Practical Foundations for Programming Languages

Robert Harper
Carnegie Mellon University

Spring, 2008

[Draft of July 8, 2008 at 11:12pm.]

Preface

This is a working draft of a book on the foundations of programming languages. The central organizing principle of the book is that programming language features may be seen as manifestations of an underlying type structure that governs its syntax and semantics. The emphasis, therefore, is on the concept of *type*, which codifies and organizes the computational universe in much the same way that the concept of *set* may be seen as an organizing principle for the mathematical universe. The purpose of this book is to explain this remark.

This is very much a work in progress, with major revisions made nearly every day. This means that there may be internal inconsistencies as revisions to one part of the book invalidate material at another part. Please bear this in mind!

Corrections, comments, and suggestions are most welcome, and should be sent to the author at `rwh@cs.cmu.edu`.

Contents

XII State 279

XIII Modalities for Effects 305

Part I

Judgements and Rules

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 Objects and Judgements

We start with the notion of a *judgement*, or *assertion*, about an *object* of study. We shall make use of many forms of judgement, including examples such as these:

$n\ \mathsf{nat}$	n is a natural number
$n = n_1 + n_2$	n is the sum of n_1 and n_2
$a\ \mathsf{ast}$	a is an abstract syntax tree
$\tau\ \mathsf{type}$	τ is a type
$e : \tau$	expression e has type τ
$e \Downarrow v$	expression e has value v

A judgement states that one or more objects have a property or stand in some relation to one another. The property or relation itself is called a *judgement form*, and the judgement that an object or objects have that property or stand in that relation is said to be an *instance* of that judgement form. A judgement form is also called a *predicate*, and the objects constituting an instance are its *subjects*.

We use the meta-variable C to stand for an unspecified judgement form, and the meta-variables a, b, and c to stand for unspecified objects. We write $a\ C$ for the judgement asserting that C holds of a. When it is not important to stress the subject of the judgement, we write J to stand for an unspecified judgement. For particular judgement forms, we freely use prefix, infix, or mixfix notation, as illustrated by the above examples, in order to enhance readability.

We are being deliberately vague about the universe of objects that may be involved in an inductive definition. The rough-and-ready rule is that any sort of finite construction of objects from other objects is permissible. For example, we shall assume that the universe of objects is closed under the formation of finite n-tuples of objects, written $(a_1, \ldots, a_n)$, where the a_i's are objects. More generally we permit the formation of trees in which nodes are operators that construct a tree from finitely many other trees constructed by a similar process. It will not be necessary to be more exacting than this, but please see Section 1.8 on page 13 for further discussion of foundational issues.

1.2 Inference Rules

An *inductive definition* consists of a collection of *rules* of the form

$$\frac{J_1 \quad \ldots \quad J_k}{J}\ , \tag{1.1}$$

where J and the J_i's are judgements. The assertions above the horizontal line are called the *premises* of the rule, and the judgement below the line is called its *conclusion*. If a rule has no premises (that is, when k is zero), the rule is an *axiom*; otherwise it is a *proper rule*.

An inference rule may be read as an implication stating that the premises are *sufficient* for the conclusion: to show J, it is enough to show $J_1, \ldots, J_k$. When k is zero, a rule states that its conclusion holds unconditionally. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

For example, the following rules constitute an inductive definition of the judgement a $\mathtt{nat}$:

$$\frac{}{\mathtt{zero}\ \mathtt{nat}} \tag{1.2a}$$

$$\frac{a\ \mathtt{nat}}{\mathtt{succ}(a)\ \mathtt{nat}} \tag{1.2b}$$

These rules specify that a $\mathtt{nat}$ holds whenever either a is $\mathtt{zero}$, or a is $\mathtt{succ}(b)$ where b $\mathtt{nat}$. Taking these rules to be exhaustive, it follows that a $\mathtt{nat}$ iff a is a natural number.

Similarly, the following rules constitute an inductive definition of the judgement a tree:

$$\frac{}{\texttt{empty}\ \texttt{tree}} \tag{1.3a}$$

$$\frac{a_1\ \texttt{tree} \quad a_2\ \texttt{tree}}{\texttt{node}(a_1;a_2)\ \texttt{tree}} \tag{1.3b}$$

These rules specify that a tree holds if either a is empty, or a is node(a_1;a_2), where a_1 tree and a_2 tree. Taking these to be exhaustive, these rules state that a is a binary tree, which is to say it is either empty, or a node consisting of two children, each of which is also a binary tree.

The judgement $a = b$ nat defining equality of a nat and b nat is inductively defined by the following rules:

$$\frac{}{\texttt{zero} = \texttt{zero}\ \texttt{nat}} \tag{1.4a}$$

$$\frac{a = b\ \texttt{nat}}{\texttt{succ}(a) = \texttt{succ}(b)\ \texttt{nat}} \tag{1.4b}$$

Strictly speaking, each of the preceding examples consists of infinitely many rules, one for each choice of object appearing in the rules. We specify an infinite family of rules using a *rule scheme* involving one or more *parameters* that range over objects. An *instance* of a rule scheme is a rule obtained by systematically replacing the parameters by objects. A rule scheme always stands for the collection of its instances.

When we say that a collection of rules constitutes an inductive definition of a judgement, we mean that the judgement holds *if and only if* it can be shown to do so by an application of these rules. In other words, the rules are *necessary and sufficient* for the judgement to hold. The natural "top down" reading of a rule corresponds to its sufficiency: *if* the premises hold, *then* the conclusion holds. Necessity is expressed by the implicit *extremal clause* stating that no other rule applies, or, in other words, *if* J holds, *then* it is only by virtue of the given rules. Sufficiency allows us to "work forward" to show that J holds by *deriving* it according to the rules. Necessity allows us to "work backwards" from the fact that J holds to show some property $\mathcal{P}$ of it by *induction*.

1.3 Derivations

To show that an inductively defined judgement holds, it is enough to exhibit a *derivation* of it. A derivation of a judgement is a composition of rules, starting with axioms and ending with that judgement. It may be thought of as a tree in which each node is a rule whose children are derivations of its premises. A derivation of J constitutes *evidence* for an inductively defined judgement J.

We usually depict derivations as trees with the conclusion at the bottom, and with the children of a node corresponding to a rule appearing above it as evidence for the premises of that rule. Thus, if

$$\frac{J_1 \quad \dots \quad J_k}{J}$$

is an inference rule and $\nabla_1, \dots, \nabla_k$ are derivations of its premises, then

$$\frac{\nabla_1 \quad \dots \quad \nabla_k}{J} \tag{1.5}$$

is a derivation of its conclusion. In particular, if $k = 0$, then the node has no children.

For example, here is a derivation of `succ(succ(succ(zero))) nat`:

$$\frac{\frac{\frac{\frac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}} \; . \tag{1.6}$$

Similarly, here is a derivation of `node(node(empty;empty);empty) tree`:

$$\frac{\frac{\frac{}{\texttt{empty tree}} \quad \frac{}{\texttt{empty tree}}}{\texttt{node(empty;empty) tree}} \quad \frac{}{\texttt{empty tree}}}{\texttt{node(node(empty;empty);empty) tree}} \; . \tag{1.7}$$

To show that an inductively defined judgement is derivable we need only find a derivation for it. There are two main methods for finding derivations, 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 conclusion, whereas backward chaining starts with the desired conclusion 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 queue 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 queue, and consider all rules whose conclusion is that judgement. For each such rule, we add the premises of that rule to the back of the queue. The process terminates when the queue 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, in general, no algorithmic method for determining in general whether the current goal is derivable. If it is not, we may futilely add more and more judgements to the goal set, never reaching a point at which all goals have been satisfied.

1.4 Rule Induction

Since an inductively defined judgement holds only if there is some derivation of it according to the rules, we may prove properties of such judgements by *rule induction*, or *induction on derivations*. We write $\mathcal{P}(J)$ to indicate that the property $\mathcal{P}$ holds whenever the judgement J is derivable. To show that $\mathcal{P}$ holds of all derivable J, it is enough to show that $\mathcal{P}$ is *closed* under the rules defining J. This means that for every rule of the form

$$\frac{J_1 \quad \dots J_k}{J} \ ,$$

we have

$$\textit{if } \mathcal{P}(J_1),\ \dots,\ \mathcal{P}(J_k),\ \textit{then } \mathcal{P}(J).$$

The conjunction of properties $\mathcal{P}(J_1), \dots, \mathcal{P}(J_k)$ is called the *inductive hypothesis*, and the proof of the implication itself is called the *inductive step*.

The principle of rule induction is an expression of the fact that the judgement, J, inductively defined by a set of rules is the *strongest* judgement closed under those rules. Put in other terms, if J is derivable, it can only be because there is some rule with J as conclusion such that each premise of that rule is also derivable. Inductively, we may assume that $\mathcal{P}$ holds of each of the premises while showing that it also holds of the conclusion. If we can do this for every rule, then $\mathcal{P}$ must hold whenever J is derivable. Note that when a rule has no premises, we are obliged to show $\mathcal{P}$ outright.

If $\mathcal{P}(J)$ is closed under a set of rules defining a judgement J, then so is the conjunction $\mathcal{Q}(J) = \mathcal{P}(J) \wedge J$, because J is automatically closed under the rules that define it. This means that in each inductive step of a proof by rule induction we may freely assume both J_i and $\mathcal{P}(J_i)$ for each of the premises of the rule to derive $\mathcal{P}(J)$ for its conclusion. We shall generally take advantage of this without explicit comment.

As a matter of notation, when J has the form $a\ C$ for some predicate C, we sometimes write $\mathcal{P}_C(a)$, or even just $\mathcal{P}(a)$, instead of $\mathcal{P}(a\ C)$ when carrying out a proof by rule induction. For specific properties $\mathcal{P}$ we often use *ad hoc* notational conventions whose meaning should be clear from context.

When specialized to Rules (1.2), the principle of rule induction states that to show $\mathcal{P}(a\ \mathsf{nat})$ whenever $a\ \mathsf{nat}$, it is enough to show:

1. $\mathcal{P}(\texttt{zero}\ \mathsf{nat})$.
2. $\mathcal{P}(\texttt{succ}(a)\ \mathsf{nat})$, assuming $\mathcal{P}(a\ \mathsf{nat})$.

This is just the familiar principle of *mathematical induction* arising as a special case of rule induction.

Similarly, rule induction for Rules (1.3) states that to show $\mathcal{P}(a\ \mathsf{tree})$ whenever $a\ \mathsf{tree}$, it is enough to show

1. $\mathcal{P}(\texttt{empty}\ \mathsf{tree})$.
2. $\mathcal{P}(\texttt{node}(a_1;a_2)\ \mathsf{tree})$, assuming $\mathcal{P}(a_1\ \mathsf{tree})$ and $\mathcal{P}(a_2\ \mathsf{tree})$.

This is called the principle of *tree induction*, and is once again an instance of rule induction.

As a simple example of a proof by rule induction, let us prove that natural number equality as defined by Rules (1.4) is reflexive:

Lemma 1.1. *If* $a\ \mathsf{nat}$*, then* $a = a\ \mathsf{nat}$*.*

Proof. By rule induction on Rules (1.2):

Rule (1.2a) Applying Rule (1.4a) we obtain $\texttt{zero} = \texttt{zero}\ \mathsf{nat}$.

Rule (1.2b) Assume that $a = a$ nat. It follows that $\texttt{succ(}a\texttt{)} = \texttt{succ(}a\texttt{)}$ nat by an application of Rule (1.4b).

□

As another example of the use of rule induction, we may show that the predecessor of a natural number is also a natural number. While this may seem self-evident, the point of the example is to show how to derive this from first principles.

Lemma 1.2. *If* $\texttt{succ(}a\texttt{)}$ nat*, then* a nat.

Proof. It is instructive to re-state the lemma in a form more suitable for inductive proof: if b nat and b is $\texttt{succ(}a\texttt{)}$ for some a, then a nat. We proceed by rule induction on Rules (1.2).

Rule (1.2a) Vacuously true, since $\texttt{zero}$ is not of the form $\texttt{succ(}-\texttt{)}$.

Rule (1.2b) We have that b is $\texttt{succ(}b'\texttt{)}$, and we may assume both that the lemma holds for b' and that b' nat. The result follows directly, since if $\texttt{succ(}b'\texttt{)} = \texttt{succ(}a\texttt{)}$ for some a, then a is b'.

□

Similarly, let us show that the successor operation is injective.

Lemma 1.3. *If* $\texttt{succ(}a_1\texttt{)} = \texttt{succ(}a_2\texttt{)}$ nat*, then* $a_1 = a_2$ nat.

Proof. It is instructive to re-state the lemma in a form more directly amenable to proof by rule induction. We are to show that if $b_1 = b_2$ nat then if b_1 is $\texttt{succ(}a_1\texttt{)}$ and b_2 is $\texttt{succ(}a_2\texttt{)}$, then $a_1 = a_2$ nat. We proceed by rule induction on Rules (1.4):

Rule (1.4a) Vacuously true, since $\texttt{zero}$ is not of the form $\texttt{succ(}-\texttt{)}$.

Rule (1.4b) Assuming the result for $b_1 = b_2$ nat, and hence that the premise $b_1 = b_2$ nat holds as well, we are to show that if $\texttt{succ(}b_1\texttt{)}$ is $\texttt{succ(}a_1\texttt{)}$ and $\texttt{succ(}b_2\texttt{)}$ is $\texttt{succ(}a_2\texttt{)}$, then $a_1 = a_2$ nat. Under these assumptions we have b_1 is a_1 and b_2 is a_2, and so $a_1 = a_2$ nat is just the premise of the rule. (We make no use of the inductive hypothesis to complete this step of the proof.)

□

Both proofs rely on some natural assumptions about the universe of objects; see Section 1.8 on page 13 for further discussion.

1.5 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. For example, the following rules, define the judgement a list stating that a is a list of natural numbers.

$$\frac{}{\texttt{nil}\ \mathsf{list}} \tag{1.8a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\texttt{cons(}a;b\texttt{)}\ \mathsf{list}} \tag{1.8b}$$

The second rule refers to the judgement a nat defined earlier.

Frequently two or more judgements are defined at once by a *simultaneous inductive definition*. In a simultaneous inductive definition the rules may involve several different judgements, none of which may be considered defined prior to or separately from the others. The principle of rule induction applies as usual, except that the property $\mathcal{P}$ applies to each of the several judgements being defined.

For example, consider the following rules, which constitute a simultaneous inductive definition of the judgements a even, stating that a is an even natural number, and a odd, stating that a is an odd natural number:

$$\frac{}{\texttt{zero}\ \mathsf{even}} \tag{1.9a}$$

$$\frac{a\ \mathsf{odd}}{\texttt{succ(}a\texttt{)}\ \mathsf{even}} \tag{1.9b}$$

$$\frac{a\ \mathsf{even}}{\texttt{succ(}a\texttt{)}\ \mathsf{odd}} \tag{1.9c}$$

The principle of rule induction for these rules states that to show simultaneously that $\mathcal{P}(a\ \mathsf{even})$ whenever a even and $\mathcal{P}(a\ \mathsf{odd})$ whenever a odd, it is enough to show the following:

1. $\mathcal{P}(\texttt{zero}\ \mathsf{even})$;

2. if $\mathcal{P}(a\ \mathsf{odd})$, then $\mathcal{P}(\texttt{succ(}a\texttt{)}\ \mathsf{even})$;

3. if $\mathcal{P}(a\ \mathsf{even})$, then $\mathcal{P}(\texttt{succ(}a\texttt{)}\ \mathsf{odd})$.

When rewritten using $\mathcal{P}_{\mathsf{even}}(a)$ and $\mathcal{P}_{\mathsf{odd}}(a)$, these conditions are as follows:

1. $\mathcal{P}_{\mathsf{even}}(\texttt{zero})$;

2. if $\mathcal{P}_{\mathsf{odd}}(a)$, then $\mathcal{P}_{\mathsf{even}}(\texttt{succ(}a\texttt{)})$.

3. if $\mathcal{P}_{\mathsf{even}}(a)$, then $\mathcal{P}_{\mathsf{odd}}(\texttt{succ(}a\texttt{)})$;

1.6 Defining Functions by Rules

Another common use of inductive definitions is to present a function by an inductive definition of its graph, which relates its input(s) to its output, then showing separately that the graph is a function. For example, one way to define the addition function on natural numbers is to define inductively the judgement $\mathsf{sum}(a,b,c)$, with the intended meaning that c is the sum of a and b, as follows:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero},b,b)} \tag{1.10a}$$

$$\frac{\mathsf{sum}(a,b,c)}{\mathsf{sum}(\mathtt{succ}(a),b,\mathtt{succ}(c))} \tag{1.10b}$$

We then show that c is uniquely determined as a function of a and b.

Theorem 1.4. *For every a nat and b nat, there exists a unique c nat such that $\mathsf{sum}(a,b,c)$.*

Proof. The proof decomposes into two parts:

1. (Existence) If a nat and b nat, then there exists c nat such that $\mathsf{sum}(a,b,c)$.

2. (Uniqueness) If a nat, b nat, c nat, c' nat, $\mathsf{sum}(a,b,c)$, and $\mathsf{sum}(a,b,c')$, then $c = c'$ nat.

For existence, let $\mathcal{P}(a\ \mathsf{nat})$ be the proposition *if b nat then there exists c nat such that* $\mathsf{sum}(a,b,c)$. We prove that if a nat then $\mathcal{P}(a\ \mathsf{nat})$ by rule induction on Rules (1.2). We have two cases to consider:

Rule (1.2a) We are to show $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$. Assuming b nat and taking c to be b, we obtain $\mathsf{sum}(\mathtt{zero},b,c)$ by Rule (1.10a).

Rule (1.2b) Assuming $\mathcal{P}(a\ \mathsf{nat})$, we are to show $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$. That is, we assume that if b nat then there exists c such that $\mathsf{sum}(a,b,c)$, and are to show that if b' nat, then there exists c' such that $\mathsf{sum}(\mathtt{succ}(a),b',c')$. To this end, suppose that b' nat. Then by induction there exists c such that $\mathsf{sum}(a,b',c)$. Taking $c' = \mathtt{succ}(c)$, and applying Rule (1.10b), we obtain $\mathsf{sum}(\mathtt{succ}(a),b',c')$, as required.

For uniqueness, we prove that *if* $\mathsf{sum}(a,b,c_1)$*, then if* $\mathsf{sum}(a,b,c_2)$*, then* $c_1 = c_2$ nat by rule induction based on Rules (1.10).

Rule (1.10a) We have $a = \mathtt{zero}$ and $c_1 = b$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(\mathtt{zero},b,c_2)$, then c_2 is b. By Lemma 1.1 on page 8 we obtain $b = b$ nat.

Rule (1.10b) We have that $a = \texttt{succ}(a')$ and $c_1 = \texttt{succ}(c_1')$, where $\mathsf{sum}(a', b, c_1')$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(a, b, c_2)$, then $c_2 = \texttt{succ}(c_2')$ nat where $\mathsf{sum}(a', b, c_2')$. By the outer inductive hypothesis $c_1' = c_2'$ nat and so $c_1 = c_2$ nat.

□

1.7 Mode Specifications

The statement that one or more arguments of a judgement is (perhaps uniquely) determined by its other arguments is called a *mode specification* for that judgement. In the case of addition we have proved that every two natural numbers have a sum, without proving that the sum is uniquely determined by its arguments. This can be concisely stated by saying that the addition judgement has the *mode* $(\forall, \forall, \exists)$, corresponding to the proposition *for all* a nat *and for all* b nat, *there exists* c nat *such that* $\mathsf{sum}(a, b, c)$. If we wish to further specify that c is *uniquely* determined by a and b, we would say that the judgement has mode $(\forall, \forall, \exists!)$, corresponding to the proposition *for all* a nat *and for all* b nat, *there exists a unique* c nat *such that* $\mathsf{sum}(a, b, c)$. This states that the sum is a *(total) function* of its two arguments. If we wish only to specify that the sum is unique, *if it exists*, then we would say that the addition judgement has mode $(\forall, \forall, \exists^{\leq 1})$, corresponding to the proposition *for all* a nat *and for all* b nat *there exists at most one* c nat *such that* $\mathsf{sum}(a, b, c)$. In other words, this mode states that the sum is a *partial function* of its arguments, which is weaker than stating that it is a *total function*. In the case of addition there is no particular reason to settle for this weaker property, but if we were, instead, to consider the quotient operation, then the best we can do is to show that the quotient is a partial function of its dividend and divisor.

As these examples illustrate, a given judgement may satisfy several different mode specifications. In general the universally quantified arguments are to be thought of as the *inputs* of the judgement, and the existentially quantified arguments are to be thought of as its *outputs*. We usually try to arrange things so that the outputs come after the inputs, but it is not essential that we do so. For example, addition also has the mode $(\forall, \exists^{\leq 1}, \forall)$, stating that the sum and the first addend uniquely determine the second addend, if there is any such addend at all. Put in other terms, addition of natural numbers has a (partial) inverse, namely subtraction! We could equally well show that addition has mode $(\exists^{\leq 1}, \forall, \forall)$, which is just another way of stating that addition has a partial inverse over the natural numbers.

Often there is an intended, or *principal*, mode of a given judgement, which we often foreshadow by choosing our notation to reflect it. For example, when giving an inductive definition of a function, we often use equations to indicate the intended input and output relationships. For example, we may re-state the inductive definition of addition (1.10) using equations.

$$\frac{a\ \mathsf{nat}}{a + \mathtt{zero} = a\ \mathsf{nat}} \tag{1.11a}$$

$$\frac{a + b = c\ \mathsf{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c)\ \mathsf{nat}} \tag{1.11b}$$

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 $a + b$ for the unique c such that $a + b = c\ \mathsf{nat}$.

1.8 Foundations

An inductively defined judgement form, such as $a\ \ \mathsf{nat}$, may be seen as "carving out" a particular class of objects from an (as yet unspecified) *universe of discourse* that is rich enough to include the objects in question. That is, among the objects in the universe, the judgement $a\ \ \mathsf{nat}$ isolates those objects of the form $\mathtt{succ}(\ldots\mathtt{succ}(\mathtt{zero})\ldots)$. But what, precisely, are these objects? And what sorts of objects are permissible in an inductive definition?

One answer to these questions is to fix in advance a particular set to serve as the universe over which all inductive definitions are to take place. This set must be proved to exist on the basis of the standard axioms of set theory, and the objects that we wish to use in our inductive definitions must be encoded as elements of this set. But what set shall we choose as our universe? And how are the various objects of interest encoded within it?

At the least we wish to include all possible *finitary trees* whose nodes are labelled by an element of an infinite set of *operators*. The object succ(succ(zero)) is a tree of height two whose root is labelled with the operator succ and whose sole child is also so labelled and has a child labelled zero. Judgements with multiple arguments, such as $a + b = c\ \mathsf{nat}$, may be handled by demanding that the universe also be closed under formation of finite tuples $(a_1, \ldots, a_n)$ of objects. One may consider other forms of objects, such as *infinitary trees*, whose nodes may have infinitely many children, or *regular trees*, whose nodes may have ancestors as children, but we shall not have need of these in our work.

To construct a set of finitary objects requires that we fix a representation of trees and tuples as certain sets. This can be done, but the results are notoriously unenlightening.[1] Instead we shall simply assert that such a set exists (*i.e.*, can be constructed from the standard axioms of set theory). The construction should ensure that we can construct any finitary tree, and, given any finitary tree, determine the operator at its root and the set of trees that are its children.

While many will feel more secure by working within set theory, it is important to keep in mind that accepting the axioms of set theory is far more dubious, foundationally speaking, than just accepting the existence of finitary trees indepdendently of their representation as sets. Moreover, there is a significant disadvantage to working with sets, because doing so complicates the argument for the computability of our constructions. If we use abstract sets to model computational phenomena, we incur the additional burden of showing that these set-theoretic constructions can all be implemented on a computer. In contrast, it is intuitively clear how to represent finitary trees on a computer, and how to compute with them by recursion, so no further explanation is required.

1.9 Exercises

1. Give an inductive definition of the judgement $\mathsf{max}(a, b, c)$, where a nat, b nat, and c nat, with the meaning that c is the larger of a and b. Prove that this judgement has the mode $(\forall, \forall, \exists!)$.

2. Consider the following rules, which define the height of a binary tree as the judgement $\mathsf{hgt}(a; b)$.

$$\frac{}{\mathsf{hgt}(\texttt{empty};\texttt{zero})} \tag{1.12a}$$

$$\frac{\mathsf{hgt}(a_1; b_1) \quad \mathsf{hgt}(a_2; b_2) \quad \mathsf{max}(b_1, b_2, b)}{\mathsf{hgt}(\texttt{node}(a_1;a_2);\texttt{succ}(b))} \tag{1.12b}$$

   Prove by tree induction that the judgement $\texttt{hgt}$ has the mode $(\forall, \exists!)$, with inputs being binary trees and outputs being natural numbers.

[1] Perhaps you have seen the definition of the natural number 0 as the empty set, $\emptyset$, and the number $n + 1$ as the set $n \cup \{ n \}$, or the definition of the ordered pair $\langle a, b \rangle = \{ a, \{ a, b \} \}$. Similar coding tricks can be used to represent any finitary tree.

3. Give an inductive definition of the judgement “∇ is a derivation of J” for an arbitrary inductively defined judgement J.

4. Give an inductive definition of the forward-chaining and backward-chaining search strategies.

Chapter 2

Hypothetical Judgements

A *categorical* judgement is an unconditional assertion about some object of the universe. The inductively defined judgements given in Chapter 1 are all categorical. A *hypothetical judgement* is made on the basis of one or more *hypotheses*, or *assumptions*, that entail a *consequent*. We will consider two forms of hypothetical judgement, the *derivability* judgement and the *admissibility* judgement, which are both defined relative to some fixed set of rules. These two forms of hypothetical judgement share a common set of *structural properties* that characterize reasoning under hypotheses. These properties are central to the extension of inductive definitions to admit rules with hypothetical judgements as premises.

2.1 Derivability

For a given set of rules defining a collection of categorical judgements, we define the *derivability* judgement, written $J \vdash K$, where J and K are categorical judgements, to mean that we may derive the judgement K from the extension of our rule set with J as a new axiom (*i.e.*, a rule without premises having J as conclusion). The assertion J is called the *hypothesis*, and K the *consequent* of the hypothetical judgement.

The hypothetical judgement is naturally extended to permit K to be hypothetical to obtain the *iterated* form

$$J_1 \vdash J_2 \vdash \dots J_n \vdash K, \tag{2.1}$$

which we abbreviate to

$$J_1, \dots, J_n \vdash K. \tag{2.2}$$

We often use Γ to stand for a finite sequence of assertions, writing $\Gamma \vdash K$ to mean that K is derivable from the judgements Γ.

There is a close correspondence between inference rules and derivability judgements. Each inference rule defining a judgement form gives rise to a valid derivability judgement. For if

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{2.3}$$

is a primitive rule, then the judgement $J_1, \ldots, J_n \vdash J$ is valid, since adding the hypotheses as axioms enables us to apply the displayed rule to derive the conclusion of that rule. Conversely, if $J_1, \ldots, J_n \vdash J$ is valid, then there is a derivation of J obtained by composing rules starting with the hypotheses J_i as axioms. Equivalently, we say that the inference rule (2.3) is *derivable* iff $J_1, \ldots, J_n \vdash J$. The derivation of J is essentially a compound inference rule with the J_i's as premises and J as conclusion.

For example, the derivability judgement

$$a \ \mathsf{nat} \vdash \mathtt{succ(succ(}a\mathtt{))} \ \mathsf{nat} \tag{2.4}$$

is valid according to Rules (1.2). For if we regard the premise a nat as a new axiom, then we may derive succ(succ(a)) nat from it according to those rules, as follows:

$$\frac{\dfrac{\dfrac{}{a \ \mathsf{nat}}}{\mathtt{succ(}a\mathtt{)} \ \mathsf{nat}}}{\mathtt{succ(succ(}a\mathtt{))} \ \mathsf{nat}} \tag{2.5}$$

This derivation consists of a composition of Rules (1.2), starting with a nat as an axiom and ending with succ(succ(a)) nat as conclusion. In other words, the rule

$$\frac{a \ \mathsf{nat}}{\mathtt{succ(succ(}a\mathtt{))} \ \mathsf{nat}} \tag{2.6}$$

is derivable.

It is interesting to observe that the derivability of this rule is entirely independent of the choice of the object a. In particular, we may choose a to be some rubbish object, say junk, and observe that

$$\mathtt{junk} \ \mathsf{nat} \vdash \mathtt{succ(succ(junk))} \ \mathsf{nat} \tag{2.7}$$

is valid. For if we treat junk nat as a new axiom, then surely we can derive succ(succ(junk)) nat by using the rules defining the natural numbers, even though we cannot derive junk nat from these rules.

Evidence for a hypothetical judgement $\Gamma \vdash J$ may be thought of as a derivation ∇_Γ for J that may contain the hypotheses Γ as unjustified axioms. These may be thought of as placeholders for a derivation that may be plugged in separately without disturbing the rest of the derivation. It follows that the hypothetical judgement enjoys certain *structural properties*, independently of the rule set under consideration:

Reflexivity Every judgement is a consequence of itself: $\Gamma, J \vdash J$. The consequent is justified because it is regarded as an axiom.

Weakening If $\Gamma \vdash J$, then $\Gamma, K \vdash J$. The derivation of J makes use of the rules and the premises Γ, and is not affected by the (unexercised) option to use K as an axiom.

Exchange If $\Gamma_1, J_1, J_2, \Gamma_2 \vdash J$, then $\Gamma_1, J_2, J_1, \Gamma_2 \vdash J$. The relative ordering of the axioms is immaterial.

Contraction If $\Gamma, J, J \vdash K$, then $\Gamma, J \vdash K$. Since we can use any assumption any number of times, stating it more than once is the same as stating it once.

Transitivity If $\Gamma, K \vdash J$ and $\Gamma \vdash K$, then $\Gamma \vdash J$. If we replace an axiom by a derivation of it, the result remains a derivation of its consequent.

The contraction and exchange properties together imply that a finite sequence of hypotheses Γ may just as well be regarded as a finite set, since set membership is not affected by duplication of elements or by the order in which elements are specified. We treat the hypotheses of an iterated hypothetical judgement as a finite set, which amounts to the tacit use of exchange and contraction as necessary.

Derivability is a relatively strong condition that is stable under extension of the set of rules defining a judgement. That is, if a rule is derivable from one set of rules, it remains derivable from any extension of that set of rules. The existence of a derivation depends only on what rules are available, and not on which rules are absent. Another characterization of derivability is explored in Exercise 1 on page 24.

2.2 Admissibility

The *admissibility* judgement, written $J \models K$, is a weaker form of hypothetical judgement whose meaning is that K is derivable from the given set of rules whenever J is derivable from the same set of rules. Equivalently, the admissibility judgement is a simple conditional assertion stating that *if* J is derivable from the rules, *then* so is K. As with derivability, we may iterate the admissibility judgement, writing $J_1, \ldots, J_n \models J$ to mean that *if* J_1 is derivable *and* ...*and* J_n is derivable, *then* J is derivable. Equivalently, we say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{2.8}$$

is *admissible* iff $J_1, \ldots, J_n \models J$.

For example, for an arbitrary object a, the admissibility judgement

$$\texttt{succ}(a)\ \texttt{nat} \models a\ \texttt{nat} \tag{2.9}$$

is valid with respect to Rules (1.2). This may be proved by rule induction, for if $\texttt{succ}(a)\ \texttt{nat}$, then this can only be by virtue of Rule (1.2b). But then the desired conclusion must hold, since it is the premise of the inference. Equivalently, we may say that the rule

$$\frac{\texttt{succ}(a)\ \texttt{nat}}{a\ \texttt{nat}} \tag{2.10}$$

is admissible.

Admissibility is, in general, strictly weaker than derivability: if $J_1, \ldots, J_n \vdash J$ is valid, then so is $J_1, \ldots, J_n \models J$, but the converse need not be the case. To see why the implication left to right holds, assume that $J_1, \ldots, J_n \vdash J$. To show $J_1, \ldots, J_n \models J$, assume further that each J_i is derivable from the original rules, which is to say that $\vdash J_1, \ldots, \vdash J_n$ are all valid derivability judgements with no hypotheses. But then by weakening and transitivity it follows that $\vdash J$, which means that J is derivable in the original set of rules. On the other hand, we have already seen that $\texttt{succ}(a)\ \texttt{nat} \models a\ \texttt{nat}$, but

$$\texttt{succ}(a)\ \texttt{nat} \nvdash a\ \texttt{nat}. \tag{2.11}$$

That is, there is no way to compose rules starting with $\texttt{succ}(a)\ \texttt{nat}$ and end up with $a\ \texttt{nat}$. To see this, take $a = \texttt{junk}$ and observe that, even with $\texttt{succ(junk)}\ \texttt{nat}$ as a new axiom, there is no way to derive $\texttt{junk}\ \texttt{nat}$.

Evidence for admissibility may be thought of as a mathematical function transforming derivations $\nabla_1, \ldots, \nabla_n$ of the hypotheses into a derivation ∇ of the consequent. Typically such a function is defined by an inductive analysis of the derivations of the hypotheses. As a consequence of this interpretation, the admissibility judgement enjoys the same structural properties as derivability.

Reflexivity If J is derivable from the original rules, then J is derivable from the original rules: $J \models J$.

Weakening If J is derivable from the original rules assuming that each of the judgements in Γ are derivable from these rules, then J must also be derivable assuming that Γ and also K are derivable from the original rules: if $\Gamma \models J$, then $\Gamma, K \models J$.

Exchange The order of assumptions in an iterated implication does not matter.

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

Transitivity If $\Gamma, K \models J$ and $\Gamma \models K$, then $\Gamma \models J$. If the assumption K is used, then we may instead appeal to the assumed derivability of K.

As with derivability we often make tacit use of exchange and contraction by stating the iterated form using finite sets, rather than sequences, of assumptions.

In contrast to derivability, admissibility is not stable under expansion of the rule set. For example, suppose we expanded Rules (1.2) with the following (fanciful) rule:

$$\frac{}{\texttt{succ(junk) nat}} \tag{2.12}$$

But relative to this expanded rule set, $\texttt{succ}(a)$ $\texttt{nat} \not\models a$ $\texttt{nat}$, even though it was valid with respect to the original. For if the premise were derived using the additional rule, there would be no derivation of `junk nat`, so the conclusion fails. In other words, admissibility is sensitive to which rules are *absent* from, as well as to which rules are *present* in, an inductive definition. At bottom a proof of admissibility amounts to an exhaustive analysis of the possible ways of deriving the premises, showing in each case that the conclusion is derivable.

Another way to compare derivability to admissibility is to note that whereas an admissibility judgement may be *vacuously* true (because the hypothesis is not derivable), a derivability judgement *never* holds vacuously (because it adds the hypothesis to the rules as a new axiom). Thus, relative to Rules (1.2), the admissibility judgement `junk nat` $\models$ `succ(junk) nat` is vacuously valid, because the hypothesis is not derivable according to those rules (as may be seen by a simple rule induction). The corresponding derivability judgement `junk nat` $\vdash$ `succ(junk) nat` is also valid, but not vacuously so! Rather, the conclusion holds because we can apply Rule (1.2b) to the hypothesis to obtain it.

2.3 Conditional Inductive Definitions

In Sections 2.1 on page 17 and 2.2 on page 19 we defined the meaning of the derivability and admissibility hypothetical judgements for an inductively defined judgement form. According to the derivability interpretation, the hypotheses of the judgement are treated as temporary axioms asserted for the purpose of deducing its consequent. According to the admissibility interpretation, we consider all the means (if any) by which the hypotheses might have been derived, and determine that the consequent must also be derivable in all such circumstances. The crucial difference is that derivability is stable under extension, whereas admissibility is not. This licenses a powerful extension to the framework of inductive definitions in which hypothetical judgements are permitted in the premises and conclusions of rules.

A *conditional inductive definition* consists of a collection of *conditional rules* of the form

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \dots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\ . \tag{2.13}$$

The hypotheses Γ are the *global hypotheses* of the rule, and the hypotheses Γ_i are the *local hypotheses* of the ith premise of the rule. Informally, this rule states that J is a derivable consequence of Γ whenever each J_i is a derivable consequence of Γ, augmented with the additional hypotheses Γ_i. Thus, one way to show that J is derivable from Γ is to show, in turn, that each J_i is derivable from $\Gamma\,\Gamma_i$. The derivation of each premise involves a "context switch" in which we extend the global hypotheses with the local hypotheses of that premise, establishing a new global hypothesis set for use within that derivation.

Often a conditional rule is given for each choice of global context, without restriction. In that case the rule is said to be *pure*, because it applies irrespective of the context in which it is used. A pure rule, being stated uniformly for all global contexts, may be given in *implicit* form, as follows:

$$\frac{\Gamma_1 \vdash J_1 \quad \dots \quad \Gamma_n \vdash J_n}{J}\ . \tag{2.14}$$

This formulation omits explicit mention of the global context in order to focus attention on the local aspects of the inference.

Sometimes it is necessary to restrict the global context of an inference, so that it applies only when the global context satisfies a specified *side condition*. Such rules are said to be *impure*. Impure rules cannot be given in implicit form, but rather take the form

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \dots \quad \Gamma\,\Gamma_n \vdash J_n \quad \mathcal{S}(\Gamma)}{\Gamma \vdash J}\ , \tag{2.15}$$

where $\mathcal{S}(\Gamma)$ is the side condition on the global context. This rule applies only when the global context Γ satisfies the condition $\mathcal{S}$.

As with an ordinary inductive definition, a conditional inductive definition is to be understood as defining the *strongest* judgement closed under the specified rules. However, we regard a conditional inductive definition as defining the hypothetical judgement $\Gamma \vdash J$ *directly* by the specified rules, rather than *indirectly* in the manner described in Section 2.1 on page 17. In other words, the rules themselves specify the meaning of $\Gamma \vdash J$, without reference to any previously given interpretation of J, or any previously given concept of derivability. We must be careful, however, to ensure that the rules are sufficient to ensure that the hypothetical judgement they define really does behave like a hypothetical judgement. For example, we would expect that $\Gamma, J \vdash J$ holds, but there is no reason to expect that this is the case for an arbitrary set of rules. (Consider the null set, for example.) After all, the rules are the rules, and we might have omitted this natural principle of reasoning.

To rule out such pathologies, we insist that the following *structural rules*, corresponding to the structural properties of the hypothetical judgement given earlier, be admissible in any conditional inductive definition:

$$\frac{}{\Gamma, J \vdash J} \tag{2.16a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{2.16b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{2.16c}$$

Rules (2.16a) and (2.16b) ensure that any hypothesis may be used as evidence for the consequent without further justification. Rule (2.16c) ensures that we may “plug in” concrete evidence for K into evidence for $\Gamma, K \vdash J$ to obtain evidence for $\Gamma \vdash J$.

Since a conditional inductive definition determines the strongest hypothetical judgement closed under the given rules, we may reason about this judgement using the principle of rule induction. Specifically, to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash J$ is derivable from a set of conditional rules, it is sufficient to show that $\mathcal{P}$ is closed under each rule. For each rule of the form

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \dots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\,, \tag{2.17}$$

we must show

$$\textit{if}\ \mathcal{P}(\Gamma\,\Gamma_1 \vdash J_1),\ \dots,\ \mathcal{P}(\Gamma\,\Gamma_n \vdash J_n),\ \textit{then}\ \mathcal{P}(\Gamma \vdash J).$$

In particular, closure under the structural rule of reflexivity means that we must show $\mathcal{P}(\Gamma, J \vdash J)$ for any set of hypotheses Γ.

We use a variety of notations to indicate that a property, $\mathcal{P}$, holds whenever $\Gamma \vdash J$. Often it is helpful to think of $\mathcal{P}$ as a family of properties indexed by Γ, writing $\mathcal{P}_\Gamma(J)$ or $\mathcal{P}(J)\,[\Gamma]$, for $\mathcal{P}(\Gamma \vdash J)$.

2.4 Exercises

1. Define $\Gamma' \vdash \Gamma$ to mean that $\Gamma' \vdash J_i$ for each J_i in Γ. Show that $\Gamma \vdash J$ iff whenever $\Gamma' \vdash \Gamma$, it follows that $\Gamma' \vdash J$. For the implication right-to-left, take $\Gamma' = \Gamma$. For the implication left-to-right, repeatedly appeal to transitivity to obtain the desired conclusion.

2. Show that it is possible to make sense of admissibility judgements in the *consequents* of inference rules by an analysis reminiscent of that used to justify derivability judgements in the consequent. Hint: make use of the interpretation of conditional rules as a simultaneous inductive definition of a family of judgement forms described in Section 2.3 on page 22.

3. Show that it is dangerous to permit admissibility judgements in the premise of a rule. Hint: show that using such rules one may "define" an inconsistent judgement form J for which we have $a\ J$ iff it is *not* the case that $a\ J$.

Chapter 3

Parametric Judgements

In Chapter 2 we introduced the concept of the derivability judgement, $\Gamma \vdash J$, whose meaning is that J is derivable if we expand the rules defining it with the hypotheses Γ as new axioms. In this chapter we consider a similar concept, called the *parametric* judgement, which permits us to expand the universe of objects with a finite set of *parameters* for the sake of a derivation. We may then construct derivations *parametrically*, or *schematically*, by giving a *derivation scheme* that involves the specified parameters. The parameters may be thought of as "names" of unspecified objects that are handled symbolically, as if they were objects of the universe. Just as conditional inductive definitions extend rules to permit hypothetical judgements, a *parametric inductive definition* permits rules that involve parametric judgements. Such rules figure prominently in the study of programming languages.

3.1 Parameterization

Let $\mathcal{X}$ be a finite collection of parameters, and let $\mathcal{J}$ be a hypothetical or categorical judgement. The parametric judgement $\mathcal{X} \mid \mathcal{J}$ expresses that the judgement $\mathcal{J}$ holds *uniformly*, or *parametrically*, in the parameters $\mathcal{X}$. Evidence for this judgement consists of a *parametric derivation*, or *derivation scheme*, $\nabla_{\mathcal{X}}$, of the judgement $\mathcal{J}$ in which the parameters in $\mathcal{X}$ may be used as objects.

For example, the parametric hypothethical judgement

$$x \mid x\ \mathtt{nat} \vdash \mathtt{succ(succ(}x\mathtt{))}\ \mathtt{nat} \tag{3.1}$$

states that $x\ \mathtt{nat} \vdash \mathtt{succ(succ(}x\mathtt{))}\ \mathtt{nat}$ holds uniformly in the parameter x. Evidence for this judgement consists of the following derivation, ∇_x, involving the parameter x:

$$\frac{\dfrac{\overline{x\ \mathsf{nat}}}{\mathtt{succ}(x)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}}\ . \tag{3.2}$$

The choice of parameter in a parametric judgement or derivation is unimportant, as it serves only as a placeholder. Consequently, we do not distinguish between parametric judgements or derivations that differ only in the choice of parameter. The judgement (3.1) is indistinguishable from the variant

$$y \mid y\ \mathsf{nat} \vdash \mathtt{succ}(\mathtt{succ}(y))\ \mathsf{nat}$$

in which the parameter y is used in place of x. Similarly, the derivation ∇_x is indistinguishable from the derivation ∇_y in which we replace x by y to obtain

$$\frac{\dfrac{\overline{y\ \mathsf{nat}}}{\mathtt{succ}(y)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(y))\ \mathsf{nat}}\ . \tag{3.3}$$

3.2 Structural Properties

The parametric judgement enjoys structural properties that are reminiscent of those enjoyed by the hypothetical judgement:

Proliferation If $\mathcal{X} \mid \mathcal{J}$ and $x \notin \mathcal{X}$, then $\mathcal{X}, x \mid \mathcal{J}$.

Swapping If $\mathcal{X}_1, x_1, x_2, \mathcal{X}_2 \mid \mathcal{J}$, then $\mathcal{X}_1, x_2, x_1, \mathcal{X}_2 \mid \mathcal{J}$.

Duplication If $\mathcal{X}, x, x \mid \mathcal{J}$, then $\mathcal{X}, x \mid \mathcal{J}$.

Renaming If $\mathcal{X}, x \mid \mathcal{J}_x$, then $\mathcal{X}, y \mid \mathcal{J}_y$, provided that $y \notin \mathcal{X}$.

Proliferation of variables corresponds to weakening, swapping corresponds to exchange, and duplication corresponds to contraction. Renaming states that a parametric judgement is invariant under renaming of parameters.

These structural properties may be stated as *structural rules* as follows:

$$\frac{\mathcal{X} \mid \mathcal{J}}{\mathcal{X}, x \mid \mathcal{J}} \tag{3.4a}$$

$$\frac{\mathcal{X}_1, x_1, x_2, \mathcal{X}_2 \mid \mathcal{J}}{\mathcal{X}_1, x_2, x_1, \mathcal{X}_2 \mid \mathcal{J}} \tag{3.4b}$$

$$\frac{\mathcal{X}, x, x \mid \mathcal{J}}{\mathcal{X}, x \mid \mathcal{J}} \tag{3.4c}$$

$$\frac{\mathcal{X}, x \mid \mathcal{J}_x}{\mathcal{X}, y \mid \mathcal{J}_y} \tag{3.4d}$$

As with the hypothetical judgement, Rules (3.4b) and (3.4c) are implicit since $\mathcal{X}$ is treated as a set of parameters.

3.3 Parametric Inductive Definitions

Just as it is useful to permit hypothetical judgements in rules, it is similarly useful to permit parametric judgements as well. Using a parametric judgement in the premise of a rule permits us to isolate parameters to particular sub-derivations of an overall derivation. This is very similar to introducing fresh hypotheses for use in the derivation of a premise, the difference being that instead of working with an unspecified derivation (represented by a "fresh" axiom), we instead work with an unspecified object (represented by a "fresh" parameter).

A *parametric inductive definition* consists of a set of *parametric rules* of the form

$$\frac{\mathcal{X}\,\mathcal{X}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \dots \quad \mathcal{X}\,\mathcal{X}_n \mid \Gamma\,\Gamma_n \vdash J_n}{\mathcal{X} \mid \Gamma \vdash J}\;. \tag{3.5}$$

The set $\mathcal{X}$ specifies the *global parameters* of the inference, and, for each $1 \leq i \leq n$, the set $\mathcal{X}_i$ specifies the *fresh local parameters* of the ith premise. The "freshness" condition is captured by the requirement that the local parameters be disjoint from the global parameters so as to avoid confusion among them. The global and local hypotheses, Γ and Γ_i, respectively, are as described in Chapter 2 for a conditional rule. The pair $\mathcal{X} \mid \Gamma$ is the *global context* of the rule, and each pair $\mathcal{X}_i \mid \Gamma_i$ is the *local context* of the ith premise of the rule.

A parametric rule is *pure* if it is stated for all choices of global context, subject only to the requirement that the global parameters be disjoint from the local parameters. Such a rule may be written in implicit form, emphasizing the local aspects of the inference, as follows:

$$\frac{\mathcal{X}_1 \mid \Gamma_1 \vdash J_1 \quad \dots \quad \mathcal{X}_n \mid \Gamma_n \vdash J_n}{J}\;. \tag{3.6}$$

This form is to be understood as standing for all rules of the form Rule (3.5) obtained by specifying the global context $\mathcal{X} \mid \Gamma$.

As may be expected, a parametric inductive definition specifies the *strongest* judgement closed under the given set of rules. To ensure that the judgement so defined behaves like a parametric hypothetical judgement, we require that Rules (3.4) be admissible for any parametric inductive definition. This is usually achieved by including some of the structural rules as part of the definition, leaving the others to be proved admissible relative to those explicitly included.

The principle of rule induction applied to a parametric inductive definition states that to show $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash J)$ whenever $\mathcal{X} \mid \Gamma \vdash J$, it is enough to show that $\mathcal{P}$ is closed under the rules comprising the definition. Specifically, for each rule of the form (3.5), we must show that

$$\textit{if}\ \mathcal{P}(\mathcal{X}\,\mathcal{X}_1 \mid \Gamma\,\Gamma_1 \vdash J_1),\ \ldots,\ \mathcal{P}(\mathcal{X}\,\mathcal{X}_n \mid \Gamma\,\Gamma_n \vdash J_n),\ \textit{then}\ \mathcal{P}(\mathcal{X} \mid \Gamma \vdash J).$$

We often use notation such as $\mathcal{P}(J)\,[\mathcal{X} \mid \Gamma]$ to emphasize that the property $\mathcal{P}$ may be thought of as a family of properties indexed by the global context of the inference.

3.4 Exercises

Chapter 4

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*, $\mapsto$. The transition judgement describes how the state of the machine evolves during execution.

4.1 Transition Systems

An *(ordinary) transition system* is specified by the following judgements:

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

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

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

4. $s \mapsto s'$, 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 $s \mapsto s'$. In general, a state s for which there is no $s' \in S$ such that $s \mapsto s'$ is said to be *stuck*, which may be indicated by writing $s \not\mapsto$. 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 initial, and $s_i \mapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff $s_n \not\mapsto$, and 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 $s \mapsto s'$, otherwise it is *non-deterministic*.

A *labelled transition system* over a set of labels, I, is a generalization of a transition system in which the single transition judgement, $s \mapsto s'$ is replaced by an I-indexed family of transition judgements, $s \overset{i}{\mapsto} s'$, where s and s' are states of the system. In typical situations the family of transition relations is given by a simultaneous inductive definition in which each rule may make reference to any member of the family.

It is often necessary to consider families of transition relations in which there is a distinguished unlabelled transition, $s \mapsto s'$, in addition to the indexed transitions. It is sometimes convenient to regard this distinguished transition as labelled by a special, anonymous label not otherwise in I. For historical reasons this distinguished label is often designated by τ or ϵ, but we will simply use an unadorned arrow. The unlabelled form is often called a *silent* transition, in contrast to the labelled forms, which announce their presence with a label.

4.2 Iterated Transition

Let $s \mapsto s'$ be a transition judgement, whether drawn from an indexed set of such judgements or not.

The *iteration* of transition judgement, $s \mapsto^* s'$, is inductively defined by the following rules:

$$\frac{}{s \mapsto^* s} \tag{4.1a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''} \tag{4.1b}$$

It is easy to show that iterated transition is transitive: if $s \mapsto^* s'$ and $s' \mapsto^* s''$, then $s \mapsto^* s''$.

The principle of rule induction for these rules states that to show that $P(s,s')$ holds whenever $s \mapsto^* s'$, it is enough to show these two properties of P:

1. $P(s,s)$.

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

The first requirement is to show that P is reflexive. The second is to show that P is *closed under head expansion*, or *converse evaluation*. Using this principle, it is easy to prove that $\mapsto^*$ is reflexive and transitive.

The *n-times iterated* transition judgement, $s \mapsto^n s'$, where $n \geq 0$, is inductively defined by the following rules.

$$\frac{}{s \mapsto^0 s} \tag{4.2a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^n s''}{s \mapsto^{n+1} s''} \tag{4.2b}$$

Theorem 4.1. *For all states s and s', $s \mapsto^* s'$ iff $s \mapsto^k s'$ for some $k \geq 0$.*

Finally, we write $\downarrow s$ to indicate that there exists some s' final such that $s \mapsto^* s'$.

4.3 Simulation and Bisimulation

A *strong simulation* between two transition systems $\mapsto_1$ and $\mapsto_2$ is given by a binary relation, $s_1 \, S \, s_2$, between their respective states such that if $s_1 \, S \, s_2$, then $s_1 \mapsto_1 s_1'$ implies $s_2 \mapsto_2 s_2'$ for some state s_2' such that $s_1' \, S \, s_2'$. Two states, s_1 and s_2, are *strongly similar* iff there is a strong simulation, S, such that $s_1 \, S \, s_2$. Two transition systems are strongly similar iff each initial state of the first is strongly similar to an initial state of the second. Finally, two states are *strongly bisimilar* iff there is a single relation S such that both S and its converse are strong simulations.

A strong simulation between two labelled transition systems over the same set, I, of labels consists of a relation S between states such that for each $i \in I$ the relation S is a strong simulation between $\overset{i}{\mapsto}_1$ and $\overset{i}{\mapsto}_2$. That is, if $s_1 \, S \, s_2$, then $s_1 \overset{i}{\mapsto}_1 s_1'$ implies that $s_2 \overset{i}{\mapsto}_2 s_2'$ for some s_2' such that $s_1' \, S \, s_2'$. In other words the simulation must preserve labels, and not just transitions.

The requirements for strong simulation are rather stringent: every step in the first system must be mimicked by a similar step in the second, up to the simulation relation in question. This means, in particular, that a sequence of steps in the first system can only be simulated by a sequence of steps of the same length in the second—there is no possibility of performing "extra" work to achieve the simulation.

A *weak simulation* between transition systems is a binary relation between states such that if $s_1 \, S \, s_2$, then $s_1 \mapsto_1 s_1'$ implies that $s_2 \mapsto_2^* s_2'$ for some s_2' such that $s_1' \, S \, s_2'$. That is, every step in the first may be matched by zero or more steps in the second. A *weak bisimulation* is such that both it and its converse are weak simulations. We say that states s_1 and s_2 are *weakly (bi)similar* iff there is a weak (bi)simulation S such that $s_1 \, S \, s_2$.

The corresponding notion of weak simulation for labelled transitions involves the silent transition. The idea is that to weakly simulate the labelled transition $s_1 \overset{i}{\longmapsto}_1 s_1'$, we do not wish to permit multiple *labelled* transitions between related states, but rather to permit any number of *unlabelled* transitions to accompany the labelled transition. A relation between states is a *weak simulation* iff it satisfies both of the following conditions whenever $s_1 \mathrel{S} s_2$:

1. If $s_1 \longmapsto_1 s_1'$, then $s_2 \longmapsto_2^* s_2'$ for some s_2' such that $s_1' \mathrel{S} s_2'$.

2. If $s_1 \overset{i}{\longmapsto}_1 s_1'$, then $s_2 \longmapsto_2^* \overset{i}{\longmapsto}_2 \longmapsto_2^* s_2'$ for some s_2' such that $s_1' \mathrel{S} s_2'$.

That is, every silent transition must be mimicked by zero or more silent transitions, and every labelled transition must be mimicked by a corresponding labelled transition, preceded and followed by any number of silent transitions. As before, a *weak bisimulation* is a relation between states such that both it and its converse are weak simulations. Finally, two states are *weakly (bi)similar* iff there is a weak (bi)simulation between them.

4.4 Exercises

1. Prove that S is a weak simulation for the ordinary transition system $\longmapsto$ iff S is a strong simulation for $\longmapsto^*$.

Part II

Levels of Syntax

Chapter 5

Basic Syntactic Objects

We will make use of two sorts of objects for representing syntax, *strings* of characters, and *abstract syntax trees*. Strings provide a convenient representation for reading and entering programs, but are all but useless for manipulating programs as objects of study. Abstract syntax trees provide a representation of programs that exposes their hierarchical structure.

5.1 Symbols

We shall have use for a variety of *symbols*, which will serve in a variety of roles as characters, variable names, names of fields, and so forth. Symbols are sometimes called *names*, or *atoms*, or *identifiers*, according to custom in particular circumstances. Symbols are to be thought of as atoms with no structure other than their identity. We write $x\ \mathsf{sym}$ to assert that x is a symbol, and we assume that there are infinitely many symbols at our disposal. The judgement $x \mathrel{\#} y$, where $x\ \mathsf{sym}$ and $y\ \mathsf{sym}$, states that x and y are distinct symbols.

We will make use of a variety of classes of symbols throughout the development. We generally assume that any two classes of symbols under consideration are disjoint from one another, so that there can be no confusion among them.

5.2 Strings Over An Alphabet

An *alphabet* is a (finite or infinite) collection of symbols, called *characters*. We write $c\ \mathsf{char}$ to indicate that c is a character, and let Σ stand for a finite set of such judgements, which is sometimes called an *alphabet*. The judgement $\Sigma \vdash s\ \mathsf{str}$, defining the strings over the alphabet Σ, is inductively defined by the following rules:

$$\frac{}{\Sigma \vdash \epsilon\ \mathsf{str}} \tag{5.1a}$$

$$\frac{\Sigma \vdash c\ \mathsf{char} \quad \Sigma \vdash s\ \mathsf{str}}{\Sigma \vdash c \cdot s\ \mathsf{str}} \tag{5.1b}$$

Thus a string is essentially a list of characters, with the null string being the empty list. We often suppress explicit mention of Σ when it is clear from context.

When specialized to Rules (5.1), the principle of rule induction states that to show $s\ P$ holds whenever $s\ \mathsf{str}$, it is enough to show

1. $\epsilon\ P$, and

2. if $s\ P$ and $c\ \mathsf{char}$, then $c \cdot s\ P$.

This is sometimes called the principle of *string induction*. It is essentially equivalent to induction over the length of a string, except that there is no need to define the length of a string in order to use it.

The following rules constitute an inductive definition of the judgement $s_1\hat{\ }s_2 = s\ \mathsf{str}$, stating that s is the result of concatenating the strings s_1 and s_2.

$$\frac{}{\epsilon\hat{\ }s = s\ \mathsf{str}} \tag{5.2a}$$

$$\frac{s_1\hat{\ }s_2 = s\ \mathsf{str}}{(c \cdot s_1)\hat{\ }s_2 = c \cdot s\ \mathsf{str}} \tag{5.2b}$$

It is easy to prove by string induction on the first argument that this judgement has mode $(\forall, \forall, \exists!)$. Thus, it determines a total function of its first two arguments.

Strings are usually written as juxtapositions of characters, writing just *abcd* for the four-letter string $a \cdot (b \cdot (c \cdot (d \cdot \epsilon)))$, for example. Concatentation is also written as juxtaposition, and individual characters are often identified with the corresponding unit-length string. This means that *abcd* can be thought of in many ways, for example as the concatenations *ab cd*, *a bcd*, or *abc d*, or even ϵ *abcd* or *abcd* ϵ, as may be convenient in a given situation.

5.3 Abtract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree in which certain symbols, called *operators*, label the nodes. A *signature*, Ω, is a finite set of judgements of the form $\mathsf{ar}(o) = n$, where o sym and n nat, assigning an *arity*, n, to an operator, o, such that if $\Omega \vdash \mathsf{ar}(o) = n$ and $\Omega \vdash \mathsf{ar}(o) = n'$, then $n = n'$ nat.

The class of abstract syntax trees over a signature, Ω, is inductively defined as follows.

$$\frac{\begin{array}{c}\Omega \vdash \mathsf{ar}(o) = n \\ a_1\ \mathsf{ast} \quad \dots \quad a_n\ \mathsf{ast}\end{array}}{o(a_1,\dots,a_n)\ \mathsf{ast}} \tag{5.3a}$$

The base case of this inductive definition is an operator of arity zero, in which case Rule (5.3a) has no premises.

5.3.1 Structural Induction

The principle of *structural induction* is the specialization of the principle of rule induction to the rules defining ast's over a signature. To show that $\mathcal{P}(a\ \mathsf{ast})$, it is enough to show that $\mathcal{P}$ is closed under Rules (5.3). That is, if $\Omega \vdash \mathsf{ar}(o) = n$, then we are to show that

$$\textit{if } \mathcal{P}(a_1\ \mathsf{ast}),\ \dots,\ \mathcal{P}(a_n\ \mathsf{ast}),\ \textit{then } \mathcal{P}(o(a_1,\dots,a_n)\ \mathsf{ast}).$$

When n is zero, this reduces to showing that $\mathcal{P}(o())$.

For example, we consider the following inductive definition of the height of an abstract syntax tree:

$$\frac{\mathsf{hgt}(a_1) = h_1 \quad \dots \quad \mathsf{hgt}(a_n) = h_n \quad \mathsf{max}(h_1,\dots,h_n) = h}{\mathsf{hgt}(o(a_1,\dots,a_n)) = \mathtt{succ}(h)} \tag{5.4a}$$

We may prove by structural induction that this judgement has mode $(\forall, \exists!)$, which is to say that every ast has a unique height. For an operator o of arity n, we may assume by induction that, for each $1 \leq i \leq n$, there is a unique h_i such that $\mathsf{hgt}(a_i) = h_i$. We may show separately that the maximum, h, of these is uniquely determined, and hence that the overall height, $\mathtt{succ}(h)$, is also uniquely determined.

5.3.2 Variables and Substitution

In practice we often wish to consider ast's with *variables* serving as placeholders for other ast's. The variables are instantiated by *substitution* of an ast for occurrences of that variable in another ast.

As a notational convenience, we let $\mathcal{X} = x_1\ \mathsf{ast}, \ldots, x_n\ \mathsf{ast}$ stand for the combined parameter set and hypothesis list $\{ x_1, \ldots, x_n \} \mid x_1\ \mathsf{ast}, \ldots, x_n\ \mathsf{ast}$, where $x_1\ \mathsf{sym}, \ldots, x_n\ \mathsf{sym}$. Moreover, we write $x \ \#\ \mathcal{X}$ to mean that $x \notin \{ x_1, \ldots, x_n \}$. Using this notation, the judgement $\mathcal{X} \vdash a\ \mathsf{ast}$ is inductively defined by the following rules:

$$\frac{}{\mathcal{X}, x\ \mathsf{ast} \vdash x\ \mathsf{ast}} \tag{5.5a}$$

$$\frac{\Omega \vdash \mathsf{ar}(o) = n \quad \mathcal{X} \vdash a_1\ \mathsf{ast} \quad \ldots \quad \mathcal{X} \vdash a_n\ \mathsf{ast}}{\mathcal{X} \vdash o(a_1, \ldots, a_n)\ \mathsf{ast}} \tag{5.5b}$$

The principle of rule induction for these rules states that to show $\mathcal{P}(\mathcal{X} \vdash a\ \mathsf{ast})$, it is enough to show

1. $\mathcal{P}(\mathcal{X}, x\ \mathsf{ast} \vdash x\ \mathsf{ast})$.
2. If $\Omega \vdash \mathsf{ar}(o) = n$, and if $\mathcal{P}(\mathcal{X} \vdash a_1\ \mathsf{ast})$, ..., $\mathcal{P}(\mathcal{X} \vdash a_n\ \mathsf{ast})$, then $\mathcal{P}(\mathcal{X} \vdash o(a_1, \ldots, a_n)\ \mathsf{ast})$.

Thus, the parameters in $\mathcal{X}$ are treated as atomic objects, each with its own abstract syntax tree.

We define the judgement $\mathcal{X} \vdash [a/x]b = c$, meaning that c is the result of substituting a for x in b, by the following rules:

$$\frac{}{\mathcal{X}, x\ \mathsf{ast} \vdash [a/x]x = a} \tag{5.6a}$$

$$\frac{x \ \#\ y}{\mathcal{X}, x\ \mathsf{ast}, y\ \mathsf{ast} \vdash [a/x]y = y} \tag{5.6b}$$

$$\frac{\mathcal{X} \vdash [a/x]b_1 = c_1 \quad \ldots \quad \mathcal{X} \vdash [a/x]b_n = c_n}{\mathcal{X} \vdash [a/x]o(b_1, \ldots, b_n) = o(c_1, \ldots, c_n)} \tag{5.6c}$$

The result of substitution is uniquely determined by its other arguments. Consequently, we write $[a/x]b$ for the unique c such that $[a/x]b = c$.

Theorem 5.1. *If $\mathcal{X} \vdash a\ \mathsf{ast}$ and $\mathcal{X}, x\ \mathsf{ast} \vdash b\ \mathsf{ast}$, where $x \ \#\ \mathcal{X}$, then there exists a unique c such that $\mathcal{X} \vdash [a/x]b = c$ and $\mathcal{X} \vdash c\ \mathsf{ast}$.*

Proof. The proof is by structural induction on b relative to the context $\mathcal{X}, x\ \mathsf{ast}$. There are three cases to consider:

1. Since $\mathcal{X}, x\ \mathsf{ast} \vdash x\ \mathsf{ast}$, we must show that there exists a unique c such that $\mathcal{X} \vdash [a/x]x = c$. Consulting Rule (5.6a), we see that choosing c to be a is both necessary and sufficient.

2. If $\mathcal{X}, x$ ast, y ast $\vdash y$ ast for some $y \# x$, then by Rule (5.6b) choosing c to be y is necessary and sufficient.

