ to the supertype τ must yield a value of type τ.

- For $\mathsf{MinML}^{i}_{<:}$, the dynamic semantics must ensure that the value of each primitive operation is defined for closed values of *any subtype* of the expected type of its arguments.

Under these conditions we may prove the Progress and Preservation Theorems for either variant of $\mathsf{MinML}_{<:}$.

Theorem 36.1 (Preservation)
For either variant of $\mathsf{MinML}_{<:}$*, under the assumption that the subtyping safety conditions hold, if* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*

Proof: By induction on the dynamic semantics, appealing to the casting condition in the case of the explicit subsumption rule of $\mathsf{MinML}^{e}_{<:}$. ■

Theorem 36.2 (Progress)
For either variant of $\mathsf{MinML}_{<:}$*, under the assumption that the subtyping safety conditions hold, if* $e : \tau$*, then either* e *is a value or there exists* e' *such that* $e \longmapsto e'$*.*

Proof: By induction on typing, appealing to the subtyping condition on primitive operations in the case of primitive instruction steps. ■

36.2 Varieties of Subtyping

In this section we will explore several different forms of subtyping in the context of extensions of MinML. To simplify the presentation of the examples, we tacitly assume that the dynamic semantics of casts is defined so that $(\tau)\,v \longmapsto v$, unless otherwise specified.

36.2.1 Arithmetic Subtyping

In informal mathematics we tacitly treat integers as real numbers, even though $\mathbb{Z} \not\subseteq \mathbb{R}$. This is justified by the observation that there is an injection $\iota : \mathbb{Z} \hookrightarrow \mathbb{R}$ that assigns a canonical representation of an integer as a real number. This injection preserves the ordering, and commutes with the arithmetic operations in the sense that $\iota(m + n) = \iota(m) + \iota(n)$, where m and n are integers, and the relevant addition operation is determined by the types of its arguments.

In most cases the real numbers are (crudely) approximated by floating point numbers. Let us therefore consider an extension of MinML with an additional base type, `float`, of floating point numbers. It is not necessary to be very specific about this extension, except to say that we enrich the language with floating point constants and arithmetic operations. We will designate the floating point operations using a decimal point, writing `+.` for floating point addition, and so forth.[2]

[2]This convention is borrowed from O'Caml.

By analogy with mathematical practice, we will consider taking the type `int` to be a subtype of `float`. The analogy is inexact, because of the limitations of computer arithmetic, but it is, nevertheless, informative to consider it.

To ensure the safety of explicit subsumption we must define how to cast an integer to a floating point number, written $(\texttt{float})\,n$. We simply postulate that this is possible, writing $n.0$ for the floating point representation of the integer n, and noting that $n.0$ has type `float`.[3]

To ensure the safety of implicit subsumption we must ensure that the floating point arithmetic operations are well-defined for integer arguments. For example, we must ensure that an expression such as $\texttt{+.}(3,4)$ has a well-defined value as a floating point number. To achieve this, we simply require that floating point operations implicitly convert any integer arguments to floating point before performing the operation. In the foregoing example evaluation proceeds as follows:

$$\texttt{+.}(3,4) \longmapsto \texttt{+.}(3.0,4.0) \longmapsto 7.0.$$

This strategy requires that the floating point operations detect the presence of integer arguments, and that it convert any such arguments to floating point before carrying out the operation. We will have more to say about this inefficiency in Section 37.2 below.

36.2.2 Function Subtyping

Suppose that `int <: float`. What subtyping relationships, if any, should hold among the following four types?

1. $\texttt{arrow}(\texttt{int},\texttt{int})$
2. $\texttt{arrow}(\texttt{int},\texttt{float})$
3. $\texttt{arrow}(\texttt{float},\texttt{int})$
4. $\texttt{arrow}(\texttt{float},\texttt{float})$

[3]We may handle the limitations of precision by allowing for a cast operation to fail in the case of overflow. We will ignore overflow here, for the sake of simplicity.

To determine the answer, keep in mind the subsumption principle, which says that a value of the subtype should be usable in a supertype context.

Suppose f : $\texttt{arrow}(\texttt{int},\texttt{int})$. If we apply f to x : $\texttt{int}$, the result has type $\texttt{int}$, and hence, by the arithmetic subtyping axiom, has type $\texttt{float}$. This suggests that

$$\texttt{arrow}(\texttt{int},\texttt{int}) \mathrel{\texttt{<:}} \texttt{arrow}(\texttt{int},\texttt{float})$$

is a valid subtype relationship. By similar reasoning, we may derive that

$$\texttt{arrow}(\texttt{float},\texttt{int}) \mathrel{\texttt{<:}} \texttt{arrow}(\texttt{float},\texttt{float})$$

is also valid.

Now suppose that f : $\texttt{arrow}(\texttt{float},\texttt{int})$. If x : $\texttt{int}$, then x : $\texttt{float}$ by subsumption, and hence we may apply f to x to obtain a result of type $\texttt{int}$. This suggests that

$$\texttt{arrow}(\texttt{float},\texttt{int}) \mathrel{\texttt{<:}} \texttt{arrow}(\texttt{int},\texttt{int})$$

is a valid subtype relationship. Since $\texttt{arrow}(\texttt{int},\texttt{int}) \mathrel{\texttt{<:}} \texttt{arrow}(\texttt{int},\texttt{float})$, it follows that

$$\texttt{arrow}(\texttt{float},\texttt{int}) \mathrel{\texttt{<:}} \texttt{arrow}(\texttt{int},\texttt{float})$$

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called *variance* principles. If a type constructor *preserves* subtyping in a given argument position, it is said to be *covariant* in that position. If, instead, it *inverts* subtyping in a given position it is said to be *contravariant* in that position. The discussion above suggests that the function space constructor is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

$$\frac{\tau_1 \mathrel{\texttt{<:}} \sigma_1 \quad \sigma_2 \mathrel{\texttt{<:}} \tau_2}{\texttt{arrow}(\sigma_1,\sigma_2) \mathrel{\texttt{<:}} \texttt{arrow}(\tau_1,\tau_2)}$$

Note well the inversion of subtyping in the domain, where the function constructor is contravariant, and the preservation of subtyping in the range, where the function constructor is covariant.

To ensure safety in the explicit case, we define the dynamic semantics of a cast operation by the following rule:

$$\frac{}{(\texttt{arrow}(\tau_1,\tau_2))\,v \longmapsto \texttt{fn}\,x\!:\!\tau_1\,\texttt{in}\,(\tau_2)\,v\texttt{(}(\sigma_1)\,x\texttt{)}}$$

Here v has type $\texttt{arrow}(\sigma_1,\sigma_2)$, $\tau_1 \texttt{<:}\ \sigma_1$, and $\sigma_2 \texttt{<:}\ \tau_2$. The argument is cast to the domain type of the function prior to the call, and its result is cast to the intended type of the application.

To ensure safety in the implicit case, we must ensure that the primitive operation of function application behaves correctly on a function of a subtype of the "expected" type. This amounts to ensuring that a function can be called with an argument of, and yields a result of, a subtype of the intended type. One way is to adopt a semantics of procedure call that is independent of the types of the arguments and results. Another is to introduce explicit run-time checks similar to those suggested for floating point arithmetic to ensure that calling conventions for different types can be met.

36.2.3 Product and Record Subtyping

In Chapter 15 we considered an extension of MinML with product types. In this section we'll consider equipping this extension with subtyping. We will work with n-ary products of the form $\tau_1 * \cdots * \tau_n$ and with n-ary records of the form $\{l_1\!:\!\tau_1, \ldots, l_n\!:\!\tau_n\}$. The tuple types have as elements n-tuples of the form $\texttt{<}e_1, \ldots, e_n\texttt{>}$ whose ith component is accessed by projection, $e\,.\,i$. Similarly, record types have as elements records of the form $\{l_1\!:\!e_1, \ldots, l_n\!:\!e_n\}$ whose lth component is accessed by field selection, $e\,.\,l$.

Using the subsumption principle as a guide, it is natural to consider a tuple type to be a subtype of any of its prefixes:

$$\frac{m > n}{\tau_1 * \cdots * \tau_m \texttt{ <: } \tau_1 * \cdots * \tau_n}$$

Given a value of type $\tau_1 * \cdots * \tau_n$, we can access its ith component, for any $1 \leq i \leq n$. If $m > n$, then we can equally well access the ith component of an m-tuple of type $\tau_1 * \cdots * \tau_m$, obtaining the same result. This is called *width subtyping* for tuples.

For records it is natural to consider a record type to be a subtype of any record type with any subset of the fields of the subtype. This may be written as follows:

$$\frac{m > n}{\{l_1 : \tau_1, \ldots, l_m : \tau_m\} \texttt{<:} \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

Bear in mind that the ordering of fields in a record type is immaterial, so this rule allows us to neglect any subset of the fields when passing to a supertype. This is called *width subtyping* for records. The justification for width subtyping is that record components are accessed by label, rather than position, and hence the projection from a supertype value will apply equally well to the subtype.

What variance principles apply to tuples and records? Applying the principle of subsumption, it is easy to see that tuples and records may be regarded as covariant in all their components. That is,

$$\frac{\forall 1 \leq i \leq n \ \sigma_i \texttt{<:} \tau_i}{\sigma_1 * \cdots * \sigma_n \texttt{<:} \tau_1 * \cdots * \tau_n}$$

and

$$\frac{\forall 1 \leq i \leq n \ \sigma_i \texttt{<:} \tau_i}{\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \texttt{<:} \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}.$$

These are called *depth subtyping* rules for tuples and records, respectively.

To ensure safety for explicit subsumption we must define the meaning of casting from a sub- to a super-type. The two forms of casting corresponding to width and depth subtyping may be consolidated into one, as follows:

$$\frac{m \geq n}{(\tau_1 * \cdots * \tau_n)\, \texttt{<}v_1, \ldots, v_m\texttt{>} \longmapsto \texttt{<}(\tau_1)\, v_1, \ldots, (\tau_n)\, v_n\texttt{>}}.$$

An analogous rule defines the semantics of casting for record types.

To ensure safety for implicit subsumption we must ensure that projection is well-defined on a subtype value. In the case of tuples this means that the operation of accessing the ith component from a tuple must be insensitive to the size of the tuple, beyond the basic requirement that it have size at least i. This can be expressed schematically as follows:

$$\texttt{<}v_1, \ldots, v_i, \ldots\texttt{>}.i \longmapsto v_i.$$

The ellision indicates that fields beyond the *i*th are not relevant to the operation. Similarly, for records we postulate that selection of the *l*th field is insensitive to the presence of any other fields:

$$\{l{:}v,\ldots\}.l \longmapsto v.$$

The ellision expresses the independence of field selection from any "extra" fields.

36.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter 29. What should be the variance rule for reference types? Suppose that r has type $\sigma\,\texttt{ref}$. We can do one of two things with r:

1. Retrieve its contents as a value of type σ.
2. Replace its contents with a value of type σ.

If σ <: τ, then retrieving the contents of r yields a value of type τ, by subsumption. This suggests that references are covariant:

$$\frac{\sigma \mathrel{\texttt{<:}} \tau}{\sigma\,\texttt{ref} \overset{?}{\texttt{<:}} \tau\,\texttt{ref}.}$$

On the other hand, if τ <: σ, then we may store a value of type τ into r. This suggests that references are contravariant:

$$\frac{\tau \mathrel{\texttt{<:}} \sigma}{\sigma\,\texttt{ref} \overset{?}{\texttt{<:}} \tau\,\texttt{ref}.}$$

Given that we may perform either operation on a reference cell, we must insist that reference types are *invariant*:

$$\frac{\sigma \mathrel{\texttt{<:}} \tau \quad \tau \mathrel{\texttt{<:}} \sigma}{\sigma\,\texttt{ref} \mathrel{\texttt{<:}} \tau\,\texttt{ref}\,.}$$

The premise of the rule is often strengthened to the requirement that σ and τ be equal:

$$\frac{\sigma = \tau}{\sigma\,\texttt{ref} \mathrel{\texttt{<:}} \tau\,\texttt{ref}}$$

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, *immutable* sequences may be safely taken to be covariant, but *mutable* sequences (arrays) must be taken to be invariant, lest safety be compromised.

Chapter 37

Implementing Subtyping

37.1 Type Checking With Subtyping

Type checking for $\mathsf{MinML}_{<:}$, in either variant, clearly requires an algorithm for deciding subtyping: given σ and τ, determine whether or not $\sigma <: \tau$. The difficulty of deciding type checking is dependent on the specific rules under consideration. In this section we will discuss type checking for $\mathsf{MinML}_{<:}$, under the assumption that we can check the subtype relation.

Consider first the explicit variant of $\mathsf{MinML}_{<:}$. Since the typing rules are syntax directed, we can proceed as for MinML, with one additional case to consider. To check whether $(\sigma)\,e$ has type τ, we must check two things:

1. Whether e has type σ.

2. Whether $\sigma <: \tau$.

The former is handled by a recursive call to the type checker, the latter by a call to the subtype checker, which we assume given.

This discussion glosses over an important point. Even in pure MinML it is not possible to determine directly whether or not $\Gamma \vdash e : \tau$. For suppose that e is an application $e_1(e_2)$. To check whether $\Gamma \vdash e : \tau$, we must find the domain type of the function, e_1, against which we must check the type of the argument, e_2. To do this we define a *type synthesis* function that determines the unique (if it exists) type τ of an expression e in a context Γ, written $\Gamma \vdash e \Rightarrow \tau$. To check whether e has type τ, we synthesize the unique type for e and check that it is τ.

This methodology applies directly to $\mathsf{MinML}^{e}_{<:}$ by using the following rule to synthesize a type for a cast:

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e \Rightarrow \tau}$$

Extending this method to $\mathsf{MinML}^{i}_{<:}$ is a bit harder, because expressions no longer have unique types! The rule of subsumption allows us to weaken the type of an expression at will, yielding many different types for the same expression. A standard approach is define a type synthesis function that determines the *principal* type, rather than the *unique* type, of an expression in a given context. The principal type of an expression e in context Γ is the *least* type (in the subtyping pre-order) for e in Γ. Not every subtype system admits principal types. But we usually strive to ensure that this is the case whenever possible in order to employ this simple type checking method.

The rules synthesizing principal types for expressions of $\mathsf{MinML}^{i}_{<:}$ are as follows:

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash n \Rightarrow \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} \Rightarrow \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} \Rightarrow \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 <: \tau_1 \quad \cdots \quad \Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n <: \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau}$$

where o is an n-ary primitive operation with arguments of type $\tau_1, \ldots, \tau_n$, and result type τ. We use subsumption to ensure that the argument types are subtypes of the required types.

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma <: \texttt{bool} \quad \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 <: \tau \quad \Gamma \vdash e_2 \Rightarrow \tau_2 \quad \tau_2 <: \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Rightarrow \tau}$$

We use subsumption to ensure that the type of the test is a subtype of `bool`. Moreover, we rely on explicit specification of the type of the two clauses of the conditional.[1]

$$\frac{\Gamma[f{:}\texttt{arrow}(\tau_1, \tau_2)][x{:}\tau_1] \vdash e \Rightarrow \tau_2}{\Gamma \vdash \texttt{fun}\, f\ (x : \tau_1) : \tau_2\ \texttt{is}\, e \Rightarrow \texttt{arrow}(\tau_1, \tau_2)}$$

[1]This may be avoided by requiring that the subtype relation have least upper bounds "whenever necessary"; we will not pursue this topic here.

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 \Rightarrow \sigma_2 \quad \sigma_2 \texttt{<:} \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau}$$

We use subsumption to check that the argument type is a subtype of the domain type of the function.

Theorem 37.1

1. *If $\Gamma \vdash e \Rightarrow \sigma$, then $\Gamma \vdash e : \sigma$.*

2. *If $\Gamma \vdash e : \tau$, then there exists σ such that $\Gamma \vdash e \Rightarrow \sigma$ and $\sigma \texttt{<:} \tau$.*

Proof:

1. By a straightforward induction on the definition of the type synthesis relation.

2. By induction on the typing relation.

■

37.2 Implementation of Subtyping

37.2.1 Coercions

The dynamic semantics of subtyping sketched above suffices to ensure type safety, but is in most cases rather impractical. Specifically,

- Arithmetic subtyping relies on run-time type recognition and conversion.

- Tuple projection depends on the insensitivity of projection to the existence of components after the point of projection.

- Record field selection depends on being able to identify the lth field in a record with numerous fields.

- Function subtyping may require run-time checks and conversions to match up calling conventions.

These costs are significant. Fortunately they can be avoided by taking a slightly different approach to the implementation of subtyping. Consider, for example, arithmetic subtyping. In order for a mixed-mode expression such as `+.(3,4)` to be well-formed, we must use subsumption to weaken the types of `3` and `4` from `int` to `float`. This means that type conversions are required exactly insofar as subsumption is used during type checking — a use of subsumption corresponds to a type conversion.

Since the subsumption rule is part of the static semantics, we can insert the appropriate conversions during type checking, and omit entirely the need to check for mixed-mode expressions during execution. This is called a *coercion interpretation* of subsumption. It is expressed formally by augmenting each subtype relation $\sigma \mathrel{\texttt{<:}} \tau$ with a function value v of type $\texttt{arrow}(\sigma,\tau)$ (in pure MinML) that *coerces* values of type σ to values of type τ. The augmented subtype relation is written $\sigma \mathrel{\texttt{<:}} \tau \rightsquigarrow v$.

Here are the rules for arithmetic subtyping augmented with coercions:

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau \rightsquigarrow \texttt{id}_\tau} \qquad \frac{\rho \mathrel{\texttt{<:}} \sigma \rightsquigarrow v \quad \sigma \mathrel{\texttt{<:}} \tau \rightsquigarrow v'}{\rho \mathrel{\texttt{<:}} \tau \rightsquigarrow v\,\texttt{;}\,v'}$$

$$\frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{float} \rightsquigarrow \texttt{to\_float}} \qquad \frac{\tau_1 \mathrel{\texttt{<:}} \sigma_1 \rightsquigarrow v_1 \quad \sigma_2 \mathrel{\texttt{<:}} \tau_2 \rightsquigarrow v_2}{\texttt{arrow}(\sigma_1,\sigma_2) \mathrel{\texttt{<:}} \texttt{arrow}(\tau_1,\tau_2) \rightsquigarrow \texttt{arrow}(v_1,v_2)}$$

These rules make use of the following auxiliary functions:

1. Primitive conversion: `to_float`.
2. Identity: $\texttt{id}_\tau = \texttt{fn}\,x\!:\!\tau\,\texttt{in}\,x$.
3. Composition: $v\,\texttt{;}\,v' = \texttt{fn}\,x\!:\!\tau\,\texttt{in}\,v'\texttt{(}v\texttt{(}x\texttt{))}$.
4. Functions: $\texttt{arrow}(v_1,v_2) =$
   $\texttt{fn}\,f\!:\!\texttt{arrow}(\sigma_1,\sigma_2)\,\texttt{in}\,\texttt{fn}\,x\!:\!\tau_1\,\texttt{in}\,v_2\texttt{(}f\texttt{(}v_1\texttt{(}x\texttt{)))}$.

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

Theorem 37.2

1. *If $\sigma \mathrel{\texttt{<:}} \tau \rightsquigarrow v$, then $\vdash^- v : \texttt{arrow}(\sigma,\tau)$.*
2. *If $\sigma \mathrel{\texttt{<:}} \tau \rightsquigarrow v_1$ and $\sigma \mathrel{\texttt{<:}} \tau \rightsquigarrow v_2$, then $\vdash^- v_1 \cong v_2 : \texttt{arrow}(\sigma,\tau)$.*

Proof:

1. By a simple induction on the rules defining the augmented subtyping relation.

2. Follows from these equations:

   (a) Composition is associative with `id` as left- and right-unit element.

   (b) $\texttt{arrow}(\texttt{id},\texttt{id}) \cong \texttt{id}$.

   (c) $(\texttt{arrow}(v_1,v_2));(\texttt{arrow}(v_1',v_2')) \cong \texttt{arrow}((v_1';v_1),(v_2;v_2'))$.

■

The type checking relation is augmented with a translation from $\mathsf{MinML}^i_{<:}$ to pure MinML that eliminates uses of subsumption by introducing coercions:

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow e' \quad \sigma <: \tau \rightsquigarrow v}{\Gamma \vdash e : \tau \rightsquigarrow v(e')}$$

The remaining rules simply commute with the translation. For example, the rule for function application becomes

$$\frac{\Gamma \vdash e_1 : \texttt{arrow}(\tau_2,\tau) \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow e_1'(e_2')}$$

Theorem 37.3

1. *If $\Gamma \vdash e : \tau \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau$ in pure MinML.*

2. *If $\Gamma \vdash e : \tau \rightsquigarrow e_1$ and $\Gamma \vdash e : \tau \rightsquigarrow e_2$, then $\Gamma \vdash e_1 \cong e_2 : \tau$ in pure MinML.*

3. *If $e : \texttt{int} \rightsquigarrow e'$ is a complete program, then $e \Downarrow n$ iff $e' \Downarrow n$.*

The coercion interpretation also applies to record subtyping. Here the problem is how to implement field selection efficiently in the presence of subsumption. Observe that in the absence of subtyping the type of a record value reveals the *exact* set of fields of a record (and their types). We can therefore implement selection efficiently by ordering the fields in some canonical manner (say, alphabetically), and compiling field selection as a projection from an offset determined statically by the field's label.

In the presence of record subtyping this simple technique breaks down, because the type no longer reveals the fields of a record, not their types. For example, every expression of record type has the record type {} with no fields whatsoever! This makes it difficult to predict statically the position of the field labelled l in a record. However, we may restore this important property by using coercions. Whenever the type of a record is weakened using subsumption, insert a function that creates a new record that exactly matches the supertype. Then use the efficient record field selection method just described.

Here, then, are the augmented rules for width and depth subtyping for records:

$$\frac{m > n}{\{l_1 : \tau_1 , \dots , l_m : \tau_m\} \texttt{ <: } \{l_1 : \tau_1 , \dots , l_n : \tau_n\} \rightsquigarrow \texttt{drop}_{m,n,l,\tau}}$$

$$\frac{\sigma_1 \texttt{ <: } \tau_1 \rightsquigarrow v_1 \quad \dots \quad \sigma_n \texttt{ <: } \tau_n \rightsquigarrow v_n}{\{l_1 : \sigma_1 , \dots , l_n : \sigma_n\} \texttt{ <: } \{l_1 : \tau_1 , \dots , l_n : \tau_n\} \rightsquigarrow \texttt{copy}_{n,l,\sigma,v}}$$

These rules make use of the following coercion functions:

$$\begin{aligned}\texttt{drop}_{m,n,l,\sigma} &= \\ &\texttt{fn}\, x : \{l_1 : \sigma_1 , \dots , l_m : \sigma_m\} \,\texttt{in}\, \{l_1 : x . l_1 , \dots , l_n : x . l_n\}\end{aligned}$$

$$\begin{aligned}\texttt{copy}_{n,l,\sigma,v} &= \\ \texttt{fn}\, x : \{l_1 : \sigma_1 , \dots , l_n : \sigma_n\} \,\texttt{in}\, &\{l_1 : v_1 (x . l_1) , \dots , l_n : v_n (x . l_n)\}\end{aligned}$$

In essence this approach represents a trade-off between the cost of subsumption and the cost of field selection. By creating a new record whenever subsumption is used, we make field selection cheap. On the other hand, we can make subsumption free, provided that we are willing to pay the cost of a search whenever a field is selected from a record.

But what if record fields are mutable? This approach to coercion is out of the question, because of *aliasing*. Suppose that a mutable record value v is bound to two variables, x and y. If coercion is applied to the binding of x, creating a new record, then future changes to y will not affect the new record, nor vice versa. In other words, uses of coercion changes the semantics of a program, which is unreasonable.

One widely-used approach is to increase slightly the cost of field selection (by a constant factor) by separating the "view" of a record from its "contents". The view determines the fields and their types that are present for each use of a record, whereas the contents is shared among all uses. In essence we represent the record type $\{l_1 : \tau_1 , \ldots , l_n : \tau_n\}$ by the product type

$$\{l_1 : \texttt{int}, \ldots , l_n : \texttt{int}\}{*}(\tau\;\texttt{array}).$$

The field selection $l\,.\,e$ becomes a two-stage process:

$$\texttt{snd(}e\texttt{)[fst(}e\texttt{).}l\texttt{]}$$

Finally, coercions copy the view, without modifying the contents. If $\sigma = \{l_1 : \sigma_1 , \ldots , l_n : \sigma_n\}$ and $\tau = \{l_1 : \texttt{int}, \ldots , l_n : \texttt{int}\}$, then

$$\texttt{drop}_{m,n,l,\sigma} = \texttt{fn}\,x\,\texttt{in (drop}_{m,n,l,\tau}\texttt{(fst(}x\texttt{)),snd(}x\texttt{)).}$$

Part XIV

Inheritance

Chapter 38

Featherweight Java

We will consider a tiny subset of the Java language, called *Featherweight Java*, or FJ, that models subtyping and inheritance in Java. We will then discuss design alternatives in the context of FJ. For example, in FJ, as in Java, the subtype relation is tightly coupled to the subclass relation. Is this necessary? Is it desirable? We will also use FJ as a framework for discussing other aspects of Java, including interfaces, privacy, and arrays.

38.1 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

$$
\begin{array}{llcl}
\textit{Classes} & C & ::= & \texttt{class}\, c\, \texttt{extends}\, c\, \{\underline{c}\, \underline{f}\,;\, k\, \underline{d}\} \\
\textit{Constructors} & k & ::= & c\,\texttt{(}\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{super(}\underline{x}\texttt{);}\,\texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\} \\
\textit{Methods} & d & ::= & c\, m\texttt{(}\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{return}\, e\texttt{;}\} \\
\textit{Types} & \tau & ::= & c \\
\textit{Expressions} & e & ::= & x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\underline{e}\texttt{)} \mid \texttt{new}\, c\texttt{(}\underline{e}\texttt{)} \mid \texttt{(}c\texttt{)}\, e
\end{array}
$$

The variable f ranges over a set of *field names*, c over a set of *class names*, m over a set of *method names*, and x over a set of *variable names*. We assume that these sets are countably infinite and pairwise disjoint. We assume that there is a distinguished class name, `Object`, standing for the root of the class hierarchy. It's role will become clear below. We assume that there is a distinguished variable `this` that cannot otherwise be declared in a program.

As a notational convenience we use "underbarring" to stand for sequences of phrases. For example, $\underline{d}$ stands for a sequence of d's, whose individual elements we designate $d_1, \ldots, d_k$, where k is the length of the sequence. We write $\underline{c}\,\underline{f}$ for the sequence $c_1\,f_1, \ldots, c_k\,f_k$, where k is the length of the sequences $\underline{c}$ and $\underline{f}$. Similar conventions govern the other uses of sequence notation.

The class expression

$$\texttt{class}\,c\,\texttt{extends}\,c'\,\{\underline{c}\,\underline{f}\,;\,k\,\underline{d}\}$$

declares the class c to be a subclass of the class c'. The subclass has additional fields $\underline{c}\,\underline{f}$, single constructor k, and method suite $\underline{d}$. The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

$$c\texttt{(}\underline{c'}\,\underline{x'}\texttt{,}\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{super(}\underline{x'}\texttt{)}\,;\,\texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\}$$

declares the constructor for class c with arguments $\underline{c'}\,\underline{x'}, \underline{c}\,\underline{x}$, corresponding to the fields of the superclass followed by those of the subclass. The variables $\underline{x'}$ and $\underline{x}$ are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments $\underline{x'}$ and of the subclass with arguments $\underline{x}$.

The method expression

$$c\,m\texttt{(}\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{return}\,e\texttt{;}\}$$

declares a method m yielding a value of class c, with arguments $\underline{x}$ of class $\underline{c}$ and body returning the value of the expression e. The variables $\underline{x}$ and `this` are bound in e.

The set of types is, for the time being, limited to the set of class names. That is, the only types are those declared by a class. In Java there are more types than just these, including the primitive types `integer` and `boolean` and the array types.