3. Finally if $b = o(b_1, \ldots, b_n)$, then by induction there exists unique $c_1, \ldots, c_n$ such that $\mathcal{X} \vdash [a/x]b_1 = c_1$, …, $\mathcal{X} \vdash [a/x]b_n = c_n$. By Rule (5.6c) the only possible choice for c, namely $o(c_1, \ldots, c_n)$, suffices.

□

5.4 Exercises

1. Give an inductive definition of the two-place judgement $|s| = n$ str, where s str and n nat, stating that a string s has length n, namely the number of symbols occurring within it. Use the principle of string induction to show that this judgement has mode $(\forall, \exists!)$, and hence defines a function.

2. Give an inductive definition of equality of strings, and show that string concatenation is associative. Specifically, define the judgement $s_1 = s_2$ str, and show that if $s_1\,\hat{}\,s_2 = s_{12}$ str, $s_{12}\,\hat{}\,s_3 = s_{123}$ str, $s_1\,\hat{}\,s_{23} = s'_{123}$ str, and $s_2\,\hat{}\,s_3 = s_{23}$ str, then $s_{123} = s'_{123}$ str.

3. Give an inductive definition of *simultaneous substitution* of a sequence of n ast's for a sequence of n distinct variables within an ast, written $\mathcal{X} \vdash [a_1, \ldots, a_n / x_1, \ldots, x_n]b = c$. Show that c is uniquely determined, and hence we may write $\mathcal{X} \vdash [a_1, \ldots, a_n / x_1, \ldots, x_n]b$ for the unique such c.

Chapter 6

Binding and Scope

Abstract syntax trees expose the hierarchical structure of syntax, dispensing with the details of how one might represent pieces of syntax on a page or a computer screen. *Abstract binding trees*, or *abt's*, enrich this representation with the concepts of *binding* and *scope*. In just about every language there is a means of associating a meaning to an identifier within a specified range of significance (perhaps the whole program, often limited regions of it). Examples include definitions, in which we introduce a name for a program phrase, or parameters to functions, in which we introduce a name for the argument to the function within its body.

Abstract binding trees enrich abstract syntax trees with a means of introducing a *fresh*, or *new*, name for use within a specified scope. Uses of the fresh name within that scope are references to the binding site. As such the particular choice of name is significant only insofar as it does not conflict with any other name currently in scope; this is the essence of what it means for the name to be "new" or "fresh."

In this chapter we introduce the concept of an abstract binding tree, including the relation of α-equivalence, which expresses the irrelevance of the choice of bound names, and the operation of *capture-avoiding substitution*, which ensures that names are not confused by substitution. While intuitively clear, the precise formalization of these concepts requires some care; experience has shown that it is surprisingly easy to get them wrong.

All of the programming languages that we shall study are represented as abstract binding trees. Consequently, we will re-use the machinery developed in this chapter many times, avoiding considerable redundancy and consolidating the effort required to make precise the notions of binding and scope.

6.1 Abstract Binding Trees

The concepts of binding and scope are formalized by the concept of an *abstract binding tree*, or *abt*. An abt is an ast in which we distinguish a name-indexed family of operators, called *abstractors*. An abstractor has the form $x\,.\,a$; it *binds* the name, x, for use in the abt, a, which is called the *scope* of the binding. The bound name x is meaningful only within a, and is, in a sense to be made precise shortly, treated as distinct from any other names that may be currently in scope.

As with abstract syntax trees, the well-formed abstract binding trees are determined by a *signature* that specifies the *arity* of each of a finite collection of operators. For ast's the arity specified only the number of arguments for each operator, but for abt's we must also specify the number of names that are bound by each operator. Thus an arity is a finite sequence $(n_1, \ldots, n_k)$ of natural numbers, with k specifying the number of arguments, and each n_i specifying the *valence*, or number of bound names, in the ith argument. The arity $(0, 0, \ldots, 0)$, of length k specifies an operator with k arguments that binds no variables in any argument; it is therefore the analogue of the arity k for an operator over abstract syntax trees.

A signature, Ω, consists of a finite set of judgements of the form $\mathsf{ar}(o) = (n_1, \ldots, n_k)$ such that no operator occurs in more than one such judgement. The well-formed abt's over a signature Ω are specified by a parametric hypothetical judgement of the form

$$\{\, x_1, \ldots, x_k \,\} \mid x_1\ \mathsf{abt}^0, \ldots, x_k\ \mathsf{abt}^0 \vdash a\ \mathsf{abt}^n$$

stating that a is an abt of *valence* n, with *parameters*, or *free names*, $x_1, \ldots, x_k$. We sometimes write just a abt as short-hand for $a\ \mathsf{abt}^0$.

We use the meta-variable $\mathcal{X}$ to range over finite sets of parameters, and the meta-variable $\mathcal{A}$ to range over finite sets of assumptions of the form $x\ \mathsf{abt}^0$, with one assumption for each $x \in \mathcal{X}$. As a notational convenience, the judgement $\mathcal{X} \mid \mathcal{A} \vdash a\ \mathsf{abt}^n$ is often abbreviated to just $\mathcal{A} \vdash a\ \mathsf{abt}^n$ when $\mathcal{X}$ is clear from context. In such cases we write $x \mathrel{\#} \mathcal{A}$ to mean that $x \notin \mathcal{X}$, where $\mathcal{X}$ is the set of parameters governed by $\mathcal{A}$.

The rules defining the well-formed abt's over a given signature are as follows:

$$\frac{}{\mathcal{X}, x \mid \mathcal{A}, x\ \mathsf{abt}^0 \vdash x\ \mathsf{abt}^0} \tag{6.1a}$$

$$\frac{\mathsf{ar}(o) = (n_1, \ldots, n_k) \quad \mathcal{X} \mid \mathcal{A} \vdash a_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{X} \mid \mathcal{A} \vdash a_k\ \mathsf{abt}^{n_k}}{\mathcal{X} \mid \mathcal{A} \vdash o(a_1, \ldots, a_k)\ \mathsf{abt}^0} \tag{6.1b}$$

$$\frac{\mathcal{X}, x' \mid \mathcal{A}, x' \ \mathsf{abt}^0 \vdash [x' \leftrightarrow x]\, a \ \mathsf{abt}^n \quad (x' \notin \mathcal{X})}{\mathcal{X} \mid \mathcal{A} \vdash x\,.\,a \ \mathsf{abt}^{n+1}} \tag{6.1c}$$

Rule (6.1c) specifies that an abstractor, $x\,.\,a$, is well-formed relative to $\mathcal{A}$, provided that its body, a, is well-formed for some variable $x' \notin \mathcal{X}$ replacing the bound variable, x, in a. The replacement of x by x' ensures that the formation of $x\,.\,a$ does not depend on whether x is already an active parameter.

According to the conventions on the names of parameters given in Chapter 3, we do not distinguish between parametric judgements that differ only in the choice of parameter names, nor do we distinguish between their corresponding parametric derivations. Consequently, Rule (6.1c) admits a dual reading as specifying that the body of the abstractor must be well-formed for some particular choice, x' of bound name not already occuring in $\mathcal{X}$, and, moreover, specifying that the same condition holds for any choice of parameter not already in $\mathcal{X}$. This means, in particular, that we may always assume that the variable, x', is chosen to satisfy any finite freshness restrictions we may wish to impose in a particular context, there being an infinite supply of names, only finitely many of which may be already in use in a given context.

6.1.1 Structural Induction With Binding and Scope

The principle of structural induction for abstract syntax trees extends to abstract binding trees. To show that $\mathcal{P}(\mathcal{X} \mid \mathcal{A} \vdash a \ \mathsf{abt}^n)$ whenever $\mathcal{X} \mid \mathcal{A} \vdash a \ \mathsf{abt}^n$, it suffices to show that $\mathcal{P}$ is closed under Rules (6.1). Specifically, we must show:

1. $\mathcal{P}(\mathcal{X}, x \mid \mathcal{A}, x \ \mathsf{abt}^0 \vdash x \ \mathsf{abt}^0)$.

2. For any operator o of arity $(m_1, \ldots, m_k)$, if $\mathcal{P}(\mathcal{X} \mid \mathcal{A} \vdash a_1 \ \mathsf{abt}^{m_1})$, $\ldots$, $\mathcal{P}(\mathcal{X} \mid \mathcal{A} \vdash a_k \ \mathsf{abt}^{m_k})$, then $\mathcal{P}(\mathcal{X} \mid \mathcal{A} \vdash o(a_1, \ldots, a_k) \ \mathsf{abt}^0)$.

3. If $\mathcal{P}(\mathcal{X}, x' \mid \mathcal{A}, x' \ \mathsf{abt}^0 \vdash [x' \leftrightarrow x]\, a \ \mathsf{abt}^n)$ for some/any $x' \notin \mathcal{X}$, then $\mathcal{P}(\mathcal{X} \mid \mathcal{A} \vdash x\,.\,a \ \mathsf{abt}^{n+1})$.

The condition on abstractors arises from the convention on parameter names described in Chapter 3. Since we identify parametric judgements that differ only in the choice of parameters, any property of a parametric judgement is constrained to respect this variation—it must not depend on the choice of parameter, only on its freshness relative to active parameters. This means that in the inductive hypothesis for abstractors, we may regard x' to be *any* fresh parameter not occurring in $\mathcal{X}$ according to our convenience.

As an example let us define the size, s, of an abt, a, of valence n by a judgement of the form $|a\ \mathsf{abt}^n| = s$. More generally, we define the parametric hypothetical judgement

$$|x_1\ \mathsf{abt}^0| = 1, \dots, |x_k\ \mathsf{abt}^0| = 1 \vdash |a\ \mathsf{abt}^n| = s,$$

with implied parameters $x_1, \dots, x_k$, by the following rules:

$$\frac{}{\mathcal{S}, |x\ \mathsf{abt}^0| = 1 \vdash |x\ \mathsf{abt}^0| = 1} \tag{6.2a}$$

$$\frac{\mathcal{S} \vdash |a_1\ \mathsf{abt}^{n_1}| = s_1 \quad \dots \quad \mathcal{S} \vdash |a_m\ \mathsf{abt}^{n_m}| = s_m \quad s = s_1 + \dots + s_m + 1}{\mathcal{S} \vdash |o(a_1, \dots, a_m)\ \mathsf{abt}^0| = s} \tag{6.2b}$$

$$\frac{\mathcal{S}, |x'\ \mathsf{abt}^0| = 1 \vdash |[x \leftrightarrow x']\, a\ \mathsf{abt}^n| = s}{\mathcal{S} \vdash |x\,.\,a\ \mathsf{abt}^{n+1}| = s + 1} \tag{6.2c}$$

Thus, the size of an abt is defined inductively counting variables as unit size, and adding one for each operator and abstractor within the abt.

Theorem 6.1. *Every well-formed abt has a unique size. If $x_1\ \mathsf{abt}^0, \dots, x_k\ \mathsf{abt}^0 \vdash a\ \mathsf{abt}^n$, then there exists a unique s nat such that*

$$|x_1\ \mathsf{abt}^0| = 1, \dots, |x_k\ \mathsf{abt}^0| = 1 \vdash |a\ \mathsf{abt}^n| = s.$$

Proof. By structural induction on the derivation of the premise. Note that the size of an abt is not sensitive to the choice of parameters, since all parameters are assigned unit size. It is straightforward to show that this property is closed under the given rules and to show that the size is uniquely determined for well-formed abt's. □

6.1.2 Apartness

The relation of a name, x, *lying apart* from an abt, a, states that a is independent of the variable x. The judgement $\mathcal{A} \vdash x \# a\ \mathsf{abt}^n$, where $\mathcal{A} \vdash a\ \mathsf{abt}^n$, is inductively defined by the following rules:

$$\frac{x \# y}{\mathcal{A} \vdash x \# y\ \mathsf{abt}^0} \tag{6.3a}$$

$$\frac{\mathcal{A} \vdash x \# a_1\ \mathsf{abt}^{n_1} \quad \dots \quad \mathcal{A} \vdash x \# a_k\ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash x \# o(a_1, \dots, a_k)\ \mathsf{abt}^0} \tag{6.3b}$$

$$\frac{\mathcal{A}, y\ \mathsf{abt}^0 \vdash x \mathbin{\#} a\ \mathsf{abt}^n}{\mathcal{A} \vdash x \mathbin{\#} y\,.\,a\ \mathsf{abt}^{n+1}} \tag{6.3c}$$

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 $x \mathbin{\#} a\ \mathsf{abt}$. We leave as an exercise to give an explicit inductive definition of this judgement.

6.1.3 Renaming of Bound Names

Two abt's are said to be *α-equivalent* iff they differ at most in the choice of bound variable names. It is inductively defined by the following rules:

$$\frac{}{\mathcal{A}, x\ \mathsf{abt}^0 \vdash x =_\alpha x\ \mathsf{abt}^0} \tag{6.4a}$$

$$\frac{\mathcal{A} \vdash a_1 =_\alpha b_1\ \mathsf{abt}^{n_1} \quad \dots \quad \mathcal{A} \vdash a_k =_\alpha b_k\ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash o(a_1, \dots, a_k) =_\alpha o(b_1, \dots, b_k)\ \mathsf{abt}^0} \tag{6.4b}$$

$$\frac{\mathcal{A}, z\ \mathsf{abt}^0 \vdash [z \leftrightarrow x]\, a =_\alpha [z \leftrightarrow y]\, b\ \mathsf{abt}^n}{\mathcal{A} \vdash x\,.\,a =_\alpha y\,.\,b\ \mathsf{abt}^{n+1}} \tag{6.4c}$$

In Rule (6.4c) we tacitly assume that the parameter z is chosen apart from those in $\mathcal{A}$.

We write $\mathcal{A} \vdash a =_\alpha b$ for $\mathcal{A} \vdash a =_\alpha b\ \mathsf{abt}^n$ for some n. Further, we sometimes write just $a =_\alpha b$ to mean $\mathcal{A} \vdash a =_\alpha b$ when the appropriate $\mathcal{A}$ is clear from context.

Lemma 6.2. *The following instance of α-equivalence, called α-*conversion*, is derivable:*

$$\mathcal{A} \vdash x\,.\,a =_\alpha y\,.\,[x \leftrightarrow y]\, a\ \mathsf{abt}^{n+1} \qquad (y \mathbin{\#} \mathcal{A}).$$

Theorem 6.3. *α-equivalence is reflexive, symmetric, and transitive.*

Proof. Reflexivity and symmetry are immediately obvious from the form of the definition. Transitivity is proved by a simultaneous induction on the heights of the derivations of $\mathcal{A} \vdash a =_\alpha b\ \mathsf{abt}^n$ and $\mathcal{A} \vdash b =_\alpha c\ \mathsf{abt}^n$. The most interesting case is when both derivations end with Rule (6.4c). We have $a = x\,.\,a'$, $b = y\,.\,b'$, $c = z\,.\,c'$, and $n = m + 1$ for some m. Moreover, $\mathcal{A}, u\ \mathsf{abt}^0 \vdash [u \leftrightarrow x]\, a' =_\alpha [u \leftrightarrow y]\, b'\ \mathsf{abt}^m$, and $\mathcal{A}, v\ \mathsf{abt}^0 \vdash [v \leftrightarrow y]\, b' =_\alpha [v \leftrightarrow z]\, c'\ \mathsf{abt}^m$, for every $u, v \mathbin{\#} \mathcal{A}$. Let $w \mathbin{\#} \mathcal{A}$ be an arbitrary name. By choosing u and v to be w, we obtain the desired result by an application of the inductive hypothesis. □

6.1.4 Capture-Avoiding Substitution

Substitution is the process of replacing all occurrences (if any) of a free name in an abt by another abt in such a way that the scopes of names are properly respected. The judgment $\mathcal{A} \vdash [a/x]b = c \ \mathsf{abt}^n$ is inductively defined by the following rules:

$$\frac{}{\mathcal{A} \vdash [a/x]x = a \ \mathsf{abt}^0} \tag{6.5a}$$

$$\frac{x \ \# \ y}{\mathcal{A} \vdash [a/x]y = y \ \mathsf{abt}^0} \tag{6.5b}$$

$$\frac{\mathcal{A} \vdash [a/x]b_1 = c_1 \ \mathsf{abt}^{n_1} \quad \dots \quad \mathcal{A} \vdash [a/x]b_k = c_k \ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash [a/x]o(b_1, \dots, b_k) = o(c_1, \dots, c_k) \ \mathsf{abt}^0} \tag{6.5c}$$

$$\frac{\mathcal{A}, y' \ \mathsf{abt}^0 \vdash [a/x]([y' \leftrightarrow y] \, b) = b' \ \mathsf{abt}^n \quad y' \ \# \ \mathcal{A} \quad y' \neq x}{\mathcal{A} \vdash [a/x]y \, . \, b = y' \, . \, b' \ \mathsf{abt}^n} \tag{6.5d}$$

In Rule (6.5d) the requirement that $y' \ \# \ \mathcal{A}$ ensures that $y' \ \# \ a$, and the requirement that $y' \neq x$ ensures that we do not confuse y' with x. Since the bound name, y, of the abstractor might well occur within $\mathcal{A}$, it may also occur in a. This necessitates that y be renamed to a fresh name y' before substituting a into the body of the abstractor. The potential confusion of an occurrence of y within a with the bound variable of the abstractor is called *capture*, and for this reason substitution as defined here is called *capture-avoiding substitution*.

The penalty for avoiding capture during substitution is that the result of performing a substitution is determined only up to α-equivalence. Observe that in the conclusion of Rule (6.5d), we have $y \, . \, [y \leftrightarrow y'] \, b' =_\alpha y' \, . \, b'$, provided that $y \ \# \ \mathcal{A}$, by Lemma 6.2 on the preceding page. If, on the contrary, y occurs within $\mathcal{A}$, then the equivalence does not apply, and, as a consequence, we cannot preserve the bound name after substitution.

Theorem 6.4. *If $\mathcal{A} \vdash a \ \mathsf{abt}^0$ and $\mathcal{A}, x \ \mathsf{abt}^0 \vdash b \ \mathsf{abt}^n$, then there exists $\mathcal{A} \vdash c \ \mathsf{abt}^n$ such that $\mathcal{A} \vdash [a/x]b = c \ \mathsf{abt}^n$. If $\mathcal{A} \vdash [a/x]b = c \ \mathsf{abt}^n$ and $\mathcal{A} \vdash [a/x]b = c' \ \mathsf{abt}^n$, then $\mathcal{A} \vdash c =_\alpha c' \ \mathsf{abt}^n$.*

Proof. The first part is proved by rule induction on $\mathcal{A}, x \ \mathsf{abt}^0 \vdash b \ \mathsf{abt}^n$, in each case constructing the required derivation of the substitution judgement. The second part is proved by simultaneous rule induction on the two premises, deriving the desired equivalence in each case. □

Even though the result is not uniquely determined, we abuse notation and write $[a/x]b$ for any c such that $[a/x]b = c$, with the understanding that c

is determined only up to choice of bound names. To ensure that this convention is sensible, we will ensure that all judgements on abt's are defined so as to respect α-equivalence—in particular, substitution itself enjoys this property.

Theorem 6.5. *If $\mathcal{A} \vdash a =_\alpha a' \ \mathsf{abt}^0$, $\mathcal{A}, x \ \mathsf{abt}^0 \vdash b =_\alpha b' \ \mathsf{abt}^n$, $\mathcal{A} \vdash [a/x]b = c \ \mathsf{abt}^n$ and $\mathcal{A} \vdash [a'/x]b' = c' \ \mathsf{abt}^n$, then $\mathcal{A} \vdash c =_\alpha c' \ \mathsf{abt}^n$.*

Proof. By rule induction on $\mathcal{A}, x \ \mathsf{abt}^0 \vdash b =_\alpha b' \ \mathsf{abt}^n$. □

More generally, we will henceforth insist that all judgements respect α-equivalence of abt's. This allows us to tacitly assume that the bound variable of an abstractor may be chosen so as to satisfy any finite constraint that we may wish to impose in a given context without further comment. Using this convention we may write the formation rule for abstractors in the simplified form

$$\frac{\mathcal{A}, x \ \mathsf{abt}^0 \vdash a \ \mathsf{abt}^n}{\mathcal{A} \vdash x\,.\,a \ \mathsf{abt}^{n+1}} \tag{6.6}$$

with the tacit understanding that x is to be chosen so that $x \ \# \ \mathcal{A}$. This convention extends the convention on parametric judgements stated in Chapter 3 to include the abt's occurring within the judgement, as well as the parameters of the judgement itself. As a consequence we may always assume that parameters and bound variables are chosen so as to be as fresh as needed in any given context, provided that only finitely many variables need be avoided by the choice.

6.2 Exercises

1. Show that the structural rule of weakening is *not* admissible for the conditional inductive definition of abstract binding trees (Rules (6.1)).

2. Suppose that `let` is an operator of arity $(0, 1)$ and that `plus` is an operator of arity $(0, 0)$. Determine whether or not each of the following α-equivalences are valid.

$$\texttt{let}(x, x\,.\,x) =_\alpha \texttt{let}(x, y\,.\,y) \tag{6.7a}$$

$$\texttt{let}(y, x\,.\,x) =_\alpha \texttt{let}(y, y\,.\,y) \tag{6.7b}$$

$$\texttt{let}(x, x\,.\,x) =_\alpha \texttt{let}(y, y\,.\,y) \tag{6.7c}$$

$$\texttt{let}(x, x\,.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, z\,.\texttt{plus}(z, y)) \tag{6.7d}$$

$$\texttt{let}(x, x\,.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, y\,.\texttt{plus}(y, y)) \tag{6.7e}$$

3. Prove that apartness respects α-equivalence.

4. Prove that substitution respects α-equivalence.

Chapter 7

Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings 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 for eliminating ambiguity, improving readability is, to a large extent, a matter of taste.

In this chapter we introduce the main methods for specifying concrete syntax, using as an example an illustrative expression language, called $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, that supports elementary arithmetic on the natural numbers and simple computations on strings. In addition, $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ includes a construct for binding the value of an expression to a variable within a specified scope.

7.1 Lexical Structure

The first phase of syntactic processing is to convert from a character-based representation to a symbol-based representation of the input. This is called *lexical analysis*, or *lexing*. The main idea is to aggregate characters into symbols that serve as tokens for subsequent phases of analysis. For example, the numeral `467` is written as a sequence of three consecutive characters, one for each digit, but is regarded as a single token, namely the number 467. Similarly, an identifier such as `temp` comprises four letters, but is treated as a single symbol representing the entire word. Moreover, many character-based representations include empty "white space" (spaces, tabs, newlines, and, perhaps, comments) that are discarded by the lexical analyzer.[1]

[1] In some languages white space *is* significant, in which case it must be converted to symbolic form for subsequent processing.

The character representation of symbols is, in most cases, conveniently described using *regular expressions*. The lexical structure of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is specified as follows:

$$
\begin{array}{lrcl}
\text{Item} & \mathsf{itm} & ::= & \mathsf{kwd} \mid \mathsf{id} \mid \mathsf{num} \mid \mathsf{lit} \mid \mathsf{spl} \\
\text{Keyword} & \mathsf{kwd} & ::= & \mathtt{l}\cdot\mathtt{e}\cdot\mathtt{t}\cdot\epsilon \mid \mathtt{b}\cdot\mathtt{e}\cdot\epsilon \mid \mathtt{i}\cdot\mathtt{n}\cdot\epsilon \\
\text{Identifier} & \mathsf{id} & ::= & \mathsf{ltr}\,(\mathsf{ltr} \mid \mathsf{dig})^{*} \\
\text{Numeral} & \mathsf{num} & ::= & \mathsf{dig}\,\mathsf{dig}^{*} \\
\text{Literal} & \mathsf{lit} & ::= & \mathsf{qum}\,(\mathsf{ltr} \mid \mathsf{dig})^{*}\mathsf{qum} \\
\text{Special} & \mathsf{spl} & ::= & \mathtt{+} \mid \mathtt{*} \mid \mathtt{\hat{}} \mid \mathtt{(} \mid \mathtt{)} \mid \mathtt{|} \\
\text{Letter} & \mathsf{ltr} & ::= & \mathtt{a} \mid \mathtt{b} \mid \ldots \\
\text{Digit} & \mathsf{dig} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots \\
\text{Quote} & \mathsf{qum} & ::= & \mathtt{"}
\end{array}
$$

A lexical item is either a keyword, an identifier, a numeral, a string literal, or a special symbol. There are three keywords, specified as sequences of characters, for emphasis. Identifiers start with a letter and may involve subsequent letters or digits. Numerals are non-empty sequences of digits. String literals are sequences of letters or digits surrounded by quotes. The special symbols, letters, digits, and quote marks are as enumerated. (Observe that we tacitly identify a character with the unit-length string consisting of that character.)

The job of the lexical analyzer is to translate character strings into token strings using the above definitions as a guide. An input string is scanned, ignoring white space, and translating lexical items into tokens, which are specified by the following rules:

$$\frac{s\ \mathsf{str}}{\mathtt{ID}[s]\ \mathsf{tok}} \tag{7.1a}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM}[n]\ \mathsf{tok}} \tag{7.1b}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT}[s]\ \mathsf{tok}} \tag{7.1c}$$

$$\frac{}{\mathtt{LET}\ \mathsf{tok}} \tag{7.1d}$$

$$\frac{}{\mathtt{BE}\ \mathsf{tok}} \tag{7.1e}$$

$$\frac{}{\mathtt{IN}\ \mathsf{tok}} \tag{7.1f}$$

$$\frac{}{\mathtt{ADD}\ \mathsf{tok}} \tag{7.1g}$$

$$\frac{}{\mathtt{MUL}\ \mathsf{tok}} \tag{7.1h}$$

$$\frac{}{\texttt{CAT}\ \mathsf{tok}} \tag{7.1i}$$

$$\frac{}{\texttt{LP}\ \mathsf{tok}} \tag{7.1j}$$

$$\frac{}{\texttt{RP}\ \mathsf{tok}} \tag{7.1k}$$

$$\frac{}{\texttt{VB}\ \mathsf{tok}} \tag{7.1l}$$

Lexical analysis is inductively defined by the following judgement forms:

$s\ \mathsf{inp} \longleftrightarrow t\ \mathsf{tokstr}$	Scan input
$s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}$	Scan an item
$s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}$	Scan a keyword
$s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}$	Scan an identifier
$s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}$	Scan a number
$s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}$	Scan a symbol
$s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}$	Scan a string literal
$s\ \mathsf{whs}$	Skip white space

The definition of these forms, which follows, makes use of several auxiliary judgements corresponding to the classifications of characters in the lexical structure of the language. For example, $s\ \mathsf{whs}$ states that the string s consists only of "white space", and $s\ \mathsf{lord}$ states that s is either an alphabetic letter or a digit, and so forth.

$$\frac{}{\epsilon\ \mathsf{inp} \longleftrightarrow \epsilon\ \mathsf{tokstr}} \tag{7.2a}$$

$$\frac{s = s_1\hat{}\,s_2\hat{}\,s_3\ \mathsf{str} \quad s_1\ \mathsf{whs} \quad s_2\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok} \quad s_3\ \mathsf{inp} \longleftrightarrow ts\ \mathsf{tokstr}}{s\ \mathsf{inp} \longleftrightarrow t \cdot ts\ \mathsf{tokstr}} \tag{7.2b}$$

$$\frac{s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2c}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2d}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2e}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2f}$$

$$\frac{s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{7.2g}$$

$$\frac{s = \texttt{l} \cdot \texttt{e} \cdot \texttt{t} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \texttt{LET}\ \mathsf{tok}} \tag{7.2h}$$

$$\frac{s = \texttt{b} \cdot \texttt{e} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \texttt{BE}\ \mathsf{tok}} \tag{7.2i}$$

$$\frac{s = \mathtt{i} \cdot \mathtt{n} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \mathtt{IN}\ \mathsf{tok}} \tag{7.2j}$$

$$\frac{s = s_1\,\hat{}\,s_2\ \mathsf{str} \quad s_1\ \mathsf{ltr} \quad s_2\ \mathsf{lord}}{s\ \mathsf{id} \longleftrightarrow \mathtt{ID[}s\mathtt{]}\ \mathsf{tok}} \tag{7.2k}$$

$$\frac{s = s_1\,\hat{}\,s_2\ \mathsf{str} \quad s_1\ \mathsf{dig} \quad s_2\ \mathsf{dgs} \quad s\ \mathsf{num} \longleftrightarrow n\ \mathsf{nat}}{s\ \mathsf{num} \longleftrightarrow \mathtt{NUM[}n\mathtt{]}\ \mathsf{tok}} \tag{7.2l}$$

$$\frac{s = s_1\,\hat{}\,s_2\,\hat{}\,s_3\ \mathsf{str} \quad s_1\ \mathsf{qum} \quad s_2\ \mathsf{lord} \quad s_3\ \mathsf{qum}}{s\ \mathsf{lit} \longleftrightarrow \mathtt{LIT[}s_2\mathtt{]}\ \mathsf{tok}} \tag{7.2m}$$

$$\frac{s = \mathtt{+} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{ADD}\ \mathsf{tok}} \tag{7.2n}$$

$$\frac{s = \mathtt{*} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{MUL}\ \mathsf{tok}} \tag{7.2o}$$

$$\frac{s = \hat{}\ \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{CAT}\ \mathsf{tok}} \tag{7.2p}$$

$$\frac{s = \mathtt{(} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{LP}\ \mathsf{tok}} \tag{7.2q}$$

$$\frac{s = \mathtt{)} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{RP}\ \mathsf{tok}} \tag{7.2r}$$

$$\frac{s = \mathtt{|} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{VB}\ \mathsf{tok}} \tag{7.2s}$$

By convention Rule (7.2k) applies only if none of Rules (7.2h) to (7.2j) apply. Technically, Rule (7.2k) has implicit premises that rule out keywords as possible identifiers.

7.2 Context-Free Grammars

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

1. The *tokens*, or *terminals*, over which the grammar is defined.

2. The *syntactic classes*, or *non-terminals*, which are disjoint from the terminals.

3. The *rules*, or *productions*, which have the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Each syntactic class is a collection of token strings. The rules determine which strings belong to which syntactic classes.

When defining a grammar, we often abbreviate a set of productions,

$$\begin{aligned} A &::= \alpha_1 \\ &\vdots \\ A &::= \alpha_n, \end{aligned}$$

each with the same left-hand side, by the *compound* production

$$A ::= \alpha_1 \mid \ldots \mid \alpha_n,$$

which specifies a set of alternatives for the syntactic class A.

A context-free grammar determines a simultaneous inductive definition of its syntactic classes. Specifically, we regard each non-terminal, A, as a judgement form, $s\ A$, over strings of terminals. To each production of the form

$$A ::= s_1\, A_1\, s_2\, \ldots\, s_n\, A_n\, s_{n+1} \tag{7.3}$$

we associate an inference rule

$$\frac{s_1'\ A_1 \quad \ldots \quad s_n'\ A_n}{s_1\, s_1'\, s_2\, \ldots\, s_n\, s_n'\, s_{n+1}\ A}\ . \tag{7.4}$$

The collection of all such rules constitutes an inductive definition of the syntactic classes of the grammar.

Recalling that juxtaposition of strings is short-hand for their concatenation, we may re-write the preceding rule as follows:

$$\frac{s_1'\ A_1 \quad \ldots \quad s_n'\ A_n \quad s = s_1\,\hat{}\,s_1'\,\hat{}\,s_2\,\hat{}\ldots s_n\,\hat{}\,s_n'\,\hat{}\,s_{n+1}}{s\ A}\ . \tag{7.5}$$

This formulation makes clear that $s\ A$ holds whenever s can be partitioned as described so that $s_i'\ A$ for each $1 \leq i \leq n$. Since string concatenation is not invertible, the decomposition is not unique, and so there may be many different ways in which the rule applies.

7.3 Grammatical Structure

The concrete syntax of $\mathcal{L}\{$`num str`$\}$ may be specified by a context-free grammar over the tokens defined in Section 7.1 on page 49. The grammar has only one syntactic class, exp, which is defined by the following compound production:

$$\begin{array}{llcl}
\text{Expression} & \mathsf{exp} & ::= & \mathsf{num} \mid \mathsf{lit} \mid \mathsf{id} \mid \mathtt{LP}\,\mathsf{exp}\,\mathtt{RP} \mid \mathsf{exp}\,\mathtt{ADD}\,\mathsf{exp} \mid \\
 & & & \mathsf{exp}\,\mathtt{MUL}\,\mathsf{exp} \mid \mathsf{exp}\,\mathtt{CAT}\,\mathsf{exp} \mid \mathtt{VB}\,\mathsf{exp}\,\mathtt{VB} \mid \\
 & & & \mathtt{LET}\,\mathsf{id}\,\mathtt{BE}\,\mathsf{exp}\,\mathtt{IN}\,\mathsf{exp} \\
\text{Number} & \mathsf{num} & ::= & \mathtt{NUM}[n] \qquad (n\ \mathsf{nat}) \\
\text{String} & \mathsf{lit} & ::= & \mathtt{LIT}[s] \qquad (s\ \mathsf{str}) \\
\text{Identifier} & \mathsf{id} & ::= & \mathtt{ID}[s] \qquad (s\ \mathsf{str})
\end{array}$$

This grammar makes use of some standard notational conventions to improve readability: we identify a token with the corresponding unit-length string, and we use juxtaposition to denote string concatenation.

Applying the interpretation of a grammar as an inductive definition, we obtain the following rules:

$$\frac{s\ \mathsf{num}}{s\ \mathsf{exp}} \tag{7.6a}$$

$$\frac{s\ \mathsf{lit}}{s\ \mathsf{exp}} \tag{7.6b}$$

$$\frac{s\ \mathsf{id}}{s\ \mathsf{exp}} \tag{7.6c}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{ADD}\,s_2\ \mathsf{exp}} \tag{7.6d}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{MUL}\,s_2\ \mathsf{exp}} \tag{7.6e}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\,\mathtt{CAT}\,s_2\ \mathsf{exp}} \tag{7.6f}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{VB}\,s\,\mathtt{VB}\ \mathsf{exp}} \tag{7.6g}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{LP}\,s\,\mathtt{RP}\ \mathsf{exp}} \tag{7.6h}$$

$$\frac{s_1\ \mathsf{id} \quad s_2\ \mathsf{exp} \quad s_3\ \mathsf{exp}}{\mathtt{LET}\,s_1\,\mathtt{BE}\,s_2\,\mathtt{IN}\,s_3\ \mathsf{exp}} \tag{7.6i}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM}[n]\ \mathsf{num}} \tag{7.6j}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT}[s]\ \mathsf{lit}} \tag{7.6k}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID}[s]\ \mathsf{id}} \tag{7.6l}$$

To emphasize the role of string concatentation, we may rewrite Rule (7.6e), for example, as follows:

$$\frac{s = s_1\,\texttt{MUL}\,s_2\ \mathsf{str}\qquad s_1\ \mathsf{exp}\qquad s_2\ \mathsf{exp}}{s\ \mathsf{exp}}\;. \tag{7.7}$$

That is, s exp is derivable if s is the concatentation of s_1, the multiplication sign, and s_2, where s_1 exp and s_2 exp.

7.4 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 token strings may be thought of as arising in several different ways. More precisely, there are token strings s for which there is more than one derivation ending with s exp according to Rules (7.6).

For example, consider the character string 1+2*3, which, after lexical analysis, is translated to the token string

$$\texttt{NUM[1]}\ \texttt{ADD}\ \texttt{NUM[2]}\ \texttt{MUL}\ \texttt{NUM[3]}.$$

Since string concatenation is associative, this token string can be thought of as arising in several ways, including

$$\texttt{NUM[1]}\ \texttt{ADD}\,{}_\wedge\texttt{NUM[2]}\ \texttt{MUL}\ \texttt{NUM[3]}$$

and

$$\texttt{NUM[1]}\ \texttt{ADD}\ \texttt{NUM[2]}\,{}_\wedge\,\texttt{MUL}\ \texttt{NUM[3]},$$

where the caret indicates the concatenation point.

One consequence of this observation is that the same token string may be seen to be grammatical according to the rules given in Section 7.3 on page 53 in two different ways. According to the first reading, the expression is principally an addition, with the first argument being a number, and the second being a multiplication of two numbers. According to the second reading, the expression is principally a multiplication, with the first argument being the addition of two numbers, and the second being a number.

Ambiguity is a *purely syntactic* property of grammars; it has nothing to do with the "meaning" of a string. For example, the token string

$$\texttt{NUM[1]}\ \texttt{ADD}\ \texttt{NUM[2]}\ \texttt{ADD}\ \texttt{NUM[3]},$$

also admits two readings. It is immaterial that both readings have the same meaning under the usual interpretation of arithmetic expressions. Moreover, nothing prevents us from interpreting the token `ADD` to mean "division," in which case the two readings would hardly coincide! Nothing in the syntax itself precludes this interpretation, so we do not regard it as relevant to whether the grammar is ambiguous.

To eliminate ambiguity the grammar of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ given in Section 7.3 on page 53 must be re-structured to ensure that every grammatical string has at most one derivation according to the rules of the grammar. The main method for achieving this is to introduce precedence and associativity conventions that ensure there is only one reading of any token string. Parenthesization may be used to override these conventions, so there is no fundamental loss of expressive power in doing so.

Precedence relationships are introduced by *layering* the grammar, which is achieved by splitting syntactic classes into several sub-classes.

$$
\begin{array}{lrcl}
\text{Factor} & \mathsf{fct} & ::= & \mathsf{num} \mid \mathsf{lit} \mid \mathsf{id} \mid \mathtt{LP}\,\mathsf{prg}\,\mathtt{RP} \\
\text{Term} & \mathsf{trm} & ::= & \mathsf{fct} \mid \mathsf{fct}\,\mathtt{MUL}\,\mathsf{trm} \mid \mathtt{VB}\,\mathsf{fct}\,\mathtt{VB} \\
\text{Expression} & \mathsf{exp} & ::= & \mathsf{trm} \mid \mathsf{trm}\,\mathtt{ADD}\,\mathsf{exp} \mid \mathsf{trm}\,\mathtt{CAT}\,\mathsf{exp} \\
\text{Program} & \mathsf{prg} & ::= & \mathsf{exp} \mid \mathtt{LET}\,\mathsf{id}\,\mathtt{BE}\,\mathsf{exp}\,\mathtt{IN}\,\mathsf{prg}
\end{array}
$$

The effect of this grammar is to ensure that `let` has the lowest precedence, addition and concatenation intermediate precedence, and multiplication and length the highest precedence. Moreover, all forms are right-associative. Other choices of rules are possible, according to taste; this grammar illustrates one way to resolve the ambiguities of the original expression grammar.

7.5 Exercises

Chapter 8

Abstract Syntax

The concrete syntax of a language is concerned with the linear representation of the phrases of a language as strings of symbols—the form in which we write them on paper, type them into a computer, and read them from a page. The main goal of concrete syntax design is to enhance the readability and writability of the language, based on subjective criteria such as similarity to other languages, ease of editing using standard tools, and so forth.

But languages are also the subjects of study, as well as the instruments of expression. As such the concrete syntax of a language is just a nuisance. When analyzing a language mathematically we are only interested in the deep structure of its phrases, not their surface representation. The *abstract syntax* of a language exposes the hierarchical and binding structure of the language, and suppresses the linear notation used to write it on the page.

Parsing is the process of translation from concrete to abstract syntax. It consists of analyzing the linear representation of a phrase in terms of the grammar of the language and transforming it into an abstract syntax tree or an abstract binding tree that reveals the deep structure of the phrase.

8.1 Abstract Syntax Trees

The abstract syntax tree representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified by the following signature:

$$\begin{aligned}
\mathsf{ar}(\texttt{num[}n\texttt{]}) &= 0 \quad (n\ \mathsf{nat})\\
\mathsf{ar}(\texttt{str[}s\texttt{]}) &= 0 \quad (s\ \mathsf{str})\\
\mathsf{ar}(\texttt{id[}s\texttt{]}) &= 0 \quad (s\ \mathsf{str})\\
\mathsf{ar}(\texttt{plus}) &= 2\\
\mathsf{ar}(\texttt{times}) &= 2\\
\mathsf{ar}(\texttt{cat}) &= 2\\
\mathsf{ar}(\texttt{len}) &= 1\\
\mathsf{ar}(\texttt{let[}s\texttt{]}) &= 2
\end{aligned}$$

Observe that each identifier is regarded as operators of arity 0, and that the `let` construct is regarded as a family of operators of arity two, indexed by the identifier that it binds.

Specializing the rules for abstract syntax trees to this signature, we obtain the following inductive definition of the abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$:

$$\frac{n\ \mathsf{nat}}{\texttt{num[}n\texttt{]}\ \mathsf{ast}} \tag{8.1a}$$

$$\frac{s\ \mathsf{str}}{\texttt{str[}s\texttt{]}\ \mathsf{ast}} \tag{8.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{id[}s\texttt{]}\ \mathsf{ast}} \tag{8.1c}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{plus(}a_1\texttt{;}a_2\texttt{)}\ \mathsf{ast}} \tag{8.1d}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{times(}a_1\texttt{;}a_2\texttt{)}\ \mathsf{ast}} \tag{8.1e}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{cat(}a_1\texttt{;}a_2\texttt{)}\ \mathsf{ast}} \tag{8.1f}$$

$$\frac{a\ \mathsf{ast}}{\texttt{len(}a\texttt{)}\ \mathsf{ast}} \tag{8.1g}$$

$$\frac{s\ \mathsf{id} \quad a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{let[}s\texttt{](}a_1\texttt{;}a_2\texttt{)}\ \mathsf{ast}} \tag{8.1h}$$

Strictly speaking, the last rule is a specialization of the rule induced by the arity assignment for `let` in which we demand that the first argument be an identifier.

8.2 Parsing Into Abstract Syntax Trees

The process of translation from concrete to abstract syntax is called *parsing*. We will define parsing as a judgement between the concrete and abstract syntax of a language. This judgement will have the mode $(\forall, \exists^{\leq 1})$ over strings and ast's, which states that the parser is a partial function of its input, being undefined for ungrammatical token strings, but otherwise uniquely determining the abstract syntax tree representation of each well-formed input.

The parsing judgements for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ follow the unambiguous grammar given in Chapter 7:

$s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}$	Parse as a program
$s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}$	Parse as an expression
$s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}$	Parse as a term
$s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}$	Parse as a factor
$s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}$	Parse as a number
$s\ \mathsf{lit} \longleftrightarrow a\ \mathsf{ast}$	Parse as a literal
$s\ \mathsf{id} \longleftrightarrow a\ \mathsf{ast}$	Parse as an identifier

These judgements are inductively defined simultaneously by the following rules:

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM[}n\mathtt{]}\ \mathsf{num} \longleftrightarrow \mathtt{num[}n\mathtt{]}\ \mathsf{ast}} \tag{8.2a}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT[}s\mathtt{]}\ \mathsf{lit} \longleftrightarrow \mathtt{str[}s\mathtt{]}\ \mathsf{ast}} \tag{8.2b}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID[}s\mathtt{]}\ \mathsf{id} \longleftrightarrow \mathtt{id[}s\mathtt{]}\ \mathsf{ast}} \tag{8.2c}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2d}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2e}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2f}$$

$$\frac{s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}}{\mathtt{LP}\,s\,\mathtt{RP}\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2g}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2h}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{trm} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\,\mathtt{MUL}\,s_2\ \mathsf{trm} \longleftrightarrow \mathtt{times(}a_1\mathtt{;}a_2\mathtt{)}\ \mathsf{ast}} \tag{8.2i}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{\mathtt{VB}\,s\,\mathtt{VB}\ \mathsf{trm} \longleftrightarrow \mathtt{len}(a)\ \mathsf{ast}} \tag{8.2j}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2k}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\,\mathtt{ADD}\,s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus}(a_1;a_2)\ \mathsf{ast}} \tag{8.2l}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\,\mathtt{CAT}\,s_2\ \mathsf{exp} \longleftrightarrow \mathtt{cat}(a_1;a_2)\ \mathsf{ast}} \tag{8.2m}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}} \tag{8.2n}$$

$$\frac{s_1\ \mathsf{id} \longleftrightarrow \mathtt{id}[s]\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast} \quad s_3\ \mathsf{prg} \longleftrightarrow a_3\ \mathsf{ast}}{\mathtt{LET}\,s_1\,\mathtt{BE}\,s_2\,\mathtt{IN}\,s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}[s](a_2;a_3)\ \mathsf{ast}} \tag{8.2o}$$

A successful parse implies that the token string must have been derived according to the rules of the unambiguous grammar and that the result is a well-formed abstract syntax tree.

Theorem 8.1. *If s $\mathsf{prg} \longleftrightarrow a$ ast, then s prg and a ast, and similarly for the other parsing judgements.*

Proof. By rule induction on Rules (8.2). □

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

Theorem 8.2. *If s prg, then there is a unique a such that s $\mathsf{prg} \longleftrightarrow a$ ast, and similarly for the other parsing judgements. That is, the parsing judgements have mode $(\forall, \exists!)$ over the class of well-formed strings and abstract syntax trees.*

Proof. By rule induction on the rules determined by reading Grammar (7.4) as an inductive definition. □

Finally, any piece of abstract syntax may be formatted as a string that parses as the given ast.

Theorem 8.3. *If a ast, then there exists a (not necessarily unique) string s such that s prg and s $\mathsf{prg} \longleftrightarrow a$ ast. That is, the parsing judgement has mode $(\exists, \forall)$.*

Proof. By rule induction on Grammar (7.4). □

The string representation of an abstract syntax tree is not unique, since we may introduce parentheses at will around any sub-expression.

8.3 Parsing Into Abstract Binding Trees

The representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using abstract syntax trees exposes the hierarchical structure of the language, but does not manage the binding and scope of variables in a `let` expression. In this section we revise the parser given in Section 8.1 on page 58 to translate from token strings (as before) to abstract binding trees to make explicit the binding and scope of identifiers in a program.

The abstract binding tree representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified by the following assignment of (generalized) arities to operators:

$$\begin{aligned}
\mathsf{ar}(\texttt{num}[n]) &= () \\
\mathsf{ar}(\texttt{str}[s]) &= () \\
\mathsf{ar}(\texttt{plus}) &= (0,0) \\
\mathsf{ar}(\texttt{times}) &= (0,0) \\
\mathsf{ar}(\texttt{cat}) &= (0,0) \\
\mathsf{ar}(\texttt{len}) &= (0) \\
\mathsf{ar}(\texttt{let}) &= (0,1)
\end{aligned}$$

The arity of the operator `let` specifies that it takes two arguments, the second of which is an abstractor of valence 1, meaning that it binds one variable in the second argument position. Observe that identifiers are no longer declared as operators; instead, identifiers are translated by the parser into variables. Similarly, parentheses are "parsed away" on passage to abstract syntax, and thus have no representation as operators.

The revised parsing judgement, $s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt}$, between strings s and abt's a, is defined by a collection of rules similar to those given in Section 8.2 on page 59. These rules take the form of a parametric inductive definition (see Chapter 2) in which the premises and conclusions of the rules involve hypothetical judgments of the form

$$\texttt{ID}[s_1]\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{abt}, \ldots, \texttt{ID}[s_n]\ \mathsf{id} \longleftrightarrow x_n\ \mathsf{abt} \vdash s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt},$$

where the x_i's are pairwise distinct variable names. The hypotheses of the judgement dictate how identifiers are to be parsed as variables, for it follows from the reflexivity of the hypothetical judgement that

$$\Gamma, \texttt{ID}[s]\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt} \vdash \texttt{ID}[s]\ \mathsf{id} \longleftrightarrow x\ \mathsf{abt}.$$

To maintain the association between identifiers and variables when parsing a `let` expression, we update the hypotheses to record the association between the bound identifier and a corresponding variable:

$$\frac{\Gamma \vdash s_1 \ \mathsf{id} \longleftrightarrow x \ \mathsf{abt} \qquad \Gamma \vdash s_2 \ \mathsf{exp} \longleftrightarrow a_2 \ \mathsf{abt} \qquad \Gamma, s_1 \ \mathsf{id} \longleftrightarrow x \ \mathsf{abt} \vdash s_3 \ \mathsf{prg} \longleftrightarrow a_3 \ \mathsf{abt}}{\Gamma \vdash \mathtt{LET}\, s_1 \, \mathtt{BE}\, s_2 \, \mathtt{IN}\, s_3 \ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_2; x\,.\,a_3) \ \mathsf{abt}} \tag{8.3a}$$

Unfortunately, this approach does not quite work properly! If an inner `let` expression binds the same identifier as an outer `let` expression, there is an ambiguity in how to parse occurrences of that identifier. Parsing such nested `let`'s will introduce two hypotheses, say $\mathtt{ID}[s] \ \mathsf{id} \longleftrightarrow x_1 \ \mathsf{abt}$ and $\mathtt{ID}[s] \ \mathsf{id} \longleftrightarrow x_2 \ \mathsf{abt}$, for the same identifier $\mathtt{ID}[s]$. By the structural property of exchange, we may choose arbitrarily which to apply to any particular occurrence of $\mathtt{ID}[s]$, and hence we may parse different occurrences differently.

To rectify this we must resort to less elegant methods. Rather than use hypotheses, we instead maintain an explicit *symbol table* to record the association between identifiers and variables. We must define explicitly the procedures for creating and extending symbol tables, and for looking up an identifier in the symbol table to determine its associated variable. This gives us the freedom to implement a *shadowing* policy for re-used identifiers, according to which the most recent binding of an identifier determines the corresponding variable.

The main change to the parsing judgement is that the hypothetical judgement

$$\Gamma \vdash s \ \mathsf{prg} \longleftrightarrow a \ \mathsf{abt}$$

is reduced to the categorical judgement

$$s \ \mathsf{prg} \longleftrightarrow a \ \mathsf{abt} \ [\sigma],$$

where σ is a symbol table. (Analogous changes must be made to the other parsing judgements.) The symbol table is now an argument to the judgement form, rather than an implicit mechanism for performing inference under hypotheses.

The rule for parsing `let` expressions is then formulated as follows:

$$\frac{s_1 \ \mathsf{id} \longleftrightarrow x \ [\sigma] \qquad s_2 \ \mathsf{exp} \longleftrightarrow a_2 \ \mathsf{abt} \ [\sigma] \qquad \sigma' = \sigma[s_1 \mapsto x] \qquad s_3 \ \mathsf{prg} \longleftrightarrow a_3 \ \mathsf{abt} \ [\sigma']}{\mathtt{LET}\, s_1 \, \mathtt{BE}\, s_2 \, \mathtt{IN}\, s_3 \ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_2; x\,.\,a_3) \ \mathsf{abt} \ [\sigma]} \tag{8.4}$$

This rule is quite similar to the hypothetical form, the difference being that we must manage the symbol table explicitly. In particular, we must include a rule for parsing identifiers, rather than relying on the reflexivity of the hypothetical judgement to do it for us.

$$\frac{\sigma(\texttt{ID[}s\texttt{]}) = x}{\texttt{ID[}s\texttt{]}\ \mathsf{id} \longleftrightarrow x\ [\sigma]} \tag{8.5}$$

The premise of this rule states that σ maps the identifier $\texttt{ID[}s\texttt{]}$ to the variable x.

Symbol tables may be defined to be finite sequences of ordered pairs of the form $(\texttt{ID[}s\texttt{]},x)$, where $\texttt{ID[}s\texttt{]}$ is an identifier and x is a variable name. Using this representation it is straightforward to define the following judgement forms:

σ symtab	well-formed symbol table
$\sigma' = \sigma[\texttt{ID[}s\texttt{]} \mapsto x]$	add new association
$\sigma(\texttt{ID[}s\texttt{]}) = x$	lookup identifier

We leave the precise definitions of these judgements as an exercise for the reader.

8.4 Syntactic Conventions

To specify a language we shall use a concise tabular notation for simultaneously specifying both its abstract and concrete syntax. Officially, the language is always a collection of abt's, but when writing examples we shall often use the concrete notation for the sake of concision and clarity. Our method of specifying the concrete syntax is sufficient for our purposes, but leaves out niggling details such as precedences of operators or the use of bracketing to disambiguate.

The method is best illustrated by example. Here is a specification of the syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ presented in the tabular style that we shall use throughout the book:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{num}$	$\texttt{num}$
		$\mid$	$\texttt{str}$	$\texttt{str}$
Expr	e	$::=$	x	x
		$\mid$	$\texttt{num[}n\texttt{]}$	n
		$\mid$	$\texttt{str[}s\texttt{]}$	$\texttt{"}s\texttt{"}$
		$\mid$	$\texttt{plus(}e_1\texttt{;}e_2\texttt{)}$	$e_1\texttt{+}e_2$
		$\mid$	$\texttt{times(}e_1\texttt{;}e_2\texttt{)}$	$e_1\texttt{*}e_2$
		$\mid$	$\texttt{cat(}e_1\texttt{;}e_2\texttt{)}$	$e_1\texttt{\^{}}e_2$
		$\mid$	$\texttt{len(}e\texttt{)}$	$\vert e \vert$
		$\mid$	$\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)}$	$\texttt{let}\,x\,\texttt{be}\,e_1\,\texttt{in}\,e_2$

This specification is to be understood as defining two judgments, τ type and τ exp, which specify two classes of abstract binding trees, one for types, the other for expressions. The abstract syntax column uses patterns ranging over abt's to determine the arities of the operators for that syntactic class. The concrete syntax column specifies the typical notational conventions used in examples. In this manner Table (8.4) defines two signatures, Ω_{type} and Ω_{expr}, that specify the operators for types and expressions, respectively. The signature for types specifies that num and str are two operators of arity $()$. The signature for expressions specifies two families of operators, $\texttt{num[}n\texttt{]}$ and $\texttt{str[}s\texttt{]}$, of arity $()$, three operators of arity $(0,0)$ corresponding to addition, multiplication, and concatenation, one operator of arity (0) for length, and one operator of arity $(0,1)$ for let-binding expressions to identifiers.

8.5 Exercises

Part III

Static and Dynamic Semantics

Chapter 9

Static Semantics

The *static semantics* of a language consists of a collection of rules for imposing constraints on the formation of programs, called a *type system*. Phrases of the language are classified by *types*, which govern how they may be used in combination with other phrases. Roughly speaking, the type of a phrase predicts the form of its value, and a phrase is said to be *well-typed* if it is constructed consistently with these predictions. For example, the sum of two expressions of numeric type is itself of numeric type, which expresses the evident fact that the sum of two numbers is itself a number. On the other hand, the sum of an expression of string type with any other expression is *ill-typed*, expressing that addition is undefined on strings.

It is rather straightforward to formulate a type system for simple calculator-like languages which do not involve variable binding. The task becomes more interesting once variables are introduced, for then we must employ parameteric hypothetical judgements to account for their types. The static semantics of such languages takes the form of a parametric inductive definition, for which we must show that the structural rules are admissible (as described in Chapter 3).

9.1 Static Semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$

Recall that the abstract syntax of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is given by Grammar (8.4), which we repeat here for convenience:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{num}$	$\mathtt{num}$
		$\mid$	$\mathtt{str}$	$\mathtt{str}$
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{num}[n]$	n
		$\mid$	$\mathtt{str}[s]$	$\texttt{"}s\texttt{"}$
		$\mid$	$\mathtt{plus}(e_1;e_2)$	$e_1\mathtt{+}e_2$
		$\mid$	$\mathtt{times}(e_1;e_2)$	$e_1\mathtt{*}e_2$
		$\mid$	$\mathtt{cat}(e_1;e_2)$	$e_1\,\hat{}\,e_2$
		$\mid$	$\mathtt{len}(e)$	$\mathtt{\vert}e\mathtt{\vert}$
		$\mid$	$\mathtt{let}(e_1;x.e_2)$	$\mathtt{let}\,x\,\mathtt{be}\,e_1\,\mathtt{in}\,e_2$

This grammar specifies two classes of abt's specified by the judgement forms τ type and e exp, as described in Chapter 8.

The role of a static semantics is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. For example, whether or not the expression $\mathtt{plus}(x;\mathtt{num}[n])$ is sensible depends on whether or not the variable x is declared to have type $\mathtt{num}$ in the surrounding context of the expression. This example is, in fact, illustrative of the general case, in that the *only* information required about the context of an expression is the type of the variables within whose scope the expression lies. Consequently, the static semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ consists of an inductive definition of parametric hypothetical judgements of the form

$$\mathcal{X} \mid \Gamma \vdash e : \tau,$$

where $\mathcal{X}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$ with $x \in \mathcal{X}$. In practice we usually omit explicit mention of the parameters, $\mathcal{X}$, of the judgement since they are determined from the form of Γ.

The rules defining the static semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{9.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{str}[s] : \mathtt{str}} \tag{9.1b}$$

$$\frac{}{\Gamma \vdash \mathtt{num}[n] : \mathtt{num}} \tag{9.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1;e_2) : \texttt{num}} \tag{9.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1;e_2) : \texttt{num}} \tag{9.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1;e_2) : \texttt{str}} \tag{9.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{9.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1;x.e_2) : \tau_2} \tag{9.1h}$$

In Rule (9.1h) we tacitly assume that the variable, x, is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the `let` expression.

It is easy to check that every expression has a unique type, if it has a type at all.

Lemma 9.1 (Unicity of Typing)**.** *For every typing context Γ and expression e, there exists at most one τ such that $\Gamma \vdash e : \tau$.*

Proof. By rule induction on Rules (9.1). □

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 properties for typing, which state that the typing rules are necessary, as well as sufficient, for each form of expression.

Lemma 9.2 (Inversion for Typing)**.** *Suppose that $\Gamma \vdash e : \tau$. If $e = \texttt{plus}(e_1;e_2)$, then $\tau = \texttt{num}$, $\Gamma \vdash e_1 : \texttt{num}$, and $\Gamma \vdash e_2 : \texttt{num}$, and similarly for the other constructs of the language.*

Proof. These may all be proved by induction on the derivation of the typing judgement $\Gamma \vdash e : \tau$. □

9.2 Structural Properties

The static semantics enjoys the structural properties of the hypothetical and parametric judgements. We will focus our attention here on two key properties, the combination of proliferation (Rule (3.4a)) and weakening (Rule (2.16b)), and substitution, which generalizes transitivity (Rule (2.16c)).

Lemma 9.3 (Proliferation and Weakening). *If* $\Gamma \vdash e' : \tau'$, *then* $\Gamma, x : \tau \vdash e' : \tau'$ *for any* $x \mathrel{\#} \Gamma$ *and any* τ type.

Proof. By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We will give one case here, for rule (9.1h). We have that $e' = \mathtt{let}(e_1;z.e_2)$, where by the conventions on parameters we may assume z is chosen such that $z \mathrel{\#} \Gamma$ and $z \mathrel{\#} x$. By induction we have

1. $\Gamma, x : \tau \vdash e_1 : \tau_1$,

2. $\Gamma, x : \tau, z : \tau_1 \vdash e_2 : \tau'$,

from which the result follows by Rule (9.1h). □

Lemma 9.4 (Substitution). *If* $\Gamma, x : \tau \vdash e' : \tau'$ *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof. By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We again consider only rule (9.1h). As in the preceding case, $e' = \mathtt{let}(e_1;z.e_2)$, where z may be chosen so that $z \mathrel{\#} x$, $z \mathrel{\#} \Gamma$, and $z \mathrel{\#} e$. We have by induction

1. $\Gamma \vdash [e/x]e_1 : \tau_1$,

2. $\Gamma, z : \tau_1 \vdash [e/x]e_2 : \tau'$.

Since we have chosen z such that $z \mathrel{\#} e$, we have

$$[e/x]\mathtt{let}(e_1;z.e_2) = \mathtt{let}([e/x]e_1;z.[e/x]e_2).$$

It follows by Rule (9.1h) that $\Gamma \vdash [e/x]\mathtt{let}(e_1;z.e_2) : \tau$, as desired. □

The substitution lemma states that an expression may be composed from separate parts by replacing a variable with any expression of the expected type. The decomposition lemma states the converse, that any expression can be decomposed into parts, with the separation mediated by a variable.

Lemma 9.5 (Decomposition). *If* $\Gamma \vdash [e/x]e' : \tau'$, *then there exists a unique type* τ *such that* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$.

Proof. This follows directly from the unicity of types (Lemma 9.1 on the preceding page), since τ is the unique type for e in the composite expression $[e/x]e'$. □

9.3 Exercises

1. Show that the expression $e = \texttt{plus(num[7];str[}abc\texttt{])}$ is ill-typed in that there is no τ such that $e : \tau$.

Chapter 10

Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. One important method for specifying dynamic semantics is called *structural semantics*, which consists of a collection of rules defining a transition system whose states are expressions with no free variables. *Contextual semantics* may be viewed as an alternative presentation of the structural semantics of a language. Another important method for specifying dynamic semantics, called *evaluation semantics*, is the subject of Chapter 12.

10.1 Structural Semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$

A structural semantics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of a transition system whose states are closed expressions, all of which are initial states. The final states are the *closed values*, as defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{10.1a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{10.1b}$$

The transition judgement, $e \mapsto e'$, is also inductively defined.

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{10.2a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}} \tag{10.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}} \tag{10.2c}$$

$$\frac{s_1\hat{\ }s_2 = s\ \mathsf{str}}{\texttt{cat(str[}s_1\texttt{];str[}s_2\texttt{])} \mapsto \texttt{str[}s\texttt{]}} \tag{10.2d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1'\texttt{;}e_2\texttt{)}} \tag{10.2e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1\texttt{;}e_2'\texttt{)}} \tag{10.2f}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{10.2g}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto \texttt{let(}e_1'\texttt{;}x\texttt{.}e_2\texttt{)}} \tag{10.2h}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (10.2a), (10.2d), and (10.2g) are *instruction transitions*, since they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order in which instructions are executed.

When defined using structural semantics, a derivation sequence has a "two-dimensional" structure, with the number of steps in the sequence being its "width" and the derivation tree for each step being its "depth." For example, consider the following evaluation sequence.

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

Each step in this sequence of transitions is justified by a derivation according to Rules (10.2). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus(num[3];num[3])} \mapsto \texttt{num[6]}}\ (10.2\text{a})}{\texttt{plus(plus(num[3];num[3]);num[4])} \mapsto \texttt{plus(num[6];num[4])}}\ (10.2\text{b})$$

The other steps are similarly justified by a composition of rules.

Since the transition judgement is inductively defined, we may reason about it using rule induction. Specifically, to show that $\mathcal{P}(e \mapsto e')$ holds whenever $e \mapsto e'$, it is sufficient to show that $\mathcal{P}$ 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.

Lemma 10.1. *If $e \mapsto e'$ and $e \mapsto e''$, then e' is e''.*

Proof. By simultaneous induction on the two premises using Rules (10.2). The key observation is that only one rule applies for a given e, from which the result follows easily by induction in each case. □

10.2 Contextual Semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$

A variant of structural semantics, called *contextual semantics*, is sometimes useful. There is no fundamental difference between the two approaches, only a difference in the style of presentation. The main idea is to isolate instruction steps as a special form of judgement, called *instruction transition*, and to formalize the process of locating the next instruction using a device called an *evaluation context*. The judgement, e val, defining whether an expression is a value, remains unchanged.

The instruction transition judgement, $e_1 \leadsto e_2$, for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p \ \mathsf{nat}}{\texttt{plus(num[}m\texttt{];num[}n\texttt{])} \leadsto \texttt{num[}p\texttt{]}} \tag{10.3a}$$

$$\frac{s\,\hat{}\,t = u \ \mathsf{str}}{\texttt{cat(str[}s\texttt{];str[}t\texttt{])} \leadsto \texttt{str[}u\texttt{]}} \tag{10.3b}$$

$$\frac{e_1 \ \mathsf{val}}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \leadsto [e_1/x]e_2} \tag{10.3c}$$

The left-hand side of each instruction is called a *redex* (that which is reduced), and the corresponding right-hand side is called its *contractum* (that to which it is contracted).

The judgement $\mathcal{E}$ ectxt determines the location of the next instruction to execute in a larger expression. The position of the next instruction step is specified by a "hole", written $\circ$, into which the next instruction is placed, as we shall detail shortly. (The rules for multiplication and length are omitted for concision, as they are handled similarly.)

$$\frac{}{\circ \ \mathsf{ectxt}} \tag{10.4a}$$

$$\frac{\mathcal{E}_1 \ \mathsf{ectxt}}{\texttt{plus(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \ \mathsf{ectxt}} \tag{10.4b}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\texttt{plus}(e_1;\mathcal{E}_2)\ \mathsf{ectxt}} \tag{10.4c}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{cat}(\mathcal{E}_1;e_2)\ \mathsf{ectxt}} \tag{10.4d}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\texttt{cat}(e_1;\mathcal{E}_2)\ \mathsf{ectxt}} \tag{10.4e}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{let}(\mathcal{E}_1;x.e_2)\ \mathsf{ectxt}} \tag{10.4f}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here", at the point of the occurrence of the hole. The remaining rules correspond one-for-one to the search rules of the structural semantics. For example, Rule (10.4c) states that in an expression $\texttt{plus}(e_1;e_2)$, if the first principal argument, e_1, is a value, then the next instruction step, if any, lies at or within the second principal argument, e_2.