The set of expressions is the minimal "interesting" set sufficient to illustrate subtyping and inheritance. The expression $e\,\texttt{.}\,f$ selects the contents of field f from instance e. The expression $e\,\texttt{.}\,m\texttt{(}\underline{e}\texttt{)}$ invokes the method m of instance e with arguments $\underline{e}$. The expression $\texttt{new}\,c\texttt{(}\underline{e}\texttt{)}$ creates a new instance of class c, passing arguments $\underline{e}$ to the constructor for c. The expression $\texttt{(}c\texttt{)}\,e$ casts the value of e to class c.

```
class Pt extends Object {
  int x;
  int y;
  Pt (int x, int y) {
     super(); this.x = x; this.y = y;
  }
  int getx () { return this.x; }
  int gety () { return this.y; }
}
class CPt extends Pt {
  color c;
  CPt (int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

Figure 38.1: A Sample FJ Program

The methods of a class may invoke one another by sending messages to `this`, standing for the instance itself. We may think of `this` as a bound variable of the instance, but we will arrange things so that renaming of `this` is never necessary to avoid conflicts.

A *class table* T is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A *program* is a pair (T, e) consisting of a class table T and an expression e. We generally suppress explicit mention of the class table, and consider programs to be expressions.

A small example of FJ code is given in Figure 38.1. In this example we assume given a class `Object` of all objects and make use of types `int` and `color` that are not, formally, part of FJ.

38.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

$\tau <: \tau'$	*subtyping*
$\Gamma \vdash e : \tau$	*expression typing*
$d\ \mathsf{ok\ in}\ c$	*well-formed method*
$C\ \mathsf{ok}$	*well-formed class*
$T\ \mathsf{ok}$	*well-formed class table*
$\mathsf{fields}(c) = \underline{c}\,\underline{f}$	*field lookup*
$\mathsf{type}(m, c) = \underline{c} \rightarrow c$	*method type*

The rules defining the static semantics follow.
Every variable must be declared:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \tag{38.1}$$

The types of fields are defined in the class table.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \mathsf{fields}(c_0) = \underline{c}\,\underline{f}}{\Gamma \vdash e_0\,.\,f_i : c_i} \tag{38.2}$$

The argument and result types of methods are defined in the class table.

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \\ \mathsf{type}(m, c_0) = \underline{c'} \rightarrow c \quad \underline{c} <: \underline{c'}\end{array}}{\Gamma \vdash e_0\,.\,m(\underline{e}) : c} \tag{38.3}$$

Instantiation must provide values for all instance variables as arguments to the constructor.

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \quad \underline{c} <: \underline{c'} \quad \mathsf{fields}(c) = \underline{c'}\,\underline{f}}{\Gamma \vdash \mathtt{new}\,c(\underline{e}) : c} \tag{38.4}$$

All casts are statically valid, but must be checked at run-time.

$$\frac{\Gamma \vdash e_0 : d}{\Gamma \vdash (c)\, e_0 : c} \tag{38.5}$$

The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

$$\frac{}{\tau <: \tau} \tag{38.6}$$

$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \tag{38.7}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\ldots\, ;\, \ldots\}}{c <: c'} \tag{38.8}$$

A well-formed class has zero or more fields, a constructor that initializes the superclass and the subclass fields, and zero or more methods. To account for method override, the typing rules for each method are relative to the class in which it is defined.

$$\frac{\begin{array}{c} k = c\,(\underline{c'}\,\underline{x'}, \underline{c}\,\underline{x})\,\{\texttt{super}\,(\underline{x'})\,;\,\texttt{this}\,.\underline{f}\texttt{=}\underline{x}\,;\} \\ \mathsf{fields}(c') = \underline{c'}\,\underline{f'} \quad \underline{d}\;\mathsf{ok\;in}\;c \end{array}}{\texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{c}\,\underline{f}\,;\, k\,\underline{d}\}\;\mathsf{ok}} \tag{38.9}$$

Method overriding takes account of the type of the method in the superclass. The subclass method must have the same argument types and result type as in the superclass.

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\ldots\, ;\, \ldots\} \\ \mathsf{type}(m, c') = \underline{c} \to c_0 \quad \underline{x}{:}\underline{c}, \texttt{this}{:}c \vdash e_0 : c_0 \end{array}}{c_0\, m\,(\underline{c}\,\underline{x})\,\{\texttt{return}\, e_0\,;\}\;\mathsf{ok\;in}\;c} \tag{38.10}$$

A class table is well-formed iff all of its classes are well-formed:

$$\frac{\forall c \in \mathrm{dom}(T)\;\; T(c)\;\mathsf{ok}}{T\;\mathsf{ok}} \tag{38.11}$$

Note that well-formedness of a class is relative to the class table!

A program is well-formed iff its method table is well-formed and the expression is well-formed:

$$\frac{T\ \mathsf{ok} \quad \emptyset \vdash e : \tau}{(T, e)\ \mathsf{ok}} \tag{38.12}$$

The auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

$$\frac{}{\mathsf{fields}(\mathtt{Object}) = \bullet} \tag{38.13}$$

$$\frac{T(c) = \mathtt{class}\, c\, \mathtt{extends}\, c'\, \{\underline{c}\, \underline{f};\, \ldots\} \quad \mathsf{fields}(c') = \underline{c'}\, \underline{f'}}{\mathsf{fields}(c) = \underline{c'}\, \underline{f'}, \underline{c}\, \underline{f}} \tag{38.14}$$

The type of a method is determined by the following rules:

$$\frac{\begin{array}{c} T(c) = \mathtt{class}\, c\, \mathtt{extends}\, c'\, \{\ldots;\, \ldots\, \underline{d}\} \\ d_i = c_i\, m(\underline{c_i}\, \underline{x})\, \{\mathtt{return}\, e;\} \end{array}}{\mathsf{type}(m_i, c) = \underline{c_i} \rightarrow c_i} \tag{38.15}$$

$$\frac{\begin{array}{c} T(c) = \mathtt{class}\, c\, \mathtt{extends}\, c'\, \{\ldots;\, \ldots\, \underline{d}\} \\ m \notin \underline{d} \quad \mathsf{type}(m, c') = \underline{c_i} \rightarrow c_i \end{array}}{\mathsf{type}(m, c) = \underline{c_i} \rightarrow c_i} \tag{38.16}$$

38.3 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. The transition relation is indexed by a class table T, which governs the semantics of casting and dynamic dispatch (which see below). In the rules below we omit explicit mention of the class table for the sake of brevity.

An instance of a class has the form $\mathtt{new}\, c(\underline{e})$, where each e_i is a value.

$$\frac{\underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e})\ \mathsf{value}} \tag{38.17}$$

Since we arrange that there be a one-to-one correspondence between instance variables and constructor arguments, an instance expression of this form carries all of the information required to determine the values of the fields of the instance. This makes clear that an instance is essentially just a labelled collection of fields. Each instance is labelled with its class, which is used to guide method dispatch.

Field selection retrieves the value of the named field from either the subclass or its superclass, as appropriate.

$$\frac{\mathsf{fields}(c) = \underline{c'}\,\underline{f'}, \underline{c}\,\underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\,c\,\mathtt{(}\underline{e'},\underline{e}\mathtt{)}\,\mathtt{.}\,f'_i \longmapsto e'_i} \tag{38.18}$$

$$\frac{\mathsf{fields}(c) = \underline{c'}\,\underline{f'}, \underline{c}\,\underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\,c\,\mathtt{(}\underline{e'},\underline{e}\mathtt{)}\,\mathtt{.}\,f_i \longmapsto e_i} \tag{38.19}$$

Message send replaces `this` by the instance itself, and replaces the method parameters by their values.

$$\frac{\mathsf{body}(m, c) = \underline{x} \to e_0 \quad \underline{e}\ \mathsf{value} \quad \underline{e'}\ \mathsf{value}}{\mathtt{new}\,c\,\mathtt{(}\underline{e}\mathtt{)}\,\mathtt{.}\,m\,\mathtt{(}\underline{e'}\mathtt{)} \longmapsto [\underline{x}\leftarrow\underline{e'}][\mathtt{this}\leftarrow\mathtt{new}\,c\,\mathtt{(}\underline{e}\mathtt{)}]e_0} \tag{38.20}$$

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

$$\frac{c \mathrel{\mathtt{<:}} c' \quad \underline{e}\ \mathsf{value}}{\mathtt{(}c'\mathtt{)}\,\mathtt{new}\,c\,\mathtt{(}\underline{e}\mathtt{)} \longmapsto \mathtt{new}\,c\,\mathtt{(}\underline{e}\mathtt{)}} \tag{38.21}$$

These rules determine the order of evaluation:

$$\frac{e_0 \longmapsto e'_0}{e_0\,\mathtt{.}\,f \longmapsto e'_0\,\mathtt{.}\,f} \tag{38.22}$$

$$\frac{e_0 \longmapsto e'_0}{e_0\,\mathtt{.}\,m\,\mathtt{(}\underline{e}\mathtt{)} \longmapsto e'_0\,\mathtt{.}\,m\,\mathtt{(}\underline{e}\mathtt{)}} \tag{38.23}$$

$$\frac{e_0\ \mathsf{value} \quad \underline{e} \longmapsto \underline{e'}}{e_0\,\mathtt{.}\,m\,\mathtt{(}\underline{e}\mathtt{)} \longmapsto e_0\,\mathtt{.}\,m\,\mathtt{(}\underline{e'}\mathtt{)}} \tag{38.24}$$

$$\frac{\underline{e} \longmapsto \underline{e'}}{\texttt{new}\,c\,\texttt{(}\underline{e}\texttt{)} \longmapsto \texttt{new}\,c\,\texttt{(}\underline{e'}\texttt{)}} \tag{38.25}$$

$$\frac{e_0 \longmapsto e_0'}{\texttt{(}c\texttt{)}\,e_0 \longmapsto \texttt{(}c\texttt{)}\,e_0'} \tag{38.26}$$

Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

$$\frac{\begin{array}{c} T(c) = \texttt{class}\,c\,\texttt{extends}\,c'\,\{\ldots\,;\ \ldots\,\underline{d}\} \\ d_i = c_i\,m_i\texttt{(}\underline{c_i}\,\underline{x}\texttt{)}\ \{\texttt{return}\,e\texttt{;}\} \end{array}}{\mathsf{body}(m_i, c) = \underline{x} \rightarrow e} \tag{38.27}$$

$$\frac{\begin{array}{c} T(c) = \texttt{class}\,c\,\texttt{extends}\,c'\,\{\ldots\,;\ \ldots\,\underline{d}\} \\ m \notin \underline{d} \quad \mathsf{body}(m, c') = \underline{x} \rightarrow e \end{array}}{\mathsf{body}(m, c) = \underline{x} \rightarrow e} \tag{38.28}$$

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (*i.e.*, consists only of values).

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_{i-1}\ \mathsf{value} \quad e_i \longmapsto e_i'}{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \longmapsto e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots, e_n} \tag{38.29}$$

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_n\ \mathsf{value}}{\underline{e}\ \mathsf{value}} \tag{38.30}$$

This completes the dynamic semantics of FJ.

38.4 Type Safety

The safety of FJ is stated in the usual manner by the Preservation and Progress Theorems.

Since the dynamic semantics of casts preserves the "true" type of an instance, the type of an expression may become "smaller" in the subtype ordering during execution.

Theorem 38.1 (Preservation)
Assume that T is a well-formed class table. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau'$ for some τ' such that $\tau' <: \tau$.

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.

Theorem 38.2 (Progress)
Assume that T is a well-formed class table. If $e : \tau$ then either

1. *v value, or*
2. *e contains an instruction of the form $(c)\,\mathtt{new}\,c'(e_0)$ with e_0 value and $c' \not<: c$, or*
3. *there exists e' such that $e \longmapsto e'$.*

It follows that if no casts occur in the source program, then the second case cannot arise. This can be sharpened somewhat to admit source-level casts for which it is known statically that the type of casted expression is a subtype of the target of the cast. However, we cannot predict, in general, statically whether a given cast will succeed or fail dynamically.

Lemma 38.3 (Canonical Forms)
If $e : c$ and e value, then e has the form $\mathtt{new}\,c'(e_0)$ with e_0 value and $c' <: c$.

38.5 Acknowledgement

This chapter is based on "Featherweight Java: A Minimal Core Calculus for Java and GJ" by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.

Chapter 39

Inheritance and Subtyping in Java

In this note we discuss the closely-related, but conceptually distinct, notions of *inheritance*, or *subclassing*, and *subtyping* as exemplified in the Java language. Inheritance is a mechanism for supporting *code re-use* through incremental extension and modification. Subtyping is a mechanism for expressing *behavioral relationships* between types that allow values of a subtype to be provided whenever a value of a supertype is required.

In Java inheritance relationships give rise to subtype relationships, but not every subtype relationship arises via inheritance. Moreover, there are languages (including some extensions of Java) for which subclasses do not give rise to subtypes, and there are languages with no classes at all, but with a rich notion of subtyping. For these reasons it is best to keep a clear distinction between subclassing and subtyping.

39.1 Inheritance Mechanisms in Java

39.1.1 Classes and Instances

The fundamental unit of inheritance in Java is the *class*. A class consists of a collection of *fields* and a collection of *methods*. Fields are assignable variables; methods are procedures acting on these variables. Fields and methods can be either *static* (per-class) or *dynamic* (per-instance).[1] Static fields are per-class data. Static methods are just ordinary functions acting on static fields.

[1]Fields and methods are assumed dynamic unless explicitly declared to be static.

Classes give rise to *instances*, or *objects*, that consist of the dynamic methods of the class together with fresh copies (or instances) of its dynamic fields. Instances of classes are created by a *constructor*, whose role is to allocate and initialize fresh copies of the dynamic fields (which are also known as *instance variables*). Constructors have the same name as their class, and are invoked by writing `new` $C(e_1, \ldots, e_n)$, where C is a class and $e_1, \ldots, e_n$ are arguments to the constructor.[2] Static methods have access only to the static fields (and methods) of its class; dynamic methods have access to both the static and dynamic fields and methods of the class.

The components of a class have a designated *visibility* attribute, either `public`, `private`, or `protected`. The public components are those that are accessible by all clients of the class. Public static components are accessible to any client with access to the class. Public dynamic components are visible to any client of any instance of the class. Protected components are "semi-private; we'll have more to say about protected components later.

The components of a class also have a *finality* attribute. Final fields are not assignable — they are read-only attributes of the class or instance. Actually, final dynamic fields can be assigned exactly once, by a constructor of the class, to initialize their values. Final methods are of interest in connection with inheritance, to which we'll return below.

The components of a class have *types*. The type of a field is the type of its binding as a (possibly assignable) variable. The type of a method specifies the types of its arguments (if any) and the type of its results. The type of a constructor specifies the types of its arguments (if any); its "result type" is the instance type of the class itself, and may not be specified explicitly. (We will say more about the type structure of Java below.)

The public static fields and methods of a class C are accessed using "dot notation". If f is a static field of C, a client may refer to it by writing $C.f$. Similarly, if m is a static method of C, a client may invoke it by writing $C.m(e_1, \ldots, e_n)$, where $e_1, \ldots, e_n$ are the argument expressions of the method. The expected type checking rules govern access to fields and invocations of methods.

The public dynamic fields and methods of an instance c of a class C are similarly accessed using "dot notation", *albeit* from the instance, rather than the class. That is, if f is a public dynamic field of C, then $c.f$ refers

[2]Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading here.

to the f field of the instance c. Since distinct instances have distinct fields, there is no essential connection between c.f and c'.f when c and c' are distinct instances of class C. If m is a public dynamic method of C, then c.m(e_1,...,e_n) invokes the method m of the instance c with the specified arguments. This is sometimes called *sending a message m to instance c with arguments $e_1, \ldots, e_n$*.

Within a dynamic method one may refer to the dynamic fields and methods of the class via the pseudo-variable `this`, which is bound to the instance itself. The methods of an instance may call one another (or themselves) by sending a message to `this`. Although Java defines conventions whereby explicit reference to `this` may be omitted, it is useful to eschew these conveniences and always use `this` to refer to the components of an instance from within code for that instance. We may think of `this` as an implicit argument to all methods that allows the method to access to object itself.

39.1.2 Subclasses

A class may be defined by *inheriting* the visible fields and methods of another class. The new class is said to be a *subclass* of the old class, the *superclass*. Consequently, inheritance is sometimes known as *subclassing*. Java supports *single inheritance* — every class has at most one superclass. That is, one can only inherit from a single class; one cannot combine two classes by inheritance to form a third. In Java the subclass is said to `extend` the superclass.

There are two forms of inheritance available in Java:

1. *Enrichment*. The subclass enriches the superclass by providing additional fields and methods not present in the superclass.

2. *Overriding*. The subclass may re-define a method in the superclass by giving it a new implementation in the subclass.

Enrichment is a relatively innocuous aspect of inheritance. The true power of inheritance lies in the ability to override methods.

Overriding, which is also known as *method specialization*, is used to "specialize" the implementation of a superclass method to suit the needs of the subclass. This is particularly important when the other methods of the class invoke the overridden method by sending a message to `this`. If a method m is overridden in a subclass D of a class C, then all methods of D that invoke m via `this` will refer to the "new" version of m defined by the override. The "old" version can still be accessed explicitly from the subclass by referring to `super.`m. The keyword `super` is a pseudo-variable that may be used to refer to the overridden methods.

Inheritance can be controlled using visibility constraints. A sub-class D of a class C automatically inherits the private fields and methods of C without the possibility of overriding, or otherwise accessing, them. The public fields and methods of the superclass are accessible to the subclass without restriction, and retain their `public` attribute in the subclass, unless overridden. A `protected` component is "semi-private" — accessible to the subclass, but not otherwise publically visible.[3]

Inheritance can also be limited using finality constraints. If a method is declared `final`, it may not be overridden in any subclass — it must be inherited as-is, without further modification. However, if a final method invokes, via `this`, a non-final method, then the behavior of the final method can still be changed by the sub-class by overriding the non-final method. By declaring an entire class to be final, no class can inherit from it. This serves to ensure that any instance of this class invokes the code from this class, and not from any subclass of it.

Instantiation of a subclass of a class proceeds in three phases:

1. The instance variables of the subclass, which include those of the superclass, are allocated.

2. The constructor of the superclass is invoked to initialize the superclass's instance variables.

3. The constructor of the subclass is invoked to initialize the subclass's instance variables.

The superclass constructor can be explicitly invoked by a subclass constructor by writing `super(`e_1`,...,`e_n`)`, but *only* as the very first statement of the subclass's constructor. This ensures proper initialization order, and avoids certain anomalies and insecurities that arise if this restriction is relaxed.

[3]Actually, Java assigns `protected` components "package scope", but since we are not discussing packages here, we will ignore this issue.

39.1.3 Abstract Classes and Interfaces

An *abstract class* is a class in which one or more methods are declared, but left unimplemented. Abstract methods may be invoked by the other methods of an abstract class by sending a message to `this`, but since their implementation is not provided, abstract classes do not themselves have instances. Instead the obligation is imposed on a subclass of the abstract class to provide implementations of the abstract methods to obtain a *concrete* class, which does have instances. Abstract classes are useful for setting up "code templates" that are instantiated by inheritance. The abstract class becomes the locus of code sharing for all concretions of that class, which inherit the shared code and provide the missing non-shared code.

Taking this idea to the extreme, an *interface* is a "fully abstract" class, which is to say that

- All its fields are `public static final` (*i.e.*, they are constants).
- All its methods are `abstract public`; they must be implemented by a subclass.

Since interfaces are a special form of abstract class, they have no instances.

The utility of interfaces stems from their role in `implements` declarations. As we mentioned above, a class may be declared to extend a *single* class to inherit from it.[4] A class may also be declared to `implement` *one or more* interfaces, meaning that the class provides the public methods of the interface, with their specified types. Since interfaces are special kinds of classes, Java is sometimes said to provide *multiple inheritance of interfaces*, but only *single inheritance of implementation*. For similar reasons an interface may be declared to extend multiple interfaces, provided that the result types of their common methods coincide.

The purpose of declaring an interface for a class is to support writing generic code that works with *any* instance providing the methods specified in the interface, *without* requiring that instance to arise from any particular position in the inheritance hierarchy. For example, we may have two unrelated classes in the class hierarchy providing a method m. If both classes are declared to implement an interface that mentions m, then code programmed against this interface will work for an instance of *either* class.

[4]Classes that do not specify a superclass implicitly extend the class `Object` of all objects.

The literature on Java emphasizes that interfaces are *descriptive* of behavior (to the extend that types alone allow), whereas classes are *prescriptive* of implementation. While this is surely a noble purpose, it is curious that interfaces are *classes* in Java, rather than *types*. In particular interfaces are unable to specify the public fields of an instance by simply stating their types, which would be natural were interfaces a form of type. Instead fields in interfaces are forced to be constants (public, static, final), precluding their use for describing the public instance variables of an object.

39.2 Subtyping in Java

The Java type system consists of the following types:

1. *Base types*, including `int`, `float`, `void`, and `boolean`.

2. *Class types* C, which classify the instances of a class C.

3. *Array types* of the form τ `[]`, where τ is a type, representing mutable arrays of values of type τ.

The basic types behave essentially as one would expect, based on previous experience with languages such as C and C++. Unlike C or C++ Java has true array types, with operations for creating and initializing an array and for accessing and assigning elements of an array. All array operations are safe in the sense that any attempt to exceed the bounds of the array results in a checked error at run-time.

Every class, whether abstract or concrete, including interfaces, has associated with it the type of its instances, called (oddly enough) the *instance type* of the class. Java blurs the distinction between the class as a program structure and the instance type determined by the class — class names serve not only to identify the class but also the instance type of that class. It may seem odd that abstract classes, and interfaces, all define instance types, even though they don't have instances. However, as will become clear below, even abstract classes have instances, indirectly through their concrete subclasses. Similarly, interfaces may be thought of as possessing instances, namely the instances of concrete classes that implement that interface.

39.2.1 Subtyping

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, $C \lhd C'$, is the reflexive and transitive closure of the *extends* relation among classes, which holds precisely when one class is declared to extend another. In other words, $C \lhd C'$ iff C either coincides with C', inherits directly from C', or inherits from a subclass of C'. Since interfaces are classes, the subclass relation also applies to interfaces, but note that multiple inheritance of interfaces means that an interface can be a subinterface (subclass) of more than one interface. The *implementation* relation, $C \blacktriangleleft I$, is defined to hold exactly when a class C is declared to implement an interface that inherits from I.

The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau} \tag{39.1}$$

$$\frac{\tau \mathrel{\texttt{<:}} \tau' \quad \tau' \mathrel{\texttt{<:}} \tau''}{\tau \mathrel{\texttt{<:}} \tau''} \tag{39.2}$$

Arrays are *covariant* type constructors, in the sense of this rule:

$$\frac{\tau \mathrel{\texttt{<:}} \tau'}{\tau\,\texttt{[]} \mathrel{\texttt{<:}} \tau'\,\texttt{[]}} \tag{39.3}$$

Inheritance implies subtyping:

$$\frac{C \lhd C'}{C \mathrel{\texttt{<:}} C'} \tag{39.4}$$

Implementation implies subtyping:

$$\frac{C \blacktriangleleft I}{C \mathrel{\texttt{<:}} I} \tag{39.5}$$

Every class is a subclass of the distinguished "root" class `Object`:

$$\frac{}{\tau \mathrel{\texttt{<:}} \texttt{Object}} \tag{39.6}$$

The array subtyping rule is a structural subtyping principle — one need not explicitly declare subtyping relationships between array types for them to hold. On the other hand, the inheritance and implementation rules of subtyping are examples of nominal subtyping — they hold when they are declared to hold at the point of definition (or are implied by further subtyping relations).

39.2.2 Subsumption

The subsumption principle tells us that if e is an expression of type τ and $\tau <: \tau'$, then e is also an expression of type τ'. In particular, if a method is declared with a parameter of type τ, then it makes sense to provide an argument of any type τ' such that $\tau' <: \tau$. Similarly, if a constructor takes a parameter of a type, then it is legitimate to provide an argument of a subtype of that type. Finally, if a method is declared to return a value of type τ, then it is legitimate to return a value of any subtype of τ.

This brings up an awkward issue in the Java type system. What should be the type of a conditional expression $e \texttt{?} e_1 \texttt{:} e_2$? Clearly e should have type `boolean`, and e_1 and e_2 should have the same type, since we cannot in general predict the outcome of the condition e. In the presence of subtyping, this amounts to the requirement that the types of e_1 and e_2 have an *upper bound* in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the *least upper bound* of the types of e_1 and e_2. Unfortunately, two types need not have a least upper bound! For example, if an interface I extends incomparable interfaces K and L, and J extends both K and L, then I and J do not have a least upper bound — both K and L are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather *ad hoc* requirement that either the type of e_1 be a subtype of the type of e_2, or *vice versa*, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is *covariant* in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If $\tau <: \tau'$ and A is an array of type $\tau\,\texttt{[]}$, then retrieving an element of A yields a value of type τ, which is by hypothesis an element of type τ'. So we are OK with respect to retrieval. Now consider array assignment. Suppose once again that $\tau <: \tau'$ and that A is an array of type $\tau\,\texttt{[]}$. Then A is also an array of type $\tau'\,\texttt{[]}$, according to the Java rule for array subtyping. This means we can assign a value x of type τ' to an element of A. But this violates the assumption that A is an array of type $\tau\,\texttt{[]}$ — one of its elements is of type τ'.

With no further provisions the language would not be type safe. It is a simple matter to contrive an example involving arrays that incurs a run-time type error ("gets stuck"). Java avoids this by a simple, but expensive, device — every array assignment incurs a "run-time type check" that ensures that the assignment does not create an unsafe situation. In the next subsection we explain how this is achieved.

39.2.3 Dynamic Dispatch

According to Java typing rules, if C is a sub-class of D, then C is a sub-type of D. Since the instances of a class C have type C, they also, by subsumption, have type D, as do the instances of class D itself. In other words, if the static type of an instance is D, it might be an instance of class C or an instance of class D. In this sense the static type of an instance is at best an approximation of its dynamic type, the class of which it is an instance.

The distinction between the static and the dynamic type of an object is fundamental to object-oriented programming. In particular method specialization is based on the dynamic type of an object, not its static type. Specifically, if C is a sub-class of D that overrides a method m, then invoking the method m of a C instance o will always refer to the overriding code in C, even if the static type of o is D. That is, method dispatch is based on the dynamic type of the instance, not on its static type. For this reason method specialization is sometimes called *dynamic dispatch*, or, less perspicuously, *late binding*.

How is this achieved? Essentially, every object is tagged with the class that created it, and this tag is used to determine which method to invoke when a message is sent to that object. The constructors of a class C "label" the objects they create with C. The method dispatch mechanism consults this label when determining which method to invoke.[5]

The same mechanism is used to ensure that array assignments do not lead to type insecurities. Suppose that the static type of A is `C []`, and that the static type of instance o is C. By covariance of array types the dynamic type of A might be `D []` for some sub-class D of C. But unless the dynamic type of o is also D, the assignment of o to an element of A should be prohibited. This is ensured by an explicit run-time check. In

[5]In practice the label is a pointer to the vector of methods of the class, and the method is accessed by indexing into this vector. But we can just as easily imagine this to be achieved by a case analysis on the class name to determine the appropriate method vector.

Java *every single array assignment incurs a run-time check* whenever the array contains objects.[6]

39.2.4 Casting

A *container class* is one whose instances "contain" instances of another class. For example, a class of lists or trees or sets would be a container class in this sense. Since the operations on containers are largely (or entirely) independent of the type of their elements, it makes sense to define containers generally, rather than defining one for each element type. In Java this is achieved by exploiting subsumption. Since every object has type `Object`, a general container is essentially a container whose elements are of type `Object`. This allows the container operations to be defined once for all element types. However, when retrieving an element from a container its static type is `Object`; we lost track of its dynamic type during type checking. If we wish to use such an object in any meaningful way, we must recover its dynamic type so that message sends are not rejected at compile time.

Java supports a safe form of *casting*, or *change of type*. A cast is written $(\tau)\ e$. The expression e is called the *subject* of the cast, and the type τ is the *target type* of the cast. The type of the cast is τ, provided that the cast makes sense, and its value is that of e. In general we cannot determine whether the cast makes sense until execution time, when the dynamic type of the expression is available for comparison with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until execution time.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. *Up casts*, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. *Down casts*, in which the static type of the expression is a *supertype* of the target type. The true type may or may not be a subtype of the

[6] Arrays of integers and floats do not incur this overhead, because numbers are not objects.

target, so a run-time check is required.

3. *Stupid casts*, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that down casts do not fail. For example, if a container is intended to contain objects of a class C, then retrieved elements of that class will typically be down cast to a sub-class of C. It is entirely up to the programmer to ensure that these casts do not fail at execution time. That is, the programmer must maintain the invariant that the retrieved element really contains an instance of the target class of the cast.

39.3 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should "behave properly" in superclass contexts — they should not be distinguishable from them.

But this isn't necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary[7] changes to the behavior of the superclass when defining the subclass. For example, we can turn a stack-like object into a queue-like object (replacing a LIFO discipline by a FIFO discipline) by inheritance, thereby changing the behavior drastically. If we

[7]Limited only by finality declarations in the superclass.

are to pass off a subclass instance as a superclass instance using subtyping, then we should refrain from making such drastic behavioral changes.

The Java type system provides only weak tools for ensuring a behavioral subtyping relationship between a subclass and its superclass. Fundamentally, the type system is not strong enough to express the desired constraints.[8]. To compensate for this Java provides the finality mechanism to limit inheritance. Final classes cannot be inherited from at all, ensuring that values of its instance type are indeed instances of that class (rather than an arbitrary subclass). Final methods cannot be overridden, ensuring that certain aspects of behavior are "frozen" by the class definition.

Nominal subtyping may also be seen as a tool for enforcing behavioral subtyping relationships. For unless a class extends a given class or is declared to implement a given interface, no subtyping relationship holds. This helps to ensure that the programmer explicitly considers the behavioral subtyping obligations that are implied by such declarations, and is therefore an aid to controlling inheritance.

[8]Nor is the type system of any other language that I am aware of, including ML

Part XV

Program Equivalence

Chapter 40

Functional Equivalence

One of the beauties of functional programming is the ease with which we may reason about equivalence of expressions. Informally, we say that two expressions e_1 and e_2 of the same type are *equivalent* iff replacing e_1 by e_2 in a complete program doesn't change its final result. By a "complete program" we mean a closed expression of type `int` or `bool`; the final result of a complete program is therefore a number or a Boolean constant. What is important here is that the final outcome be *finitely observable* — we can see immediately that the answer is `false` or 17. Since functions are essentially "infinite" objects (in the sense that the graph of a function on the integers is infinite), we would not regard functions as observable outcomes of a complete program.

We can think of a usage of an expression in a complete program as an "experiment" or "observation" performed on that expression. The idea is that the program "uses" the expression to compute an observable quantity that we can regard as a kind of test performed on that expression. For this reason the notion of equivalence just described is sometimes called *observational equivalence* — two expressions are observationally equivalent iff any experiment performed on one yields the same observable outcome as the same experiment performed on the other. This relation is also called *contextual equivalence* to emphasize that equivalence is determined by considering all contexts in which the two expressions might be used to form a complete program. In philosophical logic this relation is known as *Leibniz's Principle of Identity of Indiscernibles* — two things are equal iff we cannot tell them apart.

Observational equivalence is very difficult to handle. To determine whether or not e_1 and e_2 are observationally equivalent requires us to consider *all possible programs* that use them to compute an integer! This quickly gets out of hand. What we need are alternative criteria for establishing observational equivalences that avoid the need to explicitly consider all possible usages of the expressions in question. Unfortunately, a rigorous development of such alternatives would take us far beyond the scope of the course. We will content ourselves with stating, without proof, a collection of laws of equivalence that are useful for deriving equations between expressions such as these:

1. $x + (y + z)$ is equivalent to $(x + y) + z$.
2. `rev(rev(`x`))` is equivalent to x (where `rev` is the list reversal function).
3. If $v = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e$ and $v{:}\tau_1$ is a value, then $v\texttt{(}v_1\texttt{)}$ is equivalent to $[f, x{\leftarrow}v, v_1]e$.

In the next section we make precise the definition of observational equivalence, and state an alternative characterization of it that is often easier to handle. In the subsequent section we enumerate a collection of valid principles of equivalence for variants of MinML.

40.1 Expression Equivalence

We begin with the notion of *Kleene equality* between complete programs. Kleene equivalence captures what we mean by "same outcome" for complete programs. Two complete programs p_1 and p_2 are *Kleene equivalent*, written $p_1 \simeq p_2$, exactly when $p_1 \Downarrow v$ iff $p_2 \Downarrow v$. That is, either both p_1 and p_2 diverge (fail to halt), or both converge to the same number or Boolean constant (perhaps in very different ways, using very different amounts of time and space).

A *context* C is a complete program with a single "hole" into which we may insert an expression. That is, C has the form $\ldots \bullet \ldots$, where the $\bullet$ indicates the "hole" in the program. We write $C\{e\}$ for the result of filling the hole in C with the expression e to obtain $\ldots e \ldots$. The expression e might have free variables that are captured when inserted into the hole.

For example, if C is the program context $(\texttt{fun}\, f\; (x\texttt{:int})\texttt{:int is}\, \bullet)\texttt{(3)}$ and e is the expression $x + 5$, then $C\{x+5\}$ is the program

$$(\texttt{fun}\, f\; (x\texttt{:int})\texttt{:int is}\, x+5)\texttt{(3)}.$$

Suppose that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. We define the *observational equivalence* relation $\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ to hold iff for every program context C such that $C\{e_1\}$ and $C\{e_2\}$ are programs, $C\{e_1\} \simeq C\{e_2\}$. That is, every use of e_1 has the same observable outcome as the corresponding use of e_2, and vice-versa.

As we remarked earlier, it is rather difficult to establish that two expressions are observationally equivalent. A direct application of the definition leaves us no recourse but to consider all possible program contexts C, which quickly gets out of hand.

A more usable characterization of observational equivalence, called *applicative equivalence*, is defined as follows. For closed expressions e_1 and e_2 of type τ, we define $e_1 \cong_{app}^{closed} e_2 : \tau$ by induction on the structure of τ as follows:

- If $\tau = \texttt{int}$, or $\tau = \texttt{bool}$, then $e_1 \cong_{app}^{closed} e_2 : \tau$ iff $e_1 \simeq e_2$.
- If $\tau = \texttt{arrow}(\tau_1, \tau_2)$, then $e_1 \cong_{app}^{closed} e_2 : \tau$ iff for every $v : \tau_1$, $e_1\texttt{(}v\texttt{)} \cong_{app}^{closed} e_2\texttt{(}v\texttt{)} : \tau_2$.

This relation is extended to open expressions by substitution of closed values of appropriate type for the free variables. Let Γ be the context $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. We define $\Gamma \vdash e_1 \cong_{app} e_2 : \tau$ to hold iff

$$[x_1, \ldots, x_n{\leftarrow}v_1, \ldots, v_n]e_1 \cong_{app}^{closed} [x_1, \ldots, x_n{\leftarrow}v_1, \ldots, v_n]e_2$$

for every substitution of closed values $v_1,\ldots,v_n$ of type $\tau_1,\ldots,\tau_n$ for $x_1,\ldots,x_n$.

An important result of Milner's states that applicative and observational equivalence coincide. The proof is non-trivial, and is omitted from this brief exposition.

Theorem 40.1 (Milner's Context Lemma)
$\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ *iff* $\Gamma \vdash e_1 \cong_{app} e_2 : \tau$.

The point of the context lemma is that we may interpret the relation $\Gamma \vdash e_1 \cong_{obs} e_2 : \tau$ as expressing the universally-quantified formula

$$\forall v_1 : \tau_1 \ \ldots \ \forall v_n : \tau_n \ [x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]e_1 \cong_{app} [x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]e_2 : \tau$$

where the quantifiers range over *closed values* of the appropriate type. For example,

$$x{:}\mathtt{int}, y{:}\mathtt{int} \vdash x + y \cong_{obs} y + x : \mathtt{int}$$

means that for every m and n, $m + n \simeq n + m$.

40.2 Some Laws of Equivalence

In this section we summarize some useful principles of equivalence. These are all valid observational equivalences, but we will not prove this. What is important is to get a feeling for what are some valid principles of equivalence, and how to use them in practice. Since observational and applicative equivalence coincide, we will write $\Gamma \vdash e_1 \cong e_2 : \tau$ for equality of expressions of type τ relative to a context Γ, where we tacitly assume that $\Gamma \vdash e_i : \tau$ for $i = 1, 2$.

In the presentation of the rules, we use v to stand for an *open value*, either a variable, a constant, or an function expression (perhaps with free variables occurring within it). We admit variables as values because, in a call-by-value language, variables are only ever bound to values, and hence may be taken as standing for a fixed, but unknown, value.

It will be convenient to make use of a designated non-terminating expression of each type τ, written Ω_τ, which is defined to be the expression