An evaluation context is to be thought of as a template that is instantiated by replacing the hole with an instruction to be executed. The judgement $e' = \mathcal{E}\{e\}$ states that the expression e' is the result of filling the hole in the evaluation context $\mathcal{E}$ with the expression e. It is inductively defined by the following rules:

$$\frac{}{e = \circ\{e\}} \tag{10.5a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(\mathcal{E}_1;e_2)\{e\}} \tag{10.5b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(e_1;\mathcal{E}_2)\{e\}} \tag{10.5c}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{cat}(e_1;e_2) = \texttt{cat}(\mathcal{E}_1;e_2)\{e\}} \tag{10.5d}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{cat}(e_1;e_2) = \texttt{cat}(e_1;\mathcal{E}_2)\{e\}} \tag{10.5e}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{let}(e_1;x.e_2) = \texttt{let}(\mathcal{E}_1;x.e_2)\{e\}} \tag{10.5f}$$

There is one rule for each form of evaluation context. Filling the hole with e results in e; otherwise we proceed inductively over the structure of the evaluation context.

Finally, the dynamic semantics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined using contextual semantics by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \leadsto e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{10.6}$$

Thus, a transition from e to e' consists of (1) decomposing e into an evaluation context and an instruction, (2) execution of that instruction, and (3) replacing the instruction by the result of its execution in the same spot within e to obtain e'.

The structural and contextual semantics define the same transition relation. For the sake of the proof, let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural semantics (Rules (10.2)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual semantics (Rules (10.6)).

Theorem 10.2. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof. From left to right, proceed by rule induction on Rules (10.2). It is enough in each case to exhibit an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \leadsto e_0'$. For example, for Rule (10.2a), take $\mathcal{E} = \circ$, and observe that $e \leadsto e'$. For Rule (10.2b), we have by induction that there exists an evaluation context $\mathcal{E}_1$ such that $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_0 \leadsto e_0'$. Take $\mathcal{E} = \texttt{plus}(\mathcal{E}_1;e_2)$, and observe that $e = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0\}$ and $e' = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0'\}$ with $e_0 \leadsto e_0'$.

From right to left, observe that if $e \mapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \leadsto e_0'$. We prove by induction on Rules (10.5) that $e \mapsto_{\mathsf{s}} e'$. For example, for Rule (10.5a), e_0 is e, e_0' is e', and $e \leadsto e'$. Hence $e \mapsto_{\mathsf{s}} e'$. For Rule (10.5b), we have that $\mathcal{E} = \texttt{plus}(\mathcal{E}_1;e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathsf{s}} e_1'$. Therefore e is $\texttt{plus}(e_1;e_2)$, e' is $\texttt{plus}(e_1';e_2)$, and therefore by Rule (10.2b), $e \mapsto_{\mathsf{s}} e'$.

□

Since the two transition judgements coincide, contextual semantics may be seen as an alternative way of presenting a structural semantics. It has two advantages over structural semantics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (10.6) in the simpler form

$$\frac{e_0 \leadsto e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}} \ . \tag{10.7}$$

This formulation is simpler insofar as it leaves implicit the definition of the decomposition of the left- and right-hand sides. The deeper advantage, which we will exploit in Chapter 16, is that the transition judgement in contextual semantics applies only to closed expressions of a *fixed* type, whereas structural semantics transitions are necessarily defined over expressions of *every* type.

10.3 Exercises

1. For the structural operational semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 =_\alpha e_2$.

2. Formulate a variation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with both a by-name and a by-value `let` construct.

Chapter 11

Type Safety

Most contemporary programming languages are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ states that it will never arise that a number is to be added to a string, or that two numbers are to be concatenated, neither of which is meaningful.

In general type safety expresses the coherence between the static and the dynamic semantics. The static semantics may be seen as predicting that the value of an expression will have a certain form so that the dynamic semantics of that expression is well-defined. Consequently, evaluation cannot "get stuck" in a state for which no transition is possible, corresponding in implementation terms to the absence of "illegal instruction" errors at execution time. This is proved by showing that each step of transition preserves typability and by showing that typable states are well-defined. Consequently, evaluation can never "go off into the weeds," and hence can never encounter an illegal instruction.

More precisely, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be stated as follows:

Theorem 11.1 (Type Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

The first part, called *preservation*, says that the steps of evaluation preserve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression, e, is *stuck* iff it is not a value, yet there is no e' such that $e \mapsto e'$. It follows from the safety theorem that a stuck state is necessarily ill-typed. Or, putting it the other way around, that well-typed states do not get stuck.

11.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ defined in Chapters 9 and 10 is proved by rule induction on the transition system (rules (10.2)).

Theorem 11.2 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. We will consider two cases, leaving the rest to the reader. Consider rule (10.2b),

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)}\ .$$

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 rule (10.2g),

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2}\ .$$

Assume that $\texttt{let}(e_1;x.e_2) : \tau_2$. By the inversion lemma 9.2 on page 69, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 9.4 on page 70 $[e_1/x]e_2 : \tau_2$, as desired. □

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

The progress theorem captures the idea that well-typed programs cannot "get stuck". The proof depends crucially on the following lemma, which characterizes the values of each type.

Lemma 11.3 (Canonical Forms). *If e val and $e : \tau$, then*

1. *If $\tau = \mathtt{num}$, then $e = \mathtt{num}[n]$ for some number n.*

2. *If $\tau = \mathtt{str}$, then $e = \mathtt{str}[s]$ for some string s.*

Proof. By induction on rules (9.1) and (10.1). □

Progress is proved by rule induction on rules (9.1) defining the static semantics of the language.

Theorem 11.4 (Progress). *If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.*

Proof. The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (9.1d),

$$\frac{e_1 : \mathtt{num} \quad e_2 : \mathtt{num}}{\mathtt{plus}(e_1;e_2) : \mathtt{num}},$$

where the context is empty because we are considering only closed terms.

By induction we have that either e_1 val, or there exists e_1' such that $e_1 \mapsto e_1'$. In the latter case it follows that $\mathtt{plus}(e_1;e_2) \mapsto \mathtt{plus}(e_1';e_2)$, as required. In the former we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \mapsto e_2'$. In the latter case we have that $\mathtt{plus}(e_1;e_2) \mapsto \mathtt{plus}(e_1;e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 11.3 on the preceding page, $e_1 = \mathtt{num}[n_1]$ and $e_2 = \mathtt{num}[n_2]$, and hence

$$\mathtt{plus}(\mathtt{num}[n_1];\mathtt{num}[n_2]) \mapsto \mathtt{num}[n_1 + n_2].$$

□

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 not syntax-directed, that is, when there may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

Summing up, the combination of preservation and progress together constitute the 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 Run-Time Errors

Suppose that we wish to extend $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{div}(e_1;e_2) : \texttt{num}}\,.$$

But the expression $\texttt{div(num[3];num[0])}$ is well-typed, yet stuck! We have two options to correct this situation:

1. Enhance the type system, so that no well-typed program may divide by zero.

2. Add dynamic checks, so that division by zero signals an error as the outcome of evaluation.

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 as ill-formed, because one cannot often predict statically whether an expression will turn out to be non-zero when executed. We therefore consider the second approach, which is typical of current practice.

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 need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. On the other hand the dynamic semantics for quotient *must* check for a zero divisor, because the type system does not rule out the possibility.

One approach to modelling checked errors is to give an inductive definition of the judgment $e\ \mathsf{err}$ stating that the expression e incurs a checked run-time error, such as division by zero. Here are some representative rules that would appear in a full inductive definition of this judgement:

$$\frac{e_1\ \mathsf{val}}{\texttt{div}(e_1;\texttt{num[0]})\ \mathsf{err}} \tag{11.1a}$$

$$\frac{e_1\ \mathsf{err}}{\texttt{plus}(e_1;e_2)\ \mathsf{err}} \tag{11.1b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{err}}{\texttt{plus}(e_1;e_2)\ \mathsf{err}} \tag{11.1c}$$

Rule (11.1a) signals an error condition for division by zero. The other rules propagate this error upwards: if an evaluated sub-expression is a checked error, then so is the overall expression.

The preservation theorem is not affected by the presence of checked errors. However, the statement (and proof) of progress is modified to account for checked errors.

Theorem 11.5 (Progress With Error)**.** *If $e : \tau$, then either e err, or e val, or there exists e' such that $e \mapsto e'$.*

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. □

A disadvantage of this approach to the formalization of error checking is that it appears to require a special set of evaluation rules to check for errors. An alternative is to fold in error checking with evaluation by enriching the language with a special error expression, `error`, which signals that an error has arisen. Since an error condition aborts the computation, the static semantics assigns an arbitrary type to `error`:

$$\frac{}{\texttt{error} : \tau} \tag{11.2}$$

This rule destroys the unicity of typing property (Lemma 9.1 on page 69). This can be restored by introducing a special error expression for each type, but we shall not do so here for the sake of simplicity.

The dynamic semantics is augmented with rules that provoke a checked error (such as division by zero), plus rules that propagate the error through other language constructs.

$$\frac{e_1\ \mathsf{val}}{\texttt{div}(e_1;\texttt{num[0]}) \mapsto \texttt{error}} \tag{11.3a}$$

$$\frac{}{\texttt{plus}(\texttt{error};e_2) \mapsto \texttt{error}} \tag{11.3b}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus}(e_1;\texttt{error}) \mapsto \texttt{error}} \tag{11.3c}$$

There are similar error propagation rules for the other constructs of the language. By defining e err to hold exactly when $e = \texttt{error}$, the revised progress theorem continues to hold for this variant semantics.

11.4 Exercises

1. Complete the proof of preservation.

2. Complete the proof of progress.

Chapter 12

Evaluation Semantics

In Chapter 10 we defined the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using the method of structural semantics. This approach is useful as a foundation for proving properties of a language, but other methods are often more appropriate for other purposes, such as writing user manuals. Another method, 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. Two variants of evaluation semantics are also considered, namely *environment semantics*, which delays substitution, and *cost semantics*, which records the number of steps that are required to evaluate an expression.

12.1 Evaluation Semantics

Another method for defining the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, called *evaluation semantics*, consists of an inductive definition of the evaluation judgement, $e \Downarrow v$, stating that the closed expression, e, evaluates to the value, v.

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow \texttt{num[}n\texttt{]}} \tag{12.1a}$$

$$\frac{}{\texttt{str[}s\texttt{]} \Downarrow \texttt{str[}s\texttt{]}} \tag{12.1b}$$

$$\frac{e_1 \Downarrow \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow \texttt{num[}n_2\texttt{]} \quad n_1 + n_2 = n \ \mathsf{nat}}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \Downarrow \texttt{num[}n\texttt{]}} \tag{12.1c}$$

$$\frac{e_1 \Downarrow \texttt{str[}s_1\texttt{]} \quad e_2 \Downarrow \texttt{str[}s_2\texttt{]} \quad s_1\,\hat{}\,s_2 = s \ \mathsf{str}}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \Downarrow \texttt{str[}s\texttt{]}} \tag{12.1d}$$

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \Downarrow v_2} \tag{12.1e}$$

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.

The evaluation judgement is inductively defined, we prove properties of it by rule induction. Specifically, to show that the property $\mathcal{P}(e \Downarrow v)$ holds, it is enough to show that $\mathcal{P}$ is closed under Rules (12.1):

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

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

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

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

5. Show that $\mathcal{P}(\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \Downarrow v_2)$, if $\mathcal{P}(e_1 \Downarrow v_1)$ and $\mathcal{P}([v_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e $\texttt{exp}$, because the evaluation rules are not syntax-directed!

Lemma 12.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on Rules (12.1). All cases except Rule (12.1e) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the second premise of the evaluation rule. □

12.2 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. 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. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

Theorem 12.2. *For all closed expressions e and values v, $e \mapsto^* v$ iff $e \Downarrow v$.*

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

Lemma 12.3. *If $e \Downarrow v$, then $e \mapsto^* v$.*

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

$$\begin{array}{lll} \texttt{plus(}e_1\texttt{;}e_2\texttt{)} & \mapsto^* & \texttt{plus(num[}n_1\texttt{];}e_2\texttt{)} \\ & \mapsto^* & \texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \\ & \mapsto & \texttt{num[}n_1 + n_2\texttt{]} \end{array}$$

Therefore $\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto^* \texttt{num[}n_1 + n_2\texttt{]}$, as required. The other cases are handled similarly. □

For the converse, recall from Chapter 4 the definitions of multi-step evaluation and complete evaluation. Since $v \Downarrow v$ whenever v val, it suffices to show that evaluation is closed under head expansion.

Lemma 12.4. *If $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

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

12.3 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. 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 $\mathcal{E}$ range over finite sets of such hypotheses, which we call an *environment*. We will consider judgements of the form $\mathcal{E} \vdash e \Downarrow v$, where $\mathcal{E}$ is an environment governing some finite set of variables.

$$\frac{}{\mathcal{E}, x \Downarrow v \vdash x \Downarrow v} \tag{12.2a}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \texttt{num[}n_1\texttt{]} \quad \mathcal{E} \vdash e_2 \Downarrow \texttt{num[}n_2\texttt{]}}{\mathcal{E} \vdash \texttt{plus(}e_1\texttt{;}e_2\texttt{)} \Downarrow \texttt{num[}n_1 + n_2\texttt{]}} \tag{12.2b}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \texttt{str[}s_1\texttt{]} \quad \mathcal{E} \vdash e_2 \Downarrow \texttt{str[}s_2\texttt{]}}{\mathcal{E} \vdash \texttt{cat(}e_1\texttt{;}e_2\texttt{)} \Downarrow \texttt{str[}s_1\texttt{\^{}}s_2\texttt{]}} \tag{12.2c}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow v_1 \quad \mathcal{E}, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\mathcal{E} \vdash \texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \Downarrow v_2} \tag{12.2d}$$

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 $\mathcal{E}$ 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 12.5. *$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$.*

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. □

12.4 Cost Semantics

A structural semantics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation semantics, on the other hand, does not provide such a direct notion of complexity. Since the individual steps required to complete an evaluation are suppressed, we cannot directly read off the number of steps required to evaluate to a value. Instead we must augment the evaluation relation with a cost measure, resulting in a *cost semantics*.

Evaluation judgements have the form $e \Downarrow^k v$, with the meaning that e evaluates to v in k steps.

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow^0 \texttt{num[}n\texttt{]}} \tag{12.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow^{k_2} \texttt{num[}n_2\texttt{]}}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \Downarrow^{k_1+k_2+1} \texttt{num[}n_1+n_2\texttt{]}} \tag{12.3b}$$

$$\frac{}{\texttt{str[}s\texttt{]} \Downarrow^0 \texttt{str[}s\texttt{]}} \tag{12.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \Downarrow^{k_1+k_2+1} \texttt{str[}s_1\texttt{\^{}}s_2\texttt{]}} \tag{12.3d}$$

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad [v_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \Downarrow^{k_1+k_2+1} v_2} \tag{12.3e}$$

Theorem 12.6. *For any closed expression e and closed value v of the same type, $e \Downarrow^k v$ iff $e \mapsto^k v$.*

Proof. From left to right proceed by rule induction on the definition of the cost semantics. From right to left proceed by induction on k, with an inner rule induction on the definition of the transition semantics. □

12.5 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 11.1 on page 79) states that a language is safe iff it satisfies both preservation and progress. This formulation depends critically on the use of a transition system to specify the dynamic semantics. But what if we had instead specified the dynamic semantics as an evaluation relation, instead of using a transition system? Can we state and prove safety in such a setting?

The answer, unfortunately, is that we cannot. While there is an analogue of the preservation property for an evaluation semantics, there is no clear analogue of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. This can be readily proved by induction on the evaluation rules. But what is the analogue of progress? One might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. While this property is true for $\mathcal{L}\{\texttt{num str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num str}\}$ were extended to admit operations that may result in an error (as discussed in Section 11.3 on page 82), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude towards this situation is to simply conclude that type safety cannot be properly discussed in the context of an evaluation semantics, but only by reference to a transition semantics. Another point of view is to instrument the semantics with explicit checks for run-time type errors, and to show that any expression with a type fault must be ill-typed. Re-stated in the contrapositive, this means that a well-typed program cannot incur a type error. A difficulty with this point of view is that one must explicitly account for a class of errors solely to prove that they cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation semantics.

The main idea is to define a judgement $e \Uparrow$ stating, in the jargon of the literature, that the expression *e goes wrong* when executed. The exact definition of "going wrong" is given by a set of rules, but the intention is that it should cover all situations that correspond to type errors. The following rules are representative of the general case:

$$\frac{}{\texttt{plus(str[}s\texttt{];}e_2\texttt{)} \Uparrow} \tag{12.4a}$$

$$\frac{e_1 \ \mathsf{val}}{\texttt{plus(}e_1\texttt{;str[}s\texttt{])} \Uparrow} \tag{12.4b}$$

These rules explicitly check for the misapplication of addition to a string; similar rules govern each of the primitive constructs of the language.

Theorem 12.7. *If $e \Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (12.4). For example, for Rule (12.4a), we observe that $\texttt{str[}s\texttt{]} : \texttt{str}$, and hence $\texttt{plus(str[}s\texttt{];}e_2\texttt{)}$ is ill-typed. □

Corollary 12.8. *If $e : \tau$, then $\neg(e \Uparrow)$.*

Apart from the inconvenience of having to define the judgement $e \Uparrow$ only to show that it is irrelevant for well-typed programs, this approach suffers a very significant methodological weakness. If we should omit one or more rules defining the judgement $e \Uparrow$, the proof of Theorem 12.7 remains valid; there is nothing to ensure that we have included sufficiently many checks for run-time type errors. We can prove that the ones we define cannot arise in a well-typed program, but we cannot prove that we have covered all possible cases. By contrast the transition semantics does not specify any behavior for ill-typed expressions. Consequently, any ill-typed expression will "get stuck" without our explicit intervention, and the progress theorem rules out all such cases. Moreover, the transition system corresponds more closely to implementation—a compiler need not make any provisions for checking for run-time type errors. Instead, it relies on the static semantics to ensure that these cannot arise, and assigns no meaning to any ill-typed program. Execution is therefore more efficient, and the language definition is simpler, an elegant win-win situation for both the semantics and the implementation.

12.6 Exercises

1. Prove that if $e \Downarrow v$, then v val.

2. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

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

4. Prove preservation for the instrumented evaluation semantics, and conclude that well-typed programs cannot go wrong.

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

Chapter 13

Types and Languages

The static and dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ illustrates several fundamental organizing principles of language design on which we shall rely throughout this book. Chief among these is the central role of *types* in programming languages. The informal concept of a language "feature" is formally analyzed as a manifestation of type structure. For example, in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the type `nat` comprises the numeric literals and some arithmetic operations, and the type `str` comprises the string literals and some string operations. These types account for nearly all of the "features" of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, apart from the generic concepts of variable binding and reference, which arise from the structural properties of the parametric hypothetical typing judgement, and are not tied to particular types.

The language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ illustrates a number of important themes that recur throughout the text. In this chapter we summarize the main concepts that will be used throughout the remainder of the text.

13.1 Phase Distinction

The semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ maintains a *phase distinction* between the *static phase* and the *dynamic phase* of processing. The static semantics, or typing rules, impose constraints on the formation of programs that are sufficient to ensure that the dynamic semantics, or evaluation rules, are well-behaved. The static phase occurs prior to, and independently of, the dynamic phase. The static phase may be seen as predicting the form of the value of an expression computed during the dynamic phase. For example, by assigning the type `nat` to the addition of two expressions of the same type, the static semantics is predicting that the result of the sum will be a number. Consequently, it can be used as the argument to multiplication, for example, without fear of error.

The type safety theorem may be seen as stating that the predictions of the static semantics are true of the dynamic semantics, for otherwise the dynamic semantics would "get stuck." A counterexample to safety is a call for revision to either the static semantics—to ensure that the example is barred from consideration—or the dynamic semantics—to ensure that the error condition is checked at run-time. The purpose of proving safety is to ensure the coherence of the static and dynamic semantics.

The phase distinction also manifests itself in the syntax of a language. In most cases the syntax of types does not involve expressions, but the syntax of expressions may well involve types. This is consistent with the idea that the static phase of processing (type checking) usually occurs prior to execution, and hence is independent of it. Languages that do not respect the phase distinction usually do not maintain a clear separation between types and expressions, and consequently intermix some aspects of the dynamic and static phases of processing.

13.2 Introduction and Elimination

The primitive operations associated with a type may generally be classified as either *introduction* or *elimination forms*. The introduction operators determine the values of the type, and the elimnation operators determine the instructions for computing with those values. For example in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, the introduction forms for the type `nat` are the numerals, and those for the type `str` are the string literals. The elimination forms for the type `nat` are addition and multiplication, and those for the type `str` are concatenation and length.

The dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is based on the *inversion principle*, which, roughly speaking, states that the elimination forms are inverse to the introduction forms. This may also be thought of as a kind of *conservation principle* for computations: what can be extracted from a value by an elimination form is limited to what was put into it by the introduction form from which it is built. For example, we may think of the addition operation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ as extracting the underlying number from the numeral, performing the computation, and creating a new numeral to represent the result.

The type safety theorem may be seen as verifying the inversion principle for the language. Returning to the addition example, the type preservation theorem ensures that the values of the arguments of addition must themselves be of type `nat`, and hence by the canonical forms theorem must be numerals. This ensures that addition can make progress, yielding a numeral, which is a value of type `nat`. Had the type safety theorem failed, say by assigning the type `nat` to a string, then the addition function would have to extract the underlying number of a string literal, which it manifestly cannot do. Fortunately (rather, by design) the static semantics ensures that this situation cannot arise. This is the core idea of type safety.

The inversion principle serves as a guide for deriving the dynamic semantics of a language, but in most cases does not fully determine it. For example, suppose that we added the conditional expression $\texttt{ifz}(e;e_1;e_2)$ to $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, thought of as an elimination form for the type `nat`. The expression e of type `nat` is tested, resulting in e_1, if e evaluates to zero, and in e_2, otherwise. The dynamic semantics of this construct must specify that e is to be evaluated to determine how to proceed, but we do not wish to evaluate e_1 or e_2 in advance, only once the decision about e has been made. We say that e is a *major*, or *principal*, argument of the conditional elimination form, and that e_1 and e_2 are *minor*, or *non-principal* arguments. As a rule, the major arguments of an elimination form *must* be evaluated, but the minor arguments need not be. In the case of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ all arguments to the arithmetic and string elimination forms are principal, and hence must be evaluated before evaluation of the operation itself.

Another opportunity for discretion in the definition of the dynamic semantics arises when considering the evaluation rule for introductory forms. Suppose that we replace the numerals in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with two new primitives, `z` and $\texttt{s}(e)$, which represent zero and successor, respectively. These are both introductory forms of type `nat`, but this classification does not determine whether $\texttt{s}(e)$ should be considered a value regardless of the form of e, or only if e is itself a value. If an argument to an introduction form is required to be a value, then there is an associated search rule in the dynamic semantics to evaluate that argument; the operator is said to be *strict*, in that position. If an argument to an introduction form is not required to be a value, then the operator is said to be *non-strict* in that position. When all arguments of all introduction forms are strict, then the language itself is said to be strict, and similarly when all arguments of all introduction forms are non-strict, then the language itself is said to be non-strict.

13.3 Compositionality

The combined structural properties of substitution and transitivity for typing, which we repeat here for reference, captures an essential feature of a type system, called *compositionality*, or *modularity*.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash [e/x]e' : \tau'}$$

This rule captures the essence of *linking*. The expression e', with a free variable x of type τ, represents a client of a separately compiled component, e, which is referenced by the variable, x. The job of the linker is to combine e with e', by substitution for x, to obtain a complete compilation unit, albeit one with further free references to other units to be linked later. The result is *composed* from the shared component and the client, which gives rise to the terminology.

It is important that the client, e', is type checked independently of the implementation of the shared component, e. All that is propagated from the library to its clients is its type, and not the details of its implementation. This means, in particular, that a revised implementation of the library can be linked with the *same* client, without requiring any re-writing or other modification to the client code, so long as the type, τ, remains the same. Modular program development is the process of decomposing a program into parts whose interactions are mediated by a specification, or type, that serves as a contract between the client and the implementor. In other words, *types provide the foundation for modularity*.

13.4 Variables and Values

The typing judgement $\Gamma \vdash e : \tau$ admits two different interpretations, according to whether variables are considered to range over *values*, or over general *computations*, of their type. Which interpretation is appropriate depends on the dynamic semantics of the language. If a variable is only ever bound to (*i.e.*, replaced by) a value at execution time, then it is a *value variable*; otherwise, it is a *computation variable*.

This distinction can be expressed formally by considering carefully the meaning of the typing judgement

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

According to the interpretation given in Chapter 3, this judgement specifies that variables range over computations, and not just values. This is expressed by the rule of substitution,

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash [e/x]e' : \tau'},$$

which was proved admissible in Chapter 9. The substitution principle does not constrain the expression e to be a value, but can be any expression.

To express the restriction of variables to values, we first extend the judgement e val, which states that e is a *closed* value, to admit *open* values, which may involve free variables. The parametric hypothetical judgement

$$x_1 \;\mathsf{val}, \dots, x_n \;\mathsf{val} \vdash e \;\mathsf{val}$$

states that e is an open value, all of whose free variables are restricted to values. It is straightforward to give an inductive definition of this judgement; we need only extend the definition of e val given in Chapter 10 to open values by adding a rule of reflexivity:

$$\frac{}{x_1 \;\mathsf{val}, \dots, x_n \;\mathsf{val} \vdash x_i \;\mathsf{val}}. \tag{13.1}$$

The static semantics is then modified to constrain variables to range only over open values. The typing judgement takes the form

$$x_1 \;\mathsf{val}, \dots, x_n \;\mathsf{val}\; x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau,$$

in which each variable is constrained to be bound to values. Letting Φ range over hypotheses of the form x_1 val, $\dots$, x_n val, the substitution principle may then be stated in the following form:

$$\frac{\Phi\,\Gamma \vdash e : \tau \quad \Phi \vdash e \;\mathsf{val} \quad \Phi, x \;\mathsf{val}\,\Gamma, x : \tau \vdash e' : \tau'}{\Phi\,\Gamma \vdash [e/x]e' : \tau'},$$

To substitute for a variable, we must prove that it is a value.

13.5 Exercises

Part IV

Functions

Chapter 14

Functions

In $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ it is possible to express doubling of any given expression of type `num`, but it is not possible to express the concept of doubling in general. For this we need *functions*, which capture patterns of computation that can be instantiated to obtain specific computations. To pass from particular instances of doubling, of the form e+e for some expression e, to the general case, we replace occurrences of a fixed expression, e, by a variable, x, and then mark that variable as subject to variation using *λ-abstraction*. Thus the general pattern of doubling can be captured by the function

$$\lambda\texttt{(}x\texttt{:num.}\,x\texttt{+}x\texttt{)}.$$

The variable, x, is called the *parameter* of the function, and the expression x+x is its *body*. The parameter is bound by the λ-abstraction, and, consequently, may be renamed freely in accordance with the rules of α-equivalence. We may *apply* this function to any argument, e, of type `num` to obtain an instance of the doubling function for that choice of expression e to be doubled.

To ensure type consistency the type of the parameter of the λ-abstraction is given explicitly, and instances are restricted to arguments of that type. The type of the result is arbitrary, since we are free to use the parameter, x, in any way at all, provided only that it is type-correct. In general, the type of a λ-abstraction has the form $\sigma \rightarrow \tau$, where σ is the *domain type*, the type of its parameter, and τ is the *range type*, the type of its body. Thus, $d : \texttt{num} \rightarrow \texttt{num}$, so that if $e : \texttt{num}$, then $d\texttt{(}e\texttt{)} : \texttt{num}$ as well. Since function types are themselves types, we have *higher-order functions* with types such as

1. $\texttt{num} \rightarrow \texttt{(num} \rightarrow \texttt{num)}$,

2. $(\texttt{num} \rightarrow \texttt{num}) \rightarrow \texttt{num}$,

3. $(\texttt{num} \rightarrow \texttt{num}) \rightarrow (\texttt{num} \rightarrow \texttt{num})$.

These are, respectively,

1. the type of functions that assign a function on the natural numbers to each natural number;

2. the type of functions that assign a natural number to each function on the natural numbers;

3. the type of functions that assign a function on the natural numbers to each function on the natural numbers.

Examples of mathematical functions of each of these types are, respectively,

1. The function that, given a natural number, b, returns the exponential function to the base b, which computes b^n as a function of n.

2. The function that, given a function f, returns the sum of $f\texttt{(0)}$, $f\texttt{(1)}$, $\ldots$, $f\texttt{(10)}$.

3. The function that, given an increasing function f on the natural numbers, returns the function that, on input m, yields the first number n such that $f\texttt{(}n\texttt{)}$ is larger than m.

It is a good exercise to think of further examples of mathematical functions with these types.

A *higher-order language* is any language with higher-order function types. In this chapter we study a rudimentary example of a higher-order language, the enrichment of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with function types, which we call $\mathcal{L}\{\rightarrow\}$. In subsequent chapters we will eliminate the arithmetic and string primitives of the language in favor of more general mechanisms for defining such constructs from first principles, but for now it is useful for the sake of examples to retain them.

14.1 Syntax

The language $\mathcal{L}\{\rightarrow\}$ of pure functions is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$

The language $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\rightarrow\}$ is obtained by incorporating the types and expressions of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ as given by Grammar (8.4) into the above grammar.

As we mentioned in the introduction to this chapter, the expression $\lambda(x{:}\tau.e)$ is called a *λ-abstraction*. The variable x is the *parameter* of the abstraction, and e is its *body*. It represents the function mapping $e_0 : \tau$ to (the value of) $[e_0/x]e$. The expression $e_1(e_2)$ is called an *application*, with *function* e_1 and *argument* e_2. If e_1 evaluates to a λ-abstraction $\lambda(x{:}\tau.e)$, then the application $e_1(e_2)$ evaluates to the value of $[e_2/x]e_1$, the instance of the body obtained by replacing the parameter by the argument.

In the presence of functions we may treat `let` as a derived form by defining $\mathtt{let}[\tau](e_1;x.e_2)$ to stand for the expression

$$\mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1). \tag{14.1}$$

The dynamic semantics of this form of `let` is inherited from that given to function applications, as described in Section 14.3 on the following page.

14.2 Static Semantics

The static semantics of $\mathcal{L}\{\rightarrow\}$ is defined by a parametric inductive definition of judgements of the form $\Gamma \vdash e : \tau$, where Γ is a finite set of assumptions of the form $x : \tau$, where each x is a variable.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{14.2a}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{14.2b}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{14.2c}$$

Lemma 14.1 (Inversion). *Suppose that* $\Gamma \vdash e : \tau$.

1. *If* $e = x$, *then* $\Gamma = \Gamma', x : \tau$.
2. *If* $e = \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e\texttt{)}$, *then* $\tau = \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$.
3. *If* $e = \texttt{ap(}e_1\texttt{;}e_2\texttt{)}$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)}$ *and* $\Gamma \vdash e_2 : \tau_2$.

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 structural property of substitution holds for the typing judgement defined by the above rules.

Lemma 14.2 (Substitution). *If* $\Gamma, x : \tau \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof. By rule induction on the derivation of the first judgement. □

14.3 Dynamic Semantics

The dynamic semantics of $\mathcal{L}\{\rightarrow\}$ is given by a transition semantics on closed expressions. The judgement e val, where e is a closed expression, is inductively defined.

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{14.3a}$$

Observe that no restriction is placed on the form of e, the body of the function.

There are two forms of dynamic semantics for functions, *call-by-value* and *call-by-name*. Under call-by-value, the argument is evaluated before the function is called with the resulting value as argument; under call-by-name, the function is called with the argument in unevaluated form, deferring evaluation until it is actually needed.

The call-by-value dynamic semantics is defined by the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}} \tag{14.4a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1\texttt{;}e_2'\texttt{)}} \tag{14.4b}$$

$$\frac{e_2\ \mathsf{val}}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e_1} \tag{14.4c}$$

The call-by-name semantics is, instead, defined by the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}} \tag{14.5a}$$

$$\frac{}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e_1} \tag{14.5b}$$

In contrast to Rule (14.4c) there is no requirement on Rule (14.5b) that the argument be a value.

14.4 Safety

Theorem 14.3 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. The proof is by induction on rules (14.4), which define the dynamic semantics of the language.

Consider rule (14.4c),

$$\frac{e_2\ \mathsf{val}}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e_1}\ .$$

Suppose that $\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} : \tau_1$. By Lemma 14.1 on the preceding page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 14.2 on the facing page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 14.4 (Canonical Forms)**.** *If e $\mathsf{val}$and $e : \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$, then $e = \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)}$ for some x and e_2 such that $x : \tau_1 \vdash e_2 : \tau_2$.*

Proof. By induction on the typing rules, using the assumption e val. □

Theorem 14.5 (Progress)**.** *If $e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.*

Proof. The proof is by induction on rules (14.2). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (14.2c). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}$. Otherwise we have by induction either e_2 val or $e_2 \mapsto e_2'$. In the latter case we have $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1\texttt{;}e_2'\texttt{)}$ (bearing in mind e_1 val). Otherwise, by Lemma 14.4, we have $e_1 = \texttt{lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{)}$ for some x and e. But then $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto [e_2/x]e$, again bearing in mind that e_2 val.

□

14.5 Evaluation Semantics

An inductive definition of the evaluation judgement $e \Downarrow v$ for $\mathcal{L}\{\rightarrow\}$ is given by the following rules:

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}} \tag{14.6a}$$

$$\frac{e_1 \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \tag{14.6b}$$

It is easy to check that if $e \Downarrow v$, then v val, and that if e val, then $e \Downarrow e$.

Theorem 14.6. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (14.6). The proof makes use of a *pasting lemma* stating that, for example, if $e_1 \mapsto^* e_1'$, then $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto^* \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}$, and similarly for the other constructs of the language.

In the reverse direction we proceed by rule induction on Rules (4.1). The proof relies on a *head expansion lemma*, which states that if $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. The head expansion lemma is proved by rule induction on Rules (14.4). □

14.6 Dynamic Binding

The environment semantics of Chapter 12 uses hypothetical judgements of the form

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

to state 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 $\mathcal{L}\{\rightarrow\}$ using the following rules for functions and applications:

$$\frac{}{\mathcal{E} \vdash \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}} \tag{14.7a}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \quad \mathcal{E} \vdash e_2 \Downarrow v_2 \quad \mathcal{E}, x \Downarrow v_2 \vdash e \Downarrow v}{\mathcal{E} \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \tag{14.7b}$$

When applying a function to an argument, the parameter of the function is bound to the argument value for the duration of the evaluation of the body. It is implicit in Rule (14.7b) that the variable x lie apart from $\mathcal{E}$; this condition may always be met by choosing a suitable representative of the α-equivalence class of the function.

This extension of the environment semantics given in Chapter 10 seems to make sense, but, surprisingly, it is incorrect. Specifically, the environment semantics does not agree with the substitution semantics given by Rules (14.6), in contrast to the case of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. The culprit is the interaction between higher-order functions and environments. This is the *locus classicus* of a language design error, which therefore merits careful consideration to avoid repetition of a classical mistake in modern times.

To see what is wrong, consider the expression

$$e = \texttt{ap(lam[num]}(x\texttt{.lam[num]}(y\texttt{.}x)\texttt{);num[3])},$$

which, when written in concrete syntax, is the application

$$\lambda(x\texttt{:num}.\,\lambda(y\texttt{:num}.\,x))(3).$$

According to the substitution semantics for functions (Rules (14.6)), this expression evaluates to

$$\texttt{lam[num]}(y\texttt{.num[3]}),$$

as the reader may readily check.

Let us now evaluate e using Rules (14.7). To show that $e \Downarrow v$, it is enough to apply Rule (14.7b), and show that the judgement

$$x \Downarrow \texttt{num[3]} \vdash \texttt{lam[num]}(y\texttt{.}x) \Downarrow v$$

is derivable. Evidently, v must be $\texttt{lam[num]}(y\texttt{.}x)$, since a λ-abstraction is a value, and consequently we obtain $e \Downarrow \texttt{lam[num]}(y\texttt{.}x)$ categorically, which is to say *under no hypotheses*.

But something must be wrong, because the value v involves the free variable, x, for which we have no binding. Indeed, we can use e to build an expression that has no value in the environment semantics, but which has a definite value in the substitution semantics. Let e' be the application $\texttt{ap}(e\texttt{;num[4]})$. To show that $e' \Downarrow v'$ for some v', it is necessary and sufficient to derive the judgement

$$y \Downarrow \texttt{num[4]} \vdash x \Downarrow v'.$$

But there is no such value, because there is no hypothesis governing the variable x—the variable x has *escaped its scope*.

The source of the difficulty is the stack-like behavior of hypotheses in the environment semantics. To evaluate a function application, the body is evaluated under an additional hypothesis binding a fresh copy of the parameter to the argument value. The value of the application is the value of the body, which is returned to the original, unextended context of hypotheses. But, as the foregoing example shows, if the value of the body is a function, it may contain free occurrences of the parameter, which escape their scope when returned. Higher-order languages are therefore said to violate the *stack discipline* for binding function parameters, in contrast to first-order languages (those that lack higher-order functions), which adhere to it.

What to do? The obvious solution is to ensure that function parameters do not escape their scope. This may be achieved by substituting the argument for the parameter in the value of the function body at the point where it is returned to the surrounding context.

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \quad \mathcal{E} \vdash e_2 \Downarrow v_2 \quad \mathcal{E}, x \Downarrow v_2 \vdash e \Downarrow v}{\mathcal{E} \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow [v_2/x]v} \; . \tag{14.8}$$

The sole difference compared to Rule (14.7b) is that the returned value is $[v_2/x]v$, rather than v, so that x is replaced by its binding before v is returned to the surrounding context. A disadvantage of this solution is that it makes use of substitution, undermining the motivation for the use of environments. We shall return to this point in Section 14.7 on the facing page below.

Surprisingly, the classic "solution" to this problem is to re-characterize the bug as a feature, called *dynamic binding*. According to this view, variables are permitted to escape their scope without restriction. Whenever a binding for a variable, x, is required, we simply use whatever binding for x happens to be available in the current environment, regardless of the scoping rules given in Chapter 6. If no binding is available, the computation is aborted with an "unbound variable" error. For this approach to make any sense at all, we must rescind the policy of identifying expressions up to α-equivalence because dynamic binding is sensitive to the choice of parameter name. This does violence to the very concept of variable binding, a strong argument against it, but its advocates regard this as a good thing.

The strongest argument against dynamic binding is that it is *not type safe*. Consider the expression

$$\texttt{let}\, f\, \texttt{be}\, \lambda\texttt{(}x\texttt{:num.}\,\lambda\texttt{(}y\texttt{:num.}\,x\texttt{+}y\texttt{))(3)}\;\texttt{in}\,\lambda\texttt{(}x\texttt{:str.}\,f\texttt{(5))("}abc\texttt{")}$$

The variable f is bound to $\lambda\texttt{(}y\texttt{:num.}\,x\texttt{+}y\texttt{)}$, which was obtained with x bound to 3, but whose binding is now lost. The application $f\texttt{(5)}$ is evaluated in the presence of this binding for f, and with x bound to the string "abc". Evaluation of this application results in the evaluation of x+y with y bound to 5, but x bound to "abc", which is a run-time type error!

If we change the example to

$$\texttt{let}\,f\,\texttt{be}\,\lambda\texttt{(}x\texttt{:num.}\,\lambda\texttt{(}y\texttt{:num.}\,x\texttt{+}y\texttt{))(3)}\,\texttt{in}\,\lambda\texttt{(}x\texttt{:num.}\,f\texttt{(5))(7)}$$

then no run-time type error arises, but evaluation results in 12, rather than 8. If we further change the example by innocently renaming the bound variable in the body of the `let` expression to obtain

$$\texttt{let}\,f\,\texttt{be}\,\lambda\texttt{(}x\texttt{:num.}\,\lambda\texttt{(}y\texttt{:num.}\,x\texttt{+}y\texttt{))(3)}\,\texttt{in}\,\lambda\texttt{(}y\texttt{:num.}\,f\texttt{(5))(7)}$$

evaluation aborts with an unbound variable. The behavior is highly sensitive to the names of bound variables, and is not stable under changes to the types of bound variables, even if all uses within the scope of the binding are type correct.

How, then, can anyone advocate for dynamic binding? First, languages with dynamic binding have only one type, so that type mismatches cannot arise, albeit at the expense of incurring run-time errors such as attempting to add a string to a number. (See Chapter 23 for more information about such languages.) Second, the concept of dynamic binding may be re-interpreted so as to avoid disrupting the basic principles of binding and scope. Rather than think of *variables* as being dynamically bound, we may instead introduce a type of *symbols* that serve as keys for a dictionary data structure that maintains their bindings. (See Chapter 36 for more on this approach.)

14.7 Closures

The standard method for preventing a variable in a higher-order language from escaping its scope is *explicit substitution*. Rather than perform substitution as called for in Rule (14.8), we instead regard $[v/x]\texttt{lam[}\tau\texttt{](}y\texttt{.}e\texttt{)}$ as a form of expression representing a delayed substitution. Rather than perform the indicated substitution, we instead record the intention to do so. Whenever such an expression is applied, we reinstate the substitution as a hypothesis during the evaluation of the body of the λ-abstraction. If the value of the variable x is ever required, it is determined by this assumption.

Since λ-abstractions may be nested arbitrarily deeply, the general form of function value is an iterated delayed substitution of the form

$$[v_1/x_1]\ldots[v_k/x_k]\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)},$$

where the free variables of e are among x and $x_1, \ldots, x_k$. Collapsing the iterated substitution into a single simultaneous substitution, we obtain the standard form

$$[v_1,\ldots,v_k/x_1,\ldots,x_k]\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}.$$

This form of expression is called a *closure*, since it closes the free variables of the λ-abstraction by providing explicit bindings for them. The abstract binding tree representation of a closure has the form $\texttt{clo[}\tau\texttt{](}E;x\texttt{.}e\texttt{)}$, where E is a finite function mapping x_i to v_i for each $1 \leq i \leq k$, called the *environment* of the closure.

A correct environment semantics for $\mathcal{L}\{\rightarrow\}$ may be given in terms of closures. We consider judgements of the form $\mathcal{E} \vdash e \Downarrow v$, where $\mathcal{E}$ is $x_1 \Downarrow v_1,\ldots,x_k \Downarrow v_k$ and the free variables of e are among the $x_1,\ldots,x_k$. The value v, and the values v_i, are all closed.

$$\frac{}{\mathcal{E}, x \Downarrow v \vdash x \Downarrow v} \tag{14.9a}$$

$$\frac{E = \{\, x_1 \mapsto v_1 \ldots x_k \mapsto v_k \,\}}{x_1 \Downarrow v_1,\ldots,x_k \Downarrow v_k \vdash \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \Downarrow \texttt{clo[}\tau\texttt{](}E;x\texttt{.}e\texttt{)}} \tag{14.9b}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow \texttt{clo[}\tau\texttt{](}E;x\texttt{.}e\texttt{)} \qquad \mathcal{E} \vdash e_2 \Downarrow v \\ E = \{\, x_1 \mapsto v_1 \ldots x_k \mapsto v_k \,\} \\ x_1 \Downarrow v_1,\ldots,x_k \Downarrow v_k, x \Downarrow v \vdash e \Downarrow w\end{array}}{\mathcal{E} \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow w} \tag{14.9c}$$

In Rule (14.9b) the entire collection of evaluation hypotheses is used to form the closure that serves as the value of the λ-abstraction. In Rule (14.9c) the environment of the closure, augmented with a binding of the parameter to the argument, is installed as the hypothesis set with which to evaluate the body of the abstraction. Observe that by storing bindings for variables in a closure we are explicitly violating the stack discipline, as must be the case for a higher-order language.

To relate the environment semantics to the substitution semantics requires two auxiliary functions, $\widehat{\mathcal{E}}(e)$ and $\widehat{v}$, which, respectively, substitute expanded values for variables in an expression, and expand values for substitution into expressions. These are inductively defined as follows:

$$\frac{\begin{array}{c}\mathcal{E} = x_1 \Downarrow v_1,\ldots,x_k \Downarrow v_k \\ \widehat{v_1}{=}v_1' \qquad \ldots \qquad \widehat{v_k}{=}v_k'\end{array}}{\widehat{\mathcal{E}}(e){=}[v_1',\ldots,v_k'/x_1,\ldots,x_k]e} \tag{14.10a}$$

$$\frac{}{\widehat{\texttt{num[}n\texttt{]}}=\texttt{num[}n\texttt{]}} \tag{14.10b}$$

$$\frac{}{\widehat{\texttt{str[}s\texttt{]}}=\texttt{str[}s\texttt{]}} \tag{14.10c}$$

$$\frac{\begin{array}{c} E = \{\, x_1 \mapsto v_1 \dots x_k \mapsto v_k \,\} \\ \widehat{v_1}=v_1' \qquad \dots \qquad \widehat{v_k}=v_k' \\ [v_1',\dots,v_k'/x_1,\dots,x_k]e = e' \end{array}}{\widehat{\texttt{clo[}\tau\texttt{](}E;}x\texttt{.}e\texttt{)}=\texttt{lam[}\tau\texttt{](}x\texttt{.}e'\texttt{)}} \tag{14.10d}$$

To avoid confusion, we temporarily write e° and v° for expressions and values in $\mathcal{L}\{\rightarrow\}$ (without closures), and $e^\circ \Downarrow^\circ v^\circ$ for the evaluation relation on this language defined by Rules (14.6).

Theorem 14.7. *$\mathcal{E} \vdash e \Downarrow v$ iff $\widehat{\mathcal{E}}(e) \Downarrow^\circ \widehat{v}$.*

Proof. In the forward direction we proceed by induction on Rules (14.9), making use of the definition of expansion given by Rules (14.10). In the case of Rule (14.9a), the result follows immediately, since $\widehat{\mathcal{E}}(x)=\widehat{v}$. In the case of Rule (14.9c), we have by induction

1. $\widehat{\mathcal{E}}(e_1) \Downarrow^\circ \widehat{\texttt{clo[}\tau\texttt{](}E;}x\texttt{.}e\texttt{)}$;

2. $\widehat{\mathcal{E}}(e_2) \Downarrow^\circ \widehat{v}$; and

3. $\widehat{\mathcal{E}'}(e) \Downarrow^\circ \widehat{w}$, where $\mathcal{E}' = x_1 \Downarrow v_1, \dots, x_k \Downarrow v_k, x \Downarrow v$.

The result follows from the observation that

$$\widehat{\texttt{clo[}\tau\texttt{](}E;}x\texttt{.}e\texttt{)}=\texttt{lam[}\tau\texttt{](}x\texttt{.}[\widehat{v_1},\dots,\widehat{v_k}/x_1,\dots,x_k]e\texttt{)},$$

and an application of Rule (14.6b), making use of Rule (14.10a).

In the reverse direction we show by induction on Rules (14.6) that if $\widehat{\mathcal{E}}(e) \Downarrow^\circ v^\circ$, then $\mathcal{E} \vdash e \Downarrow v$ for some v such that $\widehat{v} = v^\circ$. Consider Rule (14.6b). We have

1. $\widehat{\mathcal{E}}(e) = \texttt{ap(}e_1^\circ\texttt{;}e_2^\circ\texttt{)}$, so $e = \texttt{ap(}e_1\texttt{;}e_2\texttt{)}$ and $\widehat{\mathcal{E}}(e_1) = e_1^\circ$ and $\widehat{\mathcal{E}}(e_2) = e_2^\circ$;

2. $e_1^\circ \Downarrow^\circ v_1^\circ$ with $v_1^\circ = \texttt{lam[}\tau\texttt{](}x\texttt{.}e^\circ\texttt{)}$;

3. $e_2^\circ \Downarrow^\circ v_2^\circ$; and

4. $[v_2^\circ/x]e^\circ \Downarrow^\circ v^\circ$.

Consequently, we have by induction

1. $\mathcal{E} \vdash e_1 \Downarrow v_1$ with $\widehat{v_1} = v_1^\circ$, so $v_1 = \texttt{clo[}\tau\texttt{](}E';x\texttt{.}e\texttt{)}$ for some E' and e such that $\mathcal{E}'$ corresponds to E' and $\widehat{\mathcal{E}}'(e) = e^\circ$;

2. $\mathcal{E} \vdash e_2 \Downarrow v_2$ with $\widehat{v_2} = v_2^\circ$.

3. $\mathcal{E}', x \Downarrow v_2 \vdash e \Downarrow v$ with $\widehat{v} = v^\circ$.

It follows from Rule (14.9c) that $\mathcal{E} \vdash e \Downarrow v$ with $\widehat{v} = v^\circ$, as required. □

14.8 Exercises

1. Formulate dynamic binding within a statically scoped language.

Chapter 15

Gödel's System T

The language $\mathcal{L}\{\texttt{nat} \rightarrow\}$, better known as *Gödel's System **T***, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\texttt{nat} \rightarrow\}$ provides a general mechanism, called *primitive recursion*, for defining functions on the natural numbers. Primitive recursion captures the essential inductive character of the natural numbers, from which we may define a wide range of functions, including elementary arithmetic.

A chief characteristic of $\mathcal{L}\{\texttt{nat} \rightarrow\}$ is that it permits the definition only of *total* functions, *i.e.*, those that assign a value in the range type to every element of the domain type. This means that programs written in $\mathcal{L}\{\texttt{nat} \rightarrow\}$ may be considered to "come equipped" with their own termination proof, in the form of typing annotations to ensure that it is well-typed. But only certain forms of proof are codifiable in this manner, with the inevitable result that some well-defined total functions on the natural numbers cannot be programmed in $\mathcal{L}\{\texttt{nat} \rightarrow\}$.

15.1 Static Semantics

The syntax of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$
		$\mid$	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{z}$	$\mathtt{z}$
		$\mid$	$\mathtt{s}(e)$	$\mathtt{s}(e)$
		$\mid$	$\mathtt{rec}[\tau](e;e_0;x.y.e_1)$	$\mathtt{rec}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y\Rightarrow e_1\}$
		$\mid$	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression

$$\mathtt{rec}[\tau](e;e_0;x.y.e_1)$$

is called *primitive recursion*. It represents the e-fold iteration of the transformation $x.y.e_1$ starting from e_0. The bound variable x represents the predecessor and the bound variable y represents the result of the x-fold iteration. The "with" clause in the concrete syntax for the recursor binds the variable y to the result of the recursive call, as will become apparent shortly.

Sometimes *iteration*, written $\mathtt{iter}[\tau](e;e_0;y.e_1)$, is considered as an alternative to primitive recursion. It has essentially the same meaning as primitive recursion, except that only the result of the recursive call is bound to y in e_1, and no binding is made for the predecessor. Clearly iteration is a special case of primitive recursion, since we can always ignore the predecessor binding. Conversely, primitive recursion is definable from iteration, provided that we have product types (Chapter 17) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The static semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}} \tag{15.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{15.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{15.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\tau](e;e_0;x.y.e_1) : \tau} \tag{15.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \texttt{lam[}\sigma\texttt{](}x\texttt{.}e\texttt{)} : \texttt{arr(}\sigma\texttt{;}\tau\texttt{)}} \tag{15.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} : \tau} \tag{15.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

Lemma 15.1. *If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$, then $\Gamma \vdash [e/x]e' : \tau'$.*

15.2 Dynamic Semantics

We will adopt a *lazy* semantics for the successor operation, and a *call-by-name* semantics for function applications. Variables range over computations, which are not necessarily values. These choices are not forced on us, but are natural and convenient in a language in which (as we shall see) every closed expression has a value.

The closed values of $\mathcal{L}\{\texttt{nat} \rightarrow\}$ are determined by the following rules:

$$\frac{}{\texttt{z}\ \mathsf{val}} \tag{15.2a}$$

$$\frac{}{\texttt{s(}e\texttt{)}\ \mathsf{val}} \tag{15.2b}$$

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{15.2c}$$

The dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightarrow\}$ is given by the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}} \tag{15.3a}$$

$$\frac{}{\texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e} \tag{15.3b}$$

$$\frac{e \mapsto e'}{\texttt{rec[}\tau\texttt{](}e\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \mapsto \texttt{rec[}\tau\texttt{](}e'\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)}} \tag{15.3c}$$

$$\frac{}{\texttt{rec[}\tau\texttt{](z;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \mapsto e_0} \tag{15.3d}$$

$$\frac{}{\texttt{rec[}\tau\texttt{](s(}e\texttt{);}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \mapsto [e, \texttt{rec[}\tau\texttt{](}e\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)}/x, y]e_1} \tag{15.3e}$$

Rules (15.3d) and (15.3e) specify the behavior of the recursor on $\texttt{z}$ and $\texttt{s(}e\texttt{)}$. In the former case the recursor evaluates e_0, and in the latter case the variable x is bound to e and the variable y is bound to a recursive call on e before evaluating e_1. Because of the lazy semantics of variable binding, if e_1 does not need y to determine its value, the recursive call is never executed.

Lemma 15.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{nat}$, then either $e = \mathtt{z}$ or $e = \mathtt{s}(e')$ for some e'.*

2. *If $\tau = \tau_1 \rightarrow \tau_2$, then $e = \lambda\,(x\!:\!\tau_1.\,e_2)$ for some e_2.*

Theorem 15.3 (Safety)**.** 1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

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

15.3 Equivalence and Termination of Expressions

One beauty of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is that it admits a very natural notion of equality of expressions that supports patterns of reasoning familiar from conventional mathematics. We shall have much more to say about this in Chapter 52, but for the time being we will make informal use of *observational equivalence*, $e_1 \cong e_2 : \tau\ [\Gamma]$, where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. This judgement states that two open expressions, e_1 and e_2, of the same type are *indistinguishable* wherever they are used in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$, and therefore may be freely interchanged with one another in any context.

Observational equivalence enjoys the following convenient properties:

1. It is *consistent* in that it does not equate zero, z, with any non-zero number, $\mathtt{s}(-)$.

2. It is a *congruence*, so that we may replace equals by equals and get equals.

3. It contains *symbolic execution*, meaning that all of the rules of the dynamic semantics are valid equivalences, even if the expressions involved have free variables.

We will rely on these properties in our reasoning below. They will be justified formally in Chapter 52.

Another important property of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is that it is impossible to define an infinite loop in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$.

Theorem 15.4. *If $e : \tau$, then there exists v val such that $e \mapsto^* v$.*

Proof. See Corollary 52.8 on page 412 in Chapter 52. □

Consequently, values of function type in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ behave like mathematical functions in that every element of the domain type of the function is mapped to a unique element of the range type.

15.4 Definability

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ is *definable* in $\mathcal{L}\{\mathtt{nat} \to\}$ iff there exists an expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{n}) \cong \overline{f(n)} : \mathtt{nat}. \tag{15.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is a expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ that accurately mimics the behavior of f on all possible inputs.

For example, the successor function is obviously definable in $\mathcal{L}\{\mathtt{nat} \to\}$ by the expression $\mathtt{succ} = \lambda(x{:}\mathtt{nat}.\,\mathtt{s}(x))$. The doubling function, $d(n) = 2 \times n$, is definable by the expression

$$e_d = \lambda(x{:}\mathtt{nat}.\,\mathtt{rec}\,x\,\{\mathtt{z} \Rightarrow \mathtt{z} \mid \mathtt{s}(u)\,\mathtt{with}\,v \Rightarrow \mathtt{s}(\mathtt{s}(v))\}).$$

To see this, observe that $e_d(\overline{0}) \cong \overline{0} : \mathtt{nat}$, and, assuming that $e_d(\overline{n}) \cong \overline{d(n)} : \mathtt{nat}$, check that

$$\begin{aligned} e_d(\overline{n+1}) &\cong \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) && (15.5)\\ &\cong \mathtt{s}(\mathtt{s}(\overline{2 \times n})) && (15.6)\\ &= \overline{2 \times (n+1)} && (15.7)\\ &= \overline{d(n+1)} && (15.8) \end{aligned}$$

As another example, consider the following function, called *Ackermann's function*, defined by the following equations:

$$\begin{aligned} A(0,n) &= n+1 && (15.9)\\ A(m+1,0) &= A(m,1) && (15.10)\\ A(m+1,n+1) &= A(m, A(m+1,n)) && (15.11) \end{aligned}$$

This function grows very quickly! For example, $A(4,2) \approx 2^{65,536}$, which is often cited as being much larger than the number of atoms in the universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of argument (m,n). On each recursive call, either m decreases, or else m remains the same, and n decreases, so inductively the recursive calls are well-defined, and hence so is $A(m,n)$.

A *first-order primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using primitive recursion, but without using any higher order functions. Ackermann's function is defined so as to grow more quickly than any first-order primitive recursive function, but if we permit ourselves to use higher-order functions, then we may give a definition of it in $\mathcal{L}\{\texttt{nat}\rightarrow\}$. The key is to observe that $A(m+1,n)$ iterates the function $A(m,-)$ for n times, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

$$\texttt{it} : (\texttt{nat} \rightarrow \texttt{nat}) \rightarrow \texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{nat}$$

to be the λ-abstraction

$$\lambda\texttt{(}f\texttt{:nat} \rightarrow \texttt{nat.}\,\lambda\texttt{(}n\texttt{:nat.}\,\texttt{rec}\,n\,\{\texttt{z}\Rightarrow\texttt{id} \mid \texttt{s(\_)}\,\texttt{with}\,g\Rightarrow f\circ g\}\texttt{))},$$

where $\texttt{id} = \lambda\texttt{(}x\texttt{:nat.}\,x\texttt{)}$ is the identity, and $f \circ g = \lambda\texttt{(}x\texttt{:nat.}\,f\texttt{(}g\texttt{(}x\texttt{)))}$ is the composition of f and g. It is easy to check that

$$\texttt{it(}f\texttt{)(}\overline{n}\texttt{)(}\overline{m}\texttt{)} \cong f^{(n)}(\overline{m}) : \texttt{nat},$$

where the latter expression is the n-fold composition of f starting with $\overline{m}$.

With this in hand we may define the Ackermann function

$$a : \texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{nat}$$

to be the λ-abstraction

$$\lambda\texttt{(}m\texttt{:nat.}\,\texttt{rec}\,m\,\{\texttt{z}\Rightarrow\texttt{succ} \mid \texttt{s(\_)}\,\texttt{with}\,f\Rightarrow\lambda\texttt{(}n\texttt{:nat.}\,\texttt{it(}f\texttt{)(}n\texttt{)(}f\texttt{(}\overline{1}\texttt{)))}\}\texttt{)}.$$

It is instructive to check that the following equivalences, which show that the Ackermann function is definable, are valid:

$$a\texttt{(}\overline{0}\texttt{)(}\overline{n}\texttt{)} \cong \texttt{s(}\overline{n}\texttt{)} \tag{15.12}$$

$$a\texttt{(}\overline{m+1}\texttt{)(}\overline{0}\texttt{)} \cong a\texttt{(}\overline{m}\texttt{)(}\overline{1}\texttt{)} \tag{15.13}$$

$$a\texttt{(}\overline{m+1}\texttt{)(}\overline{n+1}\texttt{)} \cong a\texttt{(}\overline{m}\texttt{)(}a\texttt{(s(}\overline{m}\texttt{))(}\overline{n}\texttt{))}. \tag{15.14}$$

15.5 Non-Definability

It follows directly from Theorem 15.4 on page 116 that all functions in $\mathcal{L}\{\texttt{nat}\rightarrow\}$ are total: if $f : \sigma \rightarrow \tau$ and $e : \sigma$, then $f\texttt{(}e\texttt{)}$ evaluates to a value of type τ. Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in the $\mathcal{L}\{\texttt{nat}\rightarrow\}$.

The proof of this result makes use of a technique called *Gödel-numbering*, which establishes a one-to-one representation of closed expressions of $\mathcal{L}\{\texttt{nat}\rightarrow\}$

as natural numbers. The importance of this technique is that it permits us to manipulate expressions of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ within $\mathcal{L}\{\texttt{nat}\rightarrow\}$ itself. This is not *a priori* obvious, but is an important observation of Gödel's that plays a central role in the development of his famous *incompleteness theorems* for mathematical logic. Indeed, the non-definability of certain functions on the natural numbers within $\mathcal{L}\{\texttt{nat}\rightarrow\}$ may be seen as a form of incompleteness similar to that considered by Gödel.

The essence of Gödel-numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is not difficult, the main complication being to ensure that α-equivalent expressions are assigned the same Gödel number.) Recall that a general ast, a, has the form $o(a_1,\ldots,a_k)$, where o is an operator of arity k. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let m be the index of o in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of a to be the number

$$2^m\, 3^{n_1}\, 5^{n_2}\, \ldots\, p_k^{n_k},$$

where p_k is the kth prime number (so that $p_0 = 2$, $p_1 = 3$, *etc.*.), and $n_1,\ldots,n_k$ are the Gödel numbers of $a_1,\ldots,a_k$, respectively. This obviously assigns a natural number to each ast. Conversely, given a natural number, n, we may apply the prime factorization theorem to "parse" n as a unique abstract syntax tree. (If the factorization is not of the appropriate form, which can only be because the arity of the operator does not match the number of factors, then n does not code any ast.)

Now, using this representation, we may define (mathematically!) a function $E : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ such that, for any $e : \texttt{nat} \rightarrow \texttt{nat}$, $E(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \cong \overline{n} : \texttt{nat}$. By the consistency of observational equivalence, this is equivalent to $e(\overline{m}) \mapsto^* \overline{n}$. The determinacy of the dynamic semantics together with Theorem 15.4 on page 116 ensure that E is a well-defined function. Using this we may define another mathematical function, $F : \mathbb{N} \rightarrow \mathbb{N}$, by the equation $F(m) = E(m)(m)$, so that $F(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \cong \overline{n} : \texttt{nat}$. We will show that the function F is not definable in $\mathcal{L}\{\texttt{nat}\rightarrow\}$.

Suppose for a contradiction that F were defined by the expression e_F, which means that $e_F(\overline{\ulcorner e \urcorner}) \cong e(\overline{\ulcorner e \urcorner}) : \texttt{nat}$. Let e_D be the expression

$$\lambda(x\texttt{:nat.}\,\texttt{s}(e_F(x))).$$

We then have

$$\begin{aligned} e_D(\overline{\ulcorner e_D \urcorner}) &\cong \texttt{s}(e_F(\overline{\ulcorner e_D \urcorner})) && (15.15)\\ &\cong \texttt{s}(e_D(\overline{\ulcorner e_D \urcorner})), && (15.16) \end{aligned}$$

which is a contradiction (why?).

It is crucial to the argument just given that all functions in $\mathcal{L}\{\texttt{nat} \rightarrow\}$ are total, and it is precisely this property that precludes defining the evaluation function, E, in the preceding argument. To ensure that every function is total, we must, in effect, encode the termination proof for a function into the source code of the function itself. But this rules out defining an evaluation function, for its *one* totality proof would have to encode the totality proof for *all possible programs* in $\mathcal{L}\{\texttt{nat} \rightarrow\}$, which is scarcely plausible. The foregoing argument proves that it is, indeed, impossible.

15.6 Exercises

Chapter 16

Plotkin's PCF

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, also known as *Plotkin's PCF*, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to $\mathcal{L}\{\texttt{nat} \rightarrow\}$ expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ may not terminate when evaluated; consequently, functions are partial (may be undefined for some arguments), rather than total (which explains the "partial arrow" notation for function types). Compared to $\mathcal{L}\{\texttt{nat} \rightarrow\}$, the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ moves the termination proof from the expression itself to the mind of the programmer. The type system no longer ensures termination, which permits a wider range of functions to be defined in the system, but at the cost of admitting infinite loops when the termination proof is either incorrect or absent.