```
(fun f(x:int):τ is f(x) end)(0).
```

It is easy to check that Ω_τ diverges (loops forever) when evaluated.

40.2.1 General Laws

First, equivalence is indeed an equivalence relation — it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau} \qquad (40.1)$$

$$\frac{\Gamma \vdash e_2 \cong e_1 : \tau}{\Gamma \vdash e_1 \cong e_2 : \tau} \tag{40.2}$$

$$\frac{\Gamma \vdash e_1 \cong e_2 : \tau \qquad \Gamma \vdash e_2 \cong e_3 : \tau}{\Gamma \vdash e_1 \cong e_3 : \tau} \tag{40.3}$$

Second, equivalence is a *congruence* — we may replace a sub-expression of any expression by an equivalent one to obtain an equivalent expression. This is most easily stated by a collection of rules that ensure that we may replace equivalent sub-expressions to obtain equivalent expressions. We will give just a few of these here; the rest follow a similar pattern.

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1 \quad \cdots \quad \Gamma \vdash e_n \cong e_n' : \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \cong o(e_1', \ldots, e_n') : \tau} \tag{40.4}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \texttt{arrow}(\tau_2, \tau) \qquad \Gamma \vdash e_2 \cong e_2' : \tau_2}{\Gamma \vdash e_1(e_2) \cong e_1'(e_2') : \tau} \tag{40.5}$$

$$\frac{\Gamma[f{:}\texttt{arrow}(\tau_1, \tau_2)][x{:}\tau_1] \vdash e \cong e' : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\, e \cong \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\, e' : \texttt{arrow}(\tau_1, \tau_2)} \tag{40.6}$$

Finally, equivalence is stable under substitution of *values* for free variables.

$$\frac{\Gamma[x{:}\tau] \vdash e \cong e' : \tau'}{\Gamma \vdash [x{\leftarrow}v]e \cong [x{\leftarrow}v]e' : \tau'} \tag{40.7}$$

The restriction to values is essential; this rule is *not* true for general expression substitution! A counterexample is given in the next subsection.

40.2.2 Symbolic Evaluation

Evaluation of an expression in accordance with the rules of the operational semantics results in an equivalent expression. This is called "symbolic evaluation" because the transformations may involve expressions with free variables, which are regarded as values for the purposes of these rules.

An application of a primitive operation may be simplified if we know the values of its arguments:

$$\frac{}{\Gamma \vdash o(v_1, \ldots, v_n) \cong v : \tau,} \tag{40.8}$$

where v is the result of applying o to $v_1, \ldots, v_n$.

Similarly, if we know the result of the boolean test, then a conditional may be simplified:

$$\frac{}{\Gamma \vdash \texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \cong e_1 : \tau} \tag{40.9}$$

$$\frac{}{\Gamma \vdash \texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \cong e_2 : \tau} \tag{40.10}$$

An application may be simplified if we know the function and the argument is a value. Note that either the function or argument may be *open* expressions (containing free variables)!

$$\frac{}{\Gamma \vdash v(v_1) \cong [f, x \leftarrow v, v_1]e : \tau_2} \tag{40.11}$$

where $v = \texttt{fun}\, f\ (x : \tau_1) : \tau_2\ \texttt{is}\, e$.

Exercise 40.2
Using these rules, check that `(fn x in 3 end)(z)` *is equivalent to* 3*. Show that* `(fn x in 3 end)`$(\Omega_{\texttt{int}})$ *is* not *equivalent to* 3*. Conclude that substitution of non-values for free variables does not preserve equivalence.*

40.2.3 Extensionality

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{\Gamma[x{:}\tau_1] \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \texttt{arrow}(\tau_1, \tau_2)} \tag{40.12}$$

In other words, if two functions are equal for all closed argument values, then they are equal.

40.2.4 Strictness Properties

The evaluation rules of MinML impose a *call-by-value* evaluation order on function applications and primitive operations. This can be captured equationally by a set of *strictness* equations that are defined in terms of the divergent expressions Ω_τ. We may state that an expression e of type τ diverges as an equation by stating that $e \cong \Omega_\tau$. The following rules give some conditions under which expressions are divergent.

If any argument of a primitive operation is divergent, so is the whole expression:

$$\frac{}{\Gamma \vdash o(e_1, \ldots, e_{i-1}, \Omega_{\tau_i}, e_{i+1}, \ldots, e_n) \cong \Omega_\tau : \tau} \tag{40.13}$$

If the test expression of a conditiona is divergent, so is the conditional.

$$\frac{}{\Gamma \vdash \texttt{if}\, \Omega_{\texttt{bool}}\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \cong \Omega_\tau : \tau} \tag{40.14}$$

If the function or argument of an application is divergent, so is the entire expression:

$$\frac{}{\Gamma \vdash \Omega_{\texttt{arrow}(\tau_2,\tau)}(e_2) \cong \Omega_\tau : \tau} \tag{40.15}$$

$$\frac{}{\Gamma \vdash e_1(\Omega_{\tau_2}) \cong \Omega_\tau : \tau} \tag{40.16}$$

40.2.5 Arithmetic Laws

Arithmetic and comparison operations behave as expected. For example, addition is associative and commutative, and equality test on integers is an equivalence relation. In general appropriate laws governing the primitive operations on integers hold, provided that they hold mathematically. The same could not be said for floating point (for which addition is not even associative!). Observe that these laws *fail*, in general, in the presence of effects such as writing to the screen or destructively updating a reference cell! In that case we must restrict attention to values, not general expressions.

$$\frac{}{\Gamma \vdash e_1\texttt{+}e_2 \cong e_2\texttt{+}e_1 : \texttt{int}} \tag{40.17}$$

$$\frac{}{\Gamma \vdash e_1\texttt{+}(e_2\texttt{+}e_3) \cong (e_1\texttt{+}e_2)\texttt{+}e_3 : \texttt{int}} \tag{40.18}$$

$$\frac{}{\Gamma \vdash e_1\texttt{=}e_2 \cong e_2\texttt{=}e_1 : \texttt{bool}} \tag{40.19}$$

40.2.6 Products

For the extension of MinML to product types,[1] we have the following symbolic evaluation rule:

$$\frac{}{\Gamma \vdash \texttt{split}\,(v_1,v_2)\;\texttt{as}\;(x_1:\tau_1,x_2:\tau_2)\;\texttt{in}\,e \cong [x_1,x_2{\leftarrow}v_1,v_2]e} \tag{40.20}$$

We may, in general, replace equals by equals:

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1 \qquad \Gamma \vdash e_2 \cong e_2' : \tau_2}{\Gamma \vdash (e_1,e_2) \cong (e_1',e_2') : \tau_1\texttt{*}\tau_2} \tag{40.21}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau_1\texttt{*}\tau_2 \quad \Gamma[x_1{:}\tau_1][x_2{:}\tau_2] \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash \texttt{split}\,e_1\;\texttt{as}\;(x_1:\tau_1,x_2:\tau_2)\;\texttt{in}\,e_2 \cong \texttt{split}\,e_1'\;\texttt{as}\;(x_1:\tau_1,x_2:\tau_2)\;\texttt{in}\,e_2'} \tag{40.22}$$

Finally, the expected strictness properties hold:

$$\frac{}{\Gamma \vdash (\Omega_{\tau_1},e_2) \cong \Omega_{(\tau_1,\tau_2)}} \tag{40.23}$$

$$\frac{}{\Gamma \vdash (e_1,\Omega_{\tau_2}) \cong \Omega_{(\tau_1,\tau_2)}} \tag{40.24}$$

$$\frac{}{\Gamma \vdash \texttt{split}\,\Omega_{\tau_1\texttt{*}\tau_2}\;\texttt{as}\;(x_1:\tau_1,x_2:\tau_2)\;\texttt{in}\,e \cong \Omega_{\tau'}} \tag{40.25}$$

[1]We do not consider nested or wildcard patterns, for the sake of simplicity. It is a simple matter to extend these rules to the more general case.

40.2.7 Lists

We may extend MinML with list types $\tau\,\texttt{list}$ by adding the expressions $\texttt{nil}$, $\texttt{cons(}e_1\texttt{,}e_2\texttt{)}$, and $\texttt{listcase}\,e\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\mid\texttt{cons(}x\texttt{,}y\texttt{)}\,\texttt{=>}\,e''$, with the following typing rules:

$$\frac{}{\Gamma\vdash\texttt{nil}:\tau\,\texttt{list}}\qquad(40.26)$$

$$\frac{}{\Gamma\vdash\texttt{cons(}e_1\texttt{,}e_2\texttt{)}:\tau\,\texttt{list}}\qquad(40.27)$$

$$\frac{\Gamma\vdash e:\tau\,\texttt{list}\qquad\Gamma\vdash e':\tau' \\ \Gamma[x{:}\tau][y{:}\tau\,\texttt{list}]\vdash e'':\tau'}{\Gamma\vdash\texttt{listcase}\,e\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\mid\texttt{cons(}x\texttt{,}y\texttt{)}\,\texttt{=>}\,e'':\tau'}\qquad(40.28)$$

The following symbolic evaluation and strictness rules express the evaluation of these constructs:

$$\frac{}{\Gamma\vdash\texttt{listcase}\,\texttt{nil}\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\mid\texttt{cons(}x\texttt{,}y\texttt{)}\,\texttt{=>}\,e''\cong e':\tau'}\qquad(40.29)$$

$$\frac{}{\Gamma\vdash\texttt{listcase}\,\texttt{cons(}v_1\texttt{,}v_2\texttt{)}\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\mid\texttt{cons(}x_1\texttt{,}x_2\texttt{)}\,\texttt{=>}\,e''\cong[x_1,x_2{\leftarrow}v_1,v_2]e'':\tau'}\qquad(40.30)$$

$$\frac{}{\Gamma\vdash\texttt{cons(}\Omega_\tau\texttt{,}e\texttt{)}\cong\Omega_{\tau\,\texttt{list}}:\tau\,\texttt{list}}\qquad(40.31)$$

$$\frac{}{\Gamma\vdash\texttt{cons(}e\texttt{,}\Omega_{\tau\,\texttt{list}}\texttt{)}\cong\Omega_{\tau\,\texttt{list}}:\tau\,\texttt{list}}\qquad(40.32)$$

$$\frac{}{\Gamma\vdash\texttt{listcase}\,\Omega_{\tau\,\texttt{list}}\,\texttt{of}\,\texttt{nil}\,\texttt{=>}\,e'\mid\texttt{cons(}x_1\texttt{,}x_2\texttt{)}\,\texttt{=>}\,e''\cong\Omega_{\tau'}:\tau'}\qquad(40.33)$$

Most importantly, we may prove equivalences by induction on the structure of a list. Suppose that $\Gamma[x{:}\tau\,\texttt{list}]\vdash e_i:\tau'$ ($i=1,2$). To prove that

$$\Gamma[x{:}\tau\,\texttt{list}]\vdash e_1\cong e_2:\tau',$$

it suffices to show the following two facts:

1. $\Gamma\vdash[x{\leftarrow}\texttt{nil}]e_1\cong[x{\leftarrow}\texttt{nil}]e_2:\tau'$

2. For every $\Gamma \vdash v_h : \tau$ and $\Gamma \vdash v_t : \tau\,\texttt{list}$, if $\Gamma \vdash [x{\leftarrow}v_t]e_1 \cong [x{\leftarrow}v_t]e_2 : \tau'$, then $\Gamma \vdash [x{\leftarrow}\texttt{cons}(v_h, v_t)]e_1 \cong [x{\leftarrow}\texttt{cons}(v_h, v_t)]e_2 : \tau'$.

Exercise 40.3
The list append and reversal functions are defined as follows:[2]

```
fun app(l:τ list, m:τ list):τ list is
  listcase l of nil => m | h::t => h :: app (t, m) end
end
fun rev (l:τ list) is
  listcase l of nil => nil | h::t => app (rev(t), [h]) end
end
```

Use list induction and the laws of expression equivalence to prove the following two facts:

1. $x{:}\tau\,\texttt{list}, y{:}\tau\,\texttt{list} \vdash \texttt{rev(app}(x,y)\texttt{)} \cong \texttt{app(rev(}y\texttt{),rev(}x\texttt{))} : \tau\,\texttt{list}$
2. $x{:}\tau\,\texttt{list} \vdash \texttt{rev(rev(}x\texttt{))} \cong x : \tau\,\texttt{list}$

[2]Officially, the two-argument append function is written using `bind` as follows:

```
fun app(lm:τ list * τ list):τ list is bind (l:τ list, m:τ list) to
lm in ...end end.
```

We use infix notation for the append function for the sake of clarity.

Chapter 41

Parametricity

Our original motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function give above. The idea is that if the behavior of a function *does not* depend on a choice of types, then it is useful to be able to define such "type oblivious" functions in the language. Once we have such a mechanism in hand, it can also be used to ensure that a particular piece of code *can not* depend on a choice of types by insisting that it be polymorphic in those types. In this sense polymorphism may be used to impose restrictions on a program, as well as to allow more programs to be written.

The restrictions imposed by requiring a program to be polymorphic underlie the often-observed experience when programming in ML that if the types are correct, then the program is correct. Roughly speaking, since the ML type system is polymorphic, if a function type checks with a polymorphic type, then the strictures of polymorphism vastly cut down the set of well-typed programs with that type. Since the intended program is one these (by the hypothesis that its type is "right"), you're much more likely to have written it if the set of possibilities is smaller.

The technical foundation for these remarks is called *parametricity*. The goal of this chapter is to give an account of parametricity for PolyMinML. To keep the technical details under control, we will restrict attention to the ML-like (prenex) fragment of PolyMinML. It is possibly to generalize to first-class polymorphism, but at the expense of considerable technical complexity. Nevertheless we will find it necessary to gloss over some technical details, but wherever a "pedagogic fiction" is required, I will point it out. To start with, it should be stressed that the following *does not apply* to languages with mutable references!

41.1 Informal Overview

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value f has the type $\forall t(\texttt{arrow}(t, t))$. What function could it be?

1. It could diverge when instantiated — f $[\tau]$ goes into an infinite loop. Since f is polymorphic, its behavior cannot depend on the choice of τ, so in fact f $[\tau']$ diverges for all τ' if it diverges for τ.

2. It could converge when instantiated at τ to a function g of type $\texttt{arrow}(\tau, \tau)$ that loops when applied to an argument v of type τ — *i.e.*, $g(v)$ runs forever. Since f is polymorphic, g must diverge on *every* argument v of type τ if it diverges on *some* argument of type τ.

3. It could converge when instantiated at τ to a function g of type $\texttt{arrow}(\tau, \tau)$ that, when applied to a value v of type τ returns a value v' of type τ. Since f is polymorphic, g cannot depend on the choice of v, so v' must in fact be v.

Let us call cases (1) and (2) *uninteresting*. The foregoing discussion suggests that the only *interesting* function f of type $\forall t(\texttt{arrow}(t, t))$ is the polymorphic identity function.

Suppose that f is an interesting function of type $\forall t(t)$. What function could it be? A moment's thought reveals that it cannot be interesting! That is, every function f of this type must diverge when instantiated, and hence is uninteresting. In other words, there are no interesting values of this type — it is essentially an "empty" type.

For a final example, suppose that f is an interesting function of type $\forall t(\texttt{arrow}(t\,\texttt{list}, t\,\texttt{list}))$. What function could it be?

1. The identity function that simply returns its argument.

2. The constantly-`nil` function that always returns the empty list.

3. A function that drops some elements from the list according to a predetermined (data-independent) algorithm — *e.g.*, always drops the first three elements of its argument.

4. A permutation function that reorganizes the elements of its argument.

The characteristic that these functions have in common is that their behavior is entirely determined by the *spine* of the list, and is independent of the *elements* of the list. For example, f cannot be the function that drops all "even" elements of the list — the elements might not be numbers! The point is that the type of f is polymorphic in the element type, but reveals that the argument is a list of unspecified elements. Therefore it can only depend on the "list-ness" of its argument, and never on its contents.

In general if a polymorphic function behaves the same at every type instance, we say that it is *parametric* in that type. In PolyMinML all polymorphic functions are parametric. In Standard ML most functions are, except those that involve *equality types*. The equality function is *not* parametric because the equality test depends on the type instance — testing equality of integers is different than testing equality of floating point numbers, and we cannot test equality of functions. Such "pseudo-polymorphic" operations are said to be *ad hoc*, to contrast them from *parametric*.

How can parametricity be exploited? As we will see later, parametricity is the foundation for data abstraction in a programming language. To get a sense of the relationship, let us consider a classical example of exploiting parametricity, the *polymorphic Church numerals*. Let N be the type $\forall t(\texttt{arrow}(t, \texttt{arrow}((\texttt{arrow}(t,t)), t)))$. What are the interesting functions of the type N? Given any type τ, and values $z : \tau$ and $s : \texttt{arrow}(\tau, \tau)$, the expression

$$f\,[\tau]\,(z)\,(s)$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ is by using the element z and the function s passed to it. A moment's thought reveals that the application must amount to the n-fold composition

$$s(s(\ldots s(z)\ldots)).$$

That is, the elements of N are in 1-to-1 correspondence with the natural numbers.

Let us write $\overline{n}$ for the polymorphic function of type N representing the natural number n, namely the function

```
Fun t in
    fn z:t in
       fn s:t->t in
          s(s(... s)...))
       end
    end
end
```

where there are n occurrences of s in the expression. Observe that if we instantiate $\overline{n}$ at the built-in type `int` and apply the result to 0 and `succ`, it evaluates to the number n. In general we may think of performing an "experiment" on a value of type N by instantiating it at a type whose values will constitute the observations, the applying it to operations z and s for performing the experiment, and observing the result.

Using this we can calculate with Church numerals. Let us consider how to define the addition function on N. Given $\overline{m}$ and $\overline{n}$ of type N, we wish to compute their sum $\overline{m+n}$, also of type N. That is, the addition function must look as follows:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                ...
             end
          end
      end
   end
end
```