The crucial concept embodied in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is the *fixed point* characterization of recursive definitions. In ordinary mathematical practice one may define a function f by *recursion equations* such as these:

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= (n+1) \times f(n) \end{aligned}$$

These may be viewed as simultaneous equations in the variable, f, ranging over functions on the natural numbers. The function we seek is a *solution* to these equations—a function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied. We must, of course, show that these equations have a unique solution, which is easily shown by mathematical induction on the argument to f.

The solution to such a system of equations may be characterized as the fixed point of an associated functional (operator mapping functions to functions). To see this, let us re-write these equations in another form:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Re-writing yet again, we seek f such that

$$f : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Now define the *functional* F by the equation $F(f) = f'$, where

$$f' : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Note well that the condition on f' is expressed in terms of the argument, f, to the functional F, and not in terms of f' itself! The function f we seek is then a *fixed point* of F, which is a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words f is defined to the *fix*(F), where *fix* is an operator on functionals yielding a fixed point of F.

Why does an operator such as F have a fixed point? Informally, a fixed point may be obtained as the limit of series of approximations to the desired solution obtained by iterating the functional F. This is where partial functions come into the picture. Let us say that a partial function, ϕ on the natural numbers, is an *approximation* to a total function, f, if $\phi(m) = n$ implies that $f(m) = n$. Let $\perp$: $\mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\perp(n)$ is undefined for every $n \in \mathbb{N}$. Intuitively, this is the "worst" approximation to the desired solution, f, of the recursion equations given above. Given any approximation, ϕ, of f, we may "improve" it by considering $\phi' = F(\phi)$. Intuitively, ϕ' is defined on 0 and on $m + 1$ for every $m \geq 0$ on which ϕ is defined. Continuing in this manner, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ', and hence a further improvement on ϕ. If we start with $\perp$ as the initial approximation to f, then pass to the limit

$$\lim_{i \geq 0} F^{(i)}(\perp),$$

we will obtain the least approximation to f that is defined for every $m \in \mathbb{N}$, and hence is the function f itself. Turning this around, if the limit exists, it must be the solution we seek.

This fixed point characterization of recursion equations is taken as a primitive concept in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$—we may obtain the least fixed point of *any* functional definable in the language. Using this we may solve any set of recursion equations we like, with the proviso that there is no guarantee that the solution is a *total* function. Rather, it is guaranteed to be a *partial* function that may be undefined on some, all, or no inputs. This is the price we may for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like it to be. It is our task as programmer's to ensure that the functions defined by recursion are total—*i.e.*, that all of our loops terminate.

16.1 Static Semantics

The abstract binding syntax of PCF is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{nat}$	$\texttt{nat}$
		$\mid$	$\texttt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
Expr	e	$::=$	x	x
		$\mid$	$\texttt{z}$	$\texttt{z}$
		$\mid$	$\texttt{s}(e)$	$\texttt{s}(e)$
		$\mid$	$\texttt{ifz}(e;e_0;x.e_1)$	$\texttt{ifz}\,e\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}$
		$\mid$	$\texttt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$
		$\mid$	$\texttt{fix}[\tau](x.e)$	$\texttt{fix}\,x{:}\tau\,\texttt{is}\,e$

The expression $\texttt{fix}[\tau](x.e)$ is called *general recursion*; it is discussed in more detail below. The expression $\texttt{ifz}(e;e_0;x.e_1)$ branches according to whether e evaluates to $\texttt{z}$ or not, binding the predecessor to x in the case that it is not.

The static semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{16.1a}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}} \tag{16.1b}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}} \tag{16.1c}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat} \vdash e_1 : \tau}{\Gamma \vdash \texttt{ifz}(e;e_0;x.e_1) : \tau} \tag{16.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{parr}(\tau_1;\tau_2)} \tag{16.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{16.1f}$$

$$\frac{\Gamma, \texttt{fix[}\tau\texttt{]}(x.e) : \tau \vdash [\texttt{fix[}\tau\texttt{]}(x.e)/x]e : \tau}{\Gamma \vdash \texttt{fix[}\tau\texttt{]}(x.e) : \tau} \tag{16.1g}$$

Rule (16.1g) captures the essence of recursive self-reference by replacing occurrences of x by the recursive expression itself during type checking. The reasoning is "circular" in that to check $\texttt{fix[}\tau\texttt{]}(x.e) : \tau$, we *assume* that it is so, and *deduce* that $[\texttt{fix[}\tau\texttt{]}(x.e)/x]e : \tau$.

The structural rules, including in particular substitution, are admissible for the static semantics.

Lemma 16.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

An equivalent formulation of Rule (16.1g) treats the recursive self-reference as a variable:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix[}\tau\texttt{]}(x.e) : \tau} \tag{16.2}$$

To type check a recursive expression, we *assume* that the variable, x, has type τ while checking that the body has type τ. The advantage of Rule (16.1g) is that it avoids treating x as a *variable*, since its meaning does not vary—it stands for the recursive expression itself.

To see that Rule (16.2) is admissible relative to Rules (16.1), suppose that $\Gamma, x : \tau \vdash e : \tau$. Then by substitution and weakening it follows that

$$\Gamma, \texttt{fix[}\tau\texttt{]}(x.e) : \tau \vdash [\texttt{fix[}\tau\texttt{]}(x.e)/x]e : \tau,$$

and hence by Rule (16.1g) we have $\Gamma \vdash \texttt{fix[}\tau\texttt{]}(x.e) : \tau$. Conversely, if we take Rule (16.2) as primitive, then Rule (16.1g) is admissible. To show this, suppose that $\Gamma, \texttt{fix[}\tau\texttt{]}(x.e) : \tau \vdash [\texttt{fix[}\tau\texttt{]}(x.e)/x]e : \tau$. In the derivation of the consequent replace each use of reflexivity for the indicated assumption by a use of reflexivity in the form $\Gamma, x : \tau \vdash x : \tau$. The result is a derivation of $\Gamma, x : \tau \vdash e : \tau$, from which we have $\Gamma \vdash \texttt{fix[}\tau\texttt{]}(x.e) : \tau$ by Rule (16.2).

16.2 Dynamic Semantics

The judgement $e\ \mathsf{val}$ determines which expressions are (closed) values. The definition of this judgement varies according to whether we adopt an eager or lazy interpretation of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. This judgement is defined by the following rules:

$$\frac{}{\mathsf{z}\ \mathsf{val}} \tag{16.3a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}} \tag{16.3b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{16.3c}$$

The bracketed premise is to be omitted for the lazy variant, and included for the eager variant.

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the following rules:

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\} \tag{16.4a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{16.4b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{16.4c}$$

$$\frac{}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{16.4d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{16.4e}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1;e_2')}\right\} \tag{16.4f}$$

$$\frac{\{e_2\ \mathsf{val}\}}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{16.4g}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e} \tag{16.4h}$$

The bracketed rules and premises are to be omitted for the lazy variant, and included for the eager variant of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. Rule (16.4h) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 16.2 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.

Proof. The proof of preservation is by induction on the derivation of the transition judgement. Consider Rule (16.4h). Suppose that $\mathtt{fix}[\tau](x.e) : \tau$. By inversion of typing we have $\mathtt{fix}[\tau](x.e) : \tau \vdash [\mathtt{fix}[\tau](x.e)/x]e : \tau$, from which the result follows directly by transitivity of the hypothetical judgement. The proof of progress proceeds by induction on the derivation of the typing judgement. For example, for Rule (16.1g) the result follows immediately since we may make progress by unwinding the recursion. □

16.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. The drawback is that, in contrast to primitive recursion, the termination of a recursively defined function is not intrinsic to the program itself, but rather must be proved extrinsically by the programmer. The benefit is a much greater freedom in writing programs.

Every primitive recursive function is definable in PCF. One way to see this is to first introduce notation for general recursive functions, and then use these to define primitive recursion. A general recursive function expression has the form $\mathtt{fun}[\tau_1;\tau_2](x.y.e)$, where x is a variable standing for the function itself, and y is its argument. This form generalizes ordinary λ-notation, since we may always disregard the name for the function itself. The static semantics of general recursive function expressions is given by the following rule:

$$\frac{\Gamma,\mathtt{fun}[\tau_1;\tau_2](x.y.e) : \mathtt{parr}(\tau_1;\tau_2), y : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}[\tau_1;\tau_2](x.y.e) : \mathtt{parr}(\tau_1;\tau_2)} . \tag{16.5}$$

Its dynamic semantics may be given by the following rule for application, generalized to recursive functions:

$$\frac{\{e_1\ \mathsf{val}\} \quad e = \mathtt{fun}[\tau_1;\tau_2](x.y.e')}{\mathtt{ap}(e;e_1) \mapsto [e,e_1/x,y]e'} \tag{16.6}$$

At the call site the function itself is substituted for x, the name that it has given itself, within its body.

General recursive functions are definable from general recursion and non-recursive functions. Specifically, we may take $\mathtt{fun}[\tau_1;\tau_2](x.y.e)$ to stand for the compound expression

$$\mathtt{fix}[\mathtt{parr}(\tau_1;\tau_2)](x.\mathtt{lam}[\tau_1](y.e)).$$

It is easy to check that the static and dynamic semantics of recursive functions are derivable from this definition.

Returning to primitive recursion, we may define $\mathtt{rec}[\tau](e;e_0;x.y.e_1)$ to be the expression $\mathtt{ap}(e';e)$, where e' is the recursive function

$$\mathtt{fun}[\mathtt{nat};\tau](f.u.\mathtt{ifz}(u;e_0;x.[\mathtt{ap}(f;x)/y]e_1)).$$

It is easy to check that the static and dynamic semantics of primitive recursion are derivable in $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ using this expansion.

To discuss definability in general, we proceed as in Chapter 15. First, we define the relation $e \cong e' : \tau\ [\Gamma]$ of *observational congruence* to be the coarsest consistent congruence relation—an equivalence relation such that (a) any expression may be replaced by an observationally congruent one without changing the observable behavior, and (b) is consistent in that it does not equate a terminating to a non-terminating expression. We rely here on some intuitively plausible forms of equational reasoning for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ whose justifications are made rigorous in Chapter 53.

Since expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ may not terminate, functions definable in it are necessarily partial. We say that a partial function of the natural numbers, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ iff there is an expression $e_\phi : \texttt{nat} \rightharpoonup \texttt{nat}$ such that $\phi(m) = n$ iff $e_\phi(\overline{m}) \cong \overline{n} : \texttt{nat}$. So, for example, if ϕ is the totally undefined function, then e_ϕ must be a function that loops infinitely whenever it is called.

It is informative to classify those partial functions ϕ that are definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. It turns out that these are the so-called *partial recursive functions*, which are defined to be the primitive recursive functions (definable in the first-order fragment of $\mathcal{L}\{\texttt{nat} \rightarrow\}$), augmented by closure under *minimization*: given ϕ, define $\psi(m)$ to be the least $n \geq 0$ such that (1) for $m < n$, $\phi(m)$ is defined and non-zero, and (2) $\phi(n) = 0$. If no such n exists, then $\psi(m)$ is undefined.

Theorem 16.3. *A partial function ϕ on the natural numbers is definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ iff it is partial recursive.*

Proof sketch. Minimization is readily definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, so it is at least as powerful as the class of partial recursive functions. Conversely, we may, with considerable tedium, define an evaluator for expressions of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ does not exceed the power of the class of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the class of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language currently available or that will ever be available.[1] Therefore $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is as powerful as any other programming language with respect to the class of definable functions on the natural numbers.

[1] See Chapter 22 for further discussion of Church's Law.

Let ϕ_{univ} be the partial function on the natural numbers such that

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \mapsto^* \overline{n}.$$

By Church's Law this function is definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, whereas we showed in Chapter 15 that the analogous function is not definable in $\mathcal{L}\{\texttt{nat} \rightarrow\}$. It is an instructive exercise to consider why the diagonal argument given there does not apply here.

16.4 *Contextual Semantics

Recall from Chapter 10 that a contextual semantics has only one transition rule,

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \leadsto e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto_{\mathsf{c}} e'} \tag{16.7}$$

This rule is defined in terms of the decomposition of an expression into an evaluation context and a redex, which is then replaced by its contractum.

The instruction steps for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ are inductively defined by the following inference rules:

$$\frac{}{\texttt{ifz(z;}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \leadsto e_0} \tag{16.8a}$$

$$\frac{\{e \ \mathsf{val}\}}{\texttt{ifz(s(}e\texttt{);}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \leadsto [e/x]e_1} \tag{16.8b}$$

$$\frac{\{e_2 \ \mathsf{val}\}}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{);}e_2\texttt{)} \leadsto [e_2/x]e} \tag{16.8c}$$

$$\frac{}{\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)} \leadsto [\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)}/x]e} \tag{16.8d}$$

The bracketed premises are to be omitted for the lazy variant, and included for the eager variant.

The evaluation contexts are inductively defined by the following rules:

$$\frac{}{\circ \ \mathsf{ectxt}} \tag{16.9a}$$

$$\left\{\frac{\mathcal{E} \ \mathsf{ectxt}}{\texttt{s(}\mathcal{E}\texttt{)} \ \mathsf{ectxt}}\right\} \tag{16.9b}$$

$$\frac{\mathcal{E} \ \mathsf{ectxt}}{\texttt{ifz(}\mathcal{E}\texttt{;}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \ \mathsf{ectxt}} \tag{16.9c}$$

$$\frac{\mathcal{E}_1 \ \mathsf{ectxt}}{\texttt{ap(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \ \mathsf{ectxt}} \tag{16.9d}$$

$$\left\{\frac{e_1\ \mathsf{val}\quad \mathcal{E}_2\ \mathsf{ectxt}}{\mathtt{ap}(e_1;\mathcal{E}_2)\ \mathsf{ectxt}}\right\} \tag{16.9e}$$

The bracketed rules are to be omitted for the lazy variant of the language.

It is a straightforward exercise to define the judgement $e = \mathcal{E}\{e_0\}$, which states that the result of replacing the "hole" in $\mathcal{E}$ by e_0 is e.

Let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural operational semantics, and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual semantics.

Theorem 16.4. *For any expression $e : \mathtt{nat}$ and any v val, $e \mapsto_{\mathsf{s}}^{*} v$ iff $e \mapsto_{\mathsf{c}}^{*} v$*

Proof. It suffices to that $e \mapsto_{\mathsf{s}} e'$ iff $e \mapsto_{\mathsf{c}} e'$, as in the proof of Theorem 10.2 on page 77. □

16.5 *Compactness

An important property of general recursion is called *compactness*, which implies that only finitely many unwindings of a recursive expression are needed to complete the evaluation of a program. While intuitively obvious (one cannot complete infinitely many recursive calls in a finite computation), it is rather tricky to state and prove rigorously. To get a feel for what is involved, we consider two motivating examples.

Consider the familiar factorial function, f, in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$:

$$\mathtt{fix}\,f\mathtt{:nat} \rightharpoonup \mathtt{nat}\,\mathtt{is}\,\lambda\mathtt{(}x\mathtt{:nat.ifz}\,x\,\{\mathtt{z}\Rightarrow\mathtt{s(z)} \mid \mathtt{s(}x'\mathtt{)}\Rightarrow x\mathtt{*}f\mathtt{(}x'\mathtt{)}\}\mathtt{)}.$$

Obviously evaluation of $f(\overline{n})$ requires n recursive calls to the function itself. This means that, for a given input, n, we may place a *bound*, k, on the recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the k-bounded form of factorial, $f^{(k)}$, is written

$$\mathtt{fix}^{k}\,f\mathtt{:nat} \rightharpoonup \mathtt{nat}\,\mathtt{is}\,\lambda\mathtt{(}x\mathtt{:nat.ifz}\,x\,\{\mathtt{z}\Rightarrow\mathtt{s(z)} \mid \mathtt{s(}x'\mathtt{)}\Rightarrow x\mathtt{*}f\mathtt{(}x'\mathtt{)}\}\mathtt{)}.$$

The superscript k limits the recursion to at most k unwindings, after which the computation diverges. Thus, if $f(\overline{n})$ terminates, then for some $k \geq 0$ (in fact, $k = n$ for this simple case), $f^{(k)}(\overline{n})$ also terminates.

One might expect something even stronger, namely that there is a bound that ensures termination with the *same* value. But this is not always the case. For example, in the case of a lazy dynamic semantics, we may consider the identity function defined by

$$\texttt{fix}\,i\texttt{:nat is}\,\lambda\texttt{(}x\texttt{:nat.}\,\texttt{ifz}\,x\,\{\texttt{z}\Rightarrow\texttt{z}\mid\texttt{s(}x'\texttt{)}\Rightarrow\texttt{s(}i\texttt{(}x'\texttt{))}\}\texttt{)}.$$

Applying this to a number, n, results in the successor of a recursive call to the function itself. Thus, for any non-zero input the computation terminates in three steps (unwind, apply, conditional branch), but the result is not the same when the recursion is bounded as when it is not, because the residual will contain bounded recursions whereas the original will contain unbounded recursions.

This example does not apply in the eager variant of the language, since values of type nat are numerals. But something similar does arise. Consider the addition function, a, of type $\tau = \texttt{nat} \rightharpoonup \texttt{(nat} \rightharpoonup \texttt{nat)}$, given by the expression

$$\texttt{fix}\,p\texttt{:}\tau\,\texttt{is}\,\lambda\texttt{(}x\texttt{:nat.}\,\texttt{ifz}\,x\,\{\texttt{z}\Rightarrow id\mid\texttt{s(}x'\texttt{)}\Rightarrow s\circ\texttt{(}p\texttt{(}x'\texttt{))}\}\texttt{)},$$

where $id = \lambda\texttt{(}y\texttt{:nat.}\,y\texttt{)}$ is the identity, $e' \circ e = \lambda\texttt{(}x\texttt{:}\tau\texttt{.}\,e'\texttt{(}e\texttt{(}x\texttt{)))}$ is composition, and $s = \lambda\texttt{(}x\texttt{:nat.}\,\texttt{s(}x\texttt{))}$ is the successor function. The application $a\texttt{(}\overline{m}\texttt{)}$ terminates after three transitions, regardless of the value of m, resulting in a λ-abstraction. When m is positive, the result contains a *residual* copy of a itself, which is applied to the predecessor of m as a recursive call. The corresponding k-bounded version of a, written $a^{(k)}$, also terminates in three steps, provided that $k > 0$. But the result in the case of a positive argument, m, is a λ-abstraction that contains a residual copy of $a^{(k-1)}$, not of $a^{(k)}$ or of a itself.

The proof of compactness is based on the contextual semantics given in Section 16.4 on page 128. This simplifies the proof compared to the usual structural semantics, because contextual semantics permits us to restrict attention to transitions between complete programs, whereas structural semantics requires us to consider transitions at arbitrary type to account for the premises of the rules.

As a technical convenience we will enrich the syntax of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ with *bounded recursion*, written $\texttt{fix}^k\,x\texttt{:}\tau\,\texttt{is}\,e$, where $k \geq 0$. The static semantics is the same as for general recursion, the parameter k playing no role in typing. The dynamic semantics is defined by the following primitive instruction rules:

$$\frac{}{\texttt{fix}^0\texttt{[}\tau\texttt{](}x\texttt{.}e\texttt{)} \leadsto \texttt{fix}^0\texttt{[}\tau\texttt{](}x\texttt{.}e\texttt{)}} \tag{16.10a}$$

$$\frac{}{\texttt{fix}^{k+1}\texttt{[}\tau\texttt{](}x\texttt{.}e\texttt{)} \leadsto [\texttt{fix}^{k}\texttt{[}\tau\texttt{](}x\texttt{.}e\texttt{)}/x]e} \tag{16.10b}$$

If k is positive, the recursive bound is decremented so that subsequent uses of it will be limited to one fewer unrolling. If k reaches zero, the expression steps to itself so that computation with it diverges with no result.

Let $f^{(\omega)} = \mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e_x$ be an arbitrarily chosen recursive expression such that $f^{(\omega)} : \tau$, and let $f^{(k)} = \mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e_x$ be the corresponding k-bounded recursive expression, for which we also have $f^{(k)} : \tau$. Observe that by inversion of the static semantics of recursive expressions, we have $x : \tau \vdash e_x : \tau$.

Lemma 16.5. *If $e_0 = [f^{(\omega)}/y]e_1 \rightsquigarrow e_0'$, then $e_0' = [f^{(\omega)}/y]e_1'$ for some e_1'. Moreover, if $e_0 = [f^{(k)}/y]e_1 \rightsquigarrow e_0'$, then $e_0' = [f^{(j)}/y]e_1'$ for some e_1' and $j \leq k$.*

Proof. Immediate, by inspection of Rules (16.8). □

Lemma 16.6. *If $[f^{(k)}/y]e \downarrow$, then $[f^{(k+1)}/y]e \downarrow$.*

Proof. It is enough to prove that if

$$[f^{(k)}/y]\mathcal{E}\{[f^{(k)}/y]e_0\} \downarrow,$$

then

$$[f^{(k+1)}/y]\mathcal{E}\{[f^{(k+1)}/y]e_0\} \downarrow .$$

□

Theorem 16.7 (Compactness)**.** *Suppose that $y : \tau \vdash e : \tau'$, where $y \# f^{(\omega)}$. If $[f^{(\omega)}/y]e \downarrow$, then there exists $k \geq 0$ such that $[f^{(k)}/y]e \downarrow$.*

Proof. If $[f^{(\omega)}/y]e$ val, then since e cannot be the variable y, it must itself be a value, independently of whether y val. The result then follows immediately, choosing k arbitrarily. Otherwise, suppose that $[f^{(\omega)}/y]e \mapsto_{\mathsf{c}} e' \downarrow$. That is, $[f^{(\omega)}/y]e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. Therefore $\mathcal{E}$ has the form $[f^{(\omega)}/y]\mathcal{E}_1$ for some evaluation context $\mathcal{E}_1$, $e_0 = [f^{(\omega)}/y]e_1$ for some closed expression e_1, and so $e' = [f^{(\omega)}/y]\mathcal{E}_1\{e_0'\}$.

We proceed by case analysis on whether or not e_1 is the distinguished variable, y. If so, the instruction step under consideration is precisely the unrolling of the distinguished recursive expression $f^{(\omega)}$. Consequently, $e_0 = f^{(\omega)}$, and, therefore, $e_0' = [f^{(\omega)}/x]e_x$, where we may assume without loss of generality $x \# y$. Now

$$[f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/x]e_x\} = [f^{(\omega)}/y](\mathcal{E}_1\{[y/x]e_x\}),$$

and so by induction there exists $k \geq 0$ such that

$$\begin{aligned}[f^{(k)}/y](\mathcal{E}_1\{[y/x]e_x\}) &= [f^{(k)}/y]\mathcal{E}_1\{[y/x]e_x\} \\ &= [f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/x]e_x\} \downarrow .\end{aligned}$$

Hence, by Lemma 16.6 on the preceding page, noting that $y \# [f^{(k)}/x]e_x$, and applying the dynamic semantics of bounded recursion, we have

$$\begin{aligned}[f^{(k+1)}/y](\mathcal{E}_1\{y\}) &= [f^{(k+1)}/y]\mathcal{E}_1\{f^{(k+1)}\} \\ &\mapsto_{\mathsf{c}} [f^{(k+1)}/y]\mathcal{E}_1\{[f^{(k)}/x]e_x\} \downarrow .\end{aligned}$$

This completes the proof for the case $e_1 = y$.

Otherwise, it follows from Lemma 16.5 on the previous page that $e_0' = [f^{(\omega)}/y]e_1'$ for some expression e_1'. That is, we have

$$\begin{aligned}[f^{(\omega)}/y](\mathcal{E}_1\{e_0\}) &= [f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/y]e_1\} \\ &\mapsto_{\mathsf{c}} [f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/y]e_1'\} \\ &= [f^{(\omega)}/y](\mathcal{E}_1\{e_1'\}) \downarrow .\end{aligned}$$

Therefore, by induction, there exists $k \geq 0$ such that

$$[f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1'\} = [f^{(k)}/y](\mathcal{E}_1\{e_1'\}) \downarrow .$$

It follows that

$$[f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1\} \mapsto_{\mathsf{c}} [f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1'\} \downarrow .$$

□

16.6 Exercises

Part V

Products and Sums

Chapter 17

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, we may consider the *general product*, $\prod_{i \in I} \tau_i$, where τ_i is a type for each $i \in I$. The elements of the general product type are *I-indexed tuples* whose *i*th component is an element of type τ_i. The components are accessed by *I-indexed projection* operations, generalizing the binary case. Special cases of the general product include *n-tuples*, indexed by sets of the form $I = \{ 0, \ldots, n-1 \}$, and *labelled tuples*, or *records*, indexed by sets of symbols that label the components of the tuple.

17.1 Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{unit}$	$\texttt{unit}$
		$\mid$	$\texttt{prod(}\tau_1\texttt{;}\tau_2\texttt{)}$	$\tau_1 \times \tau_2$
Expr	e	$::=$	$\texttt{triv}$	$\langle\rangle$
		$\mid$	$\texttt{pair(}e_1\texttt{;}e_2\texttt{)}$	$\langle e_1\texttt{,}e_2\rangle$
		$\mid$	$\texttt{fst(}e\texttt{)}$	$\texttt{fst(}e\texttt{)}$
		$\mid$	$\texttt{snd(}e\texttt{)}$	$\texttt{snd(}e\texttt{)}$

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 element*, 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.

$$\frac{}{\Gamma \vdash \texttt{triv} : \texttt{unit}} \tag{17.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1;e_2) : \texttt{prod}(\tau_1;\tau_2)} \tag{17.1b}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \tag{17.1c}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2} \tag{17.1d}$$

The dynamic semantics of product types is specified by the following rules:

$$\frac{}{\texttt{triv}\ \mathsf{val}} \tag{17.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{pair}(e_1;e_2)\ \mathsf{val}} \tag{17.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\texttt{pair}(e_1;e_2) \mapsto \texttt{pair}(e_1';e_2)}\right\} \tag{17.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{pair}(e_1;e_2) \mapsto \texttt{pair}(e_1;e_2')}\right\} \tag{17.2d}$$

$$\frac{e \mapsto e'}{\texttt{fst}(e) \mapsto \texttt{fst}(e')} \tag{17.2e}$$

$$\frac{e \mapsto e'}{\texttt{snd}(e) \mapsto \texttt{snd}(e')} \tag{17.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{fst}(\texttt{pair}(e_1;e_2)) \mapsto e_1} \tag{17.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{snd}(\texttt{pair}(e_1;e_2)) \mapsto e_2} \tag{17.2h}$$

The bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics of pairing.

Theorem 17.1 (Safety).

1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*
2. *If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.*

17.2 Finite Products

Finite product types are the natural generalization of nullary and binary tuples to finite index sets. The syntax of finite products makes use of *finite families* of types and expressions.

A *finite family of types* indexed by I is a finite function ϕ such that $\phi(i)$ is a type for each $i \in I$, and similarly a *finite family of expressions* indexed by I is a finite function ε such that $\varepsilon(i)$ is an expression for each $i \in I$. We often write $i \mapsto \tau_i$ for the type family ϕ such that $\phi(i) = \tau_i$ for each $i \in I$, where the index set I is understood from context. Similarly, we write $i \mapsto e_i$ for the expression family ε such that $\varepsilon(i) = e_i$ for each $i \in I$. As a notational convenience, we write $\langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ for the type family $i \mapsto \tau_i$ indexed by $I = \{ i_0, \ldots, i_{n-1} \}$, and we similarly write $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$ for the expression $i \mapsto e_i$ indexed over the same set I.

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}$	$\prod_{i \in I} \tau_i$
Expr	e	$::=$	$\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}$	$\langle e_i \rangle_{i \in I}$
		$\mid$	$\texttt{proj[}I\texttt{][}i\texttt{](}e\texttt{)}$	$e \cdot i$

Using the explicit notation for finite families, we write $\prod \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ for $\prod_{i \in I} \tau_1$, and $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$ for $\langle e_i \rangle_{i \in I}$, where $I = \{ i_0, \ldots, i_{n-1} \}$.

The static semantics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)} : \texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}} \tag{17.3a}$$

$$\frac{\Gamma \vdash e : \texttt{prod[}I\texttt{](}i \mapsto e_i\texttt{)} \quad j \in I}{\Gamma \vdash \texttt{proj[}I\texttt{][}j\texttt{](}e\texttt{)} : \tau_j} \tag{17.3b}$$

In Rule (17.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (17.3a), the index i ranges over the index set I.

The dynamic semantics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}\ \mathsf{val}} \tag{17.4a}$$

$$\frac{e_j \mapsto e'_j \quad (\forall i \neq j)\ e'_j = e_j}{\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)} \mapsto \texttt{tuple[}I\texttt{](}i \mapsto e'_i\texttt{)}} \tag{17.4b}$$

$$\frac{\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}\ \mathsf{val}}{\texttt{proj[}I\texttt{][}j\texttt{](tuple[}I\texttt{](}i \mapsto e_i\texttt{))} \mapsto e_j} \tag{17.4c}$$

Rule (17.4b) specifies that the components of a tuple are to be evaluated in *some* sequential order, without specifying the order in which they components are considered. It is straightforward, if a bit technically complicated, to impose a linear ordering on index sets that determines the evaluation order of the components of a tuple.

Theorem 17.2 (Safety). *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \mapsto e'$.*

Proof. The safety theorem may be decomposed into progress and preservation lemmas, which are proved as in Section 17.1 on page 135. □

We may define nullary and binary products as particular instances of finite products by choosing an appropriate index set. The type `unit` may be defined as the product $\prod_{\_\in\emptyset}\emptyset$ of the empty family over the empty index set, taking the expression $\langle\rangle$ to be the empty tuple, $\langle\emptyset\rangle_{\_\in\emptyset}$. Binary products $\tau_1 \times \tau_2$ may be defined as the product $\prod_{i\in\{\,1,2\,\}}\tau_i$ of the two-element family of types consisting of τ_1 and τ_2. The pair $\langle e_1, e_2\rangle$ may then be defined as the tuple $\langle e_i\rangle_{i\in\{\,1,2\,\}}$, and the projections $\mathtt{fst}(e)$ and $\mathtt{snd}(e)$ are correspondingly defined, respectively, to be $e \cdot 1$ and $e \cdot 2$.

Finite products may similarly be used to define *labelled tuples*, or *records*, whose components are accessed by symbolic names. For example, let $L = \{\,l_1, \ldots, l_n\,\}$ be a finite set of symbols serving as *field labels*. The product type $\prod\langle l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\rangle$ has as values tuples of the form $\langle l_0 : e_0, \ldots, l_{n-1} : e_{n-1}\rangle$, where $e_i : \tau_i$ for each $0 \leq i < n$. The components of such a tuple, say e, are accessed by projections of the form $e \cdot l$, where $l \in L$.

17.3 Exercises

Chapter 18

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. *Finite sums* generalize nullary and binary sums to permit an arbitrary number of cases indexed by a finite index set, including the important special case of *labelled sums*, in which the summands are designated by a symbolic label.

18.1 Binary and Nullary Sums

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{void}$	$\texttt{void}$
		$\mid$	$\texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}$	$\tau_1 + \tau_2$
Expr	e	$::=$	$\texttt{abort[}\tau\texttt{](}e\texttt{)}$	$\texttt{abort}_\tau\, e$
		$\mid$	$\texttt{in[l][}\tau\texttt{](}e\texttt{)}$	$\texttt{in[l](}e\texttt{)}$
		$\mid$	$\texttt{in[r][}\tau\texttt{](}e\texttt{)}$	$\texttt{in[r](}e\texttt{)}$
		$\mid$	$\texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)}$	$\texttt{case}\, e\, \{\texttt{in[l](}x_1\texttt{)} \Rightarrow e_1 \mid \texttt{in[r](}x_2\texttt{)} \Rightarrow e_2\}$

The type $\texttt{void}$ 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, $\texttt{abort[}\tau\texttt{](}e\texttt{)}$, aborts the computation in the event that e evaluates to a value, which it cannot do. The type

$\tau = \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}$ is the *binary sum*. The elements of the sum type are *labelled* to indicate whether they are drawn from the left or the right summand, either $\texttt{in[l][}\tau\texttt{](}e\texttt{)}$ or $\texttt{in[r][}\tau\texttt{](}e\texttt{)}$. A value of the sum type is eliminated by case analysis on the label of the value.

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

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort[}\tau\texttt{](}e\texttt{)} : \tau} \tag{18.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}}{\Gamma \vdash \texttt{in[l][}\tau\texttt{](}e\texttt{)} : \tau} \tag{18.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}}{\Gamma \vdash \texttt{in[r][}\tau\texttt{](}e\texttt{)} : \tau} \tag{18.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)} \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} : \tau} \tag{18.1d}$$

Both branches of the case analysis must have the same type. Since a 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.

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

$$\frac{e \mapsto e'}{\texttt{abort[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{abort[}\tau\texttt{](}e'\texttt{)}} \tag{18.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[l][}\tau\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{18.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[r][}\tau\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{18.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[l][}\tau\texttt{](}e\texttt{)} \mapsto \texttt{in[l][}\tau\texttt{](}e'\texttt{)}}\right\} \tag{18.2d}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[r][}\tau\texttt{](}e\texttt{)} \mapsto \texttt{in[r][}\tau\texttt{](}e'\texttt{)}}\right\} \tag{18.2e}$$

$$\frac{e \mapsto e'}{\texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto \texttt{case(}e'\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)}} \tag{18.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case(in[l][}\tau\texttt{](}e\texttt{);}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto [e/x_1]e_1} \tag{18.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case(in[r][}\tau\texttt{](}e\texttt{);}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto [e/x_2]e_2} \tag{18.2h}$$

The bracketed premises and rules are to be included for an eager semantics, and excluded for a lazy semantics.

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

Theorem 18.1 (Safety). 1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

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

Proof. The proof proceeds along standard lines, by induction on Rules (18.2) for preservation, and by induction on Rules (18.1) for progress. □

18.2 Finite Sums

Just as we may generalize nullary and binary products to finite products, so may we also generalize nullary and binary sums to finite sums. The syntax for finite sums is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{sum}[I](i \mapsto \tau_i)$	$\sum_{i \in I} \tau_i$
Expr	e	::=	$\texttt{inj}[I][j](e)$	$\texttt{in}[j](e)$
		|	$\texttt{case}[I](e; i \mapsto x_i\,.\,e_i)$	$\texttt{case}\, e\, \{\texttt{in}[i](x_i) \Rightarrow e_i\}_{i \in I}$

The abstract binding tree representation of the finite case expression involves an I-indexed family of abstractors $x_i\,.\,e_i$, but is otherwise similar to the binary form. We write $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ for $\sum_{i \in I} \tau_i$, where $I = \{\, i_0, \ldots, i_{n-1} \,\}$.

The static semantics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \texttt{inj}[I][j](e) : \texttt{sum}[I](i \mapsto \tau_i)} \tag{18.3a}$$

$$\frac{\Gamma \vdash e : \texttt{sum}[I](i \mapsto \tau_i) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \texttt{case}[I](e; i \mapsto x_i\,.\,e_i) : \tau} \tag{18.3b}$$

These rules generalize to the finite case the static semantics for nullary and binary sums given in Section 18.1 on page 139.

The dynamic semantics of finite sums is defined by the following rules:

$$\frac{\{e\ \mathsf{val}\}}{\texttt{inj}[I][j](e)\ \mathsf{val}} \tag{18.4a}$$

$$\left\{ \frac{e \mapsto e'}{\texttt{inj}[I][j](e) \mapsto \texttt{inj}[I][j](e')} \right\} \tag{18.4b}$$

$$\frac{e \mapsto e'}{\texttt{case}[I](e; i \mapsto x_i\,.\,e_i) \mapsto \texttt{case}[I](e'; i \mapsto x_i\,.\,e_i)} \tag{18.4c}$$

$$\frac{\texttt{inj}[I][j](e)\ \mathsf{val}}{\texttt{case}[I](\texttt{inj}[I][j](e); i \mapsto x_i\,.\,e_i) \mapsto [e/x_j]e_j} \tag{18.4d}$$

These again generalize the dynamic semantics of binary sums given in Section 18.1 on page 139.

Theorem 18.2 (Safety). *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \mapsto e'$.*

Proof. The proof is similar to that for the binary case, as described in Section 18.1 on page 139. □

As with products, nullary and binary sums are special cases of the finite form. The type void may be defined to be the sum type $\sum_{\_\in\emptyset} \emptyset$ of the empty family of types. The expression $\texttt{abort}(e)$ may corresponding be defined as the empty case analysis, $\texttt{case}\,e\,\{\emptyset\}$. Similarly, the binary sum type $\tau_1 + \tau_2$ may be defined as the sum $\sum_{i\in I} \tau_i$, where $I = \{\,\texttt{l},\texttt{r}\,\}$ is the two-element index set. The binary sum injections $\texttt{in[l]}(e)$ and $\texttt{in[r]}(e)$ are defined to be their counterparts, $\texttt{in[l]}(e)$ and $\texttt{in[r]}(e)$, respectively. Finally, the binary case analysis,

$$\texttt{case}\,e\,\{\texttt{in[l]}(x_{\texttt{l}}) \Rightarrow e_{\texttt{l}} \mid \texttt{in[r]}(x_{\texttt{r}}) \Rightarrow e_{\texttt{r}}\},$$

is defined to be the case analysis, $\texttt{case}\,e\,\{\texttt{in}[i](x_i) \Rightarrow \tau_i\}_{i\in I}$. It is easy to check that the static and dynamic semantics of sums given in Section 18.1 on page 139 is preserved by these definitions.

Two special cases of finite sums arise quite commonly. The *n-ary sum* corresponds to the finite sum over an index set of the form $\{\,0,\ldots,n-1\,\}$ for some $n \geq 0$. The *labelled sum* corresponds to the case of the index set being a finite set of symbols serving as symbolic indices for the injections.

18.3 Some Uses of Sum Types

Sum types have numerous uses, several of which we outline here. More interesting examples arise once we also have recursive types, which are introduced in Part VI.

18.3.1 Void and Unit

It is instructive to compare the types `unit` and `void`, which are often confused with one another. The type `unit` has exactly one element, `triv`, 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 not yield a value*; 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 what is called the `void` type in many languages is really the type `unit` because it indicates that an expression has no interesting value, not that it has no value at all!

18.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	`bool`	`bool`
Expr	e	$::=$	`tt`	`tt`
		$\mid$	`ff`	`ff`
		$\mid$	$\texttt{if}(e;e_1;e_2)$	$\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2$

The values of type `bool` are `tt` and `ff`. The expression $\texttt{if}(e;e_1;e_2)$ branches on the value of e : `bool`. We leave a precise formulation of the static and dynamic semantics of this type as an exercise for the reader.

The type `bool` is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{sum(unit;unit)} \tag{18.5a}$$

$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{18.5b}$$

$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{18.5c}$$

$$\texttt{if}(e;e_1;e_2) = \texttt{case}(e;x_1.e_1;x_2.e_2) \tag{18.5d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that x_1 # e_1 and x_2 # e_2. (We often write an underscore in place of a variable to stand for a variable that does not occur within its scope.) It is a simple matter to check that the evident static and dynamic semantics of the type `bool` is engendered by these definitions.

18.3.3 Enumerations

More generally, sum types may be used to define *finite enumeration* types, those whose values are one of an explicitly given finite set, and whose elimination form is a case analysis on the elements of that set. For example, the type `suit`, whose elements are $\clubsuit$, $\diamondsuit$, $\heartsuit$, and $\spadesuit$, has as elimination form the case analysis

$$\mathtt{case}\,e\,\{\clubsuit \Rightarrow e_0 \mid \diamondsuit \Rightarrow e_1 \mid \heartsuit \Rightarrow e_2 \mid \spadesuit \Rightarrow e_3\},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\mathtt{suit} = \sum_{\_\in I} \mathtt{unit}$, where $I = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and the type family is constant over this set. The case analysis form for a labelled sum is almost literally the desired case analysis for the given enumeration, the only difference being the binding for the uninteresting value associated with each summand, which we may ignore.

18.3.4 Options

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{opt}(\tau)$	$\tau\,\mathtt{opt}$
Expr	e	::=	$\mathtt{null}$	$\mathtt{null}$
		|	$\mathtt{just}(e)$	$\mathtt{just}(e)$
		|	$\mathtt{ifnull}[\tau](e; e_1; x.e_2)$	$\mathtt{check}\,e\{\mathtt{null} \Rightarrow e_1 \mid \mathtt{just}(x) \Rightarrow e_2\}$

The type $\mathtt{opt}(\tau)$ represents the type of "optional" values of type τ. The introductory forms are `null`, corresponding to "no value", and $\mathtt{just}(e)$, 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:

$$\mathtt{opt}(\tau) = \mathtt{sum}(\mathtt{unit}; \tau) \tag{18.6a}$$

$$\mathtt{null} = \mathtt{in[l][opt}(\tau)\mathtt{](triv)} \tag{18.6b}$$

$$\mathtt{just}(e) = \mathtt{in[r][opt}(\tau)\mathtt{]}(e) \tag{18.6c}$$

$$\mathtt{ifnull}[\tau](e; e_1; x_2.e_2) = \mathtt{case}(e; \_.e_1; x_2.e_2) \tag{18.6d}$$

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

The option type is the key to understanding a common misconception, the *null pointer fallacy*. This fallacy, which is particularly common in object-oriented languages, is based on two related errors. The first error is to deem the values of certain types to be mysterious entities called *pointers*, based on suppositions about how these values might be represented at run-time, rather than on the semantics of the type itself. The second error compounds the first. A particular value of a pointer type is distinguished as *the null pointer*, which, unlike the other elements of that type, does not designate a value of that type at all, but rather rejects all attempts to use it as such.

To help avoid such failures, such languages usually include a function, say $\mathtt{null} : \tau \to \mathtt{bool}$, that yields `tt` if its argument is null, and `ff` otherwise. This allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

$$\mathtt{if}\,\mathtt{null}(e)\,\mathtt{then}\ldots\mathit{error}\ldots\mathtt{else}\ldots\mathit{proceed}\ldots. \tag{18.7}$$

Despite this, "null pointer" exceptions at run-time are rampant, in part because it is quite easy to overlook the need for such a test, and in part because detection of a null pointer leaves little recourse other than abortion of the program.

The underlying problem may be traced to the failure to distinguish the type τ from the type $\mathtt{opt}(\tau)$. Rather than think of the elements of type τ as pointers, and thereby have to worry about the null pointer, one instead distinguishes between a *genuine* value of type τ and an *optional* value of type τ. An optional value of type τ may or may not be present, but, if it is, the underlying value is truly a value of type τ (and cannot be null). The elimination form for the option type,

$$\mathtt{ifnull}[\tau](e; e_{error}; x\,.\,e_{ok}) \tag{18.8}$$

propagates the information that e is present into the non-null branch by binding a genuine value of type τ to the variable x. The case analysis effects a change of type from "optional value of type τ" to "genuine value of type τ", so that within the non-null branch no further null checks, explicit or implicit, are required. Observe that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (18.7); the advantage of option types is precisely that it does so.

18.4 Exercises

1. Formulate general n-ary sums in terms of nullary and binary sums.
2. Explain why is makes little sense to consider self-referential sum types.

Chapter 19

Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

$$\texttt{let}\,x\,\texttt{be}\,e\,\texttt{in}\,\texttt{fst}\,(x)\texttt{+}\texttt{snd}\,(x)$$

to add the components of a pair, e, of natural numbers, we may instead write

$$\texttt{match}\,e\,\{\langle x,y\rangle \Rightarrow x\texttt{+}y\},$$

using pattern matching to name the components of the pair and refer to them directly. The first argument to the `match` expression is called the *match value* and the second argument consist of a finite sequence of *rules*, separated by vertical bars. In this example there is only one rule, but as we shall see shortly there is, in general, more than one rule in a given `match` expression. Each rule consists of a *pattern*, possibly involving variables, and an *expression* that may involve those variables (as well as any others currently in scope). The value of the `match` is determined by considering each rule in the order given to determine the first rule whose pattern matches the match value. If such a rule is found, the value of the `match` is the value of the expression part of the matching rule, with the variables of the pattern replaced by the corresponding components of the match value.

Pattern matching becomes more interesting, and useful, when combined with sums. The patterns $\texttt{in[l]}\,(x)$ and $\texttt{in[r]}\,(x)$ match the corresponding values of sum type. These may be used in combination with other patterns to express complex decisions about the structure of a value. For example, the following `match` expresses the computation that, when given a pair of type $(\texttt{unit}+\texttt{unit})\times\texttt{nat}$, either doubles or squares its second component depending on the form of its first component:

$$\texttt{match}\,e\,\{\langle\texttt{in[l]}(\langle\rangle),x\rangle \Rightarrow x\texttt{+}x \mid \langle\texttt{in[r]}(\langle\rangle),y\rangle \Rightarrow y\texttt{*}y\}. \tag{19.1}$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 17 and 18.

19.1 A Pattern Language

The main challenge in formalizing pattern matching is to manage the binding and scope of variables. The key observation is that a rule, $p \Rightarrow e$, binds variables in *both* the pattern, p, and the expression, e, simultaneously. Each rule in a sequence may bind any number of variables, independently of any preceding or succeeding rules. This gives rise to a somewhat unusual abstract syntax for sequences of rules that permits each rule to have a different, in general non-zero, valence. For example, the abstract syntax for expression (19.1) is given by

$$\texttt{match}(e;\texttt{rules[2]}(r_1;r_2)),$$

where r_1 is the rule

$$x\texttt{.rule(pair(in[l](triv);}x\texttt{);plus(}x;x\texttt{))}$$

and r_2 is the rule

$$y\texttt{.rule(pair(in[r](triv);}y\texttt{);times(}y;y\texttt{))}.$$

The salient point is that Each rule binds its own variables, in both the pattern and the expression.

The abstract syntax of pattern matching is formalized by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{match}(e;rs)$	$\texttt{match}\,e\,\{rs\}$
Rules	rs	::=	$\texttt{rules}[n](r_1;\ldots;r_n)$	$r_1 \mid \ldots \mid r_n$
Rule	r	::=	$x_1,\ldots,x_k\texttt{.rule}(p;e)$	$p \Rightarrow e$
Pattern	p	::=	$\texttt{wild}$	_
		$\mid$	x	x
		$\mid$	$\texttt{triv}$	$\langle\rangle$
		$\mid$	$\texttt{pair}(p_1;p_2)$	$\langle p_1,p_2\rangle$
		$\mid$	$\texttt{in[l]}(p)$	$\texttt{in[l]}(p)$
		$\mid$	$\texttt{in[r]}(p)$	$\texttt{in[r]}(p)$

The operator `rules[`n`]` has arity $(k_1, \ldots, k_n)$, where $n \geq 0$ and, for each $1 \leq i \leq n$, the ith rule has valence $k_i \geq 0$. Correspondingly, each rule consists of an abstractor binding $k \geq 0$ variables in a pattern and an expression. A pattern is either a variable, a *wild card* pattern, a *unit pattern* matching only the trivial element of the `unit` type, a *pair pattern*, or a *choice pattern*.

To specify the static semantics of pattern matching, we define several parametric judgements governing patterns and rules. In what follows the variable γ ranges over finite mappings from variables to types.

The judgement form

$$\mathcal{X} \mid p : \tau > \gamma$$

states that the pattern, p, is of type τ and has variables among those in $\mathcal{X}$, with the types assigned by the mapping γ. This judgement is parametric in $\mathcal{X}$, and that the three-place categorical judgment, $p : \tau > \gamma$, is intended to have mode $(\forall, \forall, \exists)$.

The judgement form

$$\mathcal{X} \mid \Gamma \vdash \gamma > e : \tau$$

states that the expression e, whose free variables are among those in $\mathcal{X}$, has type τ, provided that its variables have the types assigned by γ as well as those declared in Γ. The judgement $\gamma > e : \tau$ is also intended to have mode $(\forall, \forall, \exists)$.

The judgement form

$$\mathcal{X} \mid \Gamma \vdash r : \tau > \tau'$$

states that the rule r transforms the type τ to the type τ'. The judgement form

$$\mathcal{X} \mid \Gamma \vdash rs : \tau > \tau'$$

states the same property of a finite sequence of rules.

The judgement $\mathcal{X} \mid p : \tau > \gamma$ is inductively defined by the following rules:

$$\frac{}{\mathcal{X}, x \mid x : \tau > \langle x : \tau \rangle} \tag{19.2a}$$

$$\frac{}{\mathcal{X} \mid \texttt{wild} : \tau > \emptyset} \tag{19.2b}$$

$$\frac{}{\mathcal{X} \mid \texttt{triv} : \texttt{unit} > \emptyset} \tag{19.2c}$$

$$\frac{\mathcal{X}_1 \mid p_1 : \tau_1 > \gamma_1 \quad \mathcal{X}_2 \mid p_2 : \tau_2 > \gamma_2 \quad dom(\gamma_1) \cap dom(\gamma_2) = \emptyset}{\mathcal{X}_1\, \mathcal{X}_2 \mid \texttt{pair(}p_1\texttt{;}p_2\texttt{)} : \texttt{prod(}\tau_1\texttt{;}\tau_2\texttt{)} > \gamma_1 \otimes \gamma_2} \tag{19.2d}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_1 > \gamma_1}{\mathcal{X}_1 \mid \texttt{in[l]}(p) : \texttt{sum}(\tau_1;\tau_2) > \gamma_1} \tag{19.2e}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_2 > \gamma_2}{\mathcal{X}_2 \mid \texttt{in[r]}(p) : \texttt{sum}(\tau_1;\tau_2) > \gamma_2} \tag{19.2f}$$

The most interesting rule is Rule (19.2d), which expresses the formation of pair patterns. The disjointness requirement on γ_1 and γ_2 imposes the linearity condition on patterns: no variable may appear more than once in a pattern.

The parametric hypothetical judgement $\mathcal{X} \mid \Gamma \vdash \gamma > e : \tau$ is defined by the following rules:

$$\frac{\mathcal{X} \mid \Gamma \vdash e' : \tau'}{\mathcal{X} \mid \Gamma \vdash \emptyset > e' : \tau'} \tag{19.3a}$$

$$\frac{\mathcal{X}, x \mid \Gamma, x : \tau \vdash \gamma > e' : \tau'}{\mathcal{X} \mid \Gamma \vdash \langle x : \tau \rangle \otimes \gamma > e' : \tau'} \tag{19.3b}$$

The hypotheses Γ determine the types of the variables in scope at the point of the inference, and the mapping γ determines the types of the variables in the pattern. These rules traverse γ, introducing typing hypotheses $x : \tau$ for each variable x such that $\gamma(x) = \tau$, then checking that $e' : \tau'$ once all such hypotheses have been introduced.

The typing judgements for rules are defined as follows:

$$\frac{x_1, \ldots, x_n \mid p : \tau > \gamma \quad \mathcal{X}\, x_1, \ldots, x_n \mid \Gamma \vdash \gamma > e : \tau'}{\mathcal{X} \mid \Gamma \vdash x_1, \ldots, x_k.\texttt{rule}(p;e) : \tau > \tau'} \tag{19.4a}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \mathcal{X} \mid \Gamma \vdash r_n : \tau > \tau'}{\mathcal{X} \mid \Gamma \vdash \texttt{rules}[n](r_1;\ldots;r_n) : \tau > \tau'} \tag{19.4b}$$

In Rule (19.4a) we see that the association γ mediates between the pattern and the expression parts of a rule. The variables in the pattern must be among the abstracted variables, $x_1, \ldots, x_k$, which we require to be disjoint from $\mathcal{X}$. As usual, this condition may always be met by renaming the bound variables of the rule at the point of the inference. Note that if n is zero in Rule (19.4b), then there are no premises, and the indicated typing holds outright: if there are no rules, then the sequence may be considered to transform a value of type τ to a value of any type whatsoever, precisely because pattern matching will fail (as we shall see in the next section).

Finally, the typing rule for the `match` expression is given as follows:

$$\frac{\mathcal{X} \mid \Gamma \vdash e : \tau \quad \mathcal{X} \mid \Gamma \vdash rs : \tau > \tau'}{\mathcal{X} \mid \Gamma \vdash \texttt{match}(e;rs) : \tau'} \tag{19.5}$$

The match expression has type τ' if the rules transform any value of type τ, the type of the match expression, to a value of type τ'.

19.2 Pattern Matching

The dynamic semantics of pattern matching is defined using substitution to "guess" the bindings of the pattern variables. The dynamic semantics is given by the judgements $e \mapsto e'$, representing a step of computation, and e err, representing the checked condition of pattern matching failure.

$$\frac{e \mapsto e'}{\texttt{match(}e\texttt{;}rs\texttt{)} \mapsto \texttt{match(}e'\texttt{;}rs\texttt{)}} \tag{19.6a}$$

$$\frac{}{\texttt{match(}e\texttt{;rules[z]())}\ \mathsf{err}} \tag{19.6b}$$

$$\frac{e\ \mathsf{val} \quad [e_1,\ldots,e_k/x_1,\ldots,x_k]p_0 = e \quad [e_1,\ldots,e_k/x_1,\ldots,x_k]e_0 = e'}{\texttt{match(}e\texttt{;rules[s(}n\texttt{)](}x_1,\ldots,x_k\texttt{.rule(}p_0\texttt{;}e_0\texttt{);}rs\texttt{))} \mapsto e'} \tag{19.6c}$$

$$\frac{\begin{array}{c}\neg\exists e_1,\ldots,e_k.[e_1,\ldots,e_k/x_1,\ldots,x_k]p_0 = e \\ e\ \mathsf{val} \qquad \texttt{match(}e\texttt{;rules[}n\texttt{](}rs\texttt{))} \mapsto e'\end{array}}{\texttt{match(}e\texttt{;rules[s(}n\texttt{)](}x_1,\ldots,x_k\texttt{.rule(}p_0\texttt{;}e_0\texttt{);}rs\texttt{))} \mapsto e'} \tag{19.6d}$$

Rule (19.6b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (19.6c) specifies that the rules are to be considered in order. If the match value, e, matches the pattern, p_0 of the first rule, then the result is the corresponding instance of e_0; otherwise, the matching process continues with the remaining rules.

Theorem 19.1 (Preservation). *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof. By a straightforward induction on the derivation of $e \mapsto e'$, making use of the evident substitution lemma for the static semantics. □

While it is possible to state and prove a progress theorem, it does not have much force, because it is possible for matching to fail, a checked error. The static semantics given in Section 19.1 on page 148 does not guard against match failure, since it does not guarantee that every match value matches some rule.

We say that a sequence, rs, of rules such that $rs : \tau > \tau'$ is *exhaustive* iff for every $e : \tau$ such that e val there is some rule $p_i \Rightarrow e_i$ in rs such that e matches the pattern p_i (with respect to the variables $x_1,\ldots,x_k$). If a sequence of rules, rs, is not exhaustive, then there is a value, e, of the domain type such that $\texttt{match}\,e\,\{rs\}$ fails at run-time. Exhaustiveness can be checked statically by examining the patterns in the rules, but we shall not go into this here.

It is also useful to check that a sequence of rules is *irredundant*, meaning that no rule is completely subsumed by a preceding rule. We say that the ith rule in a sequence is *redundant* iff any value matching p_i must match the pattern p_j of some preceding rule in the sequence. Such a rule can never be considered in the dynamic semantics, because the rules are considered in order. It is also possible to ensure that all rule sequences are free of redundant rules, but we shall not go into this here.

19.3 Exercises

Part VI

Recursive Types

Chapter 20

*Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important classes of recursive types. Inductive types correspond to *least*, or *initial*, solutions of certain type isomorphism equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, inductive types are considered to be the "smallest" types containing their introduction forms; the elimination form is then a form of recursion over the introduction forms. Dually, coinductive types are considered to be the "greatest" types consistent with their elimination forms; the introduction forms are a means of presenting elements as required by the elimination forms.

The motivating example of an inductive type is the type of natural numbers. It is the least type containing the introductory forms z and s(e), where e is again an introductory form. To compute with a number we define a recursive procedure that returns a specified value on z, and, for s(e), returns a value defined in terms of the recursive call to itself on e. Other examples of inductive types are strings, lists, trees, and any other type that may be thought of as finitely generated from its introductory forms.

The motivating example of a coinductive type is the type of streams of natural numbers. Every stream may be thought of as being in the process of generation of pairs consisting of a natural number (its head) and another stream (its tail). To create a stream we define a generator that, when prompted, produces such a natural number and a co-recursive call to the generator. Other examples of coinductive types include infinite regular trees, and the so-called lazy natural numbers, which include a "point at infinity" consisting of an infinite stack of successors.

20.1 Static Semantics

20.1.1 Types and Operators

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over the class of types. The abstract syntax of inductive and coinductive types is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	t	t
		$\mid$	$\mathtt{ind}(t\,.\,\tau)$	$\mu_{\mathsf{i}}(t\,.\,\tau)$
		$\mid$	$\mathtt{coi}(t\,.\,\tau)$	$\mu_{\mathsf{f}}(t\,.\,\tau)$

The subscripts on the inductive and coinductive types are intended to indicate "initial" and "final", respectively, with the meaning that the inductive types determine least solutions to certain type equations, and the coinductive types determine greatest solutions.

We will consider *type formation* judgements of the form

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

where $t_1, \ldots, t_n$ are type names. We let Δ range over finite sets of hypotheses of the form t type, where t name is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \mid t\ \mathsf{type}} \tag{20.1a}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \tau\ \mathsf{type} \quad \Delta \mid t\,.\,\tau\ \mathsf{pos}}{\Delta \mid \mathtt{ind}(t\,.\,\tau)\ \mathsf{type}} \tag{20.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \tau\ \mathsf{type} \quad \Delta \mid t\,.\,\tau\ \mathsf{pos}}{\Delta \mid \mathtt{coi}(t\,.\,\tau)\ \mathsf{type}} \tag{20.2}$$

The premises on Rules (20.1b) and (20.2) involve a judgement of the form $t\,.\,\tau$ pos, which will be explained in Section 20.2 on the facing page.

A *type operator* is an abstractor of the form $t\,.\,\tau$ such that t $\mathsf{type} \mid \tau$ type. Thus a type operator may be thought of as a type, τ, with a distinguished free variable, t, possibly occurring in it. It follows from the meaning of the hypothetical judgement that if $t\,.\,\tau$ is a well-formed type operator, and σ type, then $[\sigma/t]\tau$ type. Thus, a type operator may also be thought of as a mapping from types to types given by substitution.

As an example of a type operator, consider the abstractor $t\,.\,\mathtt{unit} + t$, which will be used in the definition of the natural numbers as an inductive type. Other examples include $t\,.\,\mathtt{unit} + (\mathtt{nat} \times t)$, which underlies the definition of the inductive type of lists of natural numbers, and $t\,.\,\texttt{nat} \times t$, which underlies the coinductive type of streams of natural numbers.

20.1.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{in[}t\,.\,\tau\texttt{](}e\texttt{)}$	$\texttt{in(}e\texttt{)}$
		$\mid$	$\texttt{rec[}t\,.\,\tau\texttt{](}x\,.\,e;e'\texttt{)}$	$\texttt{rec(}x\,.\,e;e'\texttt{)}$
		$\mid$	$\texttt{out[}t\,.\,\tau\texttt{](}e\texttt{)}$	$\texttt{out(}e\texttt{)}$
		$\mid$	$\texttt{gen[}t\,.\,\tau\texttt{](}x\,.\,e;e'\texttt{)}$	$\texttt{gen(}x\,.\,e;e'\texttt{)}$

There is a pleasing symmetry between inductive and coinductive types that arises from the underlying duality of their semantics.

The static semantics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\texttt{ind(}t\,.\,\tau\texttt{)}/t]\tau}{\Gamma \vdash \texttt{in[}t\,.\,\tau\texttt{](}e\texttt{)} : \texttt{ind(}t\,.\,\tau\texttt{)}} \tag{20.3a}$$

$$\frac{\Gamma \vdash e' : \texttt{ind(}t\,.\,\tau\texttt{)} \quad \Gamma, x : [\rho/t]\tau \vdash e : \rho}{\Gamma \vdash \texttt{rec[}t\,.\,\tau\texttt{](}x\,.\,e;e'\texttt{)} : \rho} \tag{20.3b}$$

$$\frac{\Gamma \vdash e : \texttt{coi(}t\,.\,\tau\texttt{)}}{\Gamma \vdash \texttt{out[}t\,.\,\tau\texttt{](}e\texttt{)} : [\texttt{coi(}t\,.\,\tau\texttt{)}/t]\tau} \tag{20.3c}$$

$$\frac{\Gamma \vdash e' : \rho \quad \Gamma, x : \rho \vdash e : [\rho/t]\tau}{\Gamma \vdash \texttt{gen[}t\,.\,\tau\texttt{](}x\,.\,e;e'\texttt{)} : \texttt{coi(}t\,.\,\tau\texttt{)}} \tag{20.3d}$$

The dynamic semantics of these constructs is given in terms of the action of a positive type operator, which we now define.

20.2 Positive Type Operators

The formation of inductive and coinductive types is restricted to a special class of type operators, called the *(strictly) positive type operators*.[1] These are type operators of the form $t\,.\,\tau$ in which t is restricted so that its occurrences within τ do not lie within the domain of a function type. The prototypical

[1] A more permissive notion of *positive type operator* is sometimes considered, but we shall only have need of the strict notion.

example of a type operator that is *not* positive is the operator $t\,.\,t \to t$, in which t occurs in both the domain and the range of a function type. On the other hand, the type operator $t\,.\,\texttt{nat} \to t$ is positive, as is $t\,.\,u \to t$, where u type is some type variable other than t. Thus, the positivity restriction applies only to the bound type variable of the abstractor, and not to any other type variable.

The judgement $\Delta \mid t\,.\,\tau$ pos is inductively defined by the following rules:

$$\frac{}{\Delta \mid t\,.\,t \ \mathsf{pos}} \tag{20.4a}$$

$$\frac{\Delta \mid \tau \ \mathsf{type}}{\Delta \mid t\,.\,\tau \ \mathsf{pos}} \tag{20.4b}$$

$$\frac{\Delta \mid \tau_1 \ \mathsf{type} \quad \Delta \mid t\,.\,\tau_2 \ \mathsf{pos}}{\Delta \mid t\,.\,\tau_1 \to \tau_2 \ \mathsf{pos}} \tag{20.4c}$$

$$\frac{\Delta, u \ \mathsf{type} \mid t\,.\,\tau \ \mathsf{pos}}{\Delta \mid t\,.\,\mu_{\mathsf{i}}(u\,.\,\tau) \ \mathsf{pos}} \tag{20.4d}$$

$$\frac{\Delta, u \ \mathsf{type} \mid t\,.\,\tau \ \mathsf{pos}}{\Delta \mid t\,.\,\mu_{\mathsf{f}}(u\,.\,\tau) \ \mathsf{pos}} \tag{20.4e}$$

In the latter two rules we assume that $u \ \# \ t$, which is always achievable up to α-equivalence. Notice that in Rule (20.4c), the type variable t is not permitted to occur in τ_1, the domain type of the function type.

Positivity is preserved under substitution.

Lemma 20.1. *If $t\,.\,\sigma$ pos and $t\,.\,\tau$ pos, then $t\,.\,[\sigma/u]\tau$ pos.*

The reason to consider strictly positive type operators is that they admit a *covariant action* on types and abstractions. The action on types is given by substitution: $(t\,.\,\tau)^*(\sigma) := [\sigma/t]\tau$. The action on abstractions $x\,.\,e$, where $x : \sigma \vdash e : \sigma'$, is to transform a value of type $[\sigma/t]\tau$ into a value of type $[\sigma'/t]\tau$. This is achieved by replacing each value v of type σ at a position corresponding to an occurrences of t in τ by the expression $[v/x]e$ of type σ'. For example, if $t\,.\,\tau = t\,.\,\texttt{unit} + (\texttt{nat} \times t)$, then the action of $x\,.\,e$ is the abstractor $x'\,.\,e'$ such that

$$x' : \texttt{unit} + (\texttt{nat} \times \sigma) \vdash e' : \texttt{unit} + (\texttt{nat} \times \sigma').$$

On input $\texttt{in[l]}(\langle\rangle)$ the action yields $\texttt{in[l]}(\langle\rangle)$, and on input $\texttt{in[r]}(\langle e_1, e_2\rangle)$ it yields $\texttt{in[r]}(\langle e_1, [e_2/x]e\rangle)$. Observe that the action $t\,.\,\tau' = t\,.\,\texttt{unit} + (\sigma \times t)$ is the *same*, even though $[\sigma/t]\tau'$ has two occurrences of σ within it, because only one corresponds to an occurrence of t in τ'.