The question is: how to fill in the missing code? Think in terms of experiments. Given m and n of type N, we are to yield a value that when "probed" by supplying a type t, an element z of that type, and a function s on that type, must yield the $(m+n)$-fold composition of s with z. One way to do this is to "run" m on t, z, and s, yielding the m-fold composition of s with z, then "running" n on this value and s again to obtain the n-fold composition of s with the n-fold composition of s with z — the desired answer. Here's the code:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                n[t](m[t](z)(s))(s)
             end
          end
      end
   end
end
```

To see that it works, instantiate the result at τ, apply it to z and s, and observe the result.

41.2 Relational Parametricity

In this section we give a more precise formulation of parametricity. The main idea is that polymorphism implies that certain equations between expressions must hold. For example, if $f : \forall t(\texttt{arrow}(t,t))$, then f must be *equal to* the identity function, and if $f : N$, then f must be *equal to* some Church numeral $\overline{n}$. To make the informal idea of parametricity precise, we must clarify what we mean by equality of expressions.

The main idea is to define equality in terms of "experiments" that we carry out on expressions to "test" whether they are equal. The valid experiments on an expression are determined solely by its type. In general we say that two closed *expressions* of a type τ are equal iff either they both diverge, or they both converge to equal *values* of that type. Equality of closed values is then defined based on their type. For integers and booleans, equality is straightforward: two values are equal iff they are identical. The intuition here is that equality of numbers and booleans is directly observable. Since functions are "infinite" objects (when thought of in terms of their input/output behavior), we define equality in terms of their behavior when applied. Specifically, two functions f and g of type $\texttt{arrow}(\tau_1, \tau_2)$

are equal iff whenever they are applied to equal arguments of type τ_1, they yield equal results of type τ_2.

More formally, we make the following definitions. First, we define equality of closed expressions of type τ as follows:

$$e \cong_{exp} e' : \tau \quad \mathit{iff} \quad e \longmapsto^* v \Leftrightarrow e' \longmapsto^* v' \quad \mathit{and} v \cong_{val} v' : \tau.$$

Notice that if e and e' both diverge, then they are equal expressions in this sense. For closed values, we define equality by induction on the structure of monotypes:

$$\begin{array}{rll} v \cong_{val} v' : \texttt{bool} & \mathit{iff} & v = v' = \texttt{true} \ \mathit{or}\ v = v' = \texttt{false} \\ v \cong_{val} v' : \texttt{int} & \mathit{iff} & v = v' = n \ \mathit{for\ some}\ n \geq 0 \\ v \cong_{val} v' : \texttt{arrow}(\tau_1, \tau_2) & \mathit{iff} & v_1 \cong_{val} v_1' : \tau_1 \ \mathit{implies}\ v(v_1) \cong_{exp} v'(v_1') : \tau_2 \end{array}$$

The following lemma states two important properties of this notion of equality.

Lemma 41.1

1. *Expression and value equivalence are reflexive, symmetric, and transitive.*

2. *Expression equivalence is a* congruence*: we may replace any sub-expression of an expression* e *by an equivalent sub-expression to obtain an equivalent expression.*

So far we've considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not able to say *a priori* what we mean by equality at that type, precisely because it is "unknown". Therefore we *also* quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that R is a binary relation between values of type τ and τ'.

Here is the definition of equality of polymorphic values:

$$v \cong_{val} v' : \forall t(\sigma) \quad \mathit{iff} \quad \mathit{for\ all}\ \tau\ \mathit{and}\ \tau',\ \mathit{and\ all}\ R : \tau \leftrightarrow \tau',\ v\ [\tau] \cong_{exp} v'\ [\tau'] : \sigma$$

where we take equality at the type variable t to be the relation R (*i.e.*, $v \cong_{val} v' : t$ iff $v\, R\, v'$).

There is one important *proviso*: when quantifying over relations, we must restrict attention to what are called *admissible* relations, a sub-class of relations that, in a suitable sense, respects computation. Most natural choices of relation are admissible, but it is possible to contrive examples that are not. The rough-and-ready rule is this: a relation is admissible iff it is closed under "partial computation". Evaluation of an expression e to a value proceeds through a series of intermediate expressions $e \longmapsto e_1 \longmapsto e_2 \longmapsto \cdots e_n$. The expressions e_i may be thought of as "partial computations" of e, stopping points along the way to the value of e. If a relation relates corresponding partial computations of e and e', then, to be admissible, it must also relate e and e' — it cannot relate all partial computations, and then refuse to relate the complete expressions. We will not develop this idea any further, since to do so would require the formalization of partial computation. I hope that this informal discussion suffices to give the idea.

The following is Reynolds' Parametricity Theorem:

Theorem 41.2 (Parametricity)
If $e : \sigma$ is a closed expression, then $e \cong_{exp} e : \sigma$.

This may seem obvious, until you consider that the notion of equality between expressions of polymorphic type is very strong, requiring equivalence under *all possible* relational interpretations of the quantified type.

Using the Parametricity Theorem we may prove a result we stated informally above.

Theorem 41.3
If $f : \forall t(\texttt{arrow}(t,t))$ is an interesting value, then $f \cong_{val} id : \forall t(\texttt{arrow}(t,t))$, where id is the polymorphic identity function.

Proof: Suppose that τ and τ' are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f\ [\tau] \cong_{exp} id\ [\tau'] : \texttt{arrow}(t,t),$$

where equality at type t is taken to be the relation R.

Since f (and id) are interesting, there exists values f_τ and $id_{\tau'}$ such that

$$f\ [\tau] \longmapsto^* f_\tau$$

and

$$id\ [\tau'] \longmapsto^* id_{\tau'}.$$

We wish to show that

$$f_\tau \cong_{val} id_{\tau'} : \texttt{arrow}(t, t).$$

Suppose that $v_1 \cong_{val} v_1' : t$, which is to say $v_1\, R\, v_1'$ since equality at type t is taken to be the relation R. We are to show that

$$f_\tau(v_1) \cong_{exp} id_{\tau'}(v_1') : t$$

By the assumption that f is interesting (and the fact that *id* is interesting), there exists values v_2 and v_2' such that

$$f_\tau(v_1) \longmapsto^* v_2$$

and

$$id_{\tau'}(v_1') \longmapsto^* v_2'.$$

By the definition of *id*, it follows that $v_2' = v_1'$ (it's the identity function!). We must show that $v_2\, R\, v_1'$ to complete the proof.

Now define the relation $R' : \tau \leftrightarrow \tau$ to be the set $\{\, (v, v) \mid v\, R\, v_1' \,\}$. Since $f : \forall t(\texttt{arrow}(t, t))$, we have by the Parametricity Theorem that $f \cong_{val} f : \forall t(\texttt{arrow}(t, t))$, where equality at type t is taken to be the relation R'. Since $v_1\, R\, v_1'$, we have by definition $v_1\, R'\, v_1$. Using the definition of equality of polymorphic type, it follows that

$$f_\tau(v_1) \cong_{exp} id_{\tau'}(v_1) : t.$$

Hence $v_2\, R\, v_1'$, as required. ■

You might reasonably wonder, at this point, what the relationship $f \cong_{val} id : \forall t(\texttt{arrow}(t, t))$ has to do with f's execution behavior. It is a general fact, which we will not attempt to prove, that equivalence as we've defined it yields results about execution behavior. For example, if $f : \forall t(\texttt{arrow}(t, t))$, we can show that for every τ and every $v : \tau$, $f\ [\tau](v)$ evaluates to v. By the preceding theorem $f \cong_{val} id : \forall t(\texttt{arrow}(t, t))$. Suppose that τ is some monotype and $v : \tau$ is some closed value. Define the relation $R : \tau \leftrightarrow \tau$ by

$$v_1\, R\, v_2 \textit{ iff } v_1 = v_2 = v.$$

Then we have by the definition of equality for polymorphic values

$$f\ [\tau]\,(v) \cong_{exp} id\ [\tau]\,(v) : t,$$

where equality at t is taken to be the relation R. Since the right-hand side terminates, so must the left-hand side, and both must yield values related by R, which is to say that both sides must evaluate to v.

Chapter 42

Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given above ensure that the client of an abstract type is polymorphic in the representation type. According to our informal understanding of parametricity this means that the client's behavior is in some sense "independent" of the representation type.

More formally, we say that an (admissible) relation $R : \tau_1 \leftrightarrow \tau_2$ is a *bisimulation* between the packages

$$\mathtt{pack}\,\tau_1\,\mathtt{with}\,v_1\,\mathtt{as}\,\exists t(\sigma)$$

and

$$\mathtt{pack}\,\tau_2\,\mathtt{with}\,v_2\,\mathtt{as}\,\exists t(\sigma)$$

of type $\exists t(\sigma)$ iff $v_1 \cong_{val} v_2 : \sigma$, taking equality at type t to be the relation R. The reason for calling such a relation R a bisimulation will become apparent shortly. Two packages are said to be *bisimilar* whenever there is a bisimulation between them.

Since the client e_c of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\mathtt{arrow}(\sigma, \tau_c))$, where $t \notin \mathrm{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$[t, x \leftarrow \tau_1, v_1]e_c \cong_{exp} [t, x \leftarrow \tau_2, v_2]e_c : \tau_c$$

whenever R is such a bisimulation. Consequently,

$$\mathtt{open}\,e_1\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e_c \cong_{exp} \mathtt{open}\,e_2\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e_c : \tau_c.$$

That is, the two implementations are indistinguishable by any client of the abstraction, and hence may be regarded as equivalent. This is called *Representation Independence*; it is merely a restatement of the Parametricity Theorem in the context of existential types.

This observation licenses the following technique for proving the correctness of an ADT implementation. Suppose that we have an implementation of an abstract type $\exists t(\sigma)$ that is "clever" in some way. We wish to show that it is a correct implementation of the abstraction. Let us therefore call it a *candidate* implementation. The Representation Theorem suggests a technique for proving the candidate correct. First, we define a *reference* implementation of the same abstract type that is "obviously correct". Then we establish that the reference implementation and the candidate implementation are bisimilar. Consequently, they are equivalent, which is to say that the candidate is "equally correct as" the reference implementation.

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that `QL` and `QFB` are bisimilar, and therefore indistinguishable by any client.

Define the relation $R : \texttt{int list} \leftrightarrow \texttt{int list*int list}$ as follows:

$$R = \{\, (l,(b,f))) \mid l \cong_{val} b\texttt{@rev}(f) \,\}$$

We will show that R is a bisimulation by showing that implementations of `empty`, `insert`, and `remove` determined by the structures `QL` and `QFB` are equivalent relative to R.

To do so, we will establish the following facts:

1. `QL.empty` R `QFB.empty`.

2. Assuming that $m \cong_{val} n$: `int` and $l\,R$ `(`b`,`f`)`, show that

   $$\texttt{QL.insert((}m\texttt{,}l\texttt{))}\;R\;\texttt{QFB.insert((}n\texttt{,(}b\texttt{,}f\texttt{)))}.$$

3. Assuming that $l\,R$ `(`b`,`f`)`, show that

   $$\texttt{QL.remove(}l\texttt{)} \cong_{exp} \texttt{QFB.remove((}b\texttt{,}f\texttt{))} : \texttt{int*}t,$$

   taking t equality to be the relation R.

Observe that the latter two statements amount to the assertion that the operations *preserve* the relation R — they map related input queues to related output queues. It is in this sense that we say that R is a bisimulation, for we are showing that the operations from `QL` simulate, and are simulated by, the operations from `QFB`, up to the relationship R between their representations.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that `QL.empty` R `QFB.empty`, it suffices to show that

   $$\texttt{nil@rev(nil)} \cong_{exp} \texttt{nil} : \texttt{int list},$$

   which is obvious from the definitions of append and reverse.

2. For `insert`, we assume that $m \cong_{val} n$: `int` and $l\,R$ `(`b`,`f`)`, and prove that

   $$\texttt{QL.insert(}m\texttt{,}l\texttt{)}\;R\;\texttt{QFB.insert(}n\texttt{,(}b\texttt{,}f\texttt{))}.$$

   By the definition of `QL.insert`, the left-hand side is equivalent to m`::`l, and by the definition of `QR.insert`, the right-hand side is equivalent to `(`n`::`b`,`f`)`. It suffices to show that

   $$m\texttt{::}l \cong_{exp} (n\texttt{::}b)\texttt{@rev(}f\texttt{)} : \texttt{int list}.$$

   Calculating, we obtain

   $$\begin{array}{rcl} (n\texttt{::}b)\texttt{@rev(}f\texttt{)} & \cong_{exp} & n\texttt{::(}b\texttt{@rev(}f\texttt{))} \\ & \cong_{exp} & n\texttt{::}l \end{array}$$

   since $l \cong_{exp} b$`@rev(`f`)`. Since $m \cong_{val} n$: `int`, it follows that $m = n$, which completes the proof.

3. For `remove`, we assume that l is related by R to (b,f), which is to say that $l \cong_{exp} b\texttt{@rev}(f)$. We are to show

   $$\texttt{QL.remove}(l) \cong_{exp} \texttt{QFB.remove}((b,f)) : \texttt{int*}t,$$

   taking t equality to be the relation R. Assuming that the queue is non-empty, so that the `remove` is defined, we have $l \cong_{exp} l'\texttt{@}[m]$ for some l' and m. We proceed by cases according to whether or not f is empty. If f is non-empty, then $f \cong_{exp} n::f'$ for some n and f'. Then by the definition of `QFB.remove`,

   $$\texttt{QFB.remove}((b,f)) \cong_{exp} (n,(b,f')) : \texttt{int*}t,$$

   relative to R. We must show that

   $$(m,l') \cong_{exp} (n,(b,f')) : \texttt{int*}t,$$

   relative to R. This means that we must show that $m = n$ and $l' \cong_{exp} b\texttt{@rev}(f') : \texttt{int list}$.

   Calculating from our assumptions,

   $$\begin{array}{rcl} l & = & l'\texttt{@}[m] \\ & = & b\texttt{@rev}(f) \\ & = & b\texttt{@rev}(n::f') \\ & = & b\texttt{@}(\texttt{rev}(f')\texttt{@}[n]) \\ & = & (b\texttt{@rev}(f'))\texttt{@}[n] \end{array}$$

   From this the result follows. Finally, if f is empty, then $b \cong_{exp} b'\texttt{@}[n]$ for some b' and n. But then $\texttt{rev}(b) \cong_{exp} n::\texttt{rev}(b')$, which reduces to the case for f non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.

Part XVI

Concurrency

Chapter 43

Process Calculus

Chapter 44

Cooperative Threads

In Chapter 26 we introduced the concept of symmetric coroutines that explicitly hand-off control between them according to a a fixed pattern of interaction. A difficulty with this style of programming is that the pattern of interaction between the producer and consumer is "hardwired" into the code. But what if there are multiple producers? Or multiple consumers? How could we generalize to permit preemption or asynchronous events?

An elegant solution to these problems is to generalize the notion of a coroutine to the notion of a *cooperative thread*. As with coroutines, threads enjoy a symmetric relationship among one another, but, unlike coroutines, they do not explicitly hand off control amongst themselves. Instead threads run as coroutines of a *scheduler* that mediates interaction among the threads, deciding which to run next based on considerations such as priority relationships or availability of data. Threads yield control to the scheduler, which determines which other thread should run next, rather than explicitly handing control to another thread.

Here is a simple interface for a user-level threads package, written in Standard ML.

```
signature THREADS = sig
  exception NoMoreThreads
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> 'a
end
```

The function `fork` is called to create a new thread executing the body of the given function. The function `yield` is called to cede control to another thread, selected by the thread scheduler. The function `exit` is called to terminate a thread.

A thread is a value of type `unit cont`. The scheduler maintains a queue of threads that are ready to execute. To dispatch the scheduler dequeues a thread from the ready queue and invokes it by throwing `()` to it. Forking is implemented by creating a new thread. Yielding is achieved by enqueueing the current thread and dispatching; exiting is a simple dispatch, abandoning the current thread entirely. This implementation is suggestive of a slogan coined by Olin Shivers: "A thread is a trajectory through continuation space". During its lifetime a thread of control is represented by a succession of continuations that are processed by the scheduler.

Here is a simple implementation of the thread package just described, again written in Standard ML.

```
structure Threads :> THREADS = struct
    open SMLofNJ.Cont
    exception NoRunnableThreads
    type thread = unit cont
    val readyQueue : thread Queue.queue = Queue.mkQueue()
    fun dispatch () =
        let
            val t = Queue.dequeue readyQueue
                    handle Queue.Dequeue => raise NoRunnableThreads
        in
            throw t ()
        end
    fun exit () = dispatch()
    fun enqueue t = Queue.enqueue (readyQueue, t)
    fun fork f =
        callcc (fn parent => (enqueue parent; f (); exit()))
    fun yield () =
        callcc (fn parent => (enqueue parent; dispatch()))
end
```

Using the above thread interface we may implement a simple producer-consumer example, using a buffer variable to communicate between them, as follows:

```
structure Client = struct
    open Threads
    val buffer : int ref = ref (~1)
    fun producer (n) =
        (buffer := n ; yield () ; producer (n+1))
    fun consumer () =
        (print (Int.toString (!buffer)); yield (); consumer())
    fun run () =
        (fork (consumer); producer 0)
end
```

This example is excessively naïve, however, in that it relies on the strict FIFO ordering of threads by the scheduler, allowing careful control over the order of execution. If, for example, the producer were to run several times in a row before the consumer could run, several numbers would be omitted from the output.

Here is a better solution that avoids this problem (but does so by "busy waiting"):