We define the action of a strictly positive type operator on an abstraction by the judgement

$$(t\,.\,\tau)^*(x\,.\,e) = x'\,.\,e',$$

where $x : \sigma \vdash e : \sigma'$, by the following rules:

$$\frac{}{(t\,.\,t)^*(x\,.\,e) = x\,.\,e}\tag{20.5a}$$

$$\frac{}{(t\,.\,\tau)^*(x\,.\,e) = x\,.\,x}\tag{20.5b}$$

$$\frac{(t\,.\,\tau_2)^*(x\,.\,e) = x_2\,.\,e_2}{(t\,.\,\tau_1 \rightarrow \tau_2)^*(x\,.\,e) = x'\,.\,\lambda(x_1{:}\tau_1.\,[x'(x_1)/x_2]e_2)}\tag{20.5c}$$

$$\frac{(t\,.\,[\mu_{\mathsf{i}}(u\,.\,[\sigma'/t]\tau)/u]\tau)^*(x\,.\,e) = x'\,.\,e'}{(t\,.\,\mu_{\mathsf{i}}(u\,.\,\tau))^*(x\,.\,e) = y\,.\,\mathtt{rec}(x'\,.\,\mathtt{in}(e');y)}\tag{20.5d}$$

$$\frac{(t\,.\,[\mu_{\mathsf{f}}(u\,.\,[\sigma/t]\tau)/u]\tau)^*(x\,.\,e) = x'\,.\,e'}{(t\,.\,\mu_{\mathsf{f}}(u\,.\,\tau))^*(x\,.\,e) = y\,.\,\mathtt{gen}(x'\,.\,[\mathtt{out}(x')/x']e';y)}\tag{20.5e}$$

The covariant action on abstractors is type-consistent with its action on types in the following sense.

Lemma 20.2. *If $x : \sigma \vdash e : \sigma'$, and $(t\,.\,\tau)^*(x\,.\,e) = x'\,.\,e'$, then $x' : (t\,.\,\tau)^*(\sigma) \vdash e' : (t\,.\,\tau)^*(\sigma')$.*

20.3 Dynamic Semantics

The dynamic semantics of inductive and coinductive types is given in terms of the covariant action of the associated type operator. Specifically, we take the following axioms as the primitive steps of our semantics:

$$\frac{(t\,.\,\tau)^*(x'\,.\,\mathtt{rec}(x\,.\,e;x')) = x''\,.\,e''}{\mathtt{rec}(x\,.\,e;\mathtt{in}(e')) \mapsto [[e'/x'']e''/x]e}\tag{20.6a}$$

$$\frac{(t\,.\,\tau)^*(x'\,.\,\mathtt{gen}(x\,.\,e;x')) = x''\,.\,e''}{\mathtt{out}(\mathtt{gen}(x\,.\,e;e')) \mapsto [[e'/x]e/x'']e''}\tag{20.6b}$$

Rule (20.6a) states that to evaluate the iterator on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value, and then perform the inductive step on the result. Rule (20.6b) is simply the dual of this rule for coinductive types.

The remaining rules of the dynamic semantics are specified as follows:

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{in}(e)\ \mathsf{val}}\tag{20.7a}$$

$$\frac{\{e' \ \mathsf{val}\}}{\mathtt{gen}(x\,.\,e;e')\ \mathsf{val}} \tag{20.7b}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in}(e) \mapsto \mathtt{in}(e')}\right\} \tag{20.7c}$$

$$\frac{e' \mapsto e''}{\mathtt{rec}(x\,.\,e;e') \mapsto \mathtt{rec}(x\,.\,e;e'')} \tag{20.7d}$$

$$\frac{e \mapsto e'}{\mathtt{out}(e) \mapsto \mathtt{out}(e')} \tag{20.7e}$$

$$\left\{\frac{e' \mapsto e''}{\mathtt{gen}(x\,.\,e;e') \mapsto \mathtt{gen}(x\,.\,e;e'')}\right\} \tag{20.7f}$$

As usual, the bracketed premises and rules are to be omitted for the lazy variant, and included for the eager variant.

Lemma 20.3. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Lemma 20.4. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Although we shall not give the proof here, the language $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\rightarrow}\}$ is terminating, and all functions defined within it are total.

Theorem 20.5. *If $e : \tau$ in $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\rightarrow}\}$, then there exists v val such that $e \mapsto^* v$.*

20.4 Fixed Point Properties

Inductive and coinductive types enjoy an important property that will play a prominent role in Chapter 21, called a *fixed point property*, that characterizes them as solutions to recursive type equations. Specifically, the inductive type $\mu_{\mathsf{i}}(t\,.\,\tau)$ is isomorphic to its unrolling,

$$\mu_{\mathsf{i}}(t\,.\,\tau) \cong [\mu_{\mathsf{i}}(t\,.\,\tau)/t]\tau,$$

and, dually, the coinductive type is isomorphic to its unrolling,

$$\mu_{\mathsf{f}}(t\,.\,\tau) \cong [\mu_{\mathsf{f}}(t\,.\,\tau)/t]\tau$$

The isomorphism arises from the invertibility of $\mathtt{in}(-)$ in the inductive case and of $\mathtt{out}(-)$ in the coinductive case, with the required inverses given as follows:

$$x\,.\,\mathtt{in}^{-1}_{t\,.\,\tau}(x) = x\,.\,\mathtt{rec}_{t\,.\,\tau}((t\,.\,\tau)^*(y\,.\,\mathtt{in}(y));x) \tag{20.8}$$

$$x\,.\,\mathtt{out}^{-1}_{t\,.\,\tau}(x) = x\,.\,\mathtt{gen}_{t\,.\,\tau}((t\,.\,\tau)^*(y\,.\,\mathtt{out}(y));x) \tag{20.9}$$

(We are not yet in a position to prove that these are, respectively, inverses to $\mathtt{in}(-)$ and $\mathtt{out}(-)$, but see Chapter 52 for more on equational reasoning.)

Thus, both the inductive and the coinductive type are solutions (in X) to the type isomorphism

$$X \cong (t\,.\,\tau)^*(X) = [X/t]\tau.$$

What distinguishes the two solutions, in general, is that the inductive type is the *initial*, or *least*, solution, whereas the coinductive type is the *final*, or *greatest*, solution to the isomorphism equation. This implies, in particular, that there is an abstractor $x\,.\,e$ such that

$$x : \mu_{\mathsf{i}}(t\,.\,\tau) \vdash e : \mu_{\mathsf{f}}(t\,.\,\tau).$$

In general there is not an abstractor mapping in the opposite direction.

For the sake of comparison, let $\mathtt{nat}_{\mathsf{i}}$ be the type of inductive natural numbers, $\mu_{\mathsf{i}}(t\,.\,\mathtt{unit}+t)$, and let $\mathtt{nat}_{\mathsf{f}}$ be the type of coinductive natural numbers, $\mu_{\mathsf{f}}(t\,.\,\mathtt{unit}+t)$. Intuitively, $\mathtt{nat}_{\mathsf{i}}$ is the smallest (most restrictive) type containing zero, which is represented by

$$\mathtt{in}(\mathtt{in}[\mathtt{l}](\langle\rangle)),$$

and, if e is of type $\mathtt{nat}_{\mathsf{i}}$, its successor, which is represented by

$$\mathtt{in}(\mathtt{in}[\mathtt{r}](e)).$$

Dually, $\mathtt{nat}_{\mathsf{f}}$ is the largest (most permissive) type of expressions e such that $\mathtt{out}(e)$ is either equivalent to zero, which is represented by $\mathtt{in}[\mathtt{l}](\langle\rangle)$, or to the successor of some expression $e' : \mathtt{nat}_{\mathsf{f}}$, which is represented by $\mathtt{in}[\mathtt{r}](e')$.

It is not hard to embed the inductive natural numbers into the coinductive natural numbers, but the converse is impossible. However, the expression

$$\omega = \mathtt{gen}(x\,.\,\mathtt{in}[\mathtt{r}](x);\langle\rangle)$$

is a coinductive natural number that is greater than the embedding of all inductive natural numbers. This is because

$$\mathtt{out}(\omega) \mapsto^* \mathtt{in}[\mathtt{r}](\omega),$$

and hence is essentially an infinite composition of successors. Any embedding of the coinductive into the inductive natural numbers would place ω among the finite natural numbers, making it larger than some and smaller than others, which is impossible.

20.5 Exercises

1. Extend the covariant action to nullary and binary products and sums.

2. Prove progress and preservation.

3. Show that the required abstractor mapping the inductive to the coinductive type associated with a type operator is given by the equation

   $$x\,.\,\mathtt{gen}(y\,.\,\mathtt{in}^{-1}_{t\,.\,\tau}(y)\,;\,x).$$

   Characterize the behavior of this term when x is replaced by an element of the inductive type.

Chapter 21

Recursive Types

Recursive types are solutions to systems of type equations, much as recursive functions are solutions to systems of recursion equations. For example, the type of lists of natural numbers may be thought of as a solution to the type equation $t \cong \texttt{unit} + (\texttt{nat} \times t)$, and the type of binary trees may be thought of as a solution to the type equation $t \cong \texttt{unit} + (t \times t)$. The solution to a system of type equations is determined *up to isomorphism*, which means that there is a mutually inverse pair of functions between the two sides of the equation. Thus, a solution, τ, to the type equation for lists consists of two mutually inverse functions

$$\mathit{fold} : \texttt{unit} + (\texttt{nat} \times \tau) \rightarrow \tau$$

and

$$\mathit{unfold} : \tau \rightarrow \texttt{unit} + (\texttt{nat} \times \tau)$$

that witness the relationship between the solution and its defining condition(s).

Just as the solution to a system of recursion equations may be seen as a fixed point of an associated functional, the solution to a system of type equations is a fixed point (up to isomorphism) of an associated type operator. For example, the type operator associated to the defining equation for lists is

$$\phi_{\mathit{list}}(t) = \texttt{unit} + (\texttt{nat} \times t).$$

The type, τ, we seek is an isomorphism $\tau \cong \phi_{\mathit{list}}(\tau)$. Similarly, the type operator associated with the type of binary trees is

$$\phi_{\mathit{tree}}(t) = \texttt{unit} + (t \times t),$$

and we once again seek a solution $\tau \cong \phi_{tree}(\tau)$. In each case we seek a *fixed point* of a type operator, $\mathit{fix}(\phi_{list})$ in the case of lists, and $\mathit{fix}(\phi_{tree})$ in the case of binary trees.

A central problem in the theory of recursive types is to classify which operators admit fixed points, or, in other words, which systems of type equations have solutions? The short answer, which we shall not justify here, is that we may solve *all* type equations involving product, sum, and *partial* function types. The restriction to partial functions is essential, for a solution to an equation such as $t = t \rightarrow \texttt{bool}$ is, at the very least, suspicious since it would establish an isomorphism between a type and its "power type" (*i.e.*, the type of its characteristic functions).

Recursive types have numerous uses in programming languages, including these:

1. Representing data structures, such as lists and trees, of unbounded size.

2. Representing circular data structures, such as cyclic graphs.

3. Defining recursive functions and self-referential objects.

4. Supporting dynamic typing and dynamic type dispatch.

5. Supporting co-routines and similar control-flow constructs.

In this chapter we will study recursive types in their own right, and later use them to model more sophisticated language features.

21.1 Solving Type Equations

A *recursive type* has the form $\mu t\,.\,\tau$, where $t\,.\,\tau$ is any type operator, without restriction. It denotes the fixed point (up to isomorphism) of the given type operator, and hence provides a solution to the isomorphism equation $t \cong \tau$. The isomorphism is witnessed by the terms $\texttt{fold(}e\texttt{)}$ and $\texttt{unfold(}e\texttt{)}$ that mediate between the recursive type, $\mu t\,.\,\tau$, and its unfolding, $[\mu t\,.\,\tau/t]\tau$. In this sense the parameter, t, of the type operator is *self-referential* in that it may be considered to stand for the recursive type itself.

Recursive types are formalized by the following abstract syntax:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	t	t
		$\mid$	$\texttt{rec}(t.\tau)$	$\mu t.\tau$
Expr	e	$::=$	$\texttt{fold}[t.\tau](e)$	$\texttt{fold}(e)$
		$\mid$	$\texttt{unfold}(e)$	$\texttt{unfold}(e)$

The meta-variable t ranges over a class of *type names*, which serve as names for types. The *unfolding* of $\texttt{rec}(t.\tau)$ is the type $[\texttt{rec}(t.\tau)/t]\tau$ obtained by substituting the recursive type for t in τ.

The introduction form, $\texttt{fold}[t.\tau](e)$, introduces a value of recursive type in terms of an element of its unfolding, and the elimination form, $\texttt{unfold}(e)$, evaluates to a value of the unfolding from an element of the recursive type. In implementation terms the operation $\texttt{fold}[t.\tau](e)$ may be thought of as an abstract "pointer" to a value of the unfolded type, and the operation $\texttt{unfold}(e)$ "chases" the pointer to obtain that value from a value of the corresponding folded type.

The static semantics of $\mathcal{L}\{\mu\rightharpoonup\}$ consists of two forms of judgement, τ type, and $e : \tau$. The type formation judgement is inductively defined by a set of rules for deriving general judgements of the form

$$\Delta \mid \tau \ \mathsf{type},$$

where Δ is a finite set of assumptions of the form t_i type for some type variable t_i.

$$\frac{}{\Delta, t \ \mathsf{type} \mid t \ \mathsf{type}} \tag{21.1a}$$

$$\frac{\Delta \mid \tau_1 \ \mathsf{type} \quad \Delta \mid \tau_2 \ \mathsf{type}}{\Delta \mid \texttt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{21.1b}$$

$$\frac{\Delta, t \ \mathsf{type} \mid \tau \ \mathsf{type}}{\Delta \mid \texttt{rec}(t.\tau) \ \mathsf{type}} \tag{21.1c}$$

Note that, in contrast to Chapter 20, there is no positivity restriction on the formation of a recursive type.

Typing judgements have the form

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

where τ type and Γ consists of hypotheses of the form $x_i : \tau_i$ such that τ_i type for each $1 \leq i \leq n$. The typing rules for $\mathcal{L}\{\mu\rightharpoonup\}$ are as follows:

$$\frac{\Gamma \vdash e : [\texttt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{fold}[t.\tau](e) : \texttt{rec}(t.\tau)} \tag{21.2a}$$

$$\frac{\Gamma \vdash e : \texttt{rec(}t\texttt{.}\tau\texttt{)}}{\Gamma \vdash \texttt{unfold(}e\texttt{)} : [\texttt{rec(}t\texttt{.}\tau\texttt{)}/t]\tau} \tag{21.2b}$$

These rules express an inverse relationship stating that a recursive type is *isomorphic* to its unfolding, with the operations fold and unfold being the witnesses to the isomorphism.

Operationally, this is expressed by the following dynamic semantics rules:

$$\frac{\{e\ \mathsf{val}\}}{\texttt{fold[}t\texttt{.}\tau\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{21.3a}$$

$$\left\{\frac{e \mapsto e'}{\texttt{fold[}t\texttt{.}\tau\texttt{](}e\texttt{)} \mapsto \texttt{fold[}t\texttt{.}\tau\texttt{](}e'\texttt{)}}\right\} \tag{21.3b}$$

$$\frac{e \mapsto e'}{\texttt{unfold(}e\texttt{)} \mapsto \texttt{unfold(}e'\texttt{)}} \tag{21.3c}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{unfold(fold[}t\texttt{.}\tau\texttt{](}e\texttt{))} \mapsto e} \tag{21.3d}$$

As usual, the bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics.

It is straightforward to prove the safety of $\mathcal{L}\{\mu\rightharpoonup\}$.

Theorem 21.1 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

21.2 Recursive Data Structures

One important application of recursive types is to the representation of data structures such as lists and trees whose size and content is determined during the course of execution of a program.

One example is the type of natural numbers, which we have taken as primitive in Chapter 16. We may instead treat nat as a recursive type by thinking of it as a solution (up to isomorphism) of the type equation $t \cong 1 + t$, which is to say that every natural number is either zero or the successor of another natural number. More formally, we may define nat to be the recursive type

$$\mu t\texttt{.}\,\texttt{[z:unit,s:}t\texttt{]}, \tag{21.4}$$

which specifies that

$$\texttt{nat} \cong \texttt{[z:unit,s:nat]}.$$

The zero and successor operations are correspondingly defined by the following equations:

$$\begin{aligned} \mathtt{z} &= \mathtt{fold(in[z]}(\langle\rangle)) \\ \mathtt{s}(e) &= \mathtt{fold(in[s]}(e)). \end{aligned}$$

The conditional branch on zero is defined by the following equation:

$$\begin{aligned} &\mathtt{ifz}\,e\,\{\mathtt{z}{\Rightarrow}e_0 \mid \mathtt{s}(x){\Rightarrow}e_1\} = \\ &\qquad\qquad \mathtt{case\,unfold}(e)\,\{\mathtt{in[z]}(\_) \Rightarrow e_0 \mid \mathtt{in[s]}(x) \Rightarrow e_1\}, \end{aligned}$$

where the "underscore" indicates a variable that lies apart from e_0. It is easy to check that these definitions exhibit the expected behavior in that they correctly simulate the dynamic semantics given in Chapter 16.

As another example, the type `nat list` of lists of natural numbers may be represented by the recursive type

$$\mu t\,.\,[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times t]$$

so that we have the isomorphism

$$\mathtt{nat\,list} \cong [\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times\mathtt{nat\,list}].$$

The list formation operations are represented by the following equations:

$$\begin{aligned} \mathtt{nil} &= \mathtt{fold(in[n]}(\langle\rangle)) \\ \mathtt{cons}(e_1;e_2) &= \mathtt{fold(in[c]}(\langle e_1,e_2\rangle)). \end{aligned}$$

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned} &\mathtt{listcase}\,e\,\{\mathtt{nil} \Rightarrow e_0 \mid \mathtt{cons}(x;y) \Rightarrow e_1\} = \\ &\qquad\qquad \mathtt{case\,unfold}(e)\,\{\mathtt{in[n]}(\_) \Rightarrow e_0, \mid \mathtt{in[c]}(\langle x,y\rangle) \Rightarrow e_1\}, \end{aligned}$$

where we have used an underscore for a "don't care" variable, and used pattern-matching syntax to bind the components of a pair.

There is a natural correspondence between this representation of lists and the conventional "blackboard notation" for linked lists. We may think of `fold` as an abstract heap-allocated pointer to a tagged cell consisting of either (a) the tag `n` with no associated data, or (b) the tag `c` attached to a pair consisting of a natural number and another list, which must be an abstract pointer of the same sort.

21.3 Self-Reference

In Chapters 16 and 17 we used self-reference to implement recursion. In each case self-reference is achieved by introducing a bound variable that stands for the expression itself during its evaluation. In the case of recursive functions, the semantics of application is responsible for ensuring that the self-referential variable is replaced by the function itself before it is applied to an argument. Similarly, in the case of objects, method selection ensures that the self-referential variable is replaced by the object itself when a method is selected. General recursion, on the other hand, takes care of itself in that it unfolds the recursion implicitly whenever it is evaluated. Though the details differ in each case, they are all based on the same basic idea of *self-application*. When a self-referential expression is used, all references to "itself" are replaced by the expression itself in order to "tie the knot" of recursion. In this section we will isolate the general concept of self-reference as an application of recursive types.

The goal is to define a type of self-referential expressions of type τ. To implement self-reference, such an expression will be represented as a function that will be applied to the self-referential expression itself to effect self-reference. Thus the type we seek must have the form $t \rightharpoonup \tau$ for some type t, where the domain represents the implicit self-referential argument. This means that t must be the very type in question, which is to say that we seek a solution to the type equation

$$t \cong t \rightharpoonup \tau.$$

The recursive type

$$\tau\ \texttt{self} = \mu t\,.\,t \rightharpoonup \tau,$$

is, by definition, a solution to this type equation. It establishes the isomorphism

$$\tau\ \texttt{self} \cong \tau\ \texttt{self} \rightharpoonup \tau,$$

capturing the self-referential natural of values of this type.

The introductory form for the type $\tau\ \texttt{self}$ is the self-referential expression $\texttt{self}\ y\ \texttt{is}\ e$, where $y : \tau\ \texttt{self} \vdash e : \tau$. It is defined to be the expression

$$\texttt{fold}(\lambda(y{:}\tau\ \texttt{self}.\,e)),$$

which we note is a value, regardless of whether `fold` is eager or lazy. According to this definition, the following typing rule is derivable:

$$\frac{\Gamma, y : \tau\ \texttt{self} \vdash e : \tau}{\Gamma \vdash \texttt{self}\ y\ \texttt{is}\ e : \tau\ \texttt{self}}\ .$$

This expression has the property that

$$\texttt{unfold(self}\,y\,\texttt{is}\,e\texttt{)(self}\,y\,\texttt{is}\,e\texttt{)} \mapsto [\texttt{self}\,y\,\texttt{is}\,e/y]e.$$

Therefore let us define the eliminatory form, $\texttt{unroll(}e\texttt{)}$, where $e : \tau\ \texttt{self}$, to be the expression

$$\texttt{unfold(}e\texttt{)(}e\texttt{)}.$$

This definition gives rise to the dynamic semantics

$$\texttt{unroll(self}\,y\,\texttt{is}\,e\texttt{)} \mapsto^* [\texttt{self}\,y\,\texttt{is}\,e/y]e,$$

which implements self-reference by self-application.

We may use self-reference to represent basic objects (without method override, extension, or deletion) self-referential records:

$$\langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle := \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\ \texttt{self}.$$

Correspondingly, we use self-reference to model mutual recursion among methods:

$$\langle\!\langle l_1 = x_1 \,.\, e_1, \ldots, l_n = x_n \,.\, e_n \rangle\!\rangle := \texttt{self}\,x\,\texttt{is}\,\langle l_1 = e_1', \ldots, l_n = e_n' \rangle,$$

where for each $1 \leq i \leq n$ we define $e_i' = [x/x_i]e_i$. Method selection unfolds the self-reference and selects the appropriate record field:

$$e \cdot l_i := \texttt{unroll(}e\texttt{)} \cdot l_i.$$

It is a good exercise to check that the static and dynamic semantics of objects are derivable from these definitions.

We may also use self-reference to implement general recursion as defined in Chapter 16. Specifically, we may define $\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e$ to be the expression

$$\texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)}.$$

Observe that this expression has type τ, given that $x : \tau \vdash e : \tau$. The dynamic semantics of general recursion is derivable from this definition:

$$\begin{aligned} \texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e &= \texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)} \\ &\mapsto^* [\texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)}/x]e \\ &= [\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e/x]e. \end{aligned}$$

This construction shows that in the presence of recursive types we have general recursion at every type. Consequently, adding recursive types has a *global* effect on the language, in contrast to adding, say, list types, which do not affect the meanings of any other types. In particular, in the presence of recursive types, there is a non-terminating expression of *every* type. Consequently, function types are inherently partial; there is no analogue of System **T** with recursive types. Put in other terms, recursive types are a *non-conservative extension* of a language with total function types, because they disrupt the meaning of types that were present before the extension, as well as adding new types that were not present beforehand.

21.4 Exercises

Part VII

Dynamic Types

Chapter 22

Untyped Languages

It is customary to distinguish between *typed* and *untyped* languages, as though they were incompatible alternatives. But we shall argue that well-defined, or *safe*, type-free languages are, in fact, just a particular mode of use of types.

22.1 Untyped λ-Calculus

The premier example of an untyped programming language is the language $\mathcal{L}\{\lambda\}$, or the *untyped λ-calculus*, an elegant formalism devised by Alonzo Church in the 1930's. Its chief characteristic is that it consists of nothing but functions; there are no other forms of data in the language! 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.

The abstract syntax of $\mathcal{L}\{\lambda\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Term	u	$::=$	x	x
		$\mid$	$\lambda(x.u)$	$\lambda x.u$
		$\mid$	$\mathtt{uap}(u_1;u_2)$	$u_1(e_2)$

The second form of expression is called a *λ-abstraction*, and the third is called *application*.

The static semantics of $\mathcal{L}\{\lambda\}$ is defined by hypothetical judgements of the form $x_1\ \mathsf{ok},\ldots,x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression involving the variables x_1, …, x_n. This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x \textsf{ ok} \vdash x \textsf{ ok}} \tag{22.1a}$$

$$\frac{\Gamma \vdash u_1 \textsf{ ok} \quad \Gamma \vdash u_2 \textsf{ ok}}{\Gamma \vdash \texttt{uap}(u_1; u_2) \textsf{ ok}} \tag{22.1b}$$

$$\frac{\Gamma, x \textsf{ ok} \vdash u \textsf{ ok}}{\Gamma \vdash \lambda(x.u) \textsf{ ok}} \tag{22.1c}$$

The dynamic semantics is given by the following transition rules:

$$\frac{}{\texttt{uap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{22.2a}$$

$$\frac{u_1 \mapsto u_1'}{\texttt{uap}(u_1; u_2) \mapsto \texttt{uap}(u_1'; u_2)} \tag{22.2b}$$

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.

The dynamic semantics induces a notion of observational equivalence, written $u_1 \cong u_2$ $[\Gamma]$, stating that u_1 and u_2 are indistinguishable in the following sense. First, we define the *Church booleans*, $\mathbf{tt} = \lambda x.\,\lambda y.\,x$ and $\mathbf{ff} = \lambda x.\,\lambda y.\,y$, which satisfy $\mathbf{tt}(u_1)(u_2) \mapsto^* u_1$ and $\mathbf{ff}(u_1)(u_2) \mapsto^* u_2$. We then define Kleene equivalence, $u_1 \simeq u_2$, of two closed expressions to mean that $u_1 \mapsto^* \mathbf{tt}$ iff $u_2 \mapsto^* \mathbf{tt}$ and $u_1 \mapsto^* \mathbf{ff}$ iff $u_2 \mapsto^* \mathbf{ff}$. Finally, two terms are observationally equivalent iff whenever they are embedded in a larger expression computing a Church boolean, they either both evaluate to **tt** or to **ff**, or both diverge.

The only properties of observational equivalence that we shall need are these:

1. It is consistent: $\mathbf{tt} \not\cong \mathbf{ff}$.

2. It is a congruence: we may replace equals by equals to obtain equals.

3. It contains β-equivalence: $(\lambda x.\,u_2)(u_1) \cong [u_1/x]u_2$ $[\Gamma]$.

22.2 Definability

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. Church's Law 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 Church's Law 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. As the name implies, Church's Law is a *scientific law* in 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.[1]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 16. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. This means, in particular, that we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} = \lambda b.\, \lambda s.\, b \tag{22.3a}$$

$$\overline{n+1} = \lambda b.\, \lambda s.\, s(\overline{n}(b)(s)) \tag{22.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \cong u_2(\ldots(u_2(u_1))),$$

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 the basic functions of arithmetic. For example, successor, addition, and multiplication are de-

[1] Unfortunately, it is common in Computer Science to put forth as "laws" assertions that are not scientific laws at all. For example, Moore's Law is merely an observation about a near-term trend in microprocessor fabrication that is certainly not valid over the long term, and Amdahl's Law is but a simple truth of arithmetic. Worse, Church's Law, which is a true scientific law, is usually called *Church's Thesis*, which, to the author's ear, suggests something less than the full force of a scientific law.

fined by the following untyped λ-terms:

$$\begin{aligned} \texttt{succ} &= \lambda x.\,\lambda b.\,\lambda s.\,s\texttt{(}x\texttt{(}b\texttt{)(}s\texttt{))} && (22.4)\\ \texttt{plus} &= \lambda x.\,\lambda y.\,y\texttt{(}x\texttt{)(succ)} && (22.5)\\ \texttt{times} &= \lambda x.\,\lambda y.\,y\texttt{(}\overline{0}\texttt{)((plus(}x\texttt{)))} && (22.6) \end{aligned}$$

It is easy to check that $\texttt{succ(}\overline{n}\texttt{)} \cong \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

We may readily define $\texttt{ifz(}u\texttt{;}u_0\texttt{;}u_1\texttt{)}$ to be the application $u\texttt{(}u_0\texttt{)((}\lambda x.\,u_1\texttt{))}$, where x is chosen arbitrarily such that $x \mathrel{\#} u_1$. We can use this to define $\texttt{ifz(}u\texttt{;}u_0\texttt{;}x\texttt{.}u_1\texttt{)}$, provided that we can compute the predecessor of a natural number. Doing so requires a bit of ingenuity. We wish to find a term pred such that

$$\begin{aligned} \texttt{pred(}\overline{0}\texttt{)} &\cong \overline{0} && (22.7)\\ \texttt{pred(}\overline{n+1}\texttt{)} &\cong \overline{n}. && (22.8) \end{aligned}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ as a function of its value for $\overline{n}$. At first glance this seems straightforward—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the nth iteration the registers contain the predecessor of n and n itself, respectively. Given the result for n, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of n we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\begin{aligned} \langle u_1, u_2\rangle &= \lambda f.\,f\texttt{(}u_1\texttt{)(}u_2\texttt{)} && (22.9)\\ \texttt{fst(}u\texttt{)} &= u\texttt{((}\lambda x.\,\lambda y.\,x\texttt{))} && (22.10)\\ \texttt{snd(}u\texttt{)} &= u\texttt{((}\lambda x.\,\lambda y.\,y\texttt{))} && (22.11) \end{aligned}$$

It is easy to check that under this encoding $\texttt{fst(}\langle u_1, u_2\rangle\texttt{)} \cong u_1$, and similarly for the second projection. We may now define the required term u

representing the predecessor:

$$u'_p = \lambda x.\, x(\langle\overline{0},\overline{0}\rangle)(\lambda y.\, \langle \mathtt{snd}(y), \mathtt{s}(\mathtt{snd}(y))\rangle) \tag{22.12}$$

$$u_p = \lambda x.\, \mathtt{fst}(u(x)) \tag{22.13}$$

It is then easy to check that this gives us the required behavior.

This gives us all the apparatus of PCF, apart from general recursion. But this is also definable using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the *Y combinator*:

$$\mathbf{Y} = \lambda F.\, (\lambda f.\, F(f(f)))(\lambda f.\, F(f(f))). \tag{22.14}$$

Observe that

$$\mathbf{Y}(F) \cong F(\mathbf{Y}(F)).$$

For this reason, the term Y is called a *fixed point combinator*.

22.3 Untyped Means Uni-Typed

The untyped λ-calculus may be *faithfully embedded* in the typed language $\mathcal{L}\{\mu{\rightharpoonup}\}$, enriched with recursive types. This means that every untyped λ-term has a representation as an expression in $\mathcal{L}\{\mu{\rightharpoonup}\}$ 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 $\mathcal{L}\{\mu{\rightharpoonup}\}$, 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 $\mathcal{L}\{\mu{\rightharpoonup}\}$ (which we could surely do), but rather a direct representation of untyped λ-terms as certain typed expressions of $\mathcal{L}\{\mu{\rightharpoonup}\}$. It is for this reason that we say that untyped languages are just a special case of typed languages, provided that we have recursive types at our disposal.

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-calculus! It is not the *absence* of types that gives it its power, but rather that it has *only one* type, namely the recursive type

$$D = \mathtt{rec}(t.\mathtt{arr}(t;t)).$$

A value of type D is of the form $\mathtt{fold}[D](e)$ where e is a value of type $\mathtt{arr}(D;D)$ — 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”. As usual, a recursive type may be seen as a solution to a type isomorphism equation, which in the present case is the equation

$$D \cong \mathtt{arr}(D;D).$$

This specifies that D is a type that is isomorphic to the space of functions on D itself, something that is impossible in conventional set theory, but is feasible in the computationally-based setting of the λ-calculus.

This isomorphism leads to the following embedding, $u^\dagger$, of u into $\mathcal{L}\{\mu\rightharpoonup\}$:

$$x^\dagger = x \tag{22.15a}$$

$$\lambda\,\mathtt{(}x\,\mathtt{.}\,u\mathtt{)}^\dagger = \mathtt{fold[}D\mathtt{](lam[}D\mathtt{](}x\,\mathtt{.}\,u^\dagger\mathtt{))} \tag{22.15b}$$

$$\mathtt{uap(}u_1\mathtt{;}\,u_2\mathtt{)}^\dagger = \mathtt{ap(unfold(}u_1^\dagger\mathtt{);}\,u_2^\dagger\mathtt{)} \tag{22.15c}$$

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,

$$\begin{aligned}
\mathtt{uap(}\lambda\mathtt{(}x\,\mathtt{.}\,u_1\mathtt{);}\,u_2\mathtt{)}^\dagger &= \mathtt{ap(unfold(fold[}D\mathtt{](lam[}D\mathtt{](}x\,\mathtt{.}\,u_1^\dagger\mathtt{)));}\,u_2^\dagger\mathtt{)}\\
&\cong \mathtt{ap(lam[}D\mathtt{](}x\,\mathtt{.}\,u_1^\dagger\mathtt{);}\,u_2^\dagger\mathtt{)}\\
&\cong [u_2^\dagger/x]u_1^\dagger\\
&= ([u_2/x]u_1)^\dagger.
\end{aligned}$$

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 $u_1^\dagger \cong v_1^\dagger$, then $\mathtt{uap(}u_1\mathtt{;}\,u_2\mathtt{)}^\dagger \cong \mathtt{uap(}v_1\mathtt{;}\,u_2\mathtt{)}^\dagger$.

Thus we see that the canonical *untyped* language, $\mathcal{L}\{\lambda\}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a *special case* of these! Rather than eliminating types, an untyped language suppresses them by consolidation into a single recursive type. In Chapter 23 we will take this observation a step further and show that so-called dynamically typed languages, which admit multiple types of values but defer type checking until run-time, are also but modes of use of static typing.

22.4 Exercises

Chapter 23

Dynamic Typing

We saw in Chapter 22 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped λ-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. Obviously this property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped λ-calculus (*i.e.*, rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number.

One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be *dynamically typed*, in supposed opposition to the *statically typed* languages we have studied thus far. In this chapter we show that the supposed opposition between static and dynamic languages is fallacious: dynamic typing is but a mode of use of static typing, and, moreover, it is profitably seen as such. Dynamic typing can hardly be in opposition to that of which it is a special case!

23.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{dyn\}$. The abstract syntax of $\mathcal{L}\{dyn\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	d	$::=$	x	x
		$\mid$	$\mathtt{num}(\overline{n})$	$\overline{n}$
		$\mid$	$\mathtt{s}(d)$	$\mathtt{s}(d)$
		$\mid$	$\mathtt{ifz}(d;d_0;x.d_1)$	$\mathtt{ifz}\,d\,\{\mathtt{z}\Rightarrow d_0 \mid \mathtt{s}(x)\Rightarrow d_1\}$
		$\mid$	$\mathtt{fun}(\lambda(x.d))$	$\lambda x.\,d$
		$\mid$	$\mathtt{dap}(d_1;d_2)$	$d_1(d_2)$
		$\mid$	$\mathtt{fix}(x.d)$	$\mathtt{fix}\,x\,\mathtt{is}\,d$

The syntax is similar to that of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, the chief difference being that each value is labelled with its *class*, either num or fun. Numerals are labelled with the class num to mark them as numbers. The successor operation is now an *elimination* form acting on values of class num, rather than an *introduction* form for numbers. Untyped λ-abstractions are explicitly labelled with the class fun to mark them as functions.

Apart from formatting, the concrete syntax avoids mentioning the classes attached to values of the language. This means that they must be inserted by the parser on passage from concrete to abstract syntax. Unfortunately this invites the misapprehension that the expressions of $\mathcal{L}\{dyn\}$ are the same as those of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, but without the types. This is not the case! The classes must be present in order to define the dynamic semantics of $\mathcal{L}\{dyn\}$, but are entirely unnecessary for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

The static semantics of $\mathcal{L}\{dyn\}$ is essentially the same as for $\mathcal{L}\{\lambda\}$ given in Chapter 22; it merely checks that there are no free variables in the expression. The judgement

$$x_1\ \mathsf{ok}, \ldots x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

states that d is a well-formed expression with free variables among those in the hypothesis list.

The dynamic semantics for $\mathcal{L}\{dyn\}$ checks for errors that would never arise in a safe statically typed language. For example, function application must ensure that its first argument is a function, signaling an error in the case that it is not, and similarly the case analysis construct must ensure that its first argument is a number, signaling an error if not. The reason for having classes labelling values is precisely to make this run-time check possible. One could argue that the required check may be made by inspection of the unlabelled value itself, but this is unrealistic. At run-time both numbers and functions might be represented by machine words, the former a two's complement number, the latter an address in memory. But given an arbitrary word, one cannot determine whether it is a number or an address!

The value judgement, d val, states that d is a fully evaluated (closed) expression:

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{val}} \tag{23.1a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{val}} \tag{23.1b}$$

The dynamic semantics makes use of judgements that check the class of a value, and recover the underlying λ-abstraction in the case of a function.

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{23.2a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{is\_fun}\ \lambda(x.d)} \tag{23.2b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\texttt{num}(\_)\ \mathsf{isnt\_fun}} \tag{23.3a}$$

$$\frac{}{\texttt{fun}(\_)\ \mathsf{isnt\_num}} \tag{23.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, d err, are defined simultaneously by the following rules.

$$\frac{d \mapsto d'}{\texttt{s}(d) \mapsto \texttt{s}(d')} \tag{23.4a}$$

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\texttt{s}(d) \mapsto \texttt{num}(\texttt{s}(\overline{n}))} \tag{23.4b}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{s}(d)\ \mathsf{err}} \tag{23.4c}$$

$$\frac{d \mapsto d'}{\texttt{ifz}(d;d_0;x.d_1) \mapsto \texttt{ifz}(d';d_0;x.d_1)} \tag{23.4d}$$

$$\frac{d\ \mathsf{is\_num}\ \mathsf{z}}{\texttt{ifz}(d;d_0;x.d_1) \mapsto d_0} \tag{23.4e}$$

$$\frac{d\ \mathsf{is\_num}\ \mathsf{s}(\overline{n})}{\texttt{ifz}(d;d_0;x.d_1) \mapsto [\texttt{num}(\overline{n})/x]d_1} \tag{23.4f}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{ifz}(d;d_0;x.d_1)\ \mathsf{err}} \tag{23.4g}$$

$$\frac{d_1 \mapsto d_1'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1';d_2)} \tag{23.4h}$$

$$\frac{d_1\ \mathsf{val} \quad d_2 \mapsto d_2'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1;d_2')} \tag{23.4i}$$

$$\frac{d_1\ \mathsf{is\_fun}\ \lambda(x.d) \quad d_2\ \mathsf{val}}{\mathtt{dap}(d_1;d_2) \mapsto [d_2/x]d} \tag{23.4j}$$

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{dap}(d_1;d_2)\ \mathsf{err}} \tag{23.4k}$$

$$\frac{}{\mathtt{fix}(x.d) \mapsto [\mathtt{fix}(x.d)/x]d} \tag{23.4l}$$

Note that in Rule (23.4f) the labelled numeral $\mathtt{num}(\overline{n})$ is bound to x to maintain the invariant that variables are bound to forms of expression.

The language $\mathcal{L}\{dyn\}$ enjoys essentially the same safety properties as $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, except that there are more opportunities for errors to arise at run-time.

Theorem 23.1. *If d ok, then either d val, or d err, or there exists d' such that $d \mapsto d'$.*

Proof. By rule induction on Rules (23.4). The rules are designed so that if d ok, then some rule, possibly an error rule, applies, ensuring progress. Since well-formedness is closed under substitution, the result of a transition is always well-formed. □

This result is often promoted as an *advantage* of dynamic over static typing. Unlike static languages, essentially every piece of abstract syntax (apart from those with unbound variables) has a well-defined dynamic semantics. But this can also be seen as a *disadvantage*: errors that would be ruled out at compile time in a static language are not signalled until run time in a dynamic language.

23.2 Critique of Dynamic Typing

The dynamic semantics of $\mathcal{L}\{dyn\}$ exhibits considerable run-time overhead compared to that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. Suppose that we define addition by the $\mathcal{L}\{dyn\}$ expression

$$\lambda x.\,(\mathtt{fix}\,p\,\mathtt{is}\,\lambda y.\,\mathtt{ifz}\,y\,\{\mathtt{z}\Rightarrow x \mid \mathtt{s}(y')\Rightarrow\mathtt{s}(p(y'))\}).$$

By carefully examining the dynamics semantics, we may observe some of the hidden costs of dynamic typing.

First, observe that the body of the fixed point expression is a λ-abstraction, which is labelled by the parser with class `fun`. The semantics of the fixed point construct binds p to this (labelled) λ-abstraction, so the dynamic class check incurred by the recursive call is guaranteed to succeed. The check is redundant, but there is no way to avoid it.

Second, the result of applying the inner λ-abstraction is either x, the parameter of the outer λ-abstraction, or the result of a recursive call. The semantics of the successor operation ensures that the result of the recursive call is labelled with class `num`, so the only way the class check performed by the successor operation could fail is if x is not bound to a number at the initial call. In other words, it is a loop invariant that the result is of class `num`, so there is no need for this check within the loop, only at its entry point. But there is no way to avoid the check on each iteration.

Third, the argument, y, to the inner λ-abstraction arises either at the initial call, or as a result of a recursive call. But if the initial call binds y to a number, then so must the recursive call, because the dynamic semantics ensures that the predecessor of a number is also a number. Once again we have an unnecessary dynamic check in the inner loop of the function, but there is no way to avoid it.

Class checking and labelling is not free—storage is required for the label itself, and the marking of a value with a class takes time as well as space. While the overhead is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated whenever possible. But within $\mathcal{L}\{dyn\}$ itself there is no way to avoid the overhead, because there are no "unchecked" operations in the language—for these to be safe requires a static type system!

23.3 Hybrid Typing

Let us consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$, whose syntax extends that of the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 16 with the following additional constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{dyn}$	$\texttt{dyn}$
Expr	e	::=	$\texttt{new}[l](e)$	$l\,\texttt{!}\,e$
		|	$\texttt{cast}[l](e)$	$e\,\texttt{?}\,l$
Class	l	::=	$\texttt{num}$	$\texttt{num}$
		|	$\texttt{fun}$	$\texttt{fun}$

The type dyn represents the type of labelled values. Here we have only two classes of data object, numbers and functions. Observe that the cast operation takes as argument a class, not a type! That is, casting is concerned with an object's *class*, which is indicated by a label, not with its *type*, which is always dyn.

The static semantics for $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$ is the extension of that of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ with the following rules governing the type dyn.

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}(e) : \texttt{dyn}} \tag{23.5a}$$

$$\frac{\Gamma \vdash e : \texttt{parr(dyn;dyn)}}{\Gamma \vdash \texttt{new[fun]}(e) : \texttt{dyn}} \tag{23.5b}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[num]}(e) : \texttt{nat}} \tag{23.5c}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[fun]}(e) : \texttt{parr(dyn;dyn)}} \tag{23.5d}$$

The static semantics ensures that class labels are applied to objects of the appropriate type, namely num for natural numbers, and fun for functions defined over labelled values.

The dynamic semantics of $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$ is given by the following rules:

$$\frac{e\ \mathsf{val}}{\texttt{new}[l](e)\ \mathsf{val}} \tag{23.6a}$$

$$\frac{e \mapsto e'}{\texttt{new}[l](e) \mapsto \texttt{new}[l](e')} \tag{23.6b}$$

$$\frac{e \mapsto e'}{\texttt{cast}[l](e) \mapsto \texttt{cast}[l](e')} \tag{23.6c}$$

$$\frac{\texttt{new}[l](e)\ \mathsf{val}}{\texttt{cast}[l](\texttt{new}[l](e)) \mapsto e} \tag{23.6d}$$

$$\frac{\texttt{new}[l'](e)\ \mathsf{val} \quad l \neq l'}{\texttt{cast}[l](\texttt{new}[l'](e))\ \mathsf{err}} \tag{23.6e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 23.2 (Canonical Forms). *If e : $\mathtt{dyn}$ and e val, then $e = \mathtt{new[}l\mathtt{]}\mathtt{(}e'\mathtt{)}$ for some class l and some e' val. If $l = \mathtt{num}$, then e' : $\mathtt{nat}$, and if $l = \mathtt{fun}$, then e' : $\mathtt{parr(dyn;\,dyn)}$.*

Proof. By a straightforward rule induction on static semantics of $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$. □

Theorem 23.3 (Safety). *The language $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$ is safe:*

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

2. *If $e : \tau$, then either e val, or e err, or $e \mapsto e'$ for some e'.*

Proof. Preservation is proved by rule induction on the dynamic semantics, and progress is proved by rule induction on the static semantics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for $\mathcal{L}\{dyn\}$—a well-typed cast might fail at run-time if the class of the case does not match the class of the value. □

23.4 Optimization of Dynamic Typing

The type dyn—whether primitive or derived—supports the smooth integration of dynamic with static typing. This means that we can take full advantage of the expressive power of static types whenever possible, while permitting the flexibility of dynamic typing whenever desirable.

One application of the hybrid framework is that it permits the optimization of dynamically typed programs by taking advantage of statically evident typing constraints. Let us examine how this plays out in the case of the addition function, which is rendered in $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$ by the expression

$$\mathtt{fun}\,!\,\lambda\mathtt{(}x\mathtt{:dyn.\,fix}\,p\mathtt{:dyn\,is\,fun}\,!\,\lambda\mathtt{(}y\mathtt{:dyn.}\,e_{x,p,y}\mathtt{))},$$

where

$$x : \mathtt{dyn}, p : \mathtt{dyn}, y : \mathtt{dyn} \vdash e_{x,p,y} : \mathtt{dyn}$$

is defined to be the expression

$$\mathtt{ifz}\,\mathtt{(}y\,\mathtt{?}\,\mathtt{num)}\,\{\mathtt{z}{\Rightarrow}x \mid \mathtt{s(}y'\mathtt{)}{\Rightarrow}\mathtt{num}\,!\,\mathtt{(s(((}p\,\mathtt{?}\,\mathtt{fun)((num}\,!\,y'\mathtt{)))}\,\mathtt{?}\,\mathtt{num))}\}.$$

This is essentially an explicit form of the dynamically typed addition function given in Section 23.2 on page 182. This formulation makes explicit the checking of classes that is implicit in $\mathcal{L}\{dyn\}$. We will now show how to exploit the static type system of $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ to optimize this dynamically typed implementation of addition, whose result is of interest only in the case that the arguments are of class num.

First, note that that the body of the fix expression is an explicitly labelled function. This means that when the recursion is unwound, the variable p is bound to this value of type dyn. Consequently, the check that p is labelled with class fun is redundant, and can be eliminated. This is achieved by re-writing the function as follows:

$$\texttt{fun}\,!\,\lambda(x\texttt{:dyn.}\,\texttt{fun}\,!\,\texttt{fix}\,p\texttt{:dyn} \rightharpoonup \texttt{dyn}\,\texttt{is}\,\lambda(y\texttt{:dyn.}\,e'_{x,p,y}))\texttt{,}$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,\texttt{?}\,\texttt{num})\,\{\texttt{z}{\Rightarrow}x \mid \texttt{s}(y'){\Rightarrow}\texttt{num}\,!\,(\texttt{s}((p((\texttt{num}\,!\,y')))\,\texttt{?}\,\texttt{num}))\}.$$

We have "hoisted" the function class label out of the loop, and suppressed the cast inside the loop. Correspondingly, the type of p has changed to dyn ⇀ dyn, reflecting that the body is now a "bare function", rather than a labelled function value of type dyn.

Next, observe that the parameter y of type dyn is cast to a number on each iteration of the loop before it is tested for zero. Since this function is recursive, the bindings of y arise in one of two ways, at the initial call to the addition function, and on each recursive call. But the recursive call is made on the predecessor of y, which is a true natural number that is labelled with num at the call site, only to be removed by the class check at the conditional on the next iteration. This suggests that we hoist the check on y outside of the loop, and avoid labelling the argument to the recursive call. Doing so changes the type of the function, however, from dyn ⇀ dyn to nat ⇀ dyn. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class num, and, if so, the underlying value is incremented and labelled with class num. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the class check is redundant, because we know that it must have class num. But what if the result came from the other branch of the conditional? In that case the function returns x, which need not be of class num! However, one might reasonably insist that this is only a theoretical possibility—after all, we are defining the addition function, and its arguments might reasonably be restricted to have class num. This can be achieved by replacing x by $x\,\texttt{?}\,\texttt{num}$, which checks that x is of class `num`, and returns the underlying number.

Combining these optimizations we obtain the inner loop e''_x defined as follows:

$$\texttt{fix}\,p\texttt{:nat}\rightharpoonup\texttt{nat}\;\texttt{is}\,\lambda\texttt{(}y\texttt{:nat.}\,\texttt{ifz}\,y\,\{\texttt{z}\Rightarrow x\,\texttt{?}\,\texttt{num}\mid\texttt{s(}y'\texttt{)}\Rightarrow\texttt{s(}p\texttt{(}y'\texttt{))}\}\texttt{)}.$$

This function has type $\texttt{nat}\rightharpoonup\texttt{nat}$, and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Finally, recall that the overall goal is to define a version of addition that works on values of type `dyn`. Thus we require a value of type $\texttt{dyn}\rightharpoonup\texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat}\rightharpoonup\texttt{nat}$. This can be converted to the required form by pre-composing with a cast to `num` and post-composing with a coercion to `num`:

$$\texttt{fun}\,\texttt{!}\,\lambda\texttt{(}x\texttt{:dyn.}\,\texttt{fun}\,\texttt{!}\,\lambda\texttt{(}y\texttt{:dyn.}\,\texttt{num}\,\texttt{!}\,\texttt{(}e''_x\texttt{(}y\,\texttt{?}\,\texttt{num))))}.$$

The innermost λ-abstraction converts the function e''_x from type $\texttt{nat}\rightharpoonup\texttt{nat}$ to type $\texttt{dyn}\rightharpoonup\texttt{dyn}$ by composing it with a class check that ensures that y is a natural number at the initial call site, and applies a label to the result to restore it to type `dyn`.

23.5 Static "Versus" Dynamic Typing

There have been many attempts to explain the distinction between dynamic and static typing, most of which are misleading or wrong. For example, it is often said that static type systems associate types with variables, but dynamic type systems associate types with values. This oft-repeated characterization appears to be justified by the absence of type annotations on λ-abstractions, and the presence of classes on values. But it is based on a confusion of classes with types—the *class* of a value (`num` or `fun`) is not its *type*. Moreover, a static type system assigns types to values just as surely as it does to variables, so the description fails on this account as well. Thus, this supposed distinction between dynamic and static typing makes no sense, and is best disregarded.

A related characterization of the difference between static and dynamic languages is to say that the former check types at run-time, whereas the latter check types at compile-time. To say that static languages check types statically is a tautology; to say that dynamic languages check types at run-time is a falsehood. Dynamic languages perform *class checking*, not *type* *checking*, at run-time. For example, application checks that its first argument is labelled with `fun`; it does not type check the body of the function. Indeed, at no point does the dynamic semantics compute the *type* of a value, rather it checks its class against its expectations before proceeding. Here again, a supposed contrast between static and dynamic languages evaporates under careful analysis.

Another characterization is to assert that dynamic languages admit *heterogeneous* lists, whereas static languages admit only *homogeneous* lists. (The distinction applies to other collections as well.) To see why this description is wrong, let us consider briefly how one might add lists to $\mathcal{L}\{dyn\}$. One would add two constructs, `nil`, representing the empty list, and $\texttt{cons(}d_1\texttt{;}d_2\texttt{)}$, representing the non-empty list with head d_1 and tail d_2. The origin of the supposed distinction lies in the observation that each element of a list represented in this manner might have a different class. For example, one might form the list

$$\texttt{cons(s(z);cons(}\lambda x.\,x\texttt{;nil)),}$$

whose first element is a number, and whose second element is a function. Such a list is said to be heterogeneous. In contrast static languages commit to a single *type* for each element of the list, and hence are said to be homogeneous. But here again the supposed distinction breaks down on close inspection, because it is based on the confusion of the type of a value with its class. Every labelled value has type `dyn`, so that the lists are *type* homogeneous. But since values of type `dyn` may have different classes, lists are *class* heterogenoues—regardless of whether the language is statically or dynamically typed!

What, then, are we to make of the traditional distinction between dynamic and static languages? Rather than being in opposition to each other, we see that *dynamic languages are a mode of use of static languages*. If we have a type `dyn` in the language, then we have all of the apparatus of dynamic languages at our disposal, so there is no loss of expressive power. But there is a very significant gain from embedding dynamic typing within a static type discipline! We can avoid much of the overhead of dynamic typing by simply limiting our use of the type `dyn` in our programs, as was illustrated in Section 23.4 on page 185.

23.6 Dynamic Typing From Recursive Types

The type dyn codifies the use of dynamic typing within a static language. Its introduction form labels an object of the appropriate type, and its elimination form is a (possibly undefined) casting operation. Rather than treating dyn as primitive, we may derive it as a particular use of recursive types, according to the following definitions:[1]

$$\texttt{dyn} = \mu t\,.\,\texttt{[num:nat,fun:}t \rightharpoonup t\texttt{]} \tag{23.7}$$

$$\texttt{new[num]}(e) = \texttt{fold(in[num]}(e)\texttt{)} \tag{23.8}$$

$$\texttt{new[fun]}(e) = \texttt{fold(in[fun]}(e)\texttt{)} \tag{23.9}$$

$$\texttt{cast[num]}(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in[num]}(x) \Rightarrow x \mid \texttt{in[fun]}(x) \Rightarrow \texttt{error}\} \tag{23.10}$$

$$\texttt{cast[fun]}(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in[num]}(x) \Rightarrow \texttt{error} \mid \texttt{in[fun]}(x) \Rightarrow x\} \tag{23.11}$$

One may readily check that the static and dynamic semantics for the type dyn are derivable according to these definitions.

This observation strengthens the argument that dynamic typing is but a mode of use of static typing. This encoding shows that we need not include a special-purpose type dyn in a statically typed language in order to admit dynamic typing. Instead, one may use the general concepts of recursive types and sum types to define special-purpose dynamically typed sub-languages on a per-program basis. For example, if we wish to admit strings into our dynamic sub-language, then we may simply expand the type definition above to admit a third summand for strings, and so on for any type we may wish to consider. Classes emerge as labels of the summands of a sum type, and recursive types ensure that we can represent class-heterogeneous aggregates. Thus, not only is dynamic typing a special case of static typing, but we need make no special provision for it in a statically typed language, since we already have need of recursive types independently of this particular application.

23.7 Exercises

[1] Here we have made use of a special expression error to signal an error condition. In a richer language we would use exceptions, which are introduced in Chapter 30.

Chapter 24

Type Dynamic

24.1 Typed Values

24.2 Exercises

Part VIII

Polymorphism

Chapter 25

Girard's System F

The languages $\mathcal{L}\{\mathtt{nat}\rightarrow\}$ and $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, and their various extensions, have the property that every expression has at most one type. In particular, a function has uniquely determined domain and range types. Consequently, there is a distinct identity function for each type, $id_\tau = \lambda(x\!:\!\tau.\,x)$, and a distinct composition function for each triple of types,

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda(f\!:\!\tau_2 \rightarrow \tau_3.\,\lambda(g\!:\!\tau_1 \rightarrow \tau_2.\,\lambda(x\!:\!\tau_1.\,f(g(x))))).$$

And yet every identity function and every composition function "works the same way", regardless of the choice of types! It quickly gets tedious to write the "same" program over and over, with the sole difference being the types involved. It would clearly be advantageous to capture the underlying computation once and for all, with specific instances arising by specifying the types involved.

What is needed is a way to capture the pattern of a computation in a way that is *generic*, or *parametric*, in the types involved. This is called *polymorphism*. In this chapter we will study a language introduced by Girard under the name *System F* and by Reynolds under the name *polymorphic typed λ-calculus*.

25.1 System F

System F, or the *polymorphic λ-calculus*, or $\mathcal{L}\{\rightarrow\forall\}$, 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 syntax of System **F** is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	t	t
		$\mid$	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$
		$\mid$	$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$
		$\mid$	$\mathtt{Lam}(t.e)$	$\Lambda(t.e)$
		$\mid$	$\mathtt{App}[\tau](e)$	$e[\tau]$

The meta-variable t ranges over a class of *type variables*, and x ranges over a class of *expression variables*. The *type abstraction*, $\mathtt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* t standing for an unspecified type within e. The *type application*, or *instantiation*, $\mathtt{App}[\tau](e)$, 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*, $\mathtt{all}(t.\tau)$, that determines the type, τ, of the result as a function of the argument, t.

The static semantics of $\mathcal{L}\{\rightarrow\forall\}$ consists of two judgement forms, $\tau\ \mathsf{type}$, 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 parametric hypothetical judgements of the form

$$\mathcal{T} \mid \Delta \vdash \tau\ \mathsf{type}$$

and

$$\mathcal{T}\ \mathcal{X} \mid \Delta\ \Gamma \vdash e : \tau.$$

Here $\mathcal{T}$ consists of a finite set of *type constructor variable* declarations of the form $t\ \mathsf{cons}$ and $\mathcal{X}$ consists of a finite set of *expression variable* declarations of the form $x\ \mathsf{exp}$. The finite set of hypotheses Δ consists of assumptions of the form $t\ \mathsf{type}$ such that $\mathcal{T} \vdash t\ \mathsf{cons}$, and the finite set Γ consists of hypotheses of the form $x : \tau$, where $\mathcal{X} \vdash x\ \mathsf{exp}$ and $\mathcal{T} \mid \Delta \vdash \tau\ \mathsf{type}$. As usual, we drop explicit mention of the parameters $\mathcal{T}$ and $\mathcal{X}$, since they can be recovered from the hypotheses Δ and Γ.

The type formation rules are given as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{25.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{25.1b}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Delta \vdash \mathtt{all}(t\mathtt{.}\tau) \ \mathsf{type}} \tag{25.1c}$$

The rules for typing expressions are as follows:

$$\frac{}{\Delta \ \Gamma, x : \tau \vdash x : \tau} \tag{25.2a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta \ \Gamma \vdash \mathtt{lam}[\tau_1]\mathtt{(}x\mathtt{.}e\mathtt{)} : \mathtt{arr}(\tau_1;\tau_2)} \tag{25.2b}$$

$$\frac{\Delta \ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta \ \Gamma \vdash e_2 : \tau_2}{\Delta \ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{25.2c}$$

$$\frac{\Delta, t \ \mathsf{type} \ \Gamma \vdash e : \tau}{\Delta \ \Gamma \vdash \mathtt{Lam}(t\mathtt{.}e) : \mathtt{all}(t\mathtt{.}\tau)} \tag{25.2d}$$

$$\frac{\Delta \ \Gamma \vdash e : \mathtt{all}(t\mathtt{.}\tau') \quad \Delta \vdash \tau \ \mathsf{type}}{\Delta \ \Gamma \vdash \mathtt{App}[\tau]\mathtt{(}e\mathtt{)} : [\tau/t]\tau'} \tag{25.2e}$$

For example, the polymorphic composition function is written as follows:

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda(f{:}t_2 \rightarrow t_3.\lambda(g{:}t_1 \rightarrow t_2.\lambda(x{:}t_1.f(g(x)))))))).$$

This expression has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \rightarrow t_3) \rightarrow (t_1 \rightarrow t_2) \rightarrow (t_1 \rightarrow t_3)))).$$

The static semantics validates the expected structural rules, including substitution for both type and expression variables.

Lemma 25.1 (Substitution)**.**

1. *If Δ, t type $\vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau'$ type.*

2. *If Δ, t type $\Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau$ type, then $\Delta \ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.*

3. *If $\Delta \ \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta \ \Gamma \vdash e : \tau$, then $\Delta \ \Gamma \vdash [e/x]e' : \tau'$.*

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 $\mathcal{L}\{\to\forall\}$ is given as follows:

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{25.3a}$$

$$\frac{}{\texttt{Lam(}t\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{25.3b}$$

$$\frac{\{e_2\ \mathsf{val}\}}{\texttt{ap(lam[}\tau_1\texttt{](}x\texttt{.}e\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e} \tag{25.3c}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}} \tag{25.3d}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1\texttt{;}e_2'\texttt{)}}\right\} \tag{25.3e}$$

$$\frac{}{\texttt{App[}\tau\texttt{](Lam(}t\texttt{.}e\texttt{))} \mapsto [\tau/t]e} \tag{25.3f}$$

$$\frac{e \mapsto e'}{\texttt{App[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{App[}\tau\texttt{](}e'\texttt{)}} \tag{25.3g}$$

The bracketed premises are rules are to be omitted for the call-by-name semantics, and included for the call-by-value semantics. (There is no by-name vs by-value distinction for type applications.)

It is then a simple matter to prove safety for this language, using by-now familiar methods.

Lemma 25.2 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$, then $e = \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)}$ with $x : \tau_1 \vdash e_2 : \tau_2$.*

2. *If $\tau = \texttt{all(}t\texttt{.}\tau'\texttt{)}$, then $e = \texttt{Lam(}t\texttt{.}e'\texttt{)}$ with t $\mathsf{type} \vdash e' : \tau'$.*

Proof. By rule induction on the static semantics. □

Theorem 25.3 (Preservation)**.** *If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.*

Proof. By rule induction on the dynamic semantics. □

Theorem 25.4 (Progress)**.** *If $e : \sigma$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on the static semantics. □

25.2 Polymorphic Definability

It will be proved in Chapter 54 that every well-typed expression in $\mathcal{L}\{\to\forall\}$ evaluates to a value—there is no possibility of an infinite loop. This may seem obvious, at first glance, because there is no looping or recursion construct in $\mathcal{L}\{\to\forall\}$. But appearances can be deceiving! A rich class of types, including the natural numbers, are definable in the language. This means that primitive recursion is implicitly present in the language, even though it is not an explicit construct.

To begin with we show that lazy product and sum types are definable in the lazy variant of $\mathcal{L}\{\to\forall\}$. We then show that the natural numbers, under the lazy semantics, are definable as well.

25.2.1 Products and Sums

To show that binary products are definable means that we may fill in the following equations in such a way that the static semantics and the dynamic semantics are derivable in $\mathcal{L}\{\to\forall\}$:

$$\begin{aligned}
\texttt{prod}(\sigma;\tau) &= \ldots\\
\texttt{pair}(e_1;e_2) &= \ldots\\
\texttt{fst}(e) &= \ldots\\
\texttt{snd}(e) &= \ldots
\end{aligned}$$

The required definitions are derived from the conservation principle of Chapter 13, according to which the eliminatory forms are inverse to the introductory forms. Applying this principle to the present case, we reason that to compute an element of some type ρ from an element of type $\sigma \times \tau$, it suffices to compute that element from an element of type σ and an element of type τ. This leads to the following definitions:

$$\begin{aligned}
\sigma \times \tau &= \forall(r\,.\,(\sigma \to \tau \to r) \to r)\\
\langle e_1, e_2\rangle &= \Lambda(r\,.\,\lambda(x{:}\sigma \to \tau \to r\,.\,x(e_1)(e_2)))\\
\texttt{fst}(e) &= e[\sigma](\lambda(x{:}\sigma.\,\lambda(y{:}\tau.\,x)))\\
\texttt{snd}(e) &= e[\tau](\lambda(x{:}\sigma.\,\lambda(y{:}\tau.\,y)))
\end{aligned}$$

These encodings correspond to the lazy semantics for product types, so that $\texttt{fst}(\langle e_1, e_2\rangle) \mapsto^* e_1$ is derivable according to the lazy semantics of $\mathcal{L}\{\to\forall\}$.

The nullary product, or unit, type is similarly definable:

$$\begin{aligned} \texttt{unit} &= \forall(r\,.\,r \to r) \\ \langle\rangle &= \Lambda(r\,.\,\lambda(x\!:\!r.\,x)) \end{aligned}$$

Observe that these definitions are formally consistent with those for binary products, the difference being only in the number of components (zero, instead of two) of an element of the type.