```
structure Client = struct
    open Threads
    val buffer : int option ref = ref NONE
    fun producer (n) =
        (case !buffer
           of NONE => (buffer := SOME n ; yield() ; producer (n+1))
            | SOME _ => (yield (); producer (n)))
    fun consumer () =
        (case !buffer
           of NONE => (yield (); consumer())
            | SOME n =>
              (print (Int.toString n); buffer := NONE; yield(); consumer()))
    fun run () =
        (fork (consumer); producer 0)
end
```

There is much more to be said about threads! For now, the main idea is to give a flavor of how first-class continuations can be used to implement a user-level threads package with very little difficulty. A more complete implementation is, of course, somewhat more complex, but not much more.

We can easily provide all that is necessary for sophisticated thread programming in a few hundred lines of code.

44.1 Exercises

Chapter 45

Concurrent ML

Part XVII

Storage Management

Chapter 46

Storage Management

The dynamic semantics for MinML given in Chapter 12, and even the C-machine given in Chapter 23, ignore questions of storage management. In particular, all values, be they integers, booleans, functions, or tuples, are treated the same way. But this is unrealistic. Physical machines are capable of handling only rather "small" values, namely those that can fit into a word. Thus, while it is reasonable to treat, say, integers and booleans as values directly, it is unreasonable to do the same with "large" objects such as tuples or functions.

In this chapter we consider an extension of the C-machine to account for storage management. We proceed in two steps. First, we give an abstract machine, called the A-machine, that includes a *heap* for allocating "large" objects. This introduces the problem of *garbage*, storage that is allocated for values that are no longer needed by the program. This leads to a discussion of *automatic storage management*, or *garbage collection*, which allows us to reclaim unused storage in the heap.

46.1 The A Machine

The A-machine is defined for an extension of MinML in which we add an additional form of expression, a *location*, l, which will serve as a "reference" or "pointer" into the heap.

Values are classified into two categories, *small* and *large*, by the following rules:

$$\frac{(l \in Loc)}{l \ \mathsf{svalue}} \tag{46.1}$$

$$\frac{(n \in \mathbb{Z})}{n \ \mathsf{svalue}} \tag{46.2}$$

$$\frac{}{\texttt{true} \ \mathsf{svalue}} \tag{46.3}$$

$$\frac{}{\texttt{false} \ \mathsf{svalue}} \tag{46.4}$$

$$\frac{x \ \mathsf{var} \quad y \ \mathsf{var} \quad e \ \mathsf{exp}}{\texttt{fun}\, x\ (y\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\ \texttt{is}\, e \ \mathsf{lvalue}} \tag{46.5}$$

A state of the A-machine has the form (H, k, e), where H is a *heap*, a finite function mapping locations to large values, k is a *control stack*, and e is an expression. A heap H is said to be *self-contained* iff $\mathrm{FL}(H) \subseteq \mathrm{dom}(H)$, where $\mathrm{FL}(H)$ is the set of locations occuring free in any location in H, and $\mathrm{dom}\, H$ is the domain of H.

Stack frames are similar to those of the C-machine, but refined to account for the distinction between small and large values.

$$\frac{e_2 \ \mathsf{exp}}{+(-, e_2) \ \mathsf{frame}} \tag{46.6}$$

$$\frac{v_1 \ \mathsf{svalue}}{+(v_1, -) \ \mathsf{frame}} \tag{46.7}$$

(There are analogous frames associated with the other primitive operations.)

$$\frac{e_1 \ \mathsf{exp} \quad e_2 \ \mathsf{exp}}{\texttt{if} - \texttt{then}\, e_1\ \texttt{else}\, e_2 \ \mathsf{frame}} \tag{46.8}$$

$$\frac{e_2 \ \mathsf{exp}}{\texttt{apply(}-, e_2\texttt{)} \ \mathsf{frame}} \tag{46.9}$$

$$\frac{v_1 \ \mathsf{svalue}}{\texttt{apply(}v_1, -\texttt{)} \ \mathsf{frame}} \tag{46.10}$$

Notice that v_1 is required to be a *small* value; a function is represented by a location in the heap, which is small.

As with the C-machine, a stack is a sequence of frames:

$$\frac{}{\varepsilon \ \mathsf{stack}} \tag{46.11}$$

$$\frac{f \ \mathsf{frame} \quad k \ \mathsf{stack}}{f;k \ \mathsf{stack}} \tag{46.12}$$

The dynamic semantics of the A-machine is given by a set of rules defining the transition relation $(H,k,e) \longmapsto_{\mathsf{A}} (H',k',e')$. The rules are similar to those for the C-machine, except for the treatment of functions.

Arithmetic expressions are handled as in the C-machine:

$$(H,k,+(e_1,e_2)) \longmapsto_{\mathsf{A}} (H,+(-,e_2);k,e_1) \tag{46.13}$$

$$(H,+(-,e_2);k,v_1) \longmapsto_{\mathsf{A}} (H,+(v_1,-);k,e_2) \tag{46.14}$$

$$(H,+(n_1,-);k,n_2) \longmapsto_{\mathsf{A}} (H,k,n_1+n_2) \tag{46.15}$$

Note that the heap is simply "along for the ride" in these rules.

Booleans are also handled similarly to the C-machine:

$$\begin{array}{c}(H,k,\mathtt{if}\,e\,\mathtt{then}\,e_1\,\mathtt{else}\,e_2)\\ \longmapsto_{\mathsf{A}}\\ (H,\mathtt{if}-\mathtt{then}\,e_1\,\mathtt{else}\,e_2;k,e)\end{array} \tag{46.16}$$

$$(H,\mathtt{if}-\mathtt{then}\,e_1\,\mathtt{else}\,e_2;k,\mathtt{true}) \longmapsto_{\mathsf{A}} (H,k,e_1) \tag{46.17}$$

$$(H,\mathtt{if}-\mathtt{then}\,e_1\,\mathtt{else}\,e_2;k,\mathtt{false}) \longmapsto_{\mathsf{A}} (H,k,e_2) \tag{46.18}$$

Here again the heap plays no essential role.

The real difference between the C-machine and the A-machine is in the treatment of functions. A function expression is no longer a (small) value, but rather requires an execution step to allocate it on the heap.

$$\begin{array}{c}(H,k,\mathtt{fun}\,x\,(y{:}\tau_1){:}\tau_2\,\mathtt{is}\,e)\\ \longmapsto_{\mathsf{A}}\\ (H[l \mapsto \mathtt{fun}\,x\,(y{:}\tau_1){:}\tau_2\,\mathtt{is}\,e],k,l)\end{array} \tag{46.19}$$

where l is chosen so that $l \notin \operatorname{dom} H$.

Evaluation of the function and argument position of an application is handled similarly to the C-machine.

$$(H, k, \texttt{apply}(e_1, e_2)) \longmapsto_{\mathsf{A}} (H, \texttt{apply}(-, e_2); k, e_1) \tag{46.20}$$

$$(H, \texttt{apply}(-, e_2); k, v_1) \longmapsto_{\mathsf{A}} (H, \texttt{apply}(v_1, -); k, e_2) \tag{46.21}$$

Execution of a function call differs from the corresponding C-machine instruction in that the function must be retrieved from the heap in order to determine the appropriate instance of its body. Notice that the *location* of the function, and not the function itself, is substituted for the function variable!

$$\frac{v_1 \ \mathsf{loc} \quad H(v_1) = \texttt{fun}\, f\ (x : \tau_1) : \tau_2\ \texttt{is}\, e}{(H, \texttt{apply}(v_1, -); k, v_2) \longmapsto_{\mathsf{A}} (H, k, [f, x \leftarrow v_1, v_2]e)} \tag{46.22}$$

The A-machine preserves self-containment of the heap. This follows from observing that whenever a location is allocated, it is immediately given a binding in the heap, and that the bindings of heap locations are simply those functions that are encountered during evaluation.

Lemma 46.1
If H is self-contained and $(H, k, e) \longmapsto_{\mathsf{A}} (H', k', e')$, then H' is also self-contained. Moreover, if $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \operatorname{dom} H$, then $\mathrm{FL}(k') \cup \mathrm{FL}(e') \subseteq \operatorname{dom} H'$.

It is not too difficult to see that the A-machine and the C-machine have the same "observable behavior" in the sense that both machines determine the same value for closed expressions of integer type. However, it is somewhat technically involved to develop a precise correspondence. The main idea is to define the *heap expansion* of an A-machine state to be the C-machine state obtained by replacing all locations in the stack and expression by their values in the heap. (It is important to take care that the locations occurring in a value stored are themselves replaced by their values in the heap!) We then prove that an A-machine state reaches a final state in accordance with the transition rules of the A-machines iff its expansion does in accordance with the rules of the C-machine. Finally, we observe that the value of a final state of integer type is the same for both machines.

Formally, let $\widehat{H}(e)$ stand for the substitution

$$[l_1, \ldots, l_n \leftarrow H(l_1), \ldots, H(l_n)]e,$$

where $\operatorname{dom} H = \{ l_1, \ldots, l_n \}$. Similarly, let $\widehat{H}(k)$ denote the result of performing this substitution on every expression occurring in the stack k.

Theorem 46.2
If $(H, k, e) \longmapsto_{\mathsf{A}} (H', k', e')$, then $\widehat{H}(k) \mathbin{@} \widehat{H}(e) \longmapsto_{\mathsf{C}}^{0,1} \widehat{H'}(k') \mathbin{@} \widehat{H'}(e')$.

Notice that the allocation of a function in the A-machine corresponds to zero steps of execution on the C-machine, because in the latter case functions are values.

46.2 Garbage Collection

The purpose of the A-machine is to model the memory allocation that would be required in an implementation of MinML. This raises the question of *garbage*, storage that is no longer necessary for a computation to complete. The purpose of a *garbage collector* is to reclaim such storage for further use. Of course, in a purely abstract model there is no reason to perform garbage collection, but in practice we must contend with the limitations of finite, physical computers. For this reason we give a formal treatment of garbage collection for the A-machine.

The crucial issue for any garbage collector is to determine which locations are unnecessary for computation to complete. These are deemed garbage, and are reclaimed so as to conserve memory. But when is a location unnecessary for a computation to complete? Consider the A-machine state (H, k, e). A location $l \in \operatorname{dom}(H)$ is *unnecessary*, or *irrelevant*, for this machine state iff execution can be completed without referring to the contents of l. That is, $l \in \operatorname{dom} H$ is unnecessary iff $(H, k, e) \longmapsto_{\mathsf{A}}^{*} (H', \varepsilon, v)$ iff $(H_l, k, e) \longmapsto_{\mathsf{A}}^{*} (H'', \varepsilon, v)$, where H_l is H with the binding for l removed, and H'' is some heap.

Unfortunately, a machine cannot decide whether a location is unnecessary!

Theorem 46.3
It is mechanically undecidable whether or not a location l is unnecessary for a given state of the A*-machine.*

Intuitively, we cannot decide whether l is necessary without actually running the program. It is not hard to formulate a reduction from the halting problem to prove this theorem: simply arrange that l is used to complete a computation iff some given Turing machine diverges on blank input.

Given this fundamental limitation, practical garbage collectors must employ a *conservative approximation* to determine which locations are unnecessary in a given machine state. The most popular criterion is based on *reachability*. A location l_n is *unreachable*, or *inaccessible*, iff there is no sequence of locations $l_1, \ldots, l_n$ such that l_1 occurs in either the current expression or on the control stack, and l_i occurs in l_{i+1} for each $1 \leq i < n$.

Theorem 46.4
If a location l is unreachable in a state (H, k, e), then it is also unnecessary for that state.

Each transition depends only on the locations occurring on the control stack or in the current expression. Some steps move values from the heap onto the stack or current expression. Therefore in a multi-step sequence, execution can depend only on reachable locations in the sense of the definition above.

The set of unreachable locations in a state may be determined by *tracing*. This is easily achieved by an iterative process that maintains a finite set of of locations, called the *roots*, containing the locations that have been found to be reachable up to that point in the trace. The root set is initialized to the locations occurring in the expression and control stack. The tracing process completes when no more locations can be added. Having found the reachable locations for a given state, we then deem all other heap locations to be unreachable, and hence unnecessary for computation to proceed. For this reason the reachable locations are said to be *live*, and the unreachable are said to be *dead*.

Essentially *all* garbage collectors used in practice work by tracing. But since reachability is only a conservative approximation of necessity, *all practical collectors are conservative*! So-called conservative collectors are, in fact, *incorrect* collectors that may deem as garbage storage that is actually necessary for the computation to proceed. Calling such a collector "conservative" is misleading (actually, wrong), but it is nevertheless common practice in the literature.

The job of a garbage collector is to dispose of the unreachable locations in the heap, freeing up memory for later use. In an abstract setting where we allow for heaps of unbounded size, it is never necessary to collect garbage, but of course in practical situations we cannot afford to waste unlimited amounts of storage. We will present an abstract model of a particular form of garbage collection, called *copying collection*, that is widely used in practice. The goal is to present the main ideas of copying collection, and to prove that garbage collection is semantically "invisible" in the sense that it does not change the outcome of execution.

The main idea of copying collection is to simultaneously determine which locations are reachable, and to arrange that the contents of all reachable locations are preserved. The rest are deemed garbage, and are reclaimed. In a copying collector this is achieved by partitioning storage into two parts, called *semi-spaces*. During normal execution allocation occurs in one of the two semi-spaces until it is completely filled, at which point the collector is invoked. The collector proceeds by copying all reachable storage from the current, filled semi-space, called the *from space*, to the other semi-space, called the *to space*. Once this is accomplished, execution continues using the "to space" as the new heap, and the old "from space" is reclaimed in bulk. This exchange of roles is called a *flip*.

By copying all and only the reachable locations the collector ensures that unreachable locations are reclaimed, and that no reachable locations are lost. Since reachability is a conservative criterion, the collector may preserve more storage than is strictly necessary, but, in view of the fundamental undecidability of necessity, this is the price we pay for mechanical collection. Another important property of copying collectors is that their execution time is proportion to the size of the live data; no work is expended manipulating reclaimable storage. This is the fundamental motivation for using semi-spaces: once the reachable locations have been copied, the unreachable ones are eliminated by the simple measure of "flipping" the roles of the spaces. Since the amount of work performed is proportional to the live data, we can amortize the cost of collection across the allocation of the live storage, so that garbage collection is (asymptotically) "free". However, this benefit comes at the cost of using only half of available memory at any time, thereby doubling the overall storage required.

Copying garbage collection may be formalized as an abstract machine with states of the form (H_f, S, H_t), where H_f is the " from" space, H_t is the "to" space, and S is the *scan set*, the set of reachable locations. The initial state of the collector is $(H, S, \emptyset)$, where H is the "current" heap and $\emptyset \neq S \subseteq \text{dom}(H_f)$ is the set of locations occurring in the program or control stack. The final state of the collector is $(H_f, \emptyset, H_t)$, with an empty scan set.

The collector is invoked by adding the following instruction to the A-machine:

$$\frac{(H, \text{FL}(k) \cup \text{FL}(e), \emptyset) \longmapsto^*_{\mathsf{G}} (H'', \emptyset, H')}{(H, k, e) \longmapsto_{\mathsf{A}} (H', k, e)} \tag{46.23}$$

The scan set is initialized to the set of free locations occurring in either the current stack or the current expression. These are the locations that are immediately reachable in that state; the collector will determine those that are transitively reachable, and preserve their bindings. Once the collector has finished, the "to" space is installed as the new heap.

Note that a garbage collection can be performed at any time! This correctly models the unpredictability of collection in an implementation, but avoids specifying the exact criteria under which the collector is invoked. As mentioned earlier, this is typically because the current heap is exhausted, but in an abstract setting we impose no fixed limit on heap sizes, preferring instead to simply allow collection to be performed spontaneously according to unspecified criteria.

The collection machine is defined by the following two rules:

$$\frac{}{(H_f[l = v], S \cup \{ l \}, H_t) \longmapsto_{\mathsf{G}} (H_f, S \cup \text{FL}(v), H_t[l = v])} \tag{46.24}$$

$$\frac{}{(H_f, S \cup \{ l \}, H_t[l = v]) \longmapsto_{\mathsf{G}} (H_f, S, H_t[l = v])} \tag{46.25}$$

The first rule copies a reachable binding in the "from" space to the "to" space, and extends the scan set to include those locations occurring in the copied value. This ensures that we will correctly preserve those locations that occur in a reachable location. The second rule throws away any location in the scan set that has already been copied. This rule is necessary because when the scan set is updated by the free locations of a heap value, we may add locations that have already been copied, and we do not want to copy them twice!

The collector is governed by a number of important invariants.

1. The scan set contains only "valid" locations: $S \subseteq \text{dom}\, H_f \cup \text{dom}\, H_t$;
2. The "from" and "to" space are disjoint: $\text{dom}\, H_f \cap \text{dom}\, H_t = \emptyset$;
3. Every location in "to" space is either in "to" space, or in the scan set: $\text{FL}(H_t) \subseteq S \cup \text{dom}\, H_t$;
4. Every location in "from" space is either in "from" or "to" space: $\text{FL}(H_f) \subseteq \text{dom}\, H_f \cup \text{dom}\, H_t$.

The first two invariants are minimal "sanity" conditions; the second two are crucial to the operation of the collector. The third states that the "to" space contains only locations that are either already copied into "to" space, or will eventually be copied, because they are in the scan set, and hence in "from" space (by disjointness). The fourth states that locations in "from" space contain only locations that either have already been copied or are yet to be copied.

These invariants are easily seen to hold of the initial state of the collector, since the "to" space is empty, and the "from" space is assumed to be self-contained. Moreover, if these invariants hold of a final state, then $\text{FL}(H_t) \subseteq \text{dom}\, H_t$, since $S = \emptyset$ in that case. Thus the heap remains self-contained after collection.

Theorem 46.5 (Preservation of Invariants)
If the collector invariants hold of (H_f, S, H_t) and $(H_f, S, H_t) \longmapsto_{\mathsf{G}} (H_f', S', H_t')$, then the same invariants hold of (H_f', S', H_t').

The correctness of the collector follows from the following lemma.

Lemma 46.6
If $(H_f, S, H_t) \longmapsto_{\mathsf{G}} (H_f', S', H_t')$, then $H_f \cup H_t = H_f' \cup H_t'$ and $S \cup \text{dom}\, H_t \subseteq S' \cup \text{dom}\, H_t'$.

The first property states that the union of the semi-spaces never changes; bindings are only copied from one to the other. The second property states that the domain of the "to" space together with the scan set does not change.

From this lemma we obtain the following crucial facts about the collector. Let $S = \mathrm{FL}(k) \cup \mathrm{FL}(e)$, and suppose that

$$(H, S, \emptyset) \longmapsto^{*}_{\mathsf{G}} (H'', \emptyset, H').$$

Then we have the following properties:

1. The reachable locations are bound in H': $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \operatorname{dom} H'$. This follows from the lemma, since the inital "to" space and the final scan set are empty.

2. The reachable data is correctly copied: $H' \subseteq H$. This follows from the lemma, which yields $H = H'' \cup H'$.