The definition of binary sums proceeds by a similar analysis. To compute a element of type ρ from an element of type $\sigma + \tau$, it is enough to be able to compute that element from a value of type σ and from a value of type τ.

$$\begin{aligned} \sigma + \tau &= \forall(r\,.\,(\sigma \to r) \to (\tau \to r) \to r) \\ \texttt{in[l]}(e) &= \Lambda(r\,.\,\lambda(x\!:\!\sigma \to r.\,\lambda(y\!:\!\tau \to r.\,x(e)))) \\ \texttt{in[r]}(e) &= \Lambda(r\,.\,\lambda(x\!:\!\sigma \to r.\,\lambda(y\!:\!\tau \to r.\,y(e)))) \\ \texttt{case}\,e\,\{\texttt{in[l]}(x_1) \Rightarrow e_1 \mid \texttt{in[r]}(x_2) \Rightarrow e_2\} &= \\ e[\rho](\lambda(x_1\!:\!\sigma.\,e_1))(\lambda(x_2\!:\!\tau.\,e_2)) \end{aligned}$$

In the last equation the type ρ is the type of the case expression. It is a good exercise to check that the lazy dynamic semantics of sums is derivable under the lazy semantics for $\mathcal{L}\{\to\forall\}$.

The nullary sum, or empty, type is defined similarly:

$$\begin{aligned} \texttt{void} &= \forall(r\,.\,r) \\ \texttt{abort}(e) &= e[\rho] \end{aligned}$$

Once again, observe that this is formally consistent with the binary case, albeit for a sum of no types.

25.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\to\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \texttt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\texttt{iter}[\tau](e_0; e_1; x\,.\,e_2) : \tau}\,.$$

Since the result type τ is arbitrary, this means that if we have an iterator, then it can be used to define a 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 $\mathtt{z}$, and if given a function transforming the result for x into the result for $\mathtt{s}(x)$, 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 above chart may be completed as follows:

$$\begin{aligned}
\mathtt{nat} &= \forall(t\,.\,t \to (t \to t) \to t) \\
\mathtt{z} &= \Lambda(t\,.\,\lambda(z\!:\!t.\,\lambda(s\!:\!t \to t.\,z))) \\
\mathtt{s}(e) &= \Lambda(t\,.\,\lambda(z\!:\!t.\,\lambda(s\!:\!t \to t.\,s(e[t](z)(s))))) \\
\mathtt{iter}[\tau](e_0;e_1;x\,.\,e_2) &= e_0[\tau](e_1)(\lambda(x\!:\!\tau.\,e_2))
\end{aligned}$$

It is a straightforward exercise to check that the static semantics of these constructs is correctly derived from these definitions.

Observe that if we ignore the type abstractions, type applications, and the types ascribed to variables, then the definitions of $\mathtt{z}$ and $\mathtt{s}(e)$ in $\mathcal{L}\{\to\forall\}$ are just the same as the untyped Church numerals (Definition 22.3 on page 175). Correspondingly, the definition of iteration is the same as in the untyped case, provided that we ignore the types. The computational content is the same; the only difference here is that we are also taking care to keep track of the types of the computations performed using the natural numbers. From this point of view, the polymorphic abstraction in the Church numerals is essential, since we may use the same number to iterate several different operations at several different types. (Indeed, it is for the lack of polymorphism that there is no useful analogue of the Church numerals in System **T**, and hence the natural numbers must be taken as a primitive notion in that calculus.)

25.2.3 Expressive Power

The definability of the natural numbers implies that $\mathcal{L}\{\to\forall\}$ is at least as expressive as $\mathcal{L}\{\mathtt{nat}\to\}$. But is it more expressive? Yes, and by a wide margin! It is possible to show that the universal evaluation function, E, for $\mathcal{L}\{\mathtt{nat}\to\}$ (introduced in Chapter 15) is definable in $\mathcal{L}\{\to\forall\}$. That is, one may define an interpreter for $\mathcal{L}\{\mathtt{nat}\to\}$ in $\mathcal{L}\{\to\forall\}$. As a consequence, the diagonal function, D, which is defined in terms of the universal function,

E, is definable in $\mathcal{L}\{\to\forall\}$, but not in $\mathcal{L}\{\mathtt{nat}\to\}$—it diagonalizes out of the restricted language, but is definable within the richer one.

Put in other terms, the language $\mathcal{L}\{\to\forall\}$ is able to prove the termination of *all* expressions in $\mathcal{L}\{\mathtt{nat}\to\}$. But it cannot do this for $\mathcal{L}\{\to\forall\}$ itself, by a diagonal argument analogous to the one given in Chapter 15. Thus we see the beginnings of a hierarchy of expressiveness, with $\mathcal{L}\{\mathtt{nat}\to\}$ at the bottom, $\mathcal{L}\{\to\forall\}$ above it, and some other language, as yet to be specified, above it, and so forth. Each language in this hierarchy is total (all functions terminate), and so we may diagonalize to obtain a total function not definable within it, leading to a new language within which it may be defined, and so on *ad infinitum*. Each step increases expressive power, but nevertheless omits some functions that are only definable at higher levels of the hierarchy.

25.3 Exercises

1. Show that primitive recursion is definable in $\mathcal{L}\{\to\forall\}$ by exploiting the definability of iteration and binary products.

2. Investigate the representation of eager products and sums in eager and lazy variants of $\mathcal{L}\{\to\forall\}$.

3. Show how to write an interpreter for $\mathcal{L}\{\mathtt{nat}\to\}$ in $\mathcal{L}\{\to\forall\}$.

Chapter 26

Abstract Types

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 $\mathcal{L}\{\rightarrow\forall\}$ 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!

26.1 Existential Types

The syntax of $\mathcal{L}\{\to\forall\exists\}$ is the extension of $\mathcal{L}\{\to\forall\}$ with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Types	τ	$::=$	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$
Expr	e	$::=$	$\texttt{pack}[t.\tau;\rho](e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$
		$\mid$	$\texttt{open}[t.\tau](e_1;t,x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$, where ρ is a type and e is an expression of type $[\rho/t]\tau$. 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 $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$, 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.

26.1.1 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.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{26.1a}$$

$$\frac{\Delta\ \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}[t.\tau;\rho](e) : \texttt{some}(t.\tau)} \tag{26.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}[t.\tau](e_1; t, x.e_2) : \tau_2} \tag{26.1c}$$

Rule (26.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, τ_2, must not involve the abstract type t. This restriction 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 the type variable t.

26.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified as follows:

$$\frac{\{e\ \mathsf{val}\}}{\texttt{pack}[t.\tau;\rho](e)\ \mathsf{val}} \tag{26.2a}$$

$$\left\{\frac{e \mapsto e'}{\texttt{pack}[t.\tau;\rho](e) \mapsto \texttt{pack}[t.\tau;\rho](e')}\right\} \tag{26.2b}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{open}[t.\tau](e_1; t, x.e_2) \mapsto \texttt{open}[t.\tau](e_1'; t, x.e_2)} \tag{26.2c}$$

$$\frac{e\ \mathsf{val}}{\texttt{open}[t.\tau](\texttt{pack}[t.\tau;\rho](e); t, x.e_2) \mapsto [\rho, e/t, x]e_2} \tag{26.2d}$$

The bracketed premises and rules are to be omitted for a lazy semantics, and included for an eager semantics.

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.

26.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\to\forall\}$ to the new constructs.

Theorem 26.1 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on $e \mapsto e'$, making use of substitution for both expression- and type variables. □

Lemma 26.2 (Canonical Forms). *If e : $\mathtt{some(}t\,.\,\tau\mathtt{)}$ and e val, then $e = \mathtt{pack[}t\,.\,\tau;\rho\mathtt{](}e'\mathtt{)}$ for some type ρ and some e' val such that $e' : [\rho/t]\tau$.*

Proof. By rule induction on the static semantics, making use of the definition of closed values. □

Theorem 26.3 (Progress). *If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on $e : \tau$, making use of the canonical forms lemma. □

26.2 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, $\exists\mathtt{(}t\,.\,\sigma\mathtt{)}$, which serves as the interface of the queue abstraction:[1]

$$\exists\mathtt{(}t\,.\,\langle \mathtt{emp}:t, \mathtt{ins}:\tau\times t\to t, \mathtt{rem}:t\to\tau\times t\rangle\mathtt{)}.$$

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,

[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 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:

$$\texttt{pack}\,\tau\,\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\texttt{nil},\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\sigma),$$

where

$$e_i : \tau \times \tau\,\texttt{list} \rightarrow \tau\,\texttt{list} = \lambda(x{:}\tau \times \tau\,\texttt{list}.\,e_i'),$$

and

$$e_r : \tau\,\texttt{list} \rightarrow \tau \times \tau\,\texttt{list} = \lambda(x{:}\tau\,\texttt{list}.\,e_r').$$

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 $\tau\,\texttt{list}$, and are programmed accordingly.

It is also possible to give another implementation, e_p, of the same interface, $\exists(t.\sigma)$, 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:

$$\texttt{pack}\,\tau\,\texttt{list} \times \tau\,\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\langle\texttt{nil},\texttt{nil}\rangle,\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\sigma).$$

In this case e_i has type

$$\tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}) \rightarrow (\tau\,\texttt{list} \times \tau\,\texttt{list}),$$

and e_r has type

$$(\tau\,\texttt{list} \times \tau\,\texttt{list}) \rightarrow \tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}).$$

These operations "know" that queues are represented as values of type

$$\tau\,\texttt{list} \times \tau\,\texttt{list},$$

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 $\exists(t.\sigma)$, then a client of the abstraction has the form

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e' : \tau',$$

where the type, τ', of e' does not involve the abstract type t. Within e' the variable x has type

$$\langle \texttt{emp}:t, \texttt{ins}:\tau \times t \to t, \texttt{rem}:t \to \tau \times t\rangle,$$

in which t is unspecified — or, as is often said, *held abstract*.

Observe that *only* the type information specified in $\exists(t\,.\,\sigma)$ 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 $\exists(t\,.\,\sigma)$ 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.

26.3 Definability of Existentials in System F

Strictly speaking, it is not necessary to extend $\mathcal{L}\{\to\forall\}$ 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

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e' : \tau',$$

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:

$$\begin{aligned}
\exists(t\,.\,\sigma) &= \forall(t'\,.\,\forall(t\,.\,\sigma \to t') \to t') \\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t\,.\,\tau) &= \Lambda(t'\,.\,\lambda(x{:}\forall(t\,.\,\tau \to t')\,.\,x[\rho](e))) \\
\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e' &= e[\tau'](\Lambda(t\,.\,\lambda(x{:}\tau.\,e')))
\end{aligned}$$

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.

26.4 Exercises

Chapter 27

*Constructors and Kinds

In Chapters 25 and 26 we used quantification over types to model genericity and abstraction. While sufficient for many situations, type quantification alone is not sufficient to model many programming situations of practical interest. For example, it is natural to consider *abstract families of types*, such as τ `list`, in which we simultaneously introduce an infinite collection of types sharing a common collection of operations on them. As another example, it is natural to introduce simultaneously two abstract types, such as a type of trees, whose nodes have a forest of children, and a type of forests whose elements are trees.

Such situations may be modelled by permitting quantification over other *kinds* than just types—for example, over *type constructors*, which are functions mapping types to types, and *type structures*, which are essentially tuples of types. To support such generalizations we enrich the structure of our languages to include *(higher) kinds* classifying *constructors*, in a manner reminiscent of the familiar use of types to classify expressions. In fact, types themselves emerge as certain forms of constructor, namely those of kind `Type`. We then generalize universal and existential quantification to quantify over an arbitrary kind, recovering the original forms as quantifying over the kind `Type`.

This two-layer architecture models the *phase distinction* discussed in Chapter 13: the constructor and kind level form the *static layer*, whereas the expression and type level form the *dynamic layer*. The role of the static layer is to provide the apparatus required to define the static semantics of the language, the premier example being the class of types, which here arise as certain forms of constructor. The role of the dynamic layer is, as before, to define the facilities of the language, such as functions or data structures, that we compute with at run-time. From this point of view, constructors are the *static data* of the language, whereas expressions are the *dynamic data*.

We will exploit this interpretation in Chapter 28, wherein we introduce families of types indexed by kinds other than Type. For the present we will study the extension, $\mathcal{L}\{\rightarrow,\forall_\kappa,\exists_\kappa\}$, or $\mathcal{L}\{\rightarrow\forall\exists\}$ with higher kinds.

27.1 Static Semantics

The abstract syntax of $\mathcal{L}\{\rightarrow,\forall_\kappa,\exists_\kappa\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Kind	κ	$::=$	$\mathtt{Type}$	$\mathtt{Type}$
		$\mid$	$\mathtt{Prod(}\kappa_1\mathtt{;}\kappa_2\mathtt{)}$	$\kappa_1 \times \kappa_2$
		$\mid$	$\mathtt{Arr(}\kappa_1\mathtt{;}\kappa_2\mathtt{)}$	$\kappa_1 \rightarrow \kappa_2$
Cons	c	$::=$	t	t
		$\mid$	$\mathtt{arr}$	$\rightarrow$
		$\mid$	$\mathtt{all[}\kappa\mathtt{]}$	$\forall_\kappa$
		$\mid$	$\mathtt{some[}\kappa\mathtt{]}$	$\exists_\kappa$
		$\mid$	$\mathtt{pair(}c_1\mathtt{;}c_2\mathtt{)}$	$\langle c_1, c_2\rangle$
		$\mid$	$\mathtt{fst(}c\mathtt{)}$	$\mathtt{fst(}c\mathtt{)}$
		$\mid$	$\mathtt{snd(}c\mathtt{)}$	$\mathtt{snd(}c\mathtt{)}$
		$\mid$	$\mathtt{lambda[}\kappa\mathtt{](}t\mathtt{.}c\mathtt{)}$	$\lambda\mathtt{(}t\mathtt{::}\kappa\mathtt{.}c\mathtt{)}$
		$\mid$	$\mathtt{app(}c_1\mathtt{;}c_2\mathtt{)}$	$c_1\mathtt{[}c_2\mathtt{]}$
Type	τ	$::=$	c	c
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{lam[}\tau\mathtt{](}x\mathtt{.}e\mathtt{)}$	$\lambda\mathtt{(}x\mathtt{:}\tau\mathtt{.}e\mathtt{)}$
		$\mid$	$\mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)}$	$e_1\mathtt{(}e_2\mathtt{)}$
		$\mid$	$\mathtt{Lam[}\kappa\mathtt{](}t\mathtt{.}e\mathtt{)}$	$\Lambda\mathtt{(}t\mathtt{::}\kappa\mathtt{.}e\mathtt{)}$
		$\mid$	$\mathtt{App[}c\mathtt{](}e\mathtt{)}$	$e\mathtt{[}c\mathtt{]}$
		$\mid$	$\mathtt{pack[}\kappa\mathtt{;}c\mathtt{;}c'\mathtt{](}e\mathtt{)}$	$\mathtt{pack}\,c'\,\mathtt{with}\,e\,\mathtt{as}\,\exists_\kappa\mathtt{[}c\mathtt{]}$
		$\mid$	$\mathtt{open[}\kappa\mathtt{;}c\mathtt{](}e_1\mathtt{;}t\mathtt{,}x\mathtt{.}e_2\mathtt{)}$	$\mathtt{open}\,e_1\,\mathtt{as}\,t\mathtt{::}\kappa\,\mathtt{with}\,x\mathtt{:}c\mathtt{[}t\mathtt{]}\,\mathtt{in}\,e_2$

The first two categories constitute the *static layer* of the language, and the second two constitute the *dynamic layer*. Observe that types are just certain constructors, namely those of kind Type. The representations of arrow types and quantified types arises from the interpretation of the function type constructor and the two quantifiers as constants of higher kind, as will become clear shortly. In Section 27.3 on page 215 we will consider a formulation in which types in their role as classifiers of expressions are distinguished from types in their role as constructors of kind Type.

The static semantics of $\mathcal{L}\{\rightarrow,\forall_\kappa,\exists_\kappa\}$ consists of rules for deriving the following forms of judgement:

$\mathcal{T} \mid \Delta \vdash c :: \kappa$	constructor formation
$\mathcal{T} \mid \Delta \vdash c_1 \equiv c_2 :: \kappa$	definitional equality of constructors
$\mathcal{T} \mid \Delta \vdash \tau\ \mathsf{type}$	type formation
$\mathcal{T}\ \mathcal{X} \mid \Delta\ \Gamma \vdash e : \tau$	expression formation

The parameter set $\mathcal{T}$ specifies the *type constructor variables*, and $\mathcal{X}$ specifies the *expression variables*, in each judgement. The hypotheses in Δ have the form $t :: \kappa$, where $\mathcal{T} \vdash t\ \mathsf{cons}$, and the hypotheses in Γ have the form $x : \tau$ where $\mathcal{X} \vdash x\ \mathsf{exp}$ and $\mathcal{T} \mid \Delta \vdash \tau\ \mathsf{type}$. We omit explicit mention of $\mathcal{T}$ and $\mathcal{X}$ to avoid notational clutter.

The constructor formation judgement is analogous to the formation judgement for expression. The judgement $\Delta \vdash c :: \kappa$ states that, under hypotheses Δ, the constructor c is well-formed with kind κ. The expression formation judgement is standard, differing from $\mathcal{L}\{\rightarrow\forall\exists\}$ in that the forms of hypothesis in Δ is richer. Type formation is now subsumed by constructor formation; the judgement $\Delta \vdash \tau\ \mathsf{type}$ is synonymous with the judgement $\Delta \vdash \tau :: \mathtt{Type}$.

The judgement form of *definitional equality* is novel, and is characteristic of languages with higher kinds. Definitional equality specifies that two constructors are indistinguishable by virtue of the definitions of the concepts involved. As a case in point, it will turn out that the constructor $\mathtt{fst}(\langle c_1, c_2\rangle)$ is definitionally equivalent c_1, because it expresses directly the meaning of the first projection operation. The expression formation judgement respects definitional equality in the sense that if $e : \tau$ and τ is definitionaly equivalent to τ', then $e : \tau'$.

27.1.1 Constructor Formation

The constructor formation judgement, $\Delta \vdash c :: \kappa$, is inductively defined by the following rules:

$$\frac{}{\Delta, t :: \kappa \vdash t :: \kappa} \tag{27.1a}$$

$$\frac{}{\Delta \vdash \mathtt{arr} :: \mathtt{Arr(Type;Arr(Type;Type))}} \tag{27.1b}$$

$$\frac{}{\Delta \vdash \mathtt{all[}\kappa\mathtt{]} :: \mathtt{Arr(Arr(}\kappa\mathtt{;Type);Type)}} \tag{27.1c}$$

$$\frac{}{\Delta \vdash \mathtt{some[}\kappa\mathtt{]} :: \mathtt{Arr(Arr(}\kappa\mathtt{;Type);Type)}} \tag{27.1d}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{pair(}c_1\mathtt{;}c_2\mathtt{)} :: \mathtt{Prod(}\kappa_1\mathtt{;}\kappa_2\mathtt{)}} \tag{27.1e}$$

$$\frac{\Delta \vdash c :: \texttt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \texttt{fst}(c) :: \kappa_1} \tag{27.1f}$$

$$\frac{\Delta \vdash c :: \texttt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \texttt{snd}(c) :: \kappa_2} \tag{27.1g}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2}{\Delta \vdash \texttt{lambda[}\kappa_1\texttt{](}t\texttt{.}c_2\texttt{)} :: \texttt{Arr}(\kappa_1;\kappa_2)} \tag{27.1h}$$

$$\frac{\Delta \vdash c_1 :: \texttt{Arr}(\kappa_2;\kappa) \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \texttt{app}(c_1;c_2) :: \kappa} \tag{27.1i}$$

There is an evident correspondence between these rules and the rules for functions and products given in Chapters 15 and 17, except that we are working at the static level of constructors and kinds, rather than the dynamic level of expressions and types.

The constants arr, all[κ], and some[κ] all have functional kinds. The kind of arr is a curried function kind that specifies that the function type constructor takes two types as arguments, yielding a type. The kind of the quantifiers, Arr(Arr(κ;Type);Type), specifies that the body of a quantification over kind κ is a constructor-level function from κ to Type.

27.1.2 Definitional Equality

Definitional equality of well-formed constructors is defined to be the least congruence closed under the following rules:

$$\frac{\Delta \vdash \texttt{pair}(c_1;c_2) :: \texttt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \texttt{fst}(\texttt{pair}(c_1;c_2)) \equiv c_1 :: \kappa_1} \tag{27.2a}$$

$$\frac{\Delta \vdash \texttt{pair}(c_1;c_2) :: \texttt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \texttt{snd}(\texttt{pair}(c_1;c_2)) \equiv c_2 :: \kappa_2} \tag{27.2b}$$

$$\frac{\Delta \vdash c :: \texttt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \texttt{pair}(\texttt{fst}(c);\texttt{snd}(c)) \equiv c :: \texttt{Prod}(\kappa_1;\kappa_2)} \tag{27.2c}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \texttt{app}(\texttt{lambda[}\kappa\texttt{](}t\texttt{.}c_1\texttt{)};c_2) \equiv [c_2/t]c_1 :: \kappa_2} \tag{27.2d}$$

$$\frac{\Delta \vdash c_2 :: \texttt{Arr}(\kappa_1;\kappa_2) \quad t \,\#\, \Delta}{\Delta \vdash \texttt{lambda[}\kappa_1\texttt{](}t\texttt{.app(}c_2;t\texttt{))} \equiv c_2 :: \texttt{Arr}(\kappa_1;\kappa_2)} \tag{27.2e}$$

Rules (27.2a) and (27.2b) specify the meaning of the projection constructors when acting on a pair. Rule (27.2c) specifies that every constructor of product kind arises as a pair of projections. Rule (27.2d) specifies that the meaning of a function constructor is given by substitution of an argument into the function body. Rule (27.2e) specifies, moreover, that every constructor of function kind arises in this way.

The role of definitional equality is captured by the following rule of kind equivalence:

$$\frac{\Delta\ \Gamma \vdash e : \tau' \quad \Delta \vdash \tau \equiv \tau' :: \texttt{Type}}{\Delta\ \Gamma \vdash e : \tau} \tag{27.3}$$

In words, equivalent types (*i.e.*, constructors of kind Type) classify the same expressions.

27.2 Expression Formation

The rules for typing expressions are a straightforward generalization of those given in Chapters 25 and 26.

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{27.4a}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)} : \texttt{app(app(arr;}\tau_1\texttt{);}\tau_2\texttt{)}} \tag{27.4b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{app(app(arr;}\tau_2\texttt{);}\tau\texttt{)} \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} : \tau} \tag{27.4c}$$

$$\frac{\Delta, t :: \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \texttt{Lam[}\kappa\texttt{](}t\texttt{.}e\texttt{)} : \texttt{app(all[}\kappa\texttt{];lambda[}\kappa\texttt{](}t\texttt{.}\tau\texttt{))}} \tag{27.4d}$$

$$\frac{\Delta\ \Gamma \vdash e : \texttt{app(all[}\kappa\texttt{];}c'\texttt{)} \quad \Delta \vdash c :: \kappa}{\Delta\ \Gamma \vdash \texttt{App[}c\texttt{](}e\texttt{)} : \texttt{app(}c'\texttt{;}c\texttt{)}} \tag{27.4e}$$

$$\frac{\Delta \vdash c' :: \texttt{Arr(}\kappa\texttt{;Type)} \quad \Delta \vdash c :: \kappa \quad \Delta\ \Gamma \vdash e : \texttt{app(}c'\texttt{;}c\texttt{)}}{\Delta\ \Gamma \vdash \texttt{pack[}\kappa\texttt{;}c'\texttt{;}c\texttt{](}e\texttt{)} : \texttt{app(some[}\kappa\texttt{];}c'\texttt{)}} \tag{27.4f}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{app(some[}\kappa\texttt{];}c\texttt{)} \quad \Delta, t :: \kappa\ \Gamma, x : \texttt{app(}c\texttt{;}t\texttt{)} \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open[}\kappa\texttt{;}c\texttt{](}e_1\texttt{;}t\texttt{,}x\texttt{.}e_2\texttt{)} : \tau_2} \tag{27.4g}$$

27.3 Distinguishing Constructors from Types

The formulation of $\mathcal{L}\{\rightarrow, \forall_\kappa, \exists_\kappa\}$ identifies types with constructors of kind Type. Consequently, constructors of kind Type have a dual role:

1. As static data values that may be passed as arguments to polymorphic functions or bound into packages of existential type.

2. As classifiers of dynamic data values according to the typing rules for expressions.

The dual role of such constructors is apparent in our use of the meta-variables c and τ for constructors of kind `Type` in the static semantics.

These two roles can be separated in the syntax by making explicit the inclusion of constructors of kind `Type` into the class of types. The syntax of this variant, called $\mathcal{L}\{\texttt{typ}, \rightarrow, \forall_\kappa, \exists_\kappa\}$, is as follows:

Category	*Item*		*Abstract*	*Concrete*
Cons	c	$::=$	t	t
		$\mid$	$\texttt{arr}$	$\texttt{arr}$
		$\mid$	$\texttt{all[}\kappa\texttt{]}$	$\texttt{all[}\kappa\texttt{]}$
		$\mid$	$\texttt{some[}\kappa\texttt{]}$	$\texttt{some[}\kappa\texttt{]}$
		$\mid$	$\texttt{pair(}c_1\texttt{;}c_2\texttt{)}$	$\langle c_1, c_2\rangle$
		$\mid$	$\texttt{fst(}c\texttt{)}$	$\texttt{fst(}c\texttt{)}$
		$\mid$	$\texttt{snd(}c\texttt{)}$	$\texttt{snd(}c\texttt{)}$
		$\mid$	$\texttt{lambda[}\kappa\texttt{](}t\texttt{.}c\texttt{)}$	$\lambda\texttt{(}t\texttt{::}\kappa\texttt{.}c\texttt{)}$
		$\mid$	$\texttt{app(}c_1\texttt{;}c_2\texttt{)}$	$c_1\texttt{[}c_2\texttt{]}$
Type	τ	$::=$	$\texttt{typ(}c\texttt{)}$	$\hat{c}$
		$\mid$	$\texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$	$\tau_1 \rightarrow \tau_2$
		$\mid$	$\texttt{all[}\kappa\texttt{](}t\texttt{.}\tau\texttt{)}$	$\forall_\kappa\texttt{[}t\texttt{::}\kappa\texttt{.}\tau\texttt{]}$
		$\mid$	$\texttt{some[}\kappa\texttt{](}t\texttt{.}\tau\texttt{)}$	$\exists_\kappa\texttt{[}t\texttt{::}\kappa\texttt{.}\tau\texttt{]}$

There is a degree of redundancy in distinguishing constructors of kind `Type` from types, but the payoff is a clear separation of the two roles of types in $\mathcal{L}\{\rightarrow, \forall_\kappa, \exists_\kappa\}$: as classifiers of expressions and as arguments to polymorphic functions and as components of packages. The former is a purely static role, the latter two are dynamic.

The correspondence between the two roles of types is stated using the following axioms of definitional equality, stated with implicit contexts for brevity:

$$\frac{}{\texttt{typ(app(app(arr;}c_1\texttt{);}c_2\texttt{))} \equiv \texttt{arr(typ(}c_1\texttt{);typ(}c_2\texttt{))} :: \texttt{Type}} \qquad (27.5a)$$

$$\frac{}{\texttt{typ(app(all[}\kappa\texttt{];}c\texttt{))} \equiv \texttt{all[}\kappa\texttt{](}t\texttt{.typ(app(}c\texttt{;}t\texttt{)))} :: \texttt{Type}} \qquad (27.5b)$$

$$\frac{}{\texttt{typ(app(some[}\kappa\texttt{];}c\texttt{))} \equiv \texttt{some[}\kappa\texttt{](}t\texttt{.typ(app(}c\texttt{;}t\texttt{)))} :: \texttt{Type}} \qquad (27.5c)$$

These equivalences specify which types are designated by which constructors. For example, Rule (27.5a) specifies that the constant `arr`, when applied to constructors c_1 and c_2 of kind `Type`, designates the function space type between the types designated by c_1 and c_2, respectively. The other rules provide definitions for the constants designating the universal and existential quantifiers.

An advantage of $\mathcal{L}\{\texttt{typ}, \rightarrow, \forall_\kappa, \exists_\kappa\}$ is that it sheds light on the distinction between predicative and impredicative type quantification discussed in Chapter 25. The universal and existential quantifiers range over constructors of a specified kind. In $\mathcal{L}\{\rightarrow, \forall_\kappa, \exists_\kappa\}$ the kind $\texttt{Type}$ includes all types-as-classifiers, and hence quantification is impredicative. In $\mathcal{L}\{\texttt{typ}, \rightarrow, \forall_\kappa, \exists_\kappa\}$ we have a choice. We may either include representatives of the quantified types as constructors, in which case we obtain the impredicative fragment, or we may not include such representatives, in which case quantification is predicative. Put another way, the impredicative variant is a special case of the predicative fragment in which we ensure that quantified types have representatives as constructors.

27.4 Dynamic Semantics

Many languages admit a *type erasure* interpretation for which the transition relation of the dynamic semantics is insensitive to type information. For example, the dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ given in Chapter 16 enjoys such an interpretation. If we "erase" all the type information on an expression, we obtain the same sequence of transitions as if we had not done so. From the point of view of the dynamic semantics, the only role of the type annotations is to ensure that the original expression, as well as all of those expressions derived from it by the transition relation, are well-typed in the original source language. From an implementation point of view, these annotations may be erased, since we do not expect to type check the derivatives of an expression, only its original form as written by the programmer.

When type quantification is introduced, it appears that a type erasure interpretation interpretation is not available. For example, types appear as arguments to polymorphic functions and as components of packages, and during evaluation types are substituted for type variables within an expression. Consequently, it would appear that types play an essential role in the dynamic semantics, and hence cannot be erased prior to execution. However, we also notice that some occurrences of types are clearly negligible, such as the types attached to binding occurrences of variables. The distinction between these two forms of occurrences of types amounts to the distinction between types-as-data and types-as-classifiers. Constructors of any kind are a form of data that must be maintained and manipulated at execution time. Classifiers, on the other hand, never play a computationally significant role. Consequently, we may always erase classifiers from expressions prior to execution, but we may not always erase constructors, nor may we erase constructor abstractions or applications, nor packages or openings of packages. In short, constructors are a form of data that is manipulated at run-time, and hence cannot be eliminated from consideration.

On the other hand, a characteristic feature of $\mathcal{L}\{\rightarrow, \forall_\kappa, \exists_\kappa\}$ is that constructors play a passive role in the dynamic semantics. This property, which is related to parametricity (see Chapter 54), means that constructors are merely passed around as data items, but are never subject to any computational analysis such as dispatching on their form. In such languages it is feasible to identify all constructors of a kind at execution time, using a single token to serve as a dynamic for all elements of the kind. Some languages, however, permit non-parametric forms of computation on types, operations that distinguish types from one another based on their form (see, for example, Chapter 24). Such languages require a more sophisticated form of dynamic semantics to support run-time dispatch on the form of a type.

27.5 Exercises

Chapter 28

Indexed Families of Types

28.1 Type Families

28.2 Exercises

Part IX

Control Flow

Chapter 29

Abstract Machine for Control

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.

In this chapter we introduce an abstract machine, $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. The purpose of this machine is to make control flow explicit by introducing a control stack that maintains a record of the pending sub-computations of a computation. We then prove the equivalence of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ with the structural operational semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$.

29.1 Machine Definition

A state, s, of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ 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 $k \triangleright e$ corresponds to the evaluation of a closed expression, e, relative to a control stack, k.

2. A *return* state of the form $k \triangleleft e$, where e val, 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*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{29.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f\,;k\ \mathsf{stack}} \tag{29.1b}$$

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{29.2a}$$

$$\frac{}{\mathtt{ifz}(-;e_1;x.e_2)\ \mathsf{frame}} \tag{29.2b}$$

$$\frac{}{\mathtt{ap}(-;e_2)\ \mathsf{frame}} \tag{29.2c}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{ap}(e_1;-)\ \mathsf{frame}} \tag{29.2d}$$

The frames correspond to rules with transition premises in the dynamic semantics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. Thus, instead of relying on the structure of the transition derivation to maintain a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgement between states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \triangleright \mathtt{z} \mapsto k \triangleleft \mathtt{z}} \tag{29.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \mapsto \mathtt{s}(-)\,;k \triangleright e} \tag{29.3b}$$

$$\frac{}{\mathtt{s}(-)\,;k \triangleleft e \mapsto k \triangleleft \mathtt{s}(e)} \tag{29.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor, and evaluate e; when that returns with e', we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}(e;e_1;x.e_2) \mapsto \mathtt{ifz}(-;e_1;x.e_2)\,;k \triangleright e} \tag{29.4a}$$

$$\frac{}{\texttt{ifz(}-\texttt{;}e_1\texttt{;}x\texttt{.}e_2\texttt{)}\texttt{;}k \lhd \texttt{z} \mapsto k \rhd e_1} \tag{29.4b}$$

$$\frac{}{\texttt{ifz(}-\texttt{;}e_1\texttt{;}x\texttt{.}e_2\texttt{)}\texttt{;}k \lhd \texttt{s(}e\texttt{)} \mapsto k \rhd [e/x]e_2} \tag{29.4c}$$

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined, we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions and recursion.

$$\frac{}{k \rhd \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \mapsto k \lhd \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}} \tag{29.5a}$$

$$\frac{}{k \rhd \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}-\texttt{;}e_2\texttt{)}\texttt{;}k \rhd e_1} \tag{29.5b}$$

$$\frac{}{\texttt{ap(}-\texttt{;}e_2\texttt{)}\texttt{;}k \lhd e_1 \mapsto \texttt{ap(}e_1\texttt{;}-\texttt{)}\texttt{;}k \rhd e_2} \tag{29.5c}$$

$$\frac{e_1 = \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}}{\texttt{ap(}e_1\texttt{;}-\texttt{)}\texttt{;}k \lhd e_2 \mapsto k \rhd [e_2/x]e} \tag{29.5d}$$

$$\frac{}{k \rhd \texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)} \mapsto k \rhd [\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)}/x]e} \tag{29.5e}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated. Note that evaluation of general recursion requires no stack space! (But see Chapter 44 for more on evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\frac{}{\epsilon \rhd e \ \mathsf{initial}} \tag{29.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \lhd e \ \mathsf{final}} \tag{29.6b}$$

29.2 Safety

To define and prove safety for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ requires that we introduce a new typing judgement, $k : \tau$, stating that the stack k expects a value of type τ. This judgement is inductively defined by the following rules:

$$\frac{}{\epsilon : \tau} \tag{29.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{f\texttt{;}k : \tau} \tag{29.7b}$$

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\frac{}{\texttt{s(-)} : \texttt{nat} \Rightarrow \texttt{nat}} \tag{29.8a}$$

$$\frac{e_1 : \tau \quad x : \texttt{nat} \vdash e_2 : \tau}{\texttt{ifz(}-\texttt{;}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \texttt{nat} \Rightarrow \tau} \tag{29.8b}$$

$$\frac{e_2 : \tau_2}{\texttt{ap(}-\texttt{;}e_2\texttt{)} : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)} \Rightarrow \tau} \tag{29.8c}$$

$$\frac{e_1 : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)} \quad e_1\ \mathsf{val}}{\texttt{ap(}e_1\texttt{;}-\texttt{)} : \tau_2 \Rightarrow \tau} \tag{29.8d}$$

The two forms of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e\ \mathsf{ok}} \tag{29.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e\ \mathsf{val}}{k \triangleleft e\ \mathsf{ok}} \tag{29.9b}$$

We leave the proof of safety of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ as an exercise.

Theorem 29.1 (Safety). 1. *If s ok and $s \mapsto s'$, then s' ok.*

2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

29.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ correctly implements $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$. If we evaluate a given expression, e, using $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, and *vice versa*?

Answering this question decomposes into two conditions relating $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ to $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$:

Completeness If $e \mapsto^* e'$, where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$.

Soundness If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness it is natural to consider a proof by induction on the definition of multistep transition, which reduces the theorem to the following two lemmas:

1. If e val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e$.

2. If $e \mapsto e'$, then, for every v val, if $\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$.

The first can be proved easily by induction on the structure of e. The second requires an inductive analysis of the derivation of $e \mapsto e'$, giving rise to two complications that must be accounted for in the proof. The first complication is that we cannot restrict attention to the empty stack, for if e is, say, $\mathtt{ap}(e_1;e_2)$, then the first step of the machine is

$$\epsilon \triangleright \mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(-;e_2);\epsilon \triangleright e_1,$$

and so we must consider evaluation of e_1 on a non-empty stack.

A natural generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \mathtt{ap}(e_1;e_2)$, $e' = \mathtt{ap}(e_1';e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \mathtt{ap}(e_1';e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \mathtt{ap}(e_1';e_2) \mapsto \mathtt{ap}(-;e_2);k \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a v_1 such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

This means that we must consider the ultimate value of each sub-expression of an expression in order to complete the proof. This information is provided by the evaluation semantics described in Chapter 12, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 29.2. *If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 12.2 on page 86 for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, which states that $e \Downarrow v$ iff $e \mapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ machine state as encoding an expression, and show that $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ transitions under this encoding.

Specifically, we define a judgement, $s \looparrowright e$, stating that state s "unravels to" expression e. It will turn out that for initial states, $s = \epsilon \triangleright e$, and final states, $s = \epsilon \triangleleft e$, we have $s \looparrowright e$. Then we show that if $s \mapsto^* s'$, where s' final, $s \looparrowright e$, and $s' \looparrowright e'$, then e' val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \looparrowright e$ and s final, then e val.
2. If $s \mapsto s'$, $s \looparrowright e$, $s' \looparrowright e'$, and $e' \mapsto^* v$, where v val, then $e \mapsto^* v$.

The first is quite simple, we need only observe that the unravelling of a final state is a value. For the second, it is enough to show the following lemma.

Lemma 29.3. *If $s \mapsto s'$, $s \looparrowright e$, and $s' \looparrowright e'$, then $e \mapsto^* e'$.*

The remainder of this section is devoted to the proofs of these lemmas.

29.3.1 Completeness

Proof of Lemma 29.2 on the preceding page. The proof is by induction on an evaluation semantics for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \texttt{lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{)} \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \tag{29.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto^* k \triangleleft v$. Applying each of the three inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned}
k \triangleright \texttt{ap(}e_1\texttt{;}e_2\texttt{)} &\mapsto \texttt{ap(}-\texttt{;}e_2\texttt{)}\texttt{;}k \triangleright e_1 && (29.11)\\
&\mapsto^* \texttt{ap(}-\texttt{;}e_2\texttt{)}\texttt{;}k \triangleleft \texttt{lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{)} && (29.12)\\
&\mapsto \texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{);}-\texttt{)}\texttt{;}k \triangleright e_2 && (29.13)\\
&\mapsto^* \texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{);}-\texttt{)}\texttt{;}k \triangleleft v_2 && (29.14)\\
&\mapsto k \triangleright [v_2/x]e && (29.15)\\
&\mapsto^* k \triangleleft v. && (29.16)
\end{aligned}$$

The other cases of the proof are handled similarly. □

29.3.2 Soundness

The judgement $s \looparrowright e'$, where s is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgement $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \looparrowright e'} \tag{29.17a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \looparrowright e'} \tag{29.17b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{29.18a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{\mathtt{s}(-)\,;k \bowtie e = e'} \tag{29.18b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1;e_2;x\,.\,e_3) = e'}{\mathtt{ifz}(-;e_2;x\,.\,e_3)\,;k \bowtie e_1 = e'} \tag{29.18c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1;e_2) = e}{\mathtt{ap}(-;e_2)\,;k \bowtie e_1 = e} \tag{29.18d}$$

$$\frac{k \bowtie \mathtt{ap}(e_1;e_2) = e}{\mathtt{ap}(e_1;-)\,;k \bowtie e_2 = e} \tag{29.18e}$$

These judgements both define total functions.

Lemma 29.4. *The judgement $s \looparrowright e$ has mode $(\forall, \exists!)$, and the judgement $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique e' such that $k \bowtie e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

Lemma 29.5. *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

Proof. The proof is by rule induction on the transition $e \mapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, k.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1;e_2)$, $e' = \mathtt{ap}(e_1';e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (29.18) that $\mathtt{ap}(-;e_2)\,;k \bowtie e_1 = d$ and $\mathtt{ap}(-;e_2)\,;k \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x.e);e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = \mathtt{ap}(-;c_2)\,;k'$. It follows from Rules (29.18) that $k' \bowtie \mathtt{ap}(e;c_2) = d$ and $k' \bowtie \mathtt{ap}(e';c_2) = d'$. But by the SOS rules $\mathtt{ap}(e;c_2) \mapsto \mathtt{ap}(e';c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 29.3 on page 228.

Proof of Lemma 29.3 on page 228. The proof is by case analysis on the transitions of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$.

Suppose that $s = k \triangleright \texttt{s(}e\texttt{)}$ and $s' = \texttt{s(}-\texttt{)} \triangleright e$. Note that $k \bowtie \texttt{s(}e\texttt{)} = e'$ iff $\texttt{s(}-\texttt{)}\,\texttt{;}k \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = \texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e_1\texttt{);}-\texttt{);}k \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $\texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e_1\texttt{);}-\texttt{);}k \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} = e'$. The result follows from Lemma 29.5 on the previous page.

□

29.4 Exercises

Chapter 30

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.

30.1 Failures

To begin with let us consider a simple control mechanism, 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 $\mathcal{L}\{\rightarrow\}$ to include failures:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{fail[}\tau\texttt{]}$	$\texttt{fail}$
		$\mid$	$\texttt{catch(}e_1\texttt{;}e_2\texttt{)}$	$\texttt{try}\,e_1\,\texttt{ow}\,e_2$

The expression $\texttt{fail[}\tau\texttt{]}$ aborts the current evaluation. The expression $\texttt{catch(}e_1\texttt{;}e_2\texttt{)}$

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 failures is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail[}\tau\texttt{]} : \tau} \tag{30.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch(}e_1\texttt{;}e_2\texttt{)} : \tau} \tag{30.1b}$$

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.

The dynamic semantics of failures uses 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 using the abstract machine $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ from Chapter 29, because it makes the control stack explicit. We introduce a new form of state, $k \blacktriangleleft$, which passes a failure to the stack, k, in search of the nearest enclosing handler. A state of the form $\epsilon \blacktriangleleft$ is considered final, rather than stuck; it corresponds to an "uncaught failure" making its way to the topic of the stack.

The set of frames is extended with the following additional rule:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{catch(}-\texttt{;}e_2\texttt{)} \ \mathsf{frame}} \tag{30.2}$$

The transition rules of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail[}\tau\texttt{]} \mapsto k \blacktriangleleft} \tag{30.3a}$$

$$\frac{}{k \triangleright \texttt{catch(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{catch(}-\texttt{;}e_2\texttt{)}\texttt{;}k \triangleright e_1} \tag{30.3b}$$

$$\frac{}{\texttt{catch(}-\texttt{;}e_2\texttt{)}\texttt{;}k \triangleleft v \mapsto k \triangleleft v} \tag{30.3c}$$

$$\frac{}{\texttt{catch(}-\texttt{;}e_2\texttt{)}\texttt{;}k \blacktriangleleft \ \mapsto k \triangleright e_2} \tag{30.3d}$$

$$\frac{(f \neq \texttt{catch(}-\texttt{;}e_2\texttt{)})}{f\texttt{;}k \blacktriangleleft \ \mapsto k \blacktriangleleft} \tag{30.3e}$$

Evaluating fail[τ] propagates a failure up the stack. Evaluating catch(e_1;e_2) consists of pushing the handler onto the control stack and evaluating e_1. If a value is propagated to the handler, the handler is removed and the value continues to propagate upwards. If a failure is propagated to the handler, the stored expression is evaluated with the handler removed from the control stack. All other frames propagate failures.

The definition of initial state remains the same as for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{30.4a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}} \tag{30.4b}$$

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 29 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 result in an uncaught failure!

Theorem 30.1 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

30.2 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:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{raise}[\tau](e)$	$\texttt{raise}(\tau)e$
		$\mid$	$\texttt{handle}(e_1;x.e_2)$	$\texttt{try}\,e_1\,\texttt{ow}\,x \Rightarrow e_2$

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 dynamic semantics of exceptions is a mild generalization of that of failures given in Section 30.1 on page 231. The failure state, $k \blacktriangleleft$, is extended to permit passing a value along with the failure, $k \blacktriangleleft e$, where e val. Stack frames include these two forms:

$$\frac{}{\texttt{raise[}\tau\texttt{](}-\texttt{)}\ \mathsf{frame}} \tag{30.5a}$$

$$\frac{}{\texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}\ \mathsf{frame}} \tag{30.5b}$$

The rules for evaluating exceptions are as follows:

$$\frac{}{k \triangleright \texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{raise[}\tau\texttt{](}-\texttt{)}\texttt{;}k \triangleright e} \tag{30.6a}$$

$$\frac{}{\texttt{raise[}\tau\texttt{](}-\texttt{)}\texttt{;}k \triangleleft e \mapsto k \blacktriangleleft e} \tag{30.6b}$$

$$\frac{}{\texttt{raise[}\tau\texttt{](}-\texttt{)}\texttt{;}k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.6c}$$

$$\frac{}{k \triangleright \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto \texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}\texttt{;}k \triangleright e_1} \tag{30.6d}$$

$$\frac{}{\texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}\texttt{;}k \triangleleft e \mapsto k \triangleleft e} \tag{30.6e}$$

$$\frac{}{\texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}\texttt{;}k \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{30.6f}$$

$$\frac{(f \neq \texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)})}{f\texttt{;}k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.6g}$$

The static semantics of exceptions generalizes that of failures.

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} : \tau} \tag{30.7a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \tau} \tag{30.7b}$$

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 `str`. Then the value associated with an exception is a string that the reason for the failure. For example, one might write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. While this might be reasonable for uncaught exceptions, it is unreasonable for those that may be handled. The handler would have to parse the associated string to determine what happened and how to respond! Another well-known approach is to choose τ_{exn} to be nat, with the associated value being an "error number" according to some convention. This, too, is obviously rather primitive and error-prone, and does not support associating exception-specific data with the failure.

A more practical choice would be to distinguish a labelled sum type of the form

$$\tau_{exn} = \texttt{[div:unit,fnf:string,...]}.$$

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 ⟨⟩ ⇒ $e_{div}$
  | fnf $s$ ⇒ $e_{fnf}$
  | ... }
```

This code closely resembles the exception mechanisms found in many languages.

A significant complication remains. The type τ_{exn} must be specified on a *per-language* basis to ensure that program fragments may be combined sensibly with one another. But having to choose a single, fixed labelled sum type to serve as the type of exceptions for all possible programs is clearly absurd! Although certain low-level exceptions, such as division by zero, might reasonably be included in any program, we expect in general that the choice of exceptions is specific to the task at hand, and ought to be chosen by the programmer. This is something of a dilemma, because we must choose τ_{exn} once for all programs written in the language, yet we also expect that programmers may declare their own exceptions.

The way out of this dilemma is to define τ_{exn} to be an *extensible* labelled sum type, rather than a *fixed* labelled sum type. An extensible sum is one that permits new tags to be created *dynamically* so that the collection of possible tags on values of the type is not fixed statically, but only at runtime. The concept of extensible sum has applications beyond their use as the type of values associated with exceptions. We will discuss this type in detail in Chapter 38.

30.3 Exercises

Chapter 31

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.

31.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. The introduction 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 corresponding elimination form is $\texttt{throw}[\tau](e_1;e_2)$, which restores the value of e_1 to the control stack that is the value of e_2.[1]

To illustrate the use of these primitives, consider the problem of multiplying the first n elements of an infinite sequence q of natural numbers, where q is represented by a function of type $\texttt{nat} \rightarrow \texttt{nat}$. If zero occurs among the first n elements, we would like to effect an “early return” with the value zero, rather than perform the remaining multiplications. This problem can be solved using exceptions (we leave this as an exercise), but

[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 $\texttt{throw}[\tau](e_1;e_2)$, write $\texttt{SMLofNJ.Cont.throw } e_2\ e_1$.

we will give a solution that uses continuations in preparation for what follows.

Here is the solution in $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, without short-cutting:

```
fix ms is
  λ q : nat ⇀ nat.
    λ n : nat.
      case n {
        z ⇒ s(z)
      | s(n') ⇒ (q z) × (ms (q ∘ succ) n')
      }
```

The recursive call composes q with the successor function to shift the sequence by one step.

Here is the version with short-cutting:

```
λ q : nat ⇀ nat.
  λ n : nat.
    letcc ret : nat cont in
      let ms be
        fix ms is
          λ q : nat ⇀ nat.
            λ n : nat.
              case n {
                z ⇒ s(z)
              | s(n') ⇒
                case q z {
                  z ⇒ throw z to ret
                | s(n') ⇒ (q z) × (ms (q ∘ succ) n')
                }
              }
      in
        ms q n
```

The `letcc` binds the return point of the function to the variable `ret` for use within the main loop of the computation. If zero is encountered, control is thrown to `ret`, effecting an early return with the value zero.

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!

31.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\rightarrow\}$ expressions with these additional forms:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{cont}(\tau)$	$\tau\,\texttt{cont}$
Expr	e	::=	$\texttt{letcc}[\tau](x.e)$	$\texttt{letcc}\,x\,\texttt{in}\,e$
		|	$\texttt{throw}[\tau](e_1;e_2)$	$\texttt{throw}\,e_1\,\texttt{to}\,e_2$
		|	$\texttt{cont}(k)$	

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:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{31.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1;e_2) : \tau'} \tag{31.1b}$$

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:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{31.2}$$

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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2\ \mathsf{exp}}{\texttt{throw}[\tau](-;e_2)\ \mathsf{frame}} \tag{31.3a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{throw}[\tau](e_1;-)\ \mathsf{frame}} \tag{31.3b}$$

Every reified control stack is a value:

$$\frac{k\ \mathsf{stack}}{\texttt{cont}(k)\ \mathsf{val}} \tag{31.4}$$

The transition rules for the continuation constructs are as follows:

$$\frac{}{k \triangleright \texttt{letcc}[\tau](x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e} \tag{31.5a}$$

$$\frac{}{\texttt{throw}[\tau](v;-);k \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v} \tag{31.5b}$$

$$\frac{}{k \triangleright \texttt{throw}[\tau](e_1;e_2) \mapsto \texttt{throw}[\tau](-;e_2);k \triangleright e_1} \tag{31.5c}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{throw}[\tau](-;e_2);k \triangleleft e_1 \mapsto \texttt{throw}[\tau](e_1;-);k \triangleright e_2} \tag{31.5d}$$

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 $\mathcal{L}\{\rightarrow\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ given in Chapter 29.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \texttt{cont}(\tau)}{\texttt{throw}[\tau](-;e_2) : \tau \Rightarrow \tau'} \tag{31.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\texttt{throw}[\tau](e_1;-) : \texttt{cont}(\tau) \Rightarrow \tau'} \tag{31.6b}$$

The rest of the definitions remain as in Chapter 29.

Lemma 31.1 (Canonical Forms). *If e : $\texttt{cont}(\tau)$ and e val, then $e = \texttt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 31.2 (Safety).

1. *If s ok and $s \mapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

31.3 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

   $$\texttt{ap}(f;-)\texttt{;throw[}\tau\texttt{]}(-;k)\texttt{;}\epsilon$$

   This stack is returned from `compose`. Next, simulate the behavior of throwing a value v' to this continuation. Observe that the stack is reinstated and that v' is passed to it.

Part X

Propositions and Types

Chapter 32

The Curry-Howard Correspondence

The *Curry-Howard Correspondence* is a central organzing principle of type theory. Roughly speaking, the Curry-Howard Correspondence 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 correspondence 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 Correspondences, 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.

32.1 Constructive Logic

32.1.1 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 has neither a proof nor a refutation at the time of this writing, so we cannot at present affirm or deny its truth.

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 $\phi\ \mathsf{prop}$ and $\phi\ \mathsf{true}$ 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 $\phi\ \mathsf{true}$. 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:

$$\frac{}{\Gamma, \phi\ \mathsf{true} \vdash \phi\ \mathsf{true}} \tag{32.1a}$$

$$\frac{\Gamma \vdash \phi\ \mathsf{true} \quad \Gamma, \phi\ \mathsf{true} \vdash \psi\ \mathsf{true}}{\Gamma \vdash \psi\ \mathsf{true}} \tag{32.1b}$$

$$\frac{\Gamma \vdash \psi\ \mathsf{true}}{\Gamma, \phi\ \mathsf{true} \vdash \psi\ \mathsf{true}} \tag{32.1c}$$

$$\frac{\Gamma, \phi\ \mathsf{true}, \phi\ \mathsf{true} \vdash \theta\ \mathsf{true}}{\Gamma, \phi\ \mathsf{true} \vdash \theta\ \mathsf{true}} \tag{32.1d}$$

$$\frac{\Gamma, \psi\ \mathsf{true}, \phi\ \mathsf{true}, \Gamma' \vdash \theta\ \mathsf{true}}{\Gamma, \phi\ \mathsf{true}, \psi\ \mathsf{true}, \Gamma' \vdash \theta\ \mathsf{true}} \tag{32.1e}$$

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.

32.1.2 Propositional Logic

The syntax of propositional logic is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Prop	ϕ	$::=$	`true`	$\top$
		$\mid$	`false`	$\bot$
		$\mid$	`and(`ϕ_1`;`ϕ_2`)`	$\phi_1 \wedge \phi_2$
		$\mid$	`or(`ϕ_1`;`ϕ_2`)`	$\phi_1 \vee \phi_2$
		$\mid$	`imp(`ϕ_1`;`ϕ_2`)`	$\phi_1 \supset \phi_2$

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.

Truth Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it.

$$\frac{}{\Gamma \vdash \top\ \mathsf{true}} \tag{32.2a}$$

(no elimination rule)

(32.2b)

Conjunction Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi\ \mathsf{true} \quad \Gamma \vdash \psi\ \mathsf{true}}{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}} \tag{32.3a}$$

$$\frac{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Gamma \vdash \phi\ \mathsf{true}} \tag{32.3b}$$

$$\frac{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Gamma \vdash \psi\ \mathsf{true}} \tag{32.3c}$$

Implication Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi \ \mathsf{true} \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \supset \psi \ \mathsf{true}} \tag{32.4a}$$

$$\frac{\Gamma \vdash \phi \supset \psi \ \mathsf{true} \quad \Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{32.4b}$$

Falsehood Falsehood expresses the trivially false (refutable) proposition.

(no introduction rule) (32.5a)

$$\frac{\Gamma \vdash \bot \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{32.5b}$$

Disjunction Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{32.6a}$$

$$\frac{\Gamma \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{32.6b}$$

$$\frac{\Gamma \vdash \phi \vee \psi \ \mathsf{true} \quad \Gamma, \phi \ \mathsf{true} \vdash \theta \ \mathsf{true} \quad \Gamma, \psi \ \mathsf{true} \vdash \theta \ \mathsf{true}}{\Gamma \vdash \theta \ \mathsf{true}} \tag{32.6c}$$

32.1.3 Explicit Proofs

The key to the Curry-Howard Correspondence 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.

$$\frac{}{\Gamma \vdash \mathtt{trueI} : \top} \tag{32.7a}$$

$$\frac{\Gamma \vdash p : \phi \quad \Gamma \vdash q : \psi}{\Gamma \vdash \texttt{andI}(p;q) : \phi \wedge \psi} \tag{32.7b}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{andEl}(p) : \phi} \tag{32.7c}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{andEr}(p) : \psi} \tag{32.7d}$$

$$\frac{\Gamma, x : \phi \vdash p : \psi}{\Gamma \vdash \texttt{impI}[\phi](x.p) : \phi \supset \psi} \tag{32.7e}$$

$$\frac{\Gamma \vdash p : \phi \supset \psi \quad \Gamma \vdash q : \phi}{\Gamma \vdash \texttt{impE}(p;q) : \psi} \tag{32.7f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \texttt{falseE}[\phi](p) : \phi} \tag{32.7g}$$

$$\frac{\Gamma \vdash p : \phi}{\Gamma \vdash \texttt{orIl}[\psi](p) : \phi \vee \psi} \tag{32.7h}$$

$$\frac{\Gamma \vdash p : \psi}{\Gamma \vdash \texttt{orIr}[\phi](p) : \phi \vee \psi} \tag{32.7i}$$

$$\frac{\Gamma \vdash p : \phi \vee \psi \quad \Gamma, x : \phi \vdash q : \theta \quad \Gamma, y : \psi \vdash r : \theta}{\Gamma \vdash \texttt{orE}[\phi;\psi](p;x.q;y.r) : \theta} \tag{32.7j}$$

32.2 Propositions as Types

The Curry-Howard Correspondence emphasizes the close relationship between propositions and their proofs on one hand, and types and programs 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*
$\texttt{trueI}$	$\texttt{triv}$
$\texttt{falseE}[\phi](p)$	$\texttt{abort}[\phi^*](p^*)$
$\texttt{andI}(p;q)$	$\texttt{pair}(p^*;q^*)$
$\texttt{andEl}(p)$	$\texttt{fst}(p^*)$
$\texttt{andEr}(p)$	$\texttt{snd}(p^*)$
$\texttt{impI}[\phi](x.p)$	$\texttt{lam}[\phi^*](x.p^*)$
$\texttt{impE}(p;q)$	$\texttt{ap}(p^*;q^*)$
$\texttt{orIl}[\psi](p)$	$\texttt{in[l]}[\psi^*](p^*)$
$\texttt{orIr}[\phi](p)$	$\texttt{in[r]}[\phi^*](p^*)$
$\texttt{orE}[\phi;\psi](p;x.q;y.r)$	$\texttt{case}(p^*;x.q^*;y.r^*)$

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 32.1 (Curry-Howard Correspondence)**.**

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:

$$\begin{array}{rcl}
\texttt{andEl}(\texttt{andI}(p;q)) & \mapsto & p \\
\texttt{andEr}(\texttt{andI}(p;q)) & \mapsto & q \\
\texttt{impE}(\texttt{impI}[\phi](x.q);p) & \mapsto & [p/x]q \\
\texttt{orE}[\phi;\psi](\texttt{orIl}[\psi](p);x.q;y.r) & \mapsto & [p/x]q \\
\texttt{orE}[\phi;\psi](\texttt{orIr}[\phi](p);x.q;y.r) & \mapsto & [p/y]r
\end{array}$$

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.

The correspondence given here does not extend to general recursion, which would correspond to admitting a circular proof, one whose justification relies on its own presumed truth. Unsurprisingly, permitting circular proofs renders the logic inconsistent—one can derive a "proof" of any proposition simply by appealing to itself! However, this does not mean that there is no logical account of general recursion. Rather, it simply says that self-reference cannot be permitted as evidence for the *truth* of a proposition. But one could well imagine using self-reference in connection with a relaxed notion of truth corresponding to the isolation of effects in a monad in a programming language.

32.3 Exercises

Chapter 33

Classical Proofs and Control Operators

In Chapter 32 we saw that constructive logic is a logic of positive information in that the meaning of the judgement $\phi\ \mathsf{true}$ is that there exists a proof of ϕ. A refutation of a proposition ϕ consists of a proof of the hypothetical judgement $\phi\ \mathsf{true} \vdash \perp\ \mathsf{true}$, 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, $\phi \vee \neg\phi\ \mathsf{true}$.

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 a solution (though we may not know it), 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 possession

[1] Or, in the words of the brilliant military strategist Donald von Rumsfeld, the absence of evidence is not evidence of absence.

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.

33.1 Classical 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.

We will consider hypothetical judgements in which hypotheses have either of the first two forms; we will have no need of a hypothesis of the third form. Up to permutation, then, 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 axiomatize classical logic directly in terms of these judgement forms, we will instead give an axiomatization in which proof terms are made explicit at the outset. The proof-explicit form of the three categorical judgements of classical logic are as follows:

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.

We will consider hypothetical judgements of the form (up to permutation of hypotheses)

$$\underbrace{u_1 \div \phi_1, \ldots, u_m \div \phi_m}_{\Delta}; \underbrace{x_1 : \psi_1, \ldots, x_n : \psi_n}_{\Gamma} \vdash J,$$

where J is any of the three categorical judgements in explicit form.

Statics

A contradiction arises from the conflict between a proof and a refutation:

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash k \mathbin{\#} p} \tag{33.1a}$$

The reflexivity rules capture the meaning of hypotheses:

$$\frac{}{\Delta, u \div \phi;\Gamma \vdash u \div \phi} \tag{33.1b}$$

$$\frac{}{\Delta;\Gamma, x : \psi \vdash x : \phi} \tag{33.1c}$$

Truth and falsity are complementary:

$$\frac{\Delta, u \div \phi;\Gamma \vdash k \mathbin{\#} p}{\Delta;\Gamma \vdash \texttt{ccr}(u \div \phi\,.\,k \mathbin{\#} p) : \phi} \tag{33.1d}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash k \mathbin{\#} p}{\Delta;\Gamma \vdash \texttt{ccp}(x : \phi\,.\,k \mathbin{\#} p) \div \phi} \tag{33.1e}$$

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.

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \top} \tag{33.1f}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{abort} \div \bot} \tag{33.1g}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash q : \psi}{\Delta;\Gamma \vdash \langle p, q\rangle : \phi \wedge \psi} \tag{33.1h}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi}{\Delta;\Gamma \vdash \texttt{fst;}k \div \phi \wedge \psi} \tag{33.1i}$$

$$\frac{\Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \texttt{snd;}k \div \phi \wedge \psi} \tag{33.1j}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash p : \psi}{\Delta;\Gamma \vdash \lambda\texttt{(}x : \phi.\, p\texttt{)} : \phi \supset \psi} \tag{33.1k}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \texttt{app(}p\texttt{);}k \div \phi \supset \psi} \tag{33.1l}$$

$$\frac{\Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash \texttt{inl(}p\texttt{)} : \phi \vee \psi} \tag{33.1m}$$

$$\frac{\Delta; \Gamma \vdash p : \psi}{\Delta; \Gamma \vdash \texttt{inr}(p) : \phi \vee \psi} \tag{33.1n}$$

$$\frac{\Delta; \Gamma \vdash k \div \phi \quad \Delta; \Gamma \vdash l \div \psi}{\Delta; \Gamma \vdash \texttt{case}(k; l) \div \phi \vee \psi} \tag{33.1o}$$

$$\frac{\Delta; \Gamma \vdash k \div \phi}{\Delta; \Gamma \vdash \texttt{not}(k) : \neg\phi} \tag{33.1p}$$

$$\frac{\Delta; \Gamma \vdash p : \phi}{\Delta; \Gamma \vdash \texttt{not}(p) \div \neg\phi} \tag{33.1q}$$

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.

$$\texttt{fst};k \# \langle p, q \rangle \mapsto k \# p \tag{33.2a}$$

$$\texttt{snd};k \# \langle p, q \rangle \mapsto k \# q \tag{33.2b}$$

$$\texttt{case}(k; l) \# \texttt{inl}(p) \mapsto k \# p \tag{33.2c}$$

$$\texttt{case}(k; l) \# \texttt{inr}(q) \mapsto l \# q \tag{33.2d}$$

$$\texttt{app}(p);k \# \lambda(x{:}\phi.\, q) \mapsto k \# [p/x]q \tag{33.2e}$$

$$\texttt{not}(p) \# \texttt{not}(k) \mapsto k \# p \tag{33.2f}$$

The symmetry of the transition rule for negation is particularly elegant.

Here are the rules for the generic primitives relating truth and falsity.

$$\texttt{ccp}(x : \phi . k \# p) \# q \mapsto [q/x]k \# [q/x]p \tag{33.2g}$$

$$k \# \texttt{ccr}(u \div \phi . l \# p) \mapsto [k/u]l \# [k/u]p \tag{33.2h}$$

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 rules (33.2g) to (33.2h) overlap in the sense that there are two possible transitions for a state of the form

$$\mathtt{ccp}(x:\phi\,.\,k\;\#\;p)\;\#\;\mathtt{ccr}(u\div\phi\,.\,l\;\#\;q).$$

This state may transition either to the state

$$[r/x]k\;\#\;[r/x]p,$$

where r is $\mathtt{ccr}(u\div\phi\,.\,l\;\#\;q)$, or to the state

$$[m/u]l\;\#\;[m/u]q,$$

where m is $\mathtt{ccp}(x:\phi\,.\,k\;\#\;p)$, 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 `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 # `halt` as initial, and also final, provided that k is not a "ccp" instruction.

33.2 Exercises

Part XI

Subtypes

Chapter 34

Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) 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.

The subsumption principle relaxes the strictures of a type system to permit values of one type to be treated as values of another.

Experience shows that the subsumption principle, while useful as a general guide, can be tricky to apply correctly in practice. The key to getting it right is the principle of introduction and elimination described in Chapter 13. To determine whether a candidate subtyping relationship is sensible, it suffices to consider whether every *introductory* form of the subtype can be safely manipulated by every *eliminatory* form of the supertype. A subtyping principle makes sense only if it passes this test; the proof of the type safety theorem for a given subtyping relation ensures that this is the case.

A good way to get a subtyping principle wrong is to think of a type merely as a set of values (generated by introductory forms), and to consider whether every value of the subtype can also be considered to be a value of the supertype. The intuition behind this approach is to think of subtyping as akin to the subset relation in ordinary mathematics. But this can lead to serious errors, because it fails to take account of the operations (eliminatory forms) that one can perform on values of the supertype. It is not enough to think only of the introductory forms; one must also think of the eliminatory forms. Subtyping is a matter of *behavior*, which is a dynamic criterion, rather than *containment*, which is a static criterion.

34.1 Subsumption

A *subtyping judgement* has the form $\sigma <: \tau$, and states that σ is a subtype of τ. At a minimum we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{34.1a}$$

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \tag{34.1b}$$

In practice we either tacitly include these rules as primitive, or prove that they are admissible for a given set of subtyping rules.

The point of a subtyping relation is to enlarge the set of well-typed programs, which is achieved by the *subsumption rule*:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{34.2}$$

In contrast to most other typing rules, the rule of subsumption is *not* syntax-directed, because it does not constrain the form of e. That is, the subsumption rule may be applied to *any* form of expression. In particular, to show that $e : \tau$, we have two choices: either apply the rule appropriate to the particular form of e, or apply the subsumption rule, checking that $e : \sigma$ and $\sigma <: \tau$.

34.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 34.4 on page 268 we will examine some of these in more detail from the point of view of type safety.

34.2.1 Numbers

For languages with numeric types, our mathematical experience suggests subtyping relationships among them. For example, in a language with types `int`, `rat`, and `real`, representing, respectively, the integers, the rationals, and the reals, it is tempting to postulate the subtyping relationships

$$\texttt{int} <: \texttt{rat} <: \texttt{real}$$

by analogy with the set containments

$$\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$$

familiar from mathematical experience.

But are these subtyping relationships sensible? The answer depends on the representations and interpretations of these types! Even in mathematics, the containments just mentioned are usually not quite true—or are true only in a somewhat generalized sense. For example, rational numbers are often represented as ordered pairs (m, n) representing the ratio m/n, with $n \neq 0$ and $\gcd(m, n) = 1$. Strictly speaking, not every integer is an ordered pair, rather $\mathbb{Z}$ may be isomorphically embedded within $\mathbb{Q}$ by the inclusion mapping $n \hookrightarrow (n, 1)$. That is, the rationals contain an isomorphic copy of the integers. Similarly, the real numbers are often represented as convergent sequences of rationals, so that strictly speaking the rationals are not a subset of the reals, but rather may be embedded in them by choosing a canonical representative (*i.e.*, a particular convergent sequence) of each rational.

For mathematical purposes it is entirely reasonable to overlook fine distinctions such as that between $\mathbb{Z}$ and its embedding within $\mathbb{Q}$. This is justified because the operations on rationals restrict to the embedding in the expected manner: if we add two integers thought of as rationals in the canonical way, then the result is the rational associated with their sum. And similarly for the other operations, provided that we take some care in defining them to ensure that it all works out properly. For the purposes of computing, however, one cannot be quite so cavalier, because we must also take account of algorithmic efficiency and the finiteness of machine representations. Often what are called "real numbers" in a programming language are, in fact, finite precision floating point numbers, a small subset of the rational numbers. Not every rational can be exactly represented as a floating point number, nor does floating point arithmetic restrict to rational arithmetic even when its arguments are exactly represented.

There is a way to make sense of a subtype hierarchy such as the one suggested above, but it comes at a considerable cost. Briefly, there are exact representations of those real numbers that can be generated by a computer program. (For example, one may use computable sequences of rationals equipped with a computable modulus of convergence, but other, better-behaved, representations have also been considered.) To ensure that the specified subtype relationships hold, we may simply represent integers and rationals as real numbers in this sense (*i.e.*, as their isomorphic embeddings), so that arithmetic on reals is, by definition, the same as arithmetic on integers and rationals. In this manner we may validate the subtyping relations suggested earlier, but at the expense of making the arithmetic operations extremely inefficient compared to native machine arithmetic.

34.2.2 Products

Product types give rise to a form of subtyping based on the subsumption principle. The only elimination form applicable to a value of product type is a projection. Under mild assumptions about the dynamic semantics of projections, we may consider one product type to be a subtype of another by considering whether the projections applicable to the supertype may be validly applied to values of the subtype.

In the case of unlabelled tuple types this amounts to regarding a tuple type to be a subtype of any *prefix* of it, as specified by the following rule:

$$\frac{n \leq m}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \; . \tag{34.3}$$

This principle is called *width* subtyping for tuples; it states that a *wider* tuple is a subtype of a *narrower* one, provided that they agree on the types of their fields. This principle is justified if we may project the ith component of a record, where $1 \leq i \leq n$, without knowing whether there are more than n components in the record.

Width subtyping also extends to record (labelled tuple) types:

$$\frac{n \leq m}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \; . \tag{34.4}$$

This principle is justified if we may project the field labelled l from a record without knowing what other fields the record may possess. This may seem less easily justified than prefix subtyping. Prefix subtyping for unlabelled ordered tuples is justified if we can extract the ith component by a simple offset calculation from the front of the tuple, so that the presence or absence of subsequent fields is immaterial. But for labelled unordered tuples, it is not possible to calculate the position of the field labelled l without knowing what other fields are present. This can be remedied in several ways.

The most obvious method is to associate a mapping from labels to positions with each value of record type so that it is always possible to determine the offset of the field labelled l in any given record value. By a careful choice of hash functions one can ensure constant-time access for each field, because the labels are known in advance (new labels are not created at run-time). Another, less obvious, method is to observe that if the only elimination form is the projection, then we may coerce a record value from the subtype to the supertype by copying the fields that are retained, and dropping those that are omitted, in the supertype. In this way we ensure that the static type of a record accurately predicts its entire repertoire of fields so that offsets can be computed statically, rather than dynamically. Note, however, that this method does not scale to mutable records (those for which there is an assignment operation on each field), because copying changes the semantics of the program (mutating the copy does not affect the original). In such languages there is little recourse but to use a dynamic lookup scheme to compute projections.

Summarizing, the principle of width subtyping for finite product types is given by the following rule:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \ . \tag{34.5}$$

This rule generalizes the preceding rules, using the represetation of tuples and records as finite products described in Chapter 17.

34.2.3 Sums

The analogue of width subtyping for labelled sums states that a *narrower* range of choices is a subtype of a *wider* range of choices:

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{34.6}$$

This is justified by observing that a case analysis on the supertype accounts for all of the cases that can arise from a value of the subtype, and then some. The "extra" cases present no problems for safety, and hence we can treat the narrower sum as being a subtype of the wider. One may also consider a form of width subtyping for unlabelled n-ary sums, by considering any prefix of an n-ary sum to be a subtype of that sum. Here again the elimination form for the supertype, namely an n-ary case analysis, is prepared to handle any value of the subtype, which is enough for safety.

Observe that for width subtyping to be sensible, the representation of values of the subtype must be the *same* as their representation for the supertype. In particular one may not "optimize" representations based on the number of summands of the sum type, saying using only as many bits as necessary to represent every possible label. For then if we are to regard values of the subtype as being values of the supertype so that we may case analyze them, the representation must be sensible in the context of a wider array of options. Consequently, no optimization is possible, at least if the values of the subtype are to be passed to the elimination forms of the supertype unchanged.

As a special case of width subtyping for sums, finite enumeration types (finite sums of copies of the unit type) obey the expected subtyping relation corresponding to a naïve interpretation of the enumeration as a finite set of values. That is, a smaller enumeration is a subtype of a larger one, so that the smallest enumeration in the subtype ordering is the empty enumeration. The justification is not the set interpretation, but rather that the elimination form for a finite enumeration is a case analysis, which may be safely applied as long as all cases are covered.

The principle of width subtyping for finite sum types is given by the following rule:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j} \,. \tag{34.7}$$

Note well the reversal of the containment as compared to Rule (34.5).

34.3 Variance

In addition to basic subtyping principles such as those considered in Section 34.2 on page 262, it is also important to consider the effect of subtyping on type constructors. For example, if corresponding fields of a tuple type are subtypes of one another, then how do the tuple types themselves stand in the subtyping relation? Similar questions arise for all type constructors. A type constructor is said to be *covariant* in an argument if subtyping in that argument is preserved by the constructor. It is said to be *contravariant* if subtyping in that argument is reversed by the constructor. Finally, the constructor is said to be *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

34.3.1 Products

The tuple, record, and object type constructors are all covariant in that they preserve subtyping in each argument position. Here is the covariance principle for tuple types:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{\langle \sigma_1, \dots, \sigma_n \rangle <: \langle \tau_1, \dots, \tau_n \rangle} \,. \tag{34.8}$$

Covariance for tuple types is sometimes called *depth* subtyping, since it applies within the components of the tuple, in contrast to width subtyping, which applies across its components. A similar covariance principle governs record types:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{\langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle} \ . \tag{34.9}$$

Depth subtyping for products is justified by the subsumption principle. The only elimination form for a tuple type is projection. If e is a value of the subtype, then its ith component has type σ_i. If we regard e as a value of the supertype, then we expect that $t \cdot i$ has type τ_i, which is justified because $\sigma_i <: \tau_i$. Thus it is valid to consider the entire tuple to be of the supertype, since each component will have the component type specified there.

Summarizing, the principle of covariance for finite product types is given by the following rule:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\prod_{i \in I} \sigma_i <: \prod_{i \in I} \tau_i} \tag{34.10}$$

34.3.2 Sums

Both unlabelled and labelled sum types are covariant in each position:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{[l_1 : \sigma_1, \dots, l_n : \sigma_n] <: [l_1 : \tau_1, \dots, l_n : \tau_n]} \ . \tag{34.11}$$

A case analysis on a value of the supertype is prepared, in the ith branch, to accept a value of type τ_i. By the premises of the rule, it is sufficient to provide a value of type σ_i instead.

The principle of covariance for finite sum types is given by the following rule:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\sum_{i \in I} \sigma_i <: \sum_{i \in I} \tau_i} \tag{34.12}$$

34.3.3 Functions

The variance of the function type constructor is a bit more subtle. Let us consider first the variance of the function type in its range. Suppose that $e : \sigma \rightarrow \tau$. This means that if $e_1 : \sigma$, then $e(e_1) : \tau$. If $\tau <: \tau'$, then $e(e_1) : \tau'$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau <: \tau'}{\sigma \rightarrow \tau <: \sigma \rightarrow \tau'} \tag{34.13}$$

Every function that delivers a value of type τ must also deliver a value of type τ', provided that $\tau <: \tau'$. Thus the function type constructor is covariant in its range.

Now let us consider the variance of the function type in its domain. Suppose again that $e : \sigma \to \tau$. This means that e may be applied to any value of type σ, and hence, by the subsumption principle, it may be applied to any value of any subtype, σ', of σ. In either case it will deliver a value of type τ. Consequently, we may just as well think of e as having type $\sigma' \to \tau$.

$$\frac{\sigma' <: \sigma}{\sigma \to \tau <: \sigma' \to \tau} \tag{34.14}$$

The function type is contravariant in its domain position. Note well the reversal of the subtyping relation in the premise as compared to the conclusion of the rule!

34.4 Safety for Subtyping

Proving safety for a language with subtyping is considerably more delicate than for languages without. The rule of subsumption means that the static type of an expression reveals only partial information about the underlying value. This changes the proof of the preservation and progress theorems, and requires some care in stating and proving the auxiliary lemmas required for the proof. We will illustrate the issues that arise for record types with width and depth subtyping studied in isolation from other language features. We leave it to the reader to extend the proofs to a fuller language, taking account of the effect of subtyping on the other language constructs.

For convenience, we consolidate (and simplify) the static semantics of records as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle l_1 = e_1, \ldots, l_n = e_n \rangle : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{34.15a}$$

$$\frac{\Gamma \vdash e : \langle l : \tau \rangle}{\Gamma \vdash e \cdot l : \tau} \tag{34.15b}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{34.15c}$$

$$\frac{n \leq m}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{34.15d}$$

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{34.15e}$$

Rule (34.15b) is simplified compared to Rule (17.3b), because we can take advantage of width subtyping to focus on the one field of interest.

We state several lemmas about the static semantics that will be of use in the safety proof.

Lemma 34.1 (Structural)**.** *1. The record subtyping relation is reflexive and transitive.*

2. The typing judgement $\Gamma \vdash e : \tau$ is closed under weakening and substitution.

Lemma 34.2 (Inversion)**.** *1. If $e \cdot l : \tau$, then $e : \langle l : \tau \rangle$.*

2. If $\langle l_1 = e_1, \ldots, l_n = e_n \rangle : \tau$, then $\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \tau$ for some $\sigma_1, \ldots, \sigma_n$, such that, for each $1 \leq i \leq n$, $e_i : \sigma_i$.

3. If $\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle$, then $m \leq n$ and, for each $1 \leq i \leq m$, we have $\sigma_i <: \tau_i$.

Proof. By induction on the typing rules, paying special attention to the rule of subsumption. For example, in the proof of the first inversion principle, if $e \cdot l : \tau$ is derived by subsumption, then we have $e \cdot l : \sigma$ for some $\sigma <: \tau$, and so by induction $e : \langle l : \sigma \rangle$, and hence by covariance, $e : \langle l : \tau \rangle$. Similarly, in the proof of the second property for the case of subsumption we rely on the transitivity of the subtyping relation. □

The dynamic semantics of records is repeated here for ease of reference:

$$\frac{e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \ \mathsf{val}} \tag{34.16a}$$

$$\frac{\begin{array}{c} e_1 \ \mathsf{val} \quad e_1' = e_1 \quad \ldots \quad e_{i-1} \ \mathsf{val} \quad e_{i-1}' = e_{i-1} \\ e_i \mapsto e_i' \quad e_{i+1}' = e_{i+1} \quad \ldots \quad e_n' = e_n \end{array}}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \mapsto \langle l_1 = e_1', \ldots, l_n = e_n' \rangle} \tag{34.16b}$$

$$\frac{e \mapsto e'}{e \cdot l \mapsto e' \cdot l} \tag{34.16c}$$

$$\frac{e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val} \quad 1 \leq i \leq n}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \cdot l_i \mapsto e_i} \tag{34.16d}$$

Theorem 34.3 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By induction on Rules (34.16). For example, consider Rule (34.16d). We have by assumption $\langle l_1 = e_1, \ldots, l_n = e_n \rangle \cdot l_i : \tau$. By inversion of typing we have $\langle l_1 = e_1, \ldots, l_n = e_n \rangle : \langle l_i : \tau \rangle$, and hence by inversion of typing $\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_i : \tau \rangle$ with $e_j : \sigma_j$ for each $1 \leq j \leq n$. Therefore by inversion of subtyping there exists $1 \leq j \leq n$ such that $l_j = l_i$ and $\sigma_j <: \tau$, from which it follows that $e_i : \tau$. □

Lemma 34.4 (Canonical Forms). *If e val and $e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$, then $e = \langle l_1 = e_1, \ldots, l_m = e_m \rangle$ with $m \geq n$ and e_j val for each $1 \leq j \leq m$.*

Proof. By induction on the static semantics, taking account of the definition of values. Observe that a value of record type is, in general, larger than is predicted by its type. □

Theorem 34.5 (Progress). *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By induction on typing. Consider, for example, Rule (34.15b), with $\tau = \langle l : \sigma \rangle$. By induction either e val or $e \mapsto e'$ for some e'. In the latter case we have by Rule (34.16c) $e \cdot l \mapsto e' \cdot l$. In the former case we have by canonical forms that $e = \langle l_1 = e_1, \ldots, l_m = e_m \rangle$ where e_i val for each $1 \leq i \leq m$, and such that $l = l_j$ for some $1 \leq j \leq m$. Therefore $e \cdot l \mapsto e_j$, as required. □

34.5 Recursive Subtyping

Consider the types of labelled binary trees with natural numbers at each node,

$$\mu t\,.\,\texttt{[empty:unit,binode:}\langle\texttt{data:nat,lft:}t\texttt{,rht:}t\rangle\texttt{]},$$

and of “bare” binary trees, without labels on the nodes,

$$\mu t\,.\,\texttt{[empty:unit,binode:}\langle\texttt{lft:}t\texttt{,rht:}t\rangle\texttt{]}.$$

Is either a subtype of the other? Intuitively, one might expect the type of labelled binary trees to be a *subtype* of the type of bare binary trees, since any use of a bare binary tree can simply ignore the presence of the label.

Now consider the type of bare “two-three” trees with two sorts of nodes, those with two children, and those with three:

$$\mu t\,.\,\texttt{[empty:unit,binode:}\langle\texttt{lft:}t\texttt{,rht:}t\rangle\texttt{,trinode:}\langle\texttt{lft:}t\texttt{,mid:}t\texttt{,rht:}t\rangle\texttt{]}.$$

What subtype relationships should hold between this type and the preceding two tree types? Intuitively the type of bare two-three trees should be a *supertype* of the type of bare binary trees, since any use of a two-three tree must proceed by three-way case analysis, which covers both forms of binary tree.

To capture the pattern illustrated by these examples, we must formulate a subtyping rule for recursive types. It is tempting to consider the following rule:

$$\frac{t \mid \sigma <: \tau}{\mu t\,.\,\sigma <: \mu t\,.\,\tau}\ \textbf{??} \tag{34.17}$$

That is, to determine whether one recursive type is a subtype of the other, we simply compare their bodies, with the bound variable treated as a parameter. Notice that by reflexivity of subtyping, we have $t <: t$, and hence we may use this fact in the derivation of $\sigma <: \tau$.

Rule (34.17) validates the intuitively plausible subtyping between labelled binary tree and bare binary trees just described. To derive this reduces to checking the subtyping relationship

$$\langle \mathtt{data}:\mathtt{nat}, \mathtt{lft}:t, \mathtt{rht}:t\rangle <: \langle \mathtt{lft}:t, \mathtt{rht}:t\rangle,$$

generically in t, which is evidently the case.

Unfortunately, Rule (34.17) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\sigma = \mu t\,.\,\langle \mathtt{a}:t \rightarrow \mathtt{nat}, \mathtt{b}:t \rightarrow \mathtt{int}\rangle$$

and

$$\tau = \mu t\,.\,\langle \mathtt{a}:t \rightarrow \mathtt{int}, \mathtt{b}:t \rightarrow \mathtt{int}\rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using Rule (34.17) we may derive $\sigma <: \tau$, which we will show to be incorrect. Let $e : \sigma$ be the expression

$$\mathtt{fold}(\langle \mathtt{a}=\lambda(x{:}\sigma.\,4), \mathtt{b}=\lambda(x{:}\sigma.\,q(\mathtt{unfold}(x)\cdot\mathtt{a}(x)))\rangle),$$

where $q : \mathtt{nat} \rightarrow \mathtt{nat}$ is the discrete square root function. Since $\sigma <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(e) : \langle \mathtt{a}:\tau \rightarrow \mathtt{nat}, \mathtt{b}:\tau \rightarrow \mathtt{int}\rangle.$$

Now let $e' : \tau$ be the expression

$$\texttt{fold}(\langle \texttt{a} = \lambda(x{:}\tau.\,\texttt{-4}), \texttt{b} = \lambda(x{:}\tau.\,\texttt{0})\rangle).$$

(The important point about e' is that the a method returns a negative number; the b method is of no significance.) To finish the proof, observe that

$$\texttt{unfold}(e) \cdot \texttt{b}(e') \mapsto^{*} q(\texttt{-4}),$$

which is a stuck state. We have derived a well-typed program that "gets stuck", refuting type safety!

Rule (34.17) is therefore incorrect. But what has gone wrong? The error lies in the choice of a single parameter to stand for both recursive types, which does not correctly model self-reference. In effect we are regarding two distinct recursive types as equal while checking their bodies for a subtyping relationship. But this is clearly wrong! It fails to take account of the self-referential nature of recursive types. On the left side the bound variable stands for the subtype, whereas on the right the bound variable stands for the super-type. Confusing them leads to the unsoundness just illustrated.

As is often the case with self-reference, the solution is to *assume* what we are trying to prove, and check that this assumption can be maintained by examining the bodies of the recursive types. This leads to the following correct rule of subsumption for recursive types:

$$\frac{\mu s\,.\,\sigma <: \mu t\,.\,\tau \vdash [\mu s\,.\,\sigma/s]\sigma <: [\mu t\,.\,\tau/t]\tau}{\mu s\,.\,\sigma <: \mu t\,.\,\tau}\,. \tag{34.18}$$

Using this rule we may verify the subtypings among the tree types sketched above. Moreover, it is instructive to check that the unsound subtyping is *not* derivable using this rule! The reason is that the assumption of the subtyping relation is at odds with the contravariance of the function type in its domain.

An alternative formulation of Rule (34.18) makes use of parameters, rather than substitution.

$$\frac{s, t \mid s <: t \vdash \sigma <: \tau}{\mu s\,.\,\sigma <: \mu t\,.\,\tau}\,. \tag{34.19}$$

It is easy to verify that each rule is admissible in the presence of the other.

34.6 References[1]

Reference types interact poorly with subtyping. To see why, let us apply the principle of subsumption to derive a sound subtyping rule for references. Suppose that r has type $\sigma\,\texttt{ref}$. There are two elimination forms that may be applied to r:

1. Retrieve its contents as a value of type σ.

2. Replace its contents with a value of type σ.

If $\sigma <: \tau$, then retrieving the contents of r yields a value of type τ, by subsumption. This suggests that reference types be considered covariant:

$$\frac{\sigma <: \tau}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}}\ \textbf{??}$$

On the other hand, if $\tau <: \sigma$, then we may store a value of type τ into r. This suggests that reference types be considered contravariant:

$$\frac{\tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}}\ \textbf{??}$$

Combining these two observations, we see that reference types are *invariant*:

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \tag{34.20}$$

In practice the only mway to satisfy the premises of the rule is for σ and τ to be identical.

Similar restrictions govern mutable array types, whose components are mutable cells that can be assigned and retrieved just as for reference types. A naïve interpretation of array types would suggest that arrays be covariant, since a sequence of values of type σ is also a sequence of values of type τ, provided that $\sigma <: \tau$. But this overlooks the possibility of assigning to the elements of the array, which is inconsistent with covariance. The result is that mutable array types must be regarded as *invariant* to ensure type safety (*pace* well-known languages that stipulate otherwise).

34.7 Exercises

1. Consider the subtyping issues related to signed and unsigned, fixed precision integer types.

[1] Please see Chapter 37 for discussion of reference types.

2. Investigate "downcasting" for variant types. Makes the type of an expression of variant type more precise. When there is only one variant, case analysis is just a safe "outjection."

3. Labelled two-three trees and the associated pre-order among the four types.

4. Investigate the incorrect subtyping rule for recursive types in which the types are assumed equal.

Chapter 35

Singleton and Dependent Kinds

The expression $\texttt{let}\,e_1\texttt{:}\tau\,\texttt{be}\,x\,\texttt{in}\,e_2$ is a form of abbreviation mechanism by which we may bind e_1 to the variable x for use within e_2. In the presence of function types this expression is definable as the application $\lambda\texttt{(}x\texttt{:}\tau\texttt{.}\,e_2\texttt{)}\texttt{(}e_1\texttt{)}$, which accomplishes the same thing. It is natural to consider an analogous form of `let` expression which permits a *type expression* to be bound to a type variable within a specified scope. The expression $\texttt{let}\,t\,\texttt{be}\,\tau\,\texttt{in}\,e$ binds t to τ within e, so that one may write expressions such as

$$\texttt{let}\,t\,\texttt{be}\,\texttt{nat}\times\texttt{nat}\,\texttt{in}\,\lambda\texttt{(}x\texttt{:}t\texttt{.}\,\texttt{s(fst(}x\texttt{)))}.$$

For this expression to be type-correct the type variable t must be *synonymous* with the type $\texttt{nat}\times\texttt{nat}$, for otherwise the body of the λ-abstraction is not type correct.

Following the pattern of the expression-level `let`, we might guess that `lettype` is an abbreviation for the polymorphic instantiation $\Lambda\texttt{(}t\texttt{.}e\texttt{)}\texttt{[}\tau\texttt{]}$, which binds t to τ within e. This does, indeed, capture the dynamic semantics of type abbreviation, but it fails to validate the intended static semantics. The difficulty is that, according to this interpretation of `lettype`, the expression e is type-checked in the absence of any knowledge of the binding of t, rather than in the knowledge that t is synomous with τ. Thus, in the above example, the expression $\texttt{s(fst(}x\texttt{))}$ fails to type check, unless the binding of t were exposed.

The proposed definition of `lettype` in terms of type abstraction and type application fails. Lacking any other idea, one might argue that type abbreviation ought to be considered as a primitive concept, rather than a derived notion. The expression $\mathtt{let}\,t\,\mathtt{be}\,\tau\,\mathtt{in}\,e$ would be taken as a primitive form of expression whose static semantics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \mathtt{let}\,t\,\mathtt{be}\,\tau\,\mathtt{in}\,e : \tau'} \tag{35.1}$$

This would address the problem of supporting type abbreviations, but it does so in a rather *ad hoc* manner. One might hope for a more principled solution that arises naturally from the type structure of the language.

Our methodology of identifying language constructs with type structure suggests that we ask not how to support type abbreviations, but rather what form of type structure gives rise to type abbreviations? And what else does this type structure suggest? By following this methodology we are led to the concept of *singleton kinds*, which not only account for type abbreviations but also play a crucial role in the design of module systems.

35.1 Informal Overview

The central organizing principle of type theory is *compositionality*. To ensure that a program may be decomposed into separable parts, we ensure that the composition of a program from constituent parts is mediated by the types of those parts. Put in other terms, the only thing that one portion of a program "knows" about another is its type. For example, the formation rule for addition of natural numbers depends only on the type of its arguments (both must have type `nat`), and not on their specific form or value. But in the case of a type abbreviation of the form $\mathtt{let}\,t\,\mathtt{be}\,\tau\,\mathtt{in}\,e$, the principle of compositionality dictates that the only thing that e "knows" about the type variable t is its kind, namely `Type`, and not its binding, namely τ. This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of t to capture its identity. This may be achieved through the notion of a *singleton kind*. Informally, the kind $\mathtt{Eqv}(\tau)$ is the kind of types that are definitionally equivalent to τ. That is, up to definitional equality, this kind has only one inhabitant, namely τ. Consequently, if $u :: \mathtt{Eqv}(\tau)$ is a variable of singleton kind, then within its scope, the variable u is synonymous with τ. Thus we may represent $\mathtt{let}\,t\,\mathtt{be}\,\tau\,\mathtt{in}\,e$ by

$\Lambda(t::\texttt{Eqv}(\tau).e)[\tau]$, which correctly propagates the identity of t, namely τ, to e during type checking.

A proper treatment of singleton kinds requires some additional machinery at the constructor and kind level. First, we must capture the idea that a constructor of singleton kind is *a fortiori* a constructor of kind Type, and hence is a type. Otherwise, a variable, u, singleton kind cannot be used as a type, even though it is explicitly defined to be one! This may be captured by introducing a *subkinding* relation, $\kappa_1 :<: \kappa_2$, which is analogous to subtyping, exception at the kind level. The fundamental axiom of subkinding is $\texttt{Eqv}(\tau) :<: \texttt{Type}$, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind Type within the singleton kind $\texttt{Eqv}(\tau)$. This intermixing of the constructor and kind level means that singletons are a form of *dependent kind* in that a kind may depend on a constructor. Another way to say the same thing is that $\texttt{Eqv}(\tau)$ represents a *family of kinds* indexed by constructors of kind Type. This, in turn, implies that we must generalize the function and product kinds to *dependent functions* and *dependent products*. The dependent function kind, $\Pi\, u::\kappa_1.\kappa_2$ classifies functions that, when applied to a constructor $c_1 :: \kappa_1$, results in a constructor of kind $[c_1/u]\kappa_2$. The important point is that the kind of the result is sensitive to the argument, and not just to its kind.[1] The dependent product kind, $\Sigma\, u::\kappa_1.\kappa_2$, classifies pairs $\langle c_1, c_2\rangle$ such that $c_1 :: \kappa_1$, as might be expected, and $c_2 :: [c_1/u]\kappa_2$, in which the kind of the second component is sensitive to the first component itself, and not just its kind.

Third, it is useful to consider singletons not just of kind Type, but also of higher kinds. To support this we introduce *higher-kind singletons*, written $\texttt{Eqv}(c::\kappa)$, where κ is a kind and c is a constructor of kind k. These are definable in terms of the primitive form of singleton kind by making use of dependent function and product kinds.

[1] As we shall see in the development, the propagation of information as sketched here is managed through the use of singleton kinds.

Part XII

State

Chapter 36

Fluid Binding

In Chapter 14 we examined the concept of *dynamic binding* as a scoping discipline for variables, and found it lacking in at least two respects:

- Bound variables may no longer be identified up to consistent renaming. This does violence to the concept of scope, which is concerned with associating usages of variables with their point of definition.

- Since the scopes of variables are resolved dynamically, it is not possible to ensure type safety. Different bindings of a variable, x, at different types may govern a given user of a variable, depending on the dynamic flow of control in a program.

Nevertheless, binding does offer a useful capability that can be salvaged from this wreckage.

Dynamic binding provides a separation between the *scope* of a variable and its *extent*. The scope, as we have seen, is the static range of significance of an identifier. In a statically scoped language a variable has meaning only within a specified phrase. Within its scope the variabl serves as a reference to its binding site. Outside of its scope, the variable has no meaning whatsoever. This is crucial for modularity, since it ensures that private variables really are private, allowing a program unit to be treated as a "black box" from an external perspective.

The *extent* of a variable is its *dynamic* range of significance, the "interval" of execution during which the identifier has meaning. In a statically typed language the scope and the extent coincide. Variables are given meaning by substitution, which is defined syntactically over program phrases. What dynamic binding offers is a separation between the scope and the extent of a variable. The idea is that execution may enter the scope of a variable without associating a value to it. Later, at various points in the execution, a value is associated to the variable for use within the execution of a specific, dynamically determined, expression, which is called the *extent* of the association. Moreover, during execution the same variable may be associated with another variable for use within the evaluation of a specified expression. But once the specified expression has completed evaluation, the association is dropped, reverting it to its previous value.

The advantage of separating scope from extent is precisely that it permits evaluation within the scope of the identifier, while permitting some other, dynamically determined expression to associate a value with it. To avoid confusion, we use the term *fluid binding* for the ability to separate the scope of a fluid-bound identifier from the extent of any binding it may be given during execution. This involves two mechanisms. One is the concept of a *symbol*, which is an identifier with an associated type. The other is the concept of a *fluid let*, which associates a value of type appropriate to a symbol for use within the execution of a specific expression.

We will study a language fragment, called $\mathcal{L}\{\texttt{fluid}\}$, with fluid binding of symbols to values. In Section 36.1 we will consider the mechanisms of fluid binding for a fixed collection of symbols. Then in Section 36.2 on page 285 we will add the mechanisms required to create new symbols during execution of a program.

36.1 Fluid Binding

The syntax of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{set}[a](e_1;e_2)$	$\texttt{set}\,a\,\texttt{to}\,e_1\,\texttt{in}\,e_2$
		$\mid$	$\texttt{get}[a]$	$\texttt{get}\,a$

We assume given an infinite set of symbols, a, disjoint from the set of variables of the language. The expression $\texttt{set}\,a\,\texttt{to}\,e_1\,\texttt{in}\,e_2$, called a *fluid let*, binds the symbol a to the value e_1 for the duration of the evaluation of e_2, at which point the binding of a reverts to what it was prior to the execution. The argument a is a symbol, not a variable, and it is not introduced as a fresh variable by the fluid let. The expression $\texttt{get}\,a$ evaluates to the value of the current binding of a, if it has one, and is stuck otherwise.

The static semantics of $\mathcal{L}\{\texttt{fluid}\}$ is defined by judgements of the form $\Sigma\,\Gamma \vdash e : \tau$, where Γ is, as usual, a finite set of variable typing assumptions of the form $x : \tau$, and where Σ is a finite set of symbol typing assumptions of the form $a : \tau$, where a sym. As usual, we insist that no variable be the subject of more than one typing assumption. This is extended to symbols as well, which has the effect of ensuring that each symbol has a unique type of associated values.

As discussed in Chapter 3, the hypothetical judgement $\Sigma\,\Gamma \vdash e : \tau$ is, officially, a parametric hypothetical judgement of the form

$$\mathcal{A}\,\mathcal{X} \mid \Sigma\,\Gamma \vdash e : \tau,$$

where $\mathcal{A}$ and $\mathcal{X}$ are disjoint sets, the hypotheses Σ govern the symbols in $\mathcal{A}$, and the hypotheses Γ govern the variables in $\mathcal{X}$. (This will become significant in Section 36.2 on page 285, where we rely on the structural property of permutation of variables to manage dynamic symbol generation.) As usual we will suppress explicit mention of $\mathcal{A}$ and $\mathcal{X}$ when presenting hypothetical typing judgements.

The rules defining the static semantics of $\mathcal{L}\{\texttt{fluid}\}$ are given as follows:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \texttt{get}\,[a] : \tau} \tag{36.1a}$$

$$\frac{\Sigma \vdash a : \tau_1 \quad \Sigma\,\Gamma \vdash e_1 : \tau_1 \quad \Sigma\,\Gamma \vdash e_2 : \tau_2}{\Sigma\,\Gamma \vdash \texttt{set}\,[a]\,(e_1;e_2) : \tau_2} \tag{36.1b}$$

Rule (36.1b) treats the symbol a as given by Σ, rather than introducing a "new" bound symbol, as would be the case for a conventional `let` construct. That is, neither Σ nor Γ are extended in the premise of Rule (36.1b).

The dynamic semantics of $\mathcal{L}\{\texttt{fluid}\}$ is given by a judgement of the form $e \mapsto_\theta e'$, where θ is a finite function mapping symbols from Σ to a closed (with respect to variables) value of the type determined by Σ. If $a \in dom(\theta)$, then we may write $\theta = \theta' \otimes \langle a : e \rangle$. If $a \notin dom(\theta)$, then we shall, as a notational convenience, regard θ has having the form $\theta' \otimes \langle a : \bullet \rangle$ in which a is considered "bound" to a "black hole", $\bullet$. We will write $\langle a : \_ \rangle$ to stand ambiguously for either $\langle a : \bullet \rangle$ or $\langle a : e \rangle$ for some expression e.

The dynamic semantics of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following rules:

$$\frac{}{\texttt{get}\,[a] \mapsto_{\theta \otimes \langle a : e \rangle} e} \tag{36.2a}$$

$$\frac{e_1 \mapsto_\theta e_1'}{\texttt{set}\,[a]\,(e_1;e_2) \mapsto_\theta \texttt{set}\,[a]\,(e_1';e_2)} \tag{36.2b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto_{\theta \otimes \langle a : e_1 \rangle} e_2'}{\texttt{set}\,[a]\,(e_1;e_2) \mapsto_{\theta \otimes \langle a : \_ \rangle} \texttt{set}\,[a]\,(e_1;e_2')} \tag{36.2c}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{set}[a](e_1;e_2) \mapsto_\theta e_2} \tag{36.2d}$$

Rule (36.2a) specifies that $\mathtt{get}[a]$ evaluates to the current binding of a, if any. Rule (36.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (36.2c) evaluates e_2 in an environment in which the symbol a is bound to the value e_1, regardless of whether or not a is already bound in the environment. Rule (36.2d) eliminates the fluid binding for a once evaluation of the extent of the binding has completed. Observe that if e_2 is, say, a λ-abstraction, then it may contain unevaluated occurrences of $\mathtt{get}[a]$, which will refer to the enclosing binding for a, if any, or any subsequent binding within whose body the evaluation of the body of the λ-abstraction occurs.

The dynamic semantics specifies that there is no transition of the form $\mathtt{get}[a] \mapsto_{\theta\otimes\langle a\,:\,\bullet\rangle} e$ for any e. Since the static semantics does not rule out such states, we define the judgement $e\ \mathsf{unbound}_\theta$ by the following rules:[1]

$$\frac{}{\mathtt{get}[a]\ \mathsf{unbound}_{\theta\otimes\langle a\,:\,\bullet\rangle}} \tag{36.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\theta}{\mathtt{set}[a](e_1;e_2)\ \mathsf{unbound}_\theta} \tag{36.3b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{unbound}_\theta}{\mathtt{set}[a](e_1;e_2)\ \mathsf{unbound}_\theta} \tag{36.3c}$$

The progress theorem is stated to admit stuck states of this form, indicating that a well-typed program may incur a run-time error arising from attempting to obtain the binding of an unbound symbol.

We define $\theta : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{36.4a}$$

$$\frac{\Sigma \vdash e : \tau \quad \theta : \Sigma}{\theta \otimes \langle a : e\rangle : \Sigma, a : \tau} \tag{36.4b}$$

$$\frac{\theta : \Sigma}{\theta \otimes \langle a : \bullet\rangle : \Sigma, a : \tau} \tag{36.4c}$$

Thus we make no demands on the typing of unbound symbols, but demand that the binding of a bound symbol be of the type specified by Σ.

Theorem 36.1 (Preservation). *If $e \mapsto_\theta e'$, where $\theta : \Sigma$ and $\Sigma \vdash e : \tau$, then $\Sigma \vdash e' : \tau$.*

[1] In the presence of other constructs, such as function application, stuck states would have to be propagated through any evaluated arguments of any compound expression.

Proof. By rule induction on Rules (36.2). Rule (36.2a) is handled by the definition of $\theta : \Sigma$. Rule (36.2b) follows immediately by induction. Rule (36.2d) is handled by inversion of Rules (36.1). Finally, Rule (36.2c) is handled by inversion of Rules (36.1) and induction. □

Theorem 36.2 (Progress). *If* $\Sigma \vdash e : \tau$ *and* $\theta : \Sigma$, *then either* e *val, or* e $\texttt{unbound}_\theta$, *or there exists* e' *such that* $e \mapsto_\theta e'$.

Proof. By induction on Rules (36.1). For Rule (36.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\theta : \Sigma$, we have either $\theta(a) = \bullet$ (*i.e.*, is unbound) or $\theta(a) = e$ for some e such that $\Sigma \vdash e : \tau$. In the former case we have e $\texttt{unbound}_\theta$, and in the latter we have $\texttt{get[}a\texttt{]} \mapsto_\theta e$. For Rule (36.1b), we have by induction that either e_1 val or e_1 $\texttt{unbound}_\theta$, or $e_1 \mapsto_\theta e_1'$. In the latter two cases we may apply Rule (36.2b) or Rule (36.3b), respectively. If e_1 val, we apply induction to obtain that either e_2 val, in which case Rule (36.2d) applies; e_2 $\texttt{unbound}_\theta$, in which case Rule (36.3b) applies; or $e_2 \mapsto_\theta e_2'$, in which case Rule (36.2c) applies. □

36.2 Symbol Generation

The language $\mathcal{L}\{\texttt{fluid}\,\texttt{gen}\}$ enriches $\mathcal{L}\{\texttt{fluid}\}$ with constructs for generating fresh symbols during execution. The syntax of this extension is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{sym(}\sigma\texttt{;}\tau\texttt{)}$	$\langle\sigma\rangle\tau$
Expr	e	::=	$\texttt{new[}\sigma\texttt{](}a\texttt{.}e\texttt{)}$	$\nu(a\texttt{:}\sigma\texttt{.}e)$
		|	$\texttt{gen(}e\texttt{)}$	$\texttt{gen(}e\texttt{)}$

The type $\texttt{sym(}\sigma\texttt{;}\tau\texttt{)}$ represents the type of expressions abstracted over an unspecified symbol of type σ, yielding a value of type τ. The introductory form of type $\texttt{sym(}\sigma\texttt{;}\tau\texttt{)}$ is the symbol abstraction $\texttt{new[}\sigma\texttt{](}a\texttt{.}e\texttt{)}$, and the eliminatory form is $\texttt{gen(}e\texttt{)}$, which, when e is $\texttt{new[}\sigma\texttt{](}a\texttt{.}e\texttt{)}$, generates a fresh symbol, a', and replaces a by a' within e. Since the only property enjoyed by a symbol is its identity, the static semantics is the same for all sufficiently fresh choices of symbols. This allows us to create new symbols at run-time while still imposing a static type discipline on programs.

The static semantics of $\mathcal{L}\{\texttt{fluid}\,\texttt{gen}\}$ is given by the following rules:

$$\frac{\Sigma, a : \sigma\,\Gamma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{new[}\sigma\texttt{](}a\texttt{.}e\texttt{)} : \texttt{sym(}\sigma\texttt{;}\tau\texttt{)}} \tag{36.5a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \mathtt{sym}(\sigma;\tau)}{\Sigma\,\Gamma \vdash \mathtt{gen}(e) : \tau} \tag{36.5b}$$

Rule (36.5a) extends Σ with a new symbol whose uniqueness is guaranteed by the convention on bound variables. If a already occurs as the subject of some typing assumption in Σ, then we must rename a in $\mathtt{new}[\sigma](a.e)$ prior to applying the rule. Rule (36.5b) generates a fresh symbol to be used in place of the symbol introduced by e in accordance with Rule (36.5a).

The dynamic semantics of $\mathcal{L}\{\mathtt{fluid}\,\mathtt{gen}\}$ is given by the following rules:

$$\frac{}{\mathtt{new}[\sigma](a.e)\ \mathsf{val}} \tag{36.6a}$$

$$\frac{e \mapsto_\theta e'}{\mathtt{gen}(e) \mapsto_\theta \mathtt{gen}(e')} \tag{36.6b}$$

$$\frac{a'\ \mathit{fresh}}{\mathtt{gen}(\mathtt{new}[\sigma](a.e)) \mapsto_\theta [a'/a]e} \tag{36.6c}$$

Rule (36.6c) makes use of an informal convention regarding freshness of symbols. While intuitively clear (the symbol a' should be chosen so as to not otherwise occur in an evaluation), the dynamic semantics as given by Rules (36.6) is not properly defined.

To make precise the freshness condition in Rule (36.6c), we define a transition system on states of the form $e \mathbin{@} \nu$, where ν is a finite set of symbols and e is an expression involving at most the symbols in ν. The set ν is to be thought of as the set of *active* symbols, so that a *fresh* symbol is one that lies outside of this set.

The transition judgement, $e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'$, is defined for states $e \mathbin{@} \nu$ such that $\mathit{dom}(\theta) \subseteq \nu$. This ensures that the mapping θ governs only active symbols. An initial state has the form $e \mathbin{@} \emptyset$, which requires that e type-check with respect to the empty set of assumptions about the types of symbols. A final state has the form $e \mathbin{@} \nu$, where e val.

The rules defining this transition judgement are as follows:

$$\frac{a \in \nu}{\mathtt{get}[a] \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : e \rangle} e \mathbin{@} \nu} \tag{36.7a}$$

$$\frac{e_1 \mathbin{@} \nu \mapsto_\theta e_1' \mathbin{@} \nu'}{\mathtt{set}[a](e_1;e_2) \mathbin{@} \nu \mapsto_\theta \mathtt{set}[a](e_1';e_2) \mathbin{@} \nu'} \tag{36.7b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : e_1 \rangle} e_2' \mathbin{@} \nu'}{\mathtt{set}[a](e_1;e_2) \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : \_ \rangle} \mathtt{set}[a](e_1;e_2') \mathbin{@} \nu'} \tag{36.7c}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{set}[a]\,(e_1;e_2)\ \mathtt{@}\,\nu \mapsto_\theta e_2\ \mathtt{@}\,\nu} \tag{36.7d}$$

$$\frac{e\ \mathtt{@}\,\nu \mapsto_\theta e'\ \mathtt{@}\,\nu'}{\mathtt{gen}(e)\ \mathtt{@}\,\nu \mapsto_\theta \mathtt{gen}(e')\ \mathtt{@}\,\nu'} \tag{36.7e}$$

$$\frac{a' \notin \nu}{\mathtt{gen}(\mathtt{new}[\sigma]\,(a.e))\ \mathtt{@}\,\nu \mapsto_\theta [a'/a]e\ \mathtt{@}\,\nu \cup \{\, a' \,\}} \tag{36.7f}$$

Rule (36.7f) makes explicit that the symbol a' is chosen outside of the set ν of active symbols. Observe that the set of active symbols grows monotonically with transition: if $e\,\mathtt{@}\,\nu \mapsto_\theta e'\,\mathtt{@}\,\nu'$, then $\nu' \supseteq \nu$.

To prove safety we define a state $e\,\mathtt{@}\,\nu$ to be well-formed iff there exists a symbol typing Σ such that (a) $a \in \nu$ implies $\Sigma = \Sigma', a : \tau$ for some type τ, and (b) $\Sigma \vdash e : \sigma$ for some type σ. It is then straightforward to formulate and prove type safety, following along the lines of Section 36.1 on page 282, but treating ν as the set of active symbols. The main difference compared to the static case is that the proof of preservation for Rule (36.7f) relies on the invariance of typing under renaming of parameters, as described in Chapter 3.

36.3 Exercises

1. Complete the formalization of $\mathcal{L}\{\mathtt{fluid}\,\mathtt{gen}\}$ and prove type safety for it.

Chapter 37

Mutable Storage

Data structures constructed from sums, products, and recursive types are all *immutable* in that their structure does not change over time as a result of computation. For example, evaluation of an expression such as $\langle 2{+}3, 4{*}5\rangle$ results in the ordered pair $\langle 5, 20\rangle$, which cannot subsequently be altered by further computation. Creation of a value (of any type) is “forever” in that no subsequent computation can change it. Such data structures are said to be *persistent* in that their value persists throughout the rest of the computation. In particular if we project the components of a pair, and construct another pair from it, the original and the newly constructed pair continue to exist side-by-side. This behavior is particularly significant when working with recursive types, such as lists and trees, because the operations performed on them are *non-destructive*. Inserting an element into a persistent binary search tree does not modify the original tree; rather it constructs another tree with the new element inserted, leaving the original intact and available for further computation.

This behavior is in sharp contrast to conventional textbook treatments of data structures such as lists and trees, which are almost invariably defined by *destructive* operations that modify, or *mutate*, the data structure “in place”. Inserting an element into a binary tree changes the tree itself to include the new element; the original tree is lost in the process, and all references to it reflect the change. Such data structures are said to be *ephemeral*, in that changes to them destroy the original. In some cases ephemeral data structures are essential to the task at hand; in other cases a persistent representation would do just as well, or even better. For example, a data structure modeling a shared database accessed by many users simultaneously is naturally ephemeral in that the changes made by one user are to be immediately propagated to the computations made by another. On the other hand, data structures used internally to a body of code, such as a search tree, need no such capability and are often profitably represented in persistent form.

A good programming language should naturally support both persistent and ephemeral data structures. This is neatly achieved by making a *type distinction* between a *value* of a type and a *mutable cell* containing a value of that type. The number 3 is forever the number 3, but a mutable cell containing the number 3 may be subsequently changed to contain the number 4. Mutable cells are themselves values; one may think of them as "boxes" containing a value that we may change at will. Such boxes may appear within a data structure, so that some aspects of the data structure are mutable and other aspects are immutable. For example, a value of type $\texttt{nat} \times (\texttt{nat}\,\texttt{ref})$ is a pair consisting of a natural number and a cell containing a natural number. The pair itself cannot be changed, but the *contents* of its second component may be changed. Similarly, a value of type $\texttt{nat}\,\texttt{ref} \times \texttt{nat}\,\texttt{ref}$ is a pair both of whose components are mutable cells whose contents may change. Contrast this with a value of type $(\texttt{nat} \times \texttt{nat})\,\texttt{ref}$, which is a cell whose contents is a pair of natural numbers. Its contents may change, but any pair stored within it will not change.

As these examples illustrate, maintaining a distinction between immutable values and mutable cells greatly increases the expressive power of the language. Without mutable cells, only persistent data structures would be available. If all data structures were mutable, irrespective of type, then only ephemeral data structures would be representable. With the separation of mutable from immutable data we gain the ability to draw fine distinctions that exploit delicate combinations of mutability and immutability. The price we pay for this expressiveness is that it is more complex to reason about programs that manipulate mutable data structures. The chief complication is called *aliasing*. If variables x and y both have type $\texttt{nat}\,\texttt{ref}$, then both are bound to mutable cells, but we cannot tell from the type alone whether they are bound to the *same* cell or *different* cells. If they are bound to the same cell, then mutation of the cell bound to x affects the contents of the cell bound to y, otherwise modification of one does not affect the other. When reasoning about programs with mutation, we must always remember to consider on possible aliasing relationships to ensure that the code behaves properly in all cases. The more mutable cells there are, the more cases we have to consider, and the more opportunities for error.

37.1 Reference Cells

The language $\mathcal{L}\{\texttt{ref}\}$ of mutable cells is described by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{ref}(\tau)$	$\tau\,\texttt{ref}$
Expr	e	$::=$	l	l
		$\mid$	$\texttt{new}(e)$	$\texttt{new}(e)$
		$\mid$	$\texttt{get}(e)$	$\texttt{!}\,e$
		$\mid$	$\texttt{set}(e_1;e_2)$	$e_1 \texttt{:=} e_2$

Mutable cells are handled *by reference*; a mutable cell is represented by a *location*, which is the *name*, or *abstract address*, of the cell. The meta-variable l ranges over locations, an infinite set of parameters disjoint from the variables and from any other parameters in the language. Although a form of expression, locations arise only during evaluation as new cells are allocated, and may not be present in expressions written by the programmer. The expression $\texttt{new}(e)$ allocates a "new" reference cell with initial contents of type τ being the value of the expression e. The expression $\texttt{get}(e)$ retrieves the contents of the cell given by the value of e, and $\texttt{set}(e_1;e_2)$ sets the contents of the cell given by the value of e_1 to the value of e_2.

In practice we consider $\mathcal{L}\{\texttt{ref}\}$ as an extension to another language, such as $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, with mutable data. However, for the purposes of this chapter we study $\mathcal{L}\{\texttt{ref}\}$ in isolation from other language features. The beauty of type systems is that we may do so; the presence of a type of mutable references does not disrupt the behavior of the other constructs of the programming language in which they are embedded.

The static semantics of $\mathcal{L}\{\texttt{ref}\}$ consists of a set of rules for deriving typing judgements of the form $e : \tau$ that are parametric in two sets of parameters, one for locations and one for variables, and hypothetical in two forms of hypotheses specifying the type of the contents of a location and the type of the binding of a variable. The fully explicit form of the typing judgement for $\mathcal{L}\{\texttt{ref}\}$ is

$$\mathcal{L}\,\mathcal{X} \mid \Lambda\,\Gamma \vdash e : \tau,$$

where $\mathcal{L}$ is a finite set of locations, $\mathcal{X}$ is a finite set of variables, Λ is a finite set of assumptions of the form $l : \tau$, one for each $l \in \mathcal{L}$, and Γ is a finite set of assumptions of the form $x : \tau$, one for each $x \in \mathcal{X}$. As usual, we usually omit explicit mention of the parameters, writing just $\Lambda\,\Gamma \vdash e : \tau$.

The static semantics of $\mathcal{L}\{\mathtt{ref}\}$ is specified by the following rules:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash l : \mathtt{ref}(\tau)} \tag{37.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \mathtt{new}(e) : \mathtt{ref}(\tau)} \tag{37.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{ref}(\tau)}{\Lambda\,\Gamma \vdash \mathtt{get}(e) : \tau} \tag{37.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \mathtt{ref}(\tau_2) \quad \Lambda\,\Gamma \vdash e_2 : \tau_2}{\Lambda\,\Gamma \vdash \mathtt{set}(e_1;e_2) : \tau_2} \tag{37.1d}$$

The type of a location, l, when viewed as an expression, is a reference type, whereas the type assigned to l in the hypothesis is the type of its contents. The return type of $\mathtt{set}(e_1;e_2)$ is chosen more-or-less arbitrarily to be τ, as a technical convenience (the assignment is considered to return the assigned value as result).

A *memory* is a finite function mapping each of a finite set of locations to closed value (*i.e.*, one with no free variables). We write $\emptyset$ for the empty memory, $\langle l : e\rangle$ for the memory μ with domain $\{\, l \,\}$ such that $\mu(l) = e$, and $\mu_1 \otimes \mu_2$, where $dom(\mu_1) \cap dom(\mu_2) = \emptyset$, for the smallest memory μ such that $\mu(l) = \mu_1(l)$ if $l \in dom(\mu_1)$ and $\mu(l) = \mu_2(l)$ if $l \in dom(\mu_2)$. Whenever we write $\mu_1 \otimes \mu_2$ it is tacitly assumed that μ_1 and μ_2 are disjoint.

The dynamic semantics of $\mathcal{L}\{\mathtt{ref}\}$ consists of a transition systems between states of the form $e \mathbin{@} \mu$, where μ is a memory and e is an expression with no free variables. We will maintain the invariant that, in a state $e \mathbin{@} \mu$, the locations occurring in e, and in the contents of any cell in μ, lie within the domain of μ. An initial state has the form $e \mathbin{@} \emptyset$ in which the memory is empty and there are no locations occurring in e. A final state is one of the form $e \mathbin{@} \mu$, where e is a value.

The rules defining values and the transition judgement of the dynamic semantics of $\mathcal{L}\{\mathtt{ref}\}$ are given as follows:

$$\frac{}{l\ \mathsf{val}} \tag{37.2a}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\mathtt{new}(e) \mathbin{@} \mu \mapsto \mathtt{new}(e') \mathbin{@} \mu'} \tag{37.2b}$$

$$\frac{e\ \mathsf{val}}{\mathtt{new}(e) \mathbin{@} \mu \mapsto l \mathbin{@} \mu \otimes \langle l : e\rangle} \tag{37.2c}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\mathtt{get}(e) \mathbin{@} \mu \mapsto \mathtt{get}(e') \mathbin{@} \mu'} \tag{37.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{get}(l)\ @\ \mu \otimes \langle l : e\rangle \mapsto e\ @\ \mu \otimes \langle l : e\rangle} \tag{37.2e}$$

$$\frac{e_1\ @\ \mu \mapsto e_1'\ @\ \mu'}{\texttt{set}(e_1;e_2)\ @\ \mu \mapsto \texttt{set}(e_1';e_2)\ @\ \mu'} \tag{37.2f}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ @\ \mu \mapsto e_2'\ @\ \mu'}{\texttt{set}(e_1;e_2)\ @\ \mu \mapsto \texttt{set}(e_1;e_2')\ @\ \mu'} \tag{37.2g}$$

$$\frac{e\ \mathsf{val}}{\texttt{set}(l;e)\ @\ \mu \otimes \langle l : e'\rangle \mapsto e\ @\ \mu \otimes \langle l : e\rangle} \tag{37.2h}$$

In Rule (37.2c) it is tacitly assumed that l is chosen so as not to occur in the domain of μ. That is, l is a "new" location in the memory.

37.2 Safety

As usual, type safety is the conjunction of preservation and progress for well-formed machine states. Informally, the state $e\ @\ \mu$ is well-formed if (a) μ is well-formed relative to itself, and (b) e is well-formed relative to μ. The latter condition means that $e : \tau$ for some type τ, assuming that the locations have the types given to them by μ. The former means that the contents of each location, l, in μ has the type given to it relative to the types given to *all* the other locations by μ, including the location l itself.

This condition is reminiscent of the typing rule for recursive functions given in Chapter 16 in that we assume the typing that we are trying to prove while trying to prove it. In the case of recursive functions this is necessary to account for self-reference; in the case of memories, it is present to allow for circularities within the memory itself. One memory location, l, *depends on* another, l', in a memory μ if the contents of l in μ contains l'. It can arise that a location in a memory can depend on itself, either directly, or via an arbitrary finite chain of dependencies. Consequently, we must account for this when defining what it means for a memory to be well-formed.

The close relationship between the typing rules for memories and the typing rules for recursive functions is more than just a rough analogy. In fact we may use mutable storage to implement recursive functions, as illustrated by the following example:

```
let r be new(λ n:nat.n) in
let f be λ n:nat.ifz(n, 1, n'.n*get(r)(n')) in
let _ be set(r,f) in f
```

This expression returns a function of type $\mathtt{nat} \rightarrow \mathtt{nat}$ that is obtained by (a) allocating a reference cell initialized arbitrarily with a function of this type, (b) defining a λ-abstraction in which each "recursive call" consists of retrieving and applying the function stored in that cell, (c) assigning this function to the cell, and (d) returning that function. This technique is called *back-patching*.

The judgement $e \mathbin{@} \mu \; \mathsf{ok}$ is defined by the following rule:

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu : \Lambda}{e \mathbin{@} \mu \; \mathsf{ok}} \tag{37.3}$$

The hypotheses Λ are the types of the locations in the domain of μ. Since any location may appear in the expression e, it must be checked relative to the assumptions Λ. This is defined formally by the following rules:

$$\frac{}{\Lambda \vdash \emptyset : \emptyset} \tag{37.4a}$$

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu' : \Lambda'}{\Lambda \vdash \mu' \otimes \langle l : e \rangle : \Lambda', l : \tau} \tag{37.4b}$$

To account for circular dependencies, the contents of each location in memory is type-checked relative to the typing assumptions for all locations in memory.

Theorem 37.1 (Preservation). *If $e \mathbin{@} \mu$ ok and $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$, then $e' \mathbin{@} \mu'$ ok.*

Proof. The proof is by rule induction on Rules (37.2). For the sake of the induction we prove the following stronger result: if $\Lambda \vdash e : \tau$, $\Lambda \vdash \mu : \Lambda$, and $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$, then there exists $\Lambda' \supseteq \Lambda$ such that $\Lambda' \vdash e' : \tau$ and $\Lambda' \vdash \mu' : \Lambda'$.

Consider Rule (37.2c). We have $\mathtt{new}(e) \mathbin{@} \mu \mapsto l \mathbin{@} \mu'$, where e val and $\mu' = \mu \otimes \langle l : \sigma : e \rangle$. By inversion of typing $\Lambda \vdash e : \sigma$ and $\tau = \mathtt{ref}(\sigma)$. Taking $\Lambda' = \Lambda, l : \sigma$, observe that $\Lambda' \supseteq \Lambda$, and $\Lambda' \vdash l : \mathtt{ref}(\tau)$. Finally, we have $\Lambda' \vdash \mu' : \Lambda'$, since $\Lambda' \vdash \mu : \Lambda$ and $\Lambda' \vdash e : \sigma$ by assumption and weakening.

The other cases follow a similar pattern. □

Theorem 37.2 (Progress). *If $e \mathbin{@} \mu$ ok then either $e \mathbin{@} \mu$ is a final state or there exists $e' \mathbin{@} \mu'$ such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$.*

Proof. By rule induction on Rules (37.1). For the sake of the induction we prove the following stronger result: if $\Lambda \vdash e : \tau$ and $\Lambda \vdash \mu : \Lambda$, then either e val or there exists μ' and e' such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$.

Consider Rule (37.1c). We have $\Lambda \vdash \texttt{get}(e) : \tau$ because $\Lambda \vdash e : \tau$. By induction either $e\ \mathsf{val}$ or there exists μ' and e' such that $e \mathbin{\texttt{@}} \mu \mapsto e' \mathbin{\texttt{@}} \mu'$. In the latter case we have $\texttt{get}(e) \mathbin{\texttt{@}} \mu \mapsto \texttt{get}(e') \mathbin{\texttt{@}} \mu'$ by Rule (37.2d). In the former it follows from an analysis of Rules (37.1) that $e = l$ for some location l such that $\Lambda = \Lambda', l : \tau$. Since $\Lambda \vdash \mu : \Lambda$, it follows that $\mu = \mu' \otimes \langle l : e' \rangle$ for some μ' and e' such that $\Lambda \vdash e' : \tau$. But then by Rule (37.2e) we have $\texttt{get}(e) \mathbin{\texttt{@}} \mu \mapsto e' \mathbin{\texttt{@}} \mu$.

The remaining cases follow a similar pattern. □

37.3 Exercises

Chapter 38

Dynamic Classification

Sum types may be used to classify data values by labelling them with a class identifier that determines the type of the associated data item. For example, a sum type of the form $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ consists of n distinct classes of data, with the ith class labelling a value of type τ_i. A value of this type is introduced by the expression $\texttt{in[}i\texttt{](}e_i\texttt{)}$, where $0 \leq i < n$ and $e_i : \tau_i$, and is eliminated by an n-ary case analysis binding the variable x_i to the value of type τ_i labelled with class i.

Sum types are useful in situations where the type of a data item is determined dynamically, for example when processing input from an external data source. A class is used to label the data items so that the type of the underlying datum may be recovered from its class label. The difficulty with sum types, however, is that it requires that the programmer specify in advance the possible classes of data that may arise in a given situation. For example, the code to process the data entered into a form on a web page might yield either an integer or a string, according to what was typed by the user. This may be modelled using a sum type with two classes, one for integers, the other for strings.

The form of classification provided by finite sum types may be called *static classification*, since it requires that the labels of the summands be fixed in advance at the time the program is written. In some situations, however, it is overly restrictive to demand static classification. Instead, some form of dynamic determination of the class of data items is required, which we call *dynamic classification*. A major use of dynamic classification is to impose *privacy restrictions* on data values in which one program component may create a data item that can *only* be processed by certain other components, *even though* it may be manipulated passively by any number of other components (say, as a component of a data structure, or as an uninterpreted argument to a function).

Dynamic classification may be used to implement privacy as follows. First, the sender and receiver(s) of the data agree on a freshly generated classifier for the secret data. The sender generates a new class that is guaranteed to be distinct from all others, and communicates the identity of that class only to the intended recipients of the data. Then, when the data is ready to be transmitted, it is classified by the agreed-upon class, and propagated *without further restriction* to the recipients. For example, it could be stored in a data structure such as a hash table or passed as argument to a function, eventually making its way to an intended receiver. Knowing the class of the data, an intended receiver may "decode" the datum by dispatching on its class to recover the underlying value. Any component that is not aware of the class cannot recover this value, ensuring its secrecy.

A very common example of this sort of interaction arises when programming with exceptions, as described in Chapter 30. One may consider the value associated with an exception to be a "secret" that is shared between the program component that raises the exception and the program component that handles it. No other intervening handler may intercept the exception value; only the designated handler is permitted to process it. This behavior may be readily modelled using dynamic classification. Exception values are dynamically classified, with the class of the value known only to the raiser and to the intended handler, and to no others. For this reason, the type τ_{exn} of exception values may be usefully chosen to be the type of dynamically classified data described in the present chapter.

The role of *dynamic* classification for exception handling is not widely understood. Why not use *static* classification? One obvious reason is that to do so would require that the set of classes of exception values must be fixed globally, and in advance, by the programmer, which is evidently inconvenient. Instead, one would prefer to introduce classes of exceptions wherever they are needed, rather than all at once in a global scope. But then one immediately encounters a serious problem: how can we ensure that two different (separately developed and separately compiled) components do not accidentally use the *same* class for two *different* purposes? In the interest of modularity, it is essential to ensure that this cannot occur. Dynamic classification precludes it, even in the presence of dynamically loaded components.

In this chapter we study two separable concepts, *dynamic classification*, and *dynamic classes*. *Dynamic classification* permits the classification of data values using dynamically generated symbols (as described in Section 36.2 on page 285 of Chapter 36). A value is classified by tagging it with a symbol that determines the type of its associated value. A classified value is inspected by pattern matching against a finite set of known classes, dispatching according to whether the class of the value is among them, with a default behavior if it is not. *Dynamic classes* treat class labels as a form of dynamic data. This allows a class to be communicated between components at run-time without embedding it into another data structure. Dynamic classes, in combination with product and existential types (Chapter 26), are sufficient to implement dynamic classification.

38.1 Dynamic Classification

The language $\mathcal{L}\{\texttt{classified}\}$ of dynamically classified data employs the symbol generation mechanism described in Chapter 36 to generate fresh symbols at execution time.[1] As discussed there, the principle of implicit renaming of bound identifiers ensures that freshly generated symbols are globally unique, which ensures that no accidental collisions can occur. Symbols are used as classes, with the type of the symbol determining the type of the classified data value.

The syntax of $\mathcal{L}\{\texttt{classified}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{clsfd}$	$\texttt{clsfd}$
Expr	e	$::=$	$\texttt{in[}a\texttt{](}e\texttt{)}$	$\texttt{in[}a\texttt{](}e\texttt{)}$
		$\mid$	$\texttt{ccase(}e;e_0;r_1\texttt{,}\ldots\texttt{,}r_n\texttt{)}$	$\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e_0$
Rule	r	$::=$	$\texttt{in?[}a\texttt{](}x\texttt{.}e\texttt{)}$	$\texttt{in[}a\texttt{](}x\texttt{)} \Rightarrow e$

The expression $\texttt{in[}a\texttt{](}e\texttt{)}$ classifies the value of the expression e by labelling it with the symbol a. The expression $\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e_0$ analyzes the class of e according to the rule $r_1, \ldots, r_n$. Each rule has the form $\texttt{in[}a_i\texttt{](}x_i\texttt{)} \Rightarrow e_i$, consisting of a symbol, a_i, representing a candidate class of the analyzed value; a variable, x_i, representing the associated data value for a value of that class; and an expression, e_i, to be evaluated in the case that the analyzed expression is labelled with class a_i. If the class of the analyzed value does not match any of the rules, the default expression, e_0, is evaluated instead.

[1] Although symbol generation was introduced in Chapter 36, it is important to recognize that the mechanism of dynamic classification has nothing whatsoever to do with fluid binding!

Observe that there is, in general, no possibility for the case analysis on the class of a value to be exhaustive. Since classes are dynamically generated, and since the analyzed data value could have arisen anywhere in a program, there is no way to ensure that the class of the analyzed value is among those listed in the case analysis. For this reason it is essential to provide a default case so that evaluation has a well-defined outcome, even when none of the specified cases applies. Alternatively, one may integrate the primitive case analysis construct of $\mathcal{L}\{\mathtt{classified}\}$ into the general, non-exhaustive cdmatch expression considered in Chapter 19, using a wild card to cover the default case.

The static semantics of $\mathcal{L}\{\mathtt{classified}\}$ consists of a judgement of the form $\Sigma\,\Gamma \vdash e : \tau$, where Σ specifies the types of the symbols (as described in Chapter 36), and Γ specifies the types of the variables. The definition makes use of an auxiliary judgement of the form $\Sigma\,\Gamma \vdash r : \tau$, specifying that a rule, r, matches a classified value of the form $\mathtt{in[}a\mathtt{](}e\mathtt{)}$ and yields a value of type τ. These judgements are inductively defined by the following rules:

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau}{\Sigma\,\Gamma \vdash \mathtt{in[}a\mathtt{](}e\mathtt{)} : \mathtt{clsfd}} \tag{38.1a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \mathtt{clsfd} \quad \Sigma\,\Gamma \vdash r_1 : \tau \quad \dots \quad \Sigma\,\Gamma \vdash r_n : \tau}{\Sigma\,\Gamma \vdash \mathtt{ccase(}e;e_0;r_1\mathtt{,}\dots\mathtt{,}r_n\mathtt{)} : \tau} \tag{38.1b}$$

$$\frac{\Sigma \vdash a : \sigma \quad \Sigma\,\Gamma, x : \sigma \vdash e : \tau}{\Sigma\,\Gamma \vdash \mathtt{in?[}a\mathtt{](}x\mathtt{.}e\mathtt{)} : \tau} \tag{38.1c}$$

The dynamic semantics of these operations is an entirely straightforward extension of the semantics of dynamic symbol generation given in Section 36.2 on page 285.

$$\frac{e\ \mathsf{val}}{\mathtt{in[}a\mathtt{](}e\mathtt{)}\ \mathsf{val}} \tag{38.2a}$$

$$\frac{e \mathbin{\mathtt{@}} \nu \mapsto e_0 \mathbin{\mathtt{@}} \nu'}{\mathtt{in[}a\mathtt{](}e\mathtt{)} \mathbin{\mathtt{@}} \nu \mapsto \mathtt{in[}a\mathtt{](}e_0\mathtt{)} \mathbin{\mathtt{@}} \nu'} \tag{38.2b}$$

$$\frac{e \mathbin{\mathtt{@}} \nu \mapsto e' \mathbin{\mathtt{@}} \nu'}{\mathtt{ccase(}e;e_0;r_1\mathtt{,}\dots\mathtt{,}r_n\mathtt{)} \mathbin{\mathtt{@}} \nu \mapsto \mathtt{ccase(}e';e_0;r_1\mathtt{,}\dots\mathtt{,}r_n\mathtt{)} \mathbin{\mathtt{@}} \nu'} \tag{38.2c}$$

$$\frac{\mathtt{in[}a\mathtt{](}e\mathtt{)}\ \mathsf{val}}{\mathtt{ccase(in[}a\mathtt{](}e\mathtt{)};e_0;\epsilon\mathtt{)} \mathbin{\mathtt{@}} \nu \mapsto e_0 \mathbin{\mathtt{@}} \nu} \tag{38.2d}$$

$$\frac{\mathtt{in[}a_1\mathtt{](}e_1\mathtt{)}\ \mathsf{val}}{\begin{array}{c}\mathtt{ccase(in[}a_1\mathtt{](}e_1\mathtt{)};e_0;\mathtt{in?[}a_1\mathtt{](}x_1\mathtt{.}e_1'\mathtt{)},\dots,\mathtt{in?[}a_n\mathtt{](}x_n\mathtt{.}e_n'\mathtt{))} \mathbin{\mathtt{@}} \nu \\ \mapsto \\ [e_1/x_1]e_1' \mathbin{\mathtt{@}} \nu\end{array}} \tag{38.2e}$$

$$\frac{\texttt{in[}a\texttt{](}e\texttt{)}\ \mathsf{val} \quad a \neq a_1 \quad n > 0}{\begin{array}{c}\texttt{ccase(in[}a_1\texttt{](}e_1\texttt{);}e_0\texttt{;in?[}a_1\texttt{](}x_1\texttt{.}e_1'\texttt{),}\ldots\texttt{,in?[}a_n\texttt{](}x_n\texttt{.}e_n'\texttt{))}\ \texttt{@}\,\nu \\ \longmapsto \\ \texttt{ccase(in[}a\texttt{](}e\texttt{);}e_0\texttt{;in?[}a_2\texttt{](}x_2\texttt{.}e_2'\texttt{),}\ldots\texttt{,in?[}a_n\texttt{](}x_n\texttt{.}e_n'\texttt{))}\ \texttt{@}\,\nu\end{array}} \tag{38.2f}$$

Rule (38.2d) specifies that the default case is evaluated when all rules have been exhausted (*i.e.*, the sequence of rules is empty). Rules (38.2e) and (38.2f) specify that each rule is considered in turn, matching the class of the analyzed expression to the class of each of the successive rules of the case analysis.

The statement and proof of type safety for $\mathcal{L}\{\texttt{classified}\}$ proceeds along the lines of the safey proofs given in Chapters 18, 19, and 36.

Theorem 38.1 (Preservation)**.** *Suppose that e @$\nu \mapsto e'$ @ν', where $\Sigma \vdash e : \tau$ and $\Sigma \vdash a : \tau_a$ whenever $a \in \nu$. Then $\nu' \supseteq \nu$, and there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : \tau$ and $\Sigma' \vdash a' : \tau_{a'}$ for each $a' \in \nu'$.*

Lemma 38.2 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e :$ clsfd and e val. Then $e =$ in[a](e) for some a such that $\Sigma \vdash a : \tau$ and some e such that e val and $\Sigma \vdash e : \tau$.*

Theorem 38.3 (Progress)**.** *Suppose that $\Sigma \vdash e : \tau$, and that if $a \in \nu$, then $\Sigma \vdash a : \tau_a$ for some type τ_a. Then either e val, or e @$\nu \mapsto e'$ @ν' for some ν' and e'.*

38.2 Dynamic Classes

Dynamic classification may be used in combination with higher-order functions to provide controlled access to data among the components of a program as described in the introduction to this chapter. Given a dynamically generated (and hence globally unique) symbol, a, of type τ, one may define two functions of type $\tau \rightarrow$ `clsfd` and `clsfd` $\rightarrow \tau$ that, respectively, classify a value of type τ with class a and declassify a value classified by a, failing otherwise. Either or both of these functions may be passed out of the scope of the binder that introduced the symbol a, ensuring that knowledge of its identity is confined to these two operations. Any component with access to the classification operation may create a value with class a, and only those components with access to the declassification operation may recover the classified value.

A more direct way to enforce privacy is to treat classes themselves as values of type τ `class`, where τ is the type of data labelled by that class. The language $\mathcal{L}\{\texttt{class}\}$ consists of the dynamic symbol generation mechanism described in Chapter 36 together with the primitive operations for the type $\tau\,\mathtt{class}$. The syntax of $\mathcal{L}\{\mathtt{class}\}$ is specified by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{class}(\tau)$	$\tau\,\mathtt{class}$
Expr	e	::=	$\mathtt{cls}[a]$	$\mathtt{cls}[a]$
		$\mid$	$\mathtt{ccase}[t.\sigma](e;e_0;r_1,\ldots,r_n)$	$\mathtt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\mathtt{ow}\,e_0$
Rule	r	$\mid$	$\mathtt{cls?}[a](e)$	$\mathtt{cls}[a] \Rightarrow e$

The type $\tau\,\mathtt{class}$ represents the type of classes with associated values of type τ. A value of this type has the form $\mathtt{cls}[a]$, where a is a symbol labelling a class of values. The expression $\mathtt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\mathtt{ow}\,e_0$ is analogous to the class case construct of $\mathcal{L}\{\mathtt{classified}\}$, except that there is no data associated with each class. The abstractor, $t.\sigma$, in the syntax of the class case construct plays a critical role in the static semantics of $\mathcal{L}\{\mathtt{class}\}$.

The static semantics of $\mathcal{L}\{\mathtt{class}\}$ makes use of a technique, sometimes facetiously called the *John Major Principle*, to propagate type identity information gained during pattern matching. Suppose that e is an expression of type $\mathtt{class}(\tau)$, and that we analyze its class using a series of rules of the form $\mathtt{cls?}[a_i](e_i)$, where each symbol a_i has the corresponding type τ_i. The type of e ensures that its value is of the form $\mathtt{cls}[a]$ for some class symbol a of type τ. The typing rule for the case analysis must allow for the possibility that a is one of the a_i's, in which case we must propagate the fact that τ_i is, in fact, τ. This is achieved by assigning the case analysis the type $[\tau/t]\sigma$, and insisting that for each $1 \leq i \leq n$, we have that $e_i : [\tau_i/t]\sigma$. In the case that a is a_i, then we are, in effect, treating an expression of type $[\tau_i/t]\sigma$ as an expression of type $[\tau/t]\sigma$, which is justified by the equality of τ_i and τ.

The static semantics of $\mathcal{L}\{\mathtt{class}\}$ consists of expression typing judgements of the form $\Sigma\,\Gamma \vdash e : \tau$, and rule typing judgements of the form $\Sigma\,\Gamma \vdash r : \mathtt{class}(\tau) > \tau'$. These judgements are inductively defined by the following rules:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \mathtt{cls}[a] : \mathtt{class}(\tau)} \tag{38.3a}$$

$$\frac{\begin{array}{c}\Sigma\,\Gamma \vdash e : \mathtt{class}(\tau) \qquad \Sigma\,\Gamma \vdash e_0 : [\tau/t]\sigma \\ \Sigma\,\Gamma \vdash r_1 : \mathtt{class}(\tau_1) > [\tau_1/t]\sigma \quad \ldots \quad \Sigma\,\Gamma \vdash r_n : \mathtt{class}(\tau_n) > [\tau_n/t]\sigma\end{array}}{\Sigma\,\Gamma \vdash \mathtt{ccase}[t.\sigma](e;e_0;r_1,\ldots,r_n) : [\tau/t]\sigma} \tag{38.3b}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau'}{\Sigma\,\Gamma \vdash \mathtt{cls?}[a](e) : \mathtt{class}(\tau) > \tau'} \tag{38.3c}$$

Rule (38.3a) specifies that the class $\texttt{cls[}a\texttt{]}$ has type $\texttt{class(}\tau\texttt{)}$, where τ is the type associated to the class a by Σ. Rule (38.3b) specifies the type of a case analysis on a class of type $\texttt{class(}\tau\texttt{)}$ to be $[\tau/t]\sigma$, where each rule yields a value of type $[\tau_i/t]\sigma$, and the default case is of type $[\tau/t]\tau$.

The dynamic semantics of $\mathcal{L}\{\texttt{class}\}$ is similar to that of $\mathcal{L}\{\texttt{classified}\}$. States have the form $e\,\texttt{@}\,\nu$, where ν is a finite set of symbols. Final states are those for which e val; all states are initial states. The rules defining the judgement $e\,\texttt{@}\,\nu \mapsto e'\,\texttt{@}\,\nu'$ are easily derived from Rules (38.2), and are omitted here for the sake of brevity.

Theorem 38.4 (Preservation)**.** *Suppose that $e\,\texttt{@}\nu \mapsto e'\,\texttt{@}\nu'$, where $\Sigma \vdash e : \tau$ and $\Sigma \vdash a : \tau_a$ whenever $a \in \nu$. Then $\nu' \supseteq \nu$, and there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : \tau$ and $\Sigma' \vdash a' : \tau_{a'}$ for each $a' \in \nu'$.*

Proof. By rule induction on the dynamic semantics of $\mathcal{L}\{\texttt{class}\}$. The most interesting case arises when $e = \texttt{cls[}a\texttt{]}$ and $a = a_i$ for some rule $\texttt{cls[}a_i\texttt{]} \Rightarrow e_i$. By inversion of typing we know that $e_i : [\tau_i/t]\sigma$. We are to show that $e_i : [\tau/t]\sigma$. This follows directly from the observation that if $a = a_i$, then by unicity of typing, $\tau_i = \tau$. □

Lemma 38.5 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e : \tau\ \texttt{class}$ and e val. Then $e = \texttt{cls[}a\texttt{]}$ for some a such that $\Sigma \vdash a : \tau$.*

Proof. By rule induction on Rules (38.3), taking account of the definition of values. □

Theorem 38.6 (Progress)**.** *Suppose that $\Sigma \vdash e : \tau$, and that if $a \in \nu$, then $\Sigma \vdash a : \tau_a$ for some type τ_a. Then either e val, or $e\,\texttt{@}\nu \mapsto e'\,\texttt{@}\nu'$ for some ν' and e'.*

Proof. By rule induction on Rules (38.3). For a case analysis of the form $\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e_0$, where e val, we have by Lemma 38.5 that $e = \texttt{cls[}a\texttt{]}$ for some a. Either $a = a_i$ for some rule $\texttt{cls[}a_i\texttt{]} \Rightarrow e_i$ in $r_1, \ldots, r_n$, in which case we progress to e_i, or else we progress to e_0. □

38.3 From Classes to Classification

The language $\mathcal{L}\{\texttt{classified}\}$ is definable from the language $\mathcal{L}\{\texttt{class}\}$ augmented with existential and product types. Specifically, we may define $\texttt{clsfd}$ to be the existential type

$$\exists\texttt{(}t\,\texttt{.}\,t\,\texttt{class} \times t\texttt{)}.$$

The introductory form, $\texttt{in}[a](e)$, is defined to be the package

$$\texttt{pack}\,\tau\,\texttt{with}\,\langle\texttt{cls}[a],e\rangle\,\texttt{as}\,\exists(t.t\,\texttt{class}\times t),$$

where a is a symbol of type τ.

The eliminatory form has a slightly more complex definition. Suppose that $\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e'$ has type ρ, where r_i is the rule $\texttt{in}[a_i](x_i) \Rightarrow e_i$, with $x_i : \tau_i \vdash e_i : \rho$. We define this class case to be the expression

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle x,y\rangle : t\,\texttt{class}\times t\,\texttt{in}\,(e_{body}(y)),$$

where e_{body} will be defined shortly. This expression opens the package, e, representing the classified value, and decomposes it into a class, x, and an associate value, y. The body of the open analyzes the class x, yielding a *function* of type $t \rightarrow \rho$, where t is the abstract type introduced by the open. This function is applied to y, the value that is labelled by x in the package.

The expression e_{body} determines the function to apply by performing a case analysis on the class x. The case analysis is parameterized by the type abstractor $u\,.\,u \rightarrow \rho$, where u is not free in ρ. The overall type of the case is $[t/u]u \rightarrow \rho = t \rightarrow \rho$, which ensures that the above-mentioned application to y is well-typed. Each branch of the case analysis has type $\tau_i \rightarrow \rho$, as required by Rule (38.3b). The expression e_{body} is given by

$$\texttt{ccase}\,x\,\{r'_1 \mid \ldots \mid r'_n\}\,\texttt{ow}\,\lambda(\_:t.\,e_0),$$

where for each $1 \leq i \leq n$, we define r'_i to be the rule

$$\texttt{cls}[a_i] \Rightarrow \lambda(x_i:\tau_i.\,e_i).$$

It is easy to check that the static and dynamic semantics of $\mathcal{L}\{\texttt{classified}\}$ are derivable in $\mathcal{L}\{\texttt{class}\}$ (enriched with products and existentials) according to these definitions.

38.4 Exercises

1. Derive the Standard ML exception mechanism from the machinery developed here.

Part XIII

Modalities for Effects

Chapter 39

Monads

Computational effects, whether control effects such as exceptions or storage effects such as references, strongly influence the meanings of programs. For example, in the absence of effects the type `nat` → `nat` may be thought of as the type of mathematical functions on the natural numbers: given any natural number as argument, a function of this type evaluates to a unique natural number. General recursion weakens this to there being at most one result for each application; in some cases the function might not return. Exceptions weaken the meaning still further, because a function of this type might, on a given argument, raise an exception rather than fail to terminate or return a natural number as result. Raising an exception might be considered tantamount to not returning, but since an exception can be caught by a handler, whereas an infinite loop cannot, the analogy is not quite exact. References weaken the meaning still further, since now a "function" of this type might even return a different result from each call, even if the argument is the same! Indeed, in the absence of effects a type such as `unit` → `unit` is uninteresting (it contains only the identity function), but in the presence of storage effects, or input/output effects, many useful programs may be considered to have this type.

One attitude about this is to eschew computational effects entirely, sticking to pure functional programming in which every expression has one (or at most one) value. But this is quite unrealistic. After all, the entire purpose of running a program is to have a "side effect" on the user! Another attitude is to simply accept effects as a fact of life, and to live with the weakened guarantees afforded by the type system. However, there is a useful middle ground between these two extremes in which we maintain a distinction between *pure* and *impure* computations, being respectively those that are effect-free and those that may have effects. A pure computation is executed solely to determine its value, where as an impure computation is executed both for its value and for its effect on the context of execution.

One benefit of distinguishing pure from impure computations is that pure computations are more amenable to mathematical reasoning. For example, e_1+e_2 and e_2+e_1 are equivalent, provided that e_1 and e_2 are effect-free. If, however, either has an effect (raises an exception or modifies storage), then this natural equivalence need not hold. For example, swapping the arguments in the impure expression `raise(L)+raise(R)` changes the meaning of the expression, since one raises `L` and the other raises `R`. Storage effects give rise to even more subtle and complex dependencies on evaluation order.

Because the distinction matters to the semantics of expressions, it is useful to introduce a type discipline that makes a corresponding distinction in their syntax so that it is evident when a computation might have an effect, and when it is guaranteed to be effect-free. Such a distinction between forms of expression is, in general, called a *modality*. Here we shall be concerned with a particular modality, called a *monad*, or *lax modality*. A monad makes a distinction between two forms of expression:

1. *Pure expressions*, which have no computational effects (except, possibly, non-termination).

2. *Impure commands*, which both have a value and an effect.

The modality allows us to maintain a clear distinction between pure and impure computations.

The syntax of commands is structured to ensure that commands are executed in a fixed sequential order. The basic form of command is the *return* command, which simply returns a specified value without engendering any effects. Commands are combined using the *monadic bind* construct, which sequentializes execution of one command before another. These two forms of command are shared by all monadic effects, and give rise to a concrete syntax that is familiar from conventional imperative (command-oriented) programming languages. In this chapter we introduce a skeleton monadic language, called $\mathcal{L}\{\texttt{comm}\}$, that captures this basic structure without specifying the forms of effect. These are modelled by extending $\mathcal{L}\{\texttt{comm}\}$ with new forms of command that give rise to effects of interest, such as storage or control effects.

39.1 Monadic Framework

The syntax of $\mathcal{L}\{\texttt{comm}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{comp}(\tau)$	$\tau\,\texttt{comp}$
Expr	e	$::=$	x	x
		$\mid$	$\texttt{comp}(m)$	$\texttt{comp}(m)$
Comm	m	$::=$	$\texttt{return}(e)$	$\texttt{return}\,e$
		$\mid$	$\texttt{letcomp}(e; x\,.\,m)$	$\texttt{let}\,\texttt{comp}(x)\,\texttt{be}\,e\,\texttt{in}\,m$

The language $\mathcal{L}\{\texttt{comm}\}$ distinguishes distinguishes two *modes*, the pure (effect-free) *expressions*, and the impure (effect-capable) *commands*. The *monadic type* $\texttt{comp}(\tau)$ consists of suspended commands that, when evaluated, yield a value of type τ. The expression $\texttt{comp}(m)$ introduces an unevaluated command as a value of monadic type. The command $\texttt{return}(e)$ returns the value of e as its value, without engendering any effects. The command $\texttt{letcomp}(e; x\,.\,m)$ activates the suspended command obtained by evaluating the expression e, then continue by evaluating the command m. This form sequences evaluation of commands so that there is no ambiguity about the order in which effects occur during evaluation.

The static semantics of $\mathcal{L}\{\texttt{comm}\}$ consists of two forms of typing judgement, $e : \tau$, stating that the expression e has type τ, and $m \sim \tau$, stating that the command m only yields values of type τ. Both of these judgement forms are considered with respect to hypotheses of the form $x : \tau$, which states that a variable x has type τ. The rules defining the static semantics of $\mathcal{L}\{\texttt{comm}\}$ are as follows:

$$\frac{\Gamma \vdash m \sim \tau}{\Gamma \vdash \texttt{comp}(m) : \texttt{comp}(\tau)} \tag{39.1a}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) \sim \tau} \tag{39.1b}$$

$$\frac{\Gamma \vdash e : \texttt{comp}(\tau) \quad \Gamma, x : \tau \vdash m \sim \tau'}{\Gamma \vdash \texttt{letcomp}(e; x\,.\,m) \sim \tau'} \tag{39.1c}$$

The dynamic semantics of an instance of $\mathcal{L}\{\texttt{comm}\}$ is specified by two transition systems:

1. *Evaluation* of expressions: $e \mapsto e'$, e val.

2. *Execution* of commands: $m \mapsto m'$, m final.

The rules of expression evaluated are determined by the type structure of expressions, and are carried over to $\mathcal{L}\{\texttt{comm}\}$ unchanged.

The rules of command execution include the following *structural* rules of execution:

$$\frac{e \mapsto e'}{\texttt{return}(e) \mapsto \texttt{return}(e')} \tag{39.2a}$$

$$\frac{e \;\mathsf{val}}{\texttt{return}(e) \;\mathsf{final}} \tag{39.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e;x.m) \mapsto \texttt{letcomp}(e';x.m)} \tag{39.2c}$$

$$\frac{m_1 \mapsto m_1'}{\texttt{letcomp(comp(}m_1\texttt{);}x.m_2\texttt{)} \mapsto \texttt{letcomp(comp(}m_1'\texttt{);}x.m_2\texttt{)}} \tag{39.2d}$$

$$\frac{\texttt{return}(e) \;\mathsf{final}}{\texttt{letcomp(comp(return(}e\texttt{));}x.m\texttt{)} \mapsto [e/x]m} \tag{39.2e}$$

Rules (39.2a) and (39.2c) specify that the expression part of a `return` or `let` command is to be evaluated before execution can proceed. Rule (39.2b) specifies that a `return` command whose argument is a value is a final state of command execution. Rule (39.2d) specifies that a `let` activates an encapsulated command, and Rule (39.2e) specifies that a completed command passes its return value to the body of the `let`.

Extensions to $\mathcal{L}\{\texttt{comm}\}$ are defined by adding new forms of command, and new transition rules between commands. These rules often make use of labelled transition systems (described in Chapter 4) in which the labels express the effects of a command on the context of its execution. The structural rules given above are to be regarded as silent transitions that have no influence on the context, but are required to ensure that commands are executed in the proper order.

39.2 Programming With Monads

The monadic type, $\tau\,\texttt{comp}$, is the type of encapsulated commands yielding a value of type τ. The introductory form for this type is $\texttt{comp}(m)$, and the eliminatory form is $\texttt{let comp}(x)\;\texttt{be}\;e\;\texttt{in}\;m$. The only other form of command, other than primitives for specific effects, is the command $\texttt{return}\,e$. Consequently, a command of type τ must have the form

$$\texttt{let comp}(x_1)\;\texttt{be}\;e_1\;\texttt{in}\ldots\texttt{let comp}(x_n)\;\texttt{be}\;e_n\;\texttt{in}\,\texttt{return}\,e,$$

where $e_1 : \tau_1\,\texttt{comp}, \ldots, e_n : \tau_n\,\texttt{comp}$, and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$. This serves to sequence the encapsulated commands determined by $e_1, \ldots, e_n$ in the order specified, and returning the value of e as a result.

Observe that the only means of including a command within an expression is to encapsulate it inside the monad. Since the only means of executing an ecapsulated command is with the `let` command, there is no means of using the result of executing a command to compute the value of an expression. This is as it should be, for otherwise the evaluation of such an expression would engender effects, ruining the very distinction we seek to enforce with a monad. Put another way, it is impossible to define an *expression* $\texttt{run}\,e$ of type τ, where $e : \tau\,\texttt{comp}$, whose value is the result of running the command encapsulated in the value of e. There is, however, such a *command*, namely

$$\texttt{let}\,\texttt{comp}(x)\,\texttt{be}\,e\,\texttt{in}\,\texttt{return}\,x.$$

The only form of sequencing in $\mathcal{L}\{\texttt{comm}\}$ is the elimination form for the monadic type. This means that if we wish to execute commands m_1 and m_2 in sequence, passing the value of m_1 to m_2 via a variable x_1, then we must write

$$\texttt{let}\,\texttt{comp}(x_1)\,\texttt{be}\,\texttt{comp}(m_1)\,\texttt{in}\,m_2,$$

which encapsulates m_1 only to activate it before executing m_2. More generally, to execute a sequence of commands in this manner, we must write

$$\texttt{let}\,\texttt{comp}(x_1)\,\texttt{be}\,\texttt{comp}(m_1)\,\texttt{in}\ldots\texttt{let}\,\texttt{comp}(x_{k-1})\,\texttt{be}\,\texttt{comp}(m_{k-1})\,\texttt{in}\,m_k,$$

which quickly gets out of hand. For this reason we introduce the *`do` syntax*, which is reminiscent of the notation used in many imperative programming languages. The binary `do` construct, $\texttt{do}\,\{x \leftarrow m_1\,;\,m_2\}$, stands for the command

$$\texttt{let}\,\texttt{comp}(x)\,\texttt{be}\,\texttt{comp}(m_1)\,\texttt{in}\,m_2,$$

which executes the commands m_1 and m_2 in sequence, passing the value of m_1 to m_2 via the variable x. The general `do` construct,

$$\texttt{do}\,\{x_1 \leftarrow m_1\,;\ldots;\,x_k \leftarrow m_k\,;\,\texttt{return}\,e\},$$

is defined by iteration of the binary `do` as follows:

$$\texttt{do}\,\{x_1 \leftarrow m_1\,;\ldots\texttt{do}\,\{x_k \leftarrow m_k\,;\,\texttt{return}\,e\}\ldots\}.$$

Suppose that the expression level of $\mathcal{L}\{\texttt{comm}\}$ is enriched with function types, as in Chapter 14. Since the body of a λ-abstraction is an expression, any effects in the body of a function must be encapsulated in the monad. Such a function has a type of the form $\sigma \to \tau\,\texttt{comp}$, which may be read as saying that the function, when applied to an argument of type σ, yields a computation of type τ. The application does not activate the computation; this can only be done using the elimination form for the monad. Thus, if f is such a function, and a is an argument of appropriate type, we must write

$$\texttt{let}\,\texttt{comp}(x)\,\texttt{be}\,f(a)\,\texttt{in}\,m$$

to activate the result of the application *within another command m*. The application of f to a is an expression yielding an encapsulated command; activation of that command can only occur withn another command.

39.3 Exercises

Chapter 40

Monadic Exceptions

As we saw in Chapter 39, if an expression can raise an exception, then the order of evaluation of sub-expressions of an expression is significant. For example, the expression e_1+e_2 is not in general equivalent to e_2+e_1, even though addition is commutative. This is so because in the presence of exceptions an expression of type `nat` need not evaluate to a number—it can, instead, raise an exception. If e_1 is `raise(L)` and e_2 is `raise(R)`, then we may use a handler to distinguish the two addition expressions from each other, yielding, say, zero in the one case and one in the other.

The semantics of expressions may be preserved even in the presence of exceptions if we confine them to the monad by making the primitives for raising and handling exceptions commands, rather than expressions. In this chapter we study a variation on $\mathcal{L}\{\texttt{comm}\}$ in which exceptions are treated as an impurity to be confined to commands. It should be noted, however, that this approach is unsatisfactory for two related reasons. First, because the monad imposes a strict sequential execution order on commands, the programmer must specify an evaluation order whenever an exception might be raised. Second, if any exception can appear *somewhere* in a program, then it must be structured as though an exception could appear *anywhere*. This is because there is no means of "escaping the monad"—an impurity somewhere infects all parts of the program that depend on its result.

40.1 Monadic Exceptions

The most natural formalization of exceptions in the monadic framework is to regard an exception as an alternative outcome of evaluation of a command. That is, a command, when executed, may engender effects, and then either return a value (as in $\mathcal{L}\{\texttt{comm}\}$) or raise an exception. The language $\mathcal{L}\{\texttt{comm}\,\texttt{exc}\}$ is a modification of $\mathcal{L}\{\texttt{comm}\}$ to account for this additional outcome of execution. The following grammar specifies the characteristic features of $\mathcal{L}\{\texttt{comm}\,\texttt{exc}\}$:

Category	*Item*		*Abstract*	*Concrete*
Comm	m	$::=$	$\texttt{raise[}\tau\texttt{](}e\texttt{)}$	$\texttt{raise(}e\texttt{)}$
		$\mid$	$\texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)}$	$\texttt{let}\,\texttt{comp(}x\texttt{)}\,\texttt{be}\,e\,\texttt{in}\,m_1\,\texttt{ow(}y\texttt{)}\,\texttt{in}\,m_2$

This grammar extends that of $\mathcal{L}\{\texttt{comm}\}$ with a new primitive command, $\texttt{raise(}e\texttt{)}$, that raises an exception with value e. It also modifies the grammar of $\mathcal{L}\{\texttt{comm}\}$ to generalize the monadic bind construct to include an exception handler. The command

$$\texttt{let}\,\texttt{comp(}x\texttt{)}\,\texttt{be}\,e\,\texttt{in}\,m_1\,\texttt{ow(}y\texttt{)}\,\texttt{in}\,m_2$$

executes the encapsulated command determined by evaluation of e. If it returns normally, then the return value is bound to x and the command m_1 is executed. If, instead, it raises an exception, the exception value is bound to y and the command m_2 is executed instead. The monadic bind construct of $\mathcal{L}\{\texttt{comm}\}$ is to be regarded as short-hand for the command

$$\texttt{let}\,\texttt{comp(}x\texttt{)}\,\texttt{be}\,e\,\texttt{in}\,m\,\texttt{ow(}y\texttt{)}\,\texttt{in}\,\texttt{raise(}y\texttt{)},$$

which behaves as before in the case of a normal return, and propagates any exception in that case of an exceptional return.

The static semantics of these constructs is given by the following rules:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} \sim \tau} \tag{40.1a}$$

$$\frac{\Gamma \vdash e : \texttt{comp(}\tau\texttt{)} \quad \Gamma, x : \tau \vdash m_1 \sim \tau' \quad \Gamma, y : \tau_{exn} \vdash m_2 \sim \tau'}{\Gamma \vdash \texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)} \sim \tau'} \tag{40.1b}$$

The dynamic semantics of these commands consists of a transition system of the form $m \mapsto m'$ defined by the following rules:

$$\frac{e \mapsto e'}{\texttt{return(}e\texttt{)} \mapsto \texttt{return(}e'\texttt{)}} \tag{40.2a}$$

$$\frac{e \mapsto e'}{\texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{raise[}\tau\texttt{](}e'\texttt{)}} \tag{40.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e;x.m_1;y.m_2) \mapsto \texttt{letcomp}(e';x.m_1;y.m_2)} \tag{40.2c}$$

$$\frac{m \mapsto m'}{\texttt{letcomp(comp(}m\texttt{)};x.m_1;y.m_2) \mapsto \texttt{letcomp(comp(}m'\texttt{)};x.m_1;y.m_2)} \tag{40.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp(comp(return(}e\texttt{))};x.m_1;y.m_2) \mapsto [e/x]m_1} \tag{40.2e}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp(comp(raise[}\tau\texttt{](}e\texttt{))};x.m_1;y.m_2) \mapsto [e/y]m_2} \tag{40.2f}$$

40.2 Programming With Monadic Exceptions

The chief virtue of monadic exceptions is also its chief vice. A value of type $\texttt{nat} \rightarrow \texttt{nat}$ remains a function that, when applied to a natural number, returns a natural number (or, in the case of partial functions, may diverge). If a function can raise an exception when called, then it must be given the weaker type $\texttt{nat} \rightarrow \texttt{nat comp}$, which specifies that, when applied, it yields an encapsulated computation that, when executed, may raise an exception. Two such functions cannot be directly compose, since their types are no longer compatible. Instead we must explicitly sequence their execution. For example, to compose f and g of this type, we may write

$$\lambda\texttt{(}x\texttt{:nat.do}\,\{y \leftarrow \texttt{run}\,g\texttt{(}x\texttt{)}\ ;\ z \leftarrow \texttt{run}\,f\texttt{(}y\texttt{)};\texttt{return}\,z\}\texttt{)}.$$

Here we have used the do syntax introduced in Chapter 39, which according to our conventions above, implicitly propagates exceptions arising from the application of f and g to their surrounding context.

This distinction may be regarded as either a virtue or a vice, depending on how important it is to indicate in the type whether a function might raise an exception when called. For programmer-defined exceptions one may wish to draw the distinction, but the situation is less clear for other forms of run-time errors. For example, if division by zero is to be regarded as a form of exception, then the type of division must be

$$\texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{nat comp}$$

to reflect this possibility. But then one cannot then use division in an ordinary arithmetic expression, because its result is not a number, but an encapsulated command. One response to this might be to consider division by zero, and other related faults, not as handle-able exceptions, but rather as fatal errors that abort computation. In that case there is no difference between such an error and divergence: the computation never terminates, and this condition cannot be detected during execution. Consequently, operations such as division may be regarded as partial functions, and may therefore be used freely in expressions without taking special pains to manage any errors that may arise.

40.3 Exercises

Chapter 41

Monadic State

In Chapter 37 we introduced the type of cells of a given type so as to distinguish mutable from immutable data structures. In that chapter we left open the question of how to integrate mutation into a full-scale language. There are two main methods of doing so, one that permits great flexibility in the use of mutable storage at the expense of weakening the meaning of the typing judgement considerably, and one that retains the meaning of the typing judgement, but impairs the use of storage effects considerably. As this description suggest, each approach has its benefits and drawbacks, with neither being clearly preferable to the other in all circumstances.

The simplest, and most obvious, approach, which we will call the *integral* style, is to enrich a purely functional language, such as $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ or its extensions, with the mechanisms of $\mathcal{L}\{\texttt{ref}\}$. This results in an integration of imperative and functional programming in which the programmer may, at will, use or eschew mutation at any point within a program. For example, if we start with a purely functional data structure such as a tree structure represented as a recursive type, and then we wish to instrument this structure with, say, a reference count for profiling purposes, we may simply revise the definition of the type to, say, attach a mutable cell to each node that maintains the profiling information. It is usually straightforward to extend the implementation of the tree operations to account for the additional information at the nodes.

The chief drawback of the integral approach is that the meaning of the typing judgement changes drastically. The judgement $e : \tau$ no longer means "if e evaluates to a value v, then v is a value of type τ." Instead, the judgement now means that, in addition, that evaluation of e can engender arbitrary *side effects* on any data structure to which e has (direct or indirect) access. (Indeed, side effects are so-called precisely because they act "on the side," without their influence being reflected in the type of the expression.) Consequently, the type `nat` → `nat` can no longer be understood as the type of partial functions on the natural numbers, but must also admit the possibility of side effects during its execution. As a case in point, in a language without mutation the type `unit` → `unit` is quite trivial, containing only the identity and the divergent function, whereas in a language with integral mutation, this type contains arbitrarily complex functions that mutate storage, with the type revealing nothing about this behavior.

The integral approach works best with a strict language, in which the order of evaluation of sub-expressions is fully determined by its form, and is not sensitive to the evolution of the computation. Any form of laziness complicates the integral approach because it makes it much harder to predict when expressions are evaluated. In the absence of storage effects this is of no concern (at least for correctness, if not efficiency), but in the presence of storage effects, the indeterminacy of evaluation order is disastrous. It is difficult to tell exactly when, or how often, a cell will be allocated or assigned, greatly complicating reasoning about program correctness.

The *monadic* approach to storage effects is to confine operations that may affect storage to the command level of $\mathcal{L}\{\texttt{comm}\}$. This ensures that the expression level remains pure, so that it is compatible with both an eager and a lazy interpretation. The chief benefit of the monadic style is that it makes explicit in the types any reliance on storage effects. The chief drawback of the monadic style is tat it makes explicit in the types any reliance on storage effects. While it can be useful to document the use of storage effects, it can also be a hindrance to program development. For example, if we wish to instrument a piece of pure code with code for profiling, then we must restructure it to permit modifications to the store for profiling purposes, even though its functionality has not changed.

41.1 Storage Effects

The language $\mathcal{L}\{\texttt{comm}\,\texttt{ref}\}$ is an extension of $\mathcal{L}\{\texttt{comm}\}$ (described in Chapter 39) with mutable references into $\mathcal{L}\{\texttt{comm}\}$. The syntax of $\mathcal{L}\{\texttt{comm}\,\texttt{ref}\}$

extends that of $\mathcal{L}\{\mathtt{comm}\}$ with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{ref}(\tau)$	$\tau\,\mathtt{ref}$
Expr	e	$::=$	l	l
Comm	m	$::=$	$\mathtt{new}(e)$	$\mathtt{new}(e)$
		$\mid$	$\mathtt{get}(e)$	$\mathtt{!}\,e$
		$\mid$	$\mathtt{set}(e_1;e_2)$	$e_1 \mathtt{:=} e_2$

Locations are pure expressions, whereas the primitives for reference cells are forms of command.

The static semantics of $\mathcal{L}\{\mathtt{comm\,ref}\}$ extends that of $\mathcal{L}\{\mathtt{comm}\}$ with the following rules:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash l : \mathtt{ref}(\tau)} \tag{41.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \mathtt{new}(e) \sim \mathtt{ref}(\tau)} \tag{41.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{ref}(\tau)}{\Lambda\,\Gamma \vdash \mathtt{get}(e) \sim \tau} \tag{41.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \mathtt{ref}(\tau) \quad \Lambda\,\Gamma \vdash e_2 : \tau}{\Lambda\,\Gamma \vdash \mathtt{set}(e_1;e_2) \sim \tau} \tag{41.1d}$$

Here we make explicit the location typing assumptions, Λ, as well as the variable typing assumptions, Γ.

The dynamic semantics of $\mathcal{L}\{\mathtt{comm\,ref}\}$ is structured into two parts:

1. A transition relation $e \mapsto e'$ for expressions.

2. A transition relation $m \mathbin{@} \mu \mapsto m' \mathbin{@} \mu'$ for commands.

Expressions are evaluated without regard to context, since they engender no effects, whereas commands are evaluated relative to a memory, on which they may have an effect.

The rules defining the dynamic semantics of the monad constructs are as follows.

$$\frac{}{\mathtt{comp}(m)\ \mathsf{val}} \tag{41.2a}$$

$$\frac{e \mapsto e'}{\mathtt{return}(e) \mathbin{@} \mu \mapsto \mathtt{return}(e') \mathbin{@} \mu} \tag{41.2b}$$

$$\frac{e \mapsto e'}{\mathtt{letcomp}(e;x.m) \mathbin{@} \mu \mapsto \mathtt{letcomp}(e';x.m) \mathbin{@} \mu} \tag{41.2c}$$

$$\frac{m_1 \mathbin{@} \mu \mapsto m_1' \mathbin{@} \mu'}{\mathtt{letcomp(comp(}m_1\mathtt{);}x\mathtt{.}m_2\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{letcomp(comp(}m_1'\mathtt{);}x\mathtt{.}m_2\mathtt{)} \mathbin{@} \mu'} \quad (41.2d)$$

$$\frac{e \;\mathsf{val}}{\mathtt{letcomp(comp(return(}e\mathtt{));}x\mathtt{.}m\mathtt{)} \mathbin{@} \mu \mapsto [e/x]m \mathbin{@} \mu} \quad (41.2e)$$

The transition rules for the monadic elimination form is somewhat unusual. First, the expression e is evaluated to obtain a suspended command. Once such a command 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 suspended command is a `return` statement, in which case this value is passed to the body of the `letcomp`.

The transition rules for evaluation of storage commands are as follows:

$$\frac{}{l \;\mathsf{val}} \quad (41.3a)$$

$$\frac{e \mapsto e'}{\mathtt{new(}e\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{new(}e'\mathtt{)} \mathbin{@} \mu} \quad (41.3b)$$

$$\frac{e \;\mathsf{val}}{\mathtt{new(}e\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{return(}l\mathtt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \quad (41.3c)$$

$$\frac{e \mapsto e'}{\mathtt{get(}e\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{get(}e'\mathtt{)} \mathbin{@} \mu} \quad (41.3d)$$

$$\frac{e \;\mathsf{val}}{\mathtt{get(}l\mathtt{)} \mathbin{@} \mu \otimes \langle l : e \rangle \mapsto \mathtt{return(}e\mathtt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \quad (41.3e)$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{set(}e_1\mathtt{;}e_2\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{set(}e_1'\mathtt{;}e_2\mathtt{)} \mathbin{@} \mu} \quad (41.3f)$$

$$\frac{e_1 \;\mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{set(}e_1\mathtt{;}e_2\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{set(}e_1\mathtt{;}e_2'\mathtt{)} \mathbin{@} \mu} \quad (41.3g)$$

$$\frac{e \;\mathsf{val}}{\mathtt{set(}l\mathtt{;}e\mathtt{)} \mathbin{@} \mu \otimes \langle l : e' \rangle \mapsto \mathtt{return(}e\mathtt{)} \mathbin{@} \mu \otimes \langle l : e \rangle} \quad (41.3h)$$

Type safety for $\mathcal{L}\{\mathtt{comm\,ref}\}$ is stated and proved much as it is for $\mathcal{L}\{\mathtt{ref}\}$.

41.2 Integral versus Monadic Effects

The chief motivation for introducing monads is to make explicit in the types any reliance on computational effects. In the case of storage effects this is not always an advantage. The problem is that any use of storage forces the computation to be within the monad, and there is no way to get back out—once in the monad, always in the monad. This rules out the use of so-called *benign effects*, which may be used internally in some computation that is, for all outward purposes, entirely pure. One example of this is provided by splay trees, which may be used to implement a functional dictionary abstraction, but which rely heavily on mutation for their implementation in order to ensure efficiency. A simpler example, which we consider in detail, is provided by the use of backpatching to implement recursion as described in Chapter 37.

When formulated using monads to expose the use of storage effects, the backpatching implementation, *fact*, of the factorial function is as follows:

```
do {
  r ← new (λ n:nat. comp(return (n)))
; f ← return (λ n:nat. ...)
; _ ← set (r, f)
; return f
}
```

where the elided λ-abstraction is given as follows:

```
λ(n:nat.
  ifz(n,
      comp(return(1)),
      n'.comp(
            do {
              f'← get(r)
            ; return (n*f'(n'))
            })))
```

Observe that each branch of the conditional test returns a suspended command. In the case that the argument is zero, the command simply returns the value 1. Otherwise, it fetches the contents of the associated reference cell, applies this to the predecessor, and returns the result of the appropriate calculation.

We may check that that *fact* $\sim$ `nat` $\rightarrow$ `(nat comp)`, which exposes two aspects of this code:

1. The command that builds the recursive factorial function is impure, because it allocates and assigns to the reference cell used to implement backpatching.

2. The body of the factorial function is itself impure, because it accesses the reference cell to effect the recursive call.

The consequence is that the factorial function may no longer be used as a (pure) function! In particular, we cannot apply *fact* to an argument in an expression; it must be executed as a command. We must write

```
do {
  f ← fact
; x ← let comp (x:nat) be f(n) in return x
; return x
}
```

to bind the function computed by the expression *fact* to the variable `f`; apply this to n, yielding the result; and return this to the caller.

The difficulty is that the use of a reference cell to implement recursion is a benign effect, one that does not affect the purity of the function expression itself, nor of its applications. But the type system for effects studied here is incapable of recognizing this fact, and for good reason. It is extremely difficult, in general, to determine whether or not the use of effects in some region of a program is benign. As a stop-gap measure, one way around this is to introduce an operation of type $\tau\ \texttt{comp} \rightarrow \tau$, which may be used to exit the monad. But this ruins the very distinction we are trying to enforce, to segregate pure expressions from impure commands!

41.3 Exercises

Chapter 42

Comonads

Monads arise naturally for managing effects that both *influence* and are *influenced by* the context in which they arise. This is particularly clear for storage effects, whose context is a memory mapping locations to values. The semantics of the storage primitives makes reference to the memory (to retrieve the contents of a location) and makes changes to the memory (to change the contents of a location or allocate a new location). These operations must be sequentialized in order to be meaningul (*i.e.*, the precise order of execution matters), and we cannot expect to escape the context since locations are values that give rise to dependencies on the context. As we shall see in Chapter 48 other forms of effect, such as input/output or interprocess communication, are naturally expressed in the context of a monad.

By contrast the use of monads for exceptions as in Chapter 40 is rather less natural. Raising an exception does not influence the context, but rather imposes the requirement on it that a handler be present to ensure that the command is meaningful even when an exception is raised. One might argue that installing a handler influences the context, but it does so in a nested, or stack-like, manner. A new handler is installed for the duration of execution of a command, and restored afterwards. The handler does not persist across commands in the same sense that locations persist across commands in the case of the state monad. Moreover, installing a handler may be seen as restoring purity in that it catches any exceptions that may be raised and, assuming that the handler does not itself raise an exception, yields a pure value. A similar situation arises with fluid binding (as described in Chapter 36). A reference to a symbol imposes the demand on the context to provide a binding for it. The binding of a symbol may be changed, but only for the duration of execution of a command, and not persistently. Moreover, the reliance on symbol bindings within a specified scope confines the impurity to that scope.

The concept of a *comonad* captures the concept of an effect that *imposes a requirement* on its context of execution, but that does not persistently alter that context beyond its execution. Computations that rely on the context to provide some capability may be thought of as impure, but the impurity is confined to the extent of the reliance—outside of this context the computation may be once again regarded as pure. One may say that monads are appropriate for *global*, or *persistent*, effects, whereas comonads are appropriate for *local*, or *ephemeral*, effects.

42.1 A Comonadic Framework

The central concept of the comonadic framework for effects is the *constrained typing judgement*, $e : \tau\,[\chi]$, which states that an expression e has type τ (as usual) provided that the context of its evaluation satisfies the constraint χ. The nature of constraints varies from one situation to another, but will include at least the trivially true constraint, $\top$, and the conjunction of constraints, $\chi_1 \wedge \chi_2$. We sometimes write $e : \tau$ to mean $e : \tau\top$, which states that expression e has type τ under no constraints.

The syntax of the comonadic framework, $\mathcal{L}\{\texttt{comon}\}$, is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{box[}\chi\texttt{](}\tau\texttt{)}$	$\Box_\chi\,\tau$
Const	χ	$::=$	$\texttt{tt}$	$\top$
		$\mid$	$\texttt{and(}\chi_1\texttt{;}\chi_2\texttt{)}$	$\chi_1 \wedge \chi_2$
Expr	e	$::=$	$\texttt{box(}e\texttt{)}$	$\texttt{box(}e\texttt{)}$
		$\mid$	$\texttt{unbox(}e\texttt{)}$	$\texttt{unbox(}e\texttt{)}$

A type of the form $\Box_\chi\,\tau$ is called a *comonad*; it represents the type of unevaluated expressions that impose constraint χ on their context of execution. The constraint $\top$ is the trivially true constraint, and the constraint $\chi_1 \wedge \chi_2$ is the conjunction of two constraints. The expression $\texttt{box(}e\texttt{)}$ is the introduction form for the comonad, and the expression $\texttt{unbox(}e\texttt{)}$ is the corresponding elimination form.

The judgement χ $\texttt{true}$ expresses that the constraint χ is satisfied. This judgement is partially defined by the following rules, which specify the meanings of the trivially true constraint and the conjunction of constraints.

$$\frac{}{\texttt{tt}\ \mathsf{true}} \tag{42.1a}$$

$$\frac{\chi_1\ \mathsf{true} \quad \chi_2\ \mathsf{true}}{\texttt{and}(\chi_1;\chi_2)\ \mathsf{true}} \tag{42.1b}$$

$$\frac{\texttt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_1\ \mathsf{true}} \tag{42.1c}$$

$$\frac{\texttt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_2\ \mathsf{true}} \tag{42.1d}$$

We will make use of hypothetical judgements of the form $\chi_1\ \mathsf{true}, \ldots, \chi_n\ \mathsf{true} \vdash \chi\ \mathsf{true}$, where $n \geq 0$, expressing that χ is derivable from $\chi_1, \ldots, \chi_n$, as usual.

The static semantics is specified by parametric hypothetical judgements of the form

$$x_1 : \tau_1\,[\chi_1], \ldots, x_n : \tau_n\,[\chi_n] \vdash e : \tau\,[\chi].$$

As usual we write Γ for a finite set of hypotheses of the above form.

The static semantics of the core constructs of $\mathcal{L}\{\texttt{comon}\}$ is defined by the following rules:

$$\frac{\chi' \vdash \chi}{\Gamma, x : \tau\,[\chi] \vdash x : \tau\,[\chi']} \tag{42.2a}$$

$$\frac{\Gamma \vdash e : \tau\,[\chi]}{\Gamma \vdash \texttt{box}(e) : \Box_\chi\,\tau\,[\chi']} \tag{42.2b}$$

$$\frac{\Gamma \vdash e : \Box_\chi\,\tau\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \texttt{unbox}(e) : \tau\,[\chi']} \tag{42.2c}$$

Rule (42.2b) states that a boxed computation has comonadic type under an arbitrary constraint. This is valid because a boxed computation is a value, and hence imposes no constraint on its context of evaluation. Rule (42.2c) states that a boxed computation may be activated provided that the ambient constraint, χ', is at least as strong as the constraint χ of the boxed computation. That is, any requirement imposed by the boxed computation must be met at the point at which it is unboxed.

Rules (42.2) are formulated to ensure that the constraint on a typing judgement may be strengthened arbitrarily.

Lemma 42.1 (Constraint Strengthening). *If* $\Gamma \vdash e : \tau\,[\chi]$ *and* $\chi' \vdash \chi$*, then* $\Gamma \vdash e : \tau\,[\chi']$.

Proof. By rule induction on Rules (42.2). □

Intuitively, if a typing holds under a weaker constraint, then it also holds under any stronger constraint as well.

At this level of abstraction the dynamic semantics of $\mathcal{L}\{\texttt{comon}\}$ is trivial.

$$\frac{}{\texttt{box(}e\texttt{)}\ \mathsf{val}} \tag{42.3a}$$

$$\frac{e \mapsto e'}{\texttt{unbox(}e\texttt{)} \mapsto \texttt{unbox(}e'\texttt{)}} \tag{42.3b}$$

$$\frac{}{\texttt{unbox(box(}e\texttt{))} \mapsto e} \tag{42.3c}$$

In specific applications of $\mathcal{L}\{\texttt{comon}\}$ the dynamic semantics will also specify the context of evaluation with respect to which constraints are to interpreted.

The role of the comonadic type in $\mathcal{L}\{\texttt{comon}\}$ is explained by considering how one might extend the language with, say, function types. The crucial idea is that the comonad isolates the dependence of a computation on its context of evaluation so that such constraints do not affect the other type constructors. For example, here are the rules for function types expressed in the context of $\mathcal{L}\{\texttt{comon}\}$:

$$\frac{\Gamma, x : \tau_1\,[\texttt{tt}] \vdash e_2 : \tau_2\,[\texttt{tt}]}{\Gamma \vdash \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)} : \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}\,[\chi]} \tag{42.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau\,[\chi] \quad \Gamma \vdash e_2 : \tau_2\,[\chi]}{\Gamma \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} : \tau\,[\chi]} \tag{42.4b}$$

These rules are formulated so as to ensure that constraint strengthening remains admissible. Rule (42.4a) states that a λ-abstraction has type $\tau_1 \rightarrow \tau_2$ under any constraint χ provided that its body has type τ_2 under the trivially true constraint, assuming that its argument has type τ_1 under the trivially true constraint. By demanding that the body be well-formed under no constraints we are, in effect, insisting that its body be boxed if it is to impose a constraint on the context at the point of application. Under a call-by-value evaluation order, the argument x will always be a value, and hence imposes no constraints on its context.

Let the expression $\texttt{unbox\_app(}e_1\texttt{;}e_2\texttt{)}$ be an abbreviation for $\texttt{unbox(ap(}e_1\texttt{;}e_2\texttt{))}$, which applies e_1 to e_2, then activates the result. The derived static semantics for this construct is given by the following rule:

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \Box_\chi\,\tau\,[\chi'] \quad \Gamma \vdash e_2 : \tau_2\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \texttt{unbox\_app(}e_1\texttt{;}e_2\texttt{)} : \tau\,[\chi']} \tag{42.5}$$

In words, to apply a function with impure body to an argument, the ambient constraint must be strong enough to type the function and its argument, and must be at least as strong as the requirements imposed by the body of the function. We may view a type of the form $\tau_1 \rightarrow \Box_\chi \tau_2$ as the type of functions that, when applied to a value of type τ_1, yield a value of type τ_2 engendering local effects with requirements specified by χ.

Similar principles govern the extension of $\mathcal{L}\{\texttt{comon}\}$ with other types such as products or sums.

42.2 Comonadic Effects

In this section we discuss two applications of $\mathcal{L}\{\texttt{comon}\}$ to managing local effects. The first application is to exceptions, using constraints to specify whether or not an exception handler must be installed to evaluate an expression so as to avoid an uncaught exception error. The second is to fluid binding, using constraints to specify which symbols must be bound during execution so as to avoid accessing an unbound symbol. The first may be considered to be an instance of the second, in which we think of the exception handler as a distinguished symbol whose binding is the current exception continuation.

42.2.1 Exceptions

To model exceptions we extend $\mathcal{L}\{\texttt{comon}\}$ as follows:

Category	*Item*		*Abstract*	*Concrete*
Const	χ	::=	$\uparrow$	$\uparrow$
Expr	e	::=	$\texttt{raise[}\tau\texttt{](}e\texttt{)}$	$\texttt{raise(}e\texttt{)}$
		|	$\texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)}$	$\texttt{try}\,e_1\,\texttt{ow}\,x \Rightarrow e_2$

The constraint $\uparrow$ specifies that an expression may raise an exception, and hence that its context is required to provide a handler for it.

The static semantics of $\mathcal{L}\{\texttt{comon}\}$ is extended with the following rules:

$$\frac{\Gamma \vdash e : \tau_{exn}\,[\chi] \quad \chi \vdash \uparrow}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} : \tau\,[\chi]} \tag{42.6a}$$

$$\frac{\Gamma \vdash e_1 : \tau\,[\chi \wedge \uparrow] \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau\,[\chi]}{\Gamma \vdash \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \tau\,[\chi]} \tag{42.6b}$$

Rule (42.6a) imposes the requirement for a handler on the context of a `raise` expression, in addition to any other conditions that may be imposed by its argument. (The rule is formulated so as to ensure that constraint strengthening remains admissible.) Rule (42.6b) transforms an expression that requires a handler into one that may or may not require one, according to the demands of the handling expression. If e_2 does not demand a handler, then χ may be taken to be the trivial constraint, in which case the overall expression is pure, even though e_1 is impure (may raise an exception).

The dynamic semantics of exceptions is as given in Chapter 30. The interesting question is to explore the additional assurances given by the comonadic type system given by Rules (42.6). Intuitively, we may think of a stack as a constraint transformer that turns a constraint χ into a constraint χ' by composing frames, including handler frames. Then if e is an expression of type τ imposing constraint χ and k is a τ-accepting stack transforming constraint χ into constraint $\top$, then evaluation of e on k cannot yield an uncaught exception. In this sense the constraints reflect the reality of the execution behavior of expressions.

To make this precise, we define the judgement $k : \tau\,[\chi]$ to mean that k is stack that is suitable as an execution context for an expression $e : \tau\,[\chi]$. The typing rules for stacks are as follows:

$$\frac{}{\epsilon : \tau\,[\top]} \tag{42.7a}$$

$$\frac{k : \tau'\,[\chi'] \quad f : \tau\,[\chi] \Rightarrow \tau'\,[\chi']}{f\,;k : \tau\,[\chi]} \tag{42.7b}$$

Rule (42.7a) states that the empty stack must not impose any constraints on its context, which is to say that there must be no uncaught exceptions at the end of execution. Rule (42.7b) simply specifies that a stack is a composition of frames. The typing rules for frames are easily derived from the static semantics of $\mathcal{L}\{\texttt{comon}\}$. For example,

$$\frac{x : \tau_{exn} \vdash e : \tau\,[\chi]}{\texttt{handle}(-;x\,.\,e) : \tau\,[\chi \wedge \uparrow] \Rightarrow \tau\,[\chi]} \tag{42.8}$$

This rule states that a handler frame transforms an expression of type τ demanding a handler into an expression of type τ that may, or may not, demand a handler, according to the form of the handling expression.

The formation of states is defined essentially as in Chapter 29.

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi]}{k \rhd e\ \mathsf{ok}} \tag{42.9a}$$

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi] \quad e\ \mathsf{val}}{k \lhd e\ \mathsf{ok}} \tag{42.9b}$$

Observe that a state of the form $\epsilon \triangleright \mathtt{raise}(e)$, where e val, is ill-formed, because the empty stack is well-formed only under no constraints on the context.

Safety ensures that no uncaught exceptions can arise. This is expressed by defining final states to be only those returning a value to the empty stack.

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{42.10}$$

In contrast to Chapter 30, we *do not* consider an uncaught exception state to be final!

Theorem 42.2 (Safety).

1. *If s ok and $s \mapsto s'$, then s' ok.*
2. *If s ok then either s final or there exists s' such that $s \mapsto s'$.*

Proof. These are proved by rule induction on the dynamic semantics and on the static semantics, respectively, proceeding along standard lines. □

42.2.2 Fluid Binding

Using comonads we may devise a type system for fluid binding that ensures that no unbound symbols are accessed during execution. This is achieved by regarding the mapping of symbols to their values to be the context of execution, and introducing a form of constraint stating that a specified symbol must be bound in the context.

Let us consider a comonadic static semantics for $\mathcal{L}\{\mathtt{fluid}\}$ defined in Chapter 36. For this purpose we consider atomic constraints of the form $\mathtt{bd}(a)$, stating that the symbol a has a binding.

The static semantics of fluid binding consists of judgements of the form $\Sigma\,\Gamma \vdash e : \tau\,[\chi]$, where Σ consists of hypotheses of the form $a : \tau$ assigning a type to a symbol.

$$\frac{\Sigma \vdash a : \tau \quad \chi \vdash \mathtt{bd}(a)}{\Sigma\,\Gamma \vdash \mathtt{get[}a\mathtt{]} : \tau\,[\chi]} \tag{42.11a}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e_1 : \tau\,[\chi] \quad \Sigma\,\Gamma \vdash e_2 : \tau\,[\chi \wedge \mathtt{bd}(a)]}{\Sigma\,\Gamma \vdash \mathtt{set[}a\mathtt{](}e_1\mathtt{;}e_2\mathtt{)} : \tau\,[\chi]} \tag{42.11b}$$

Rule (42.11a) records the demand for a binding for the symbol a incurred by retrieving its value. Rule (42.11b) propagates the fact that the symbol a is bound to the body of the fluid binding.

The dynamic semantics is as specified in Chapter 36. The safety theorem for the comonadic type system for fluid binding states that no unbound symbol error may ever arise during execution. We define the judgement $\theta \models \chi$ to mean that $a \in dom(\theta)$ whenever $\chi \vdash \mathtt{bd}(a)$.

Theorem 42.3 (Safety). *1. If $e : \tau\,[\chi]$ and $e \mapsto_\theta e'$, then $e' : \theta\,[\chi]$.*

2. If $e : \tau\,[\chi]$ and $\theta \models \chi$, then either e val or there exists e' such that $e \mapsto_\theta e'$.

The comonadic static semantics for $\mathcal{L}\{\mathtt{fluid}\}$ may be extended to $\mathcal{L}\{\mathtt{fluid\,gen}\}$, which also permits dynamic symbol generation. The main difficulty is to manage the interaction between the scopes of symbols and their occurrences in types. First, it is straightforward to define the judgement $\Sigma \vdash \chi$ constr to mean that χ is a constraint involving only those symbols a such that $\Sigma \vdash a : \tau$ for some τ. Using this we may also define the judgement $\Sigma \vdash \tau$ type analogously. This judgement is used to impose a restriction on symbol generation to ensure that symbols do not escape their scope:

$$\frac{\Sigma, a : \sigma\,\Gamma \vdash e : \tau \quad \Sigma \vdash \tau\ \mathsf{type}}{\Sigma\,\Gamma \vdash \mathtt{new}[\sigma]\,(a.e) : \langle\sigma\rangle\tau} \tag{42.12}$$

This imposes the requirement that the result type of a computation involving a dynamically generated symbol must not mention that symbol. Otherwise the type $\langle\sigma\rangle\tau$ would involve a symbol that makes no sense with respect to the ambient symbol context, Σ. In practical terms this means that the expression, e, must ensure that its type imposes no residual requirements involving the symbol a introduced by the binder.

For example, an expression such as

$$\mathtt{gen}(\nu(a\mathtt{:nat.set}\,a\,\mathtt{to\,z\,in}\,\lambda(x\mathtt{:nat.box}(\ldots\mathtt{get}\,a\ldots))))$$

is necessarily ill-typed. The type of the λ-abstraction must be of the form $\mathtt{nat} \to \Box_\chi\,\tau$, where $\chi \vdash \mathtt{bd}(a)$, reflecting the dependence of the body of the function on the binding of a. This type is propagated through the fluid binding for a, since it holds only for the duration of evaluation of the λ-abstraction itself, which is immediately returned as its value. Since the type of the λ-abstraction involves the symbol a, the second premise of Rule (42.12) is not met, and the expression is ill-typed. This is as it should be, for we cannot guarantee that the dynamically generated symbol replacing a during evaluation will, in fact, be bound when the body of the function is executed.

However, if we move the binding for a into the scope of the λ-abstraction,

$$\mathtt{gen}(\nu(a\mathtt{:nat}.\lambda(x\mathtt{:nat.box}(\mathtt{set}\,a\,\mathtt{to\,z\,in}\ \ldots\mathtt{get}\,a\ldots)))),$$

then the type of the λ-abstraction may have the form `nat` $\rightarrow \Box_{\chi}\,\tau$, where χ need not constrain a to be bound. The reason is that the fluid binding for a discharges the obligation to bind a within the body of the function. Consequently, the condition on Rule (42.12) is met, and the expression is well-typed. Indeed, each evaluation of the body of the λ-abstraction initializes the fresh copy of a generated during evaluation, so no unbound symbol error can arise during execution.

42.3 Exercises

Part XIV

Laziness

Chapter 43

Eagerness and Laziness

A fundamental distinction between *eager*, or *strict*, and *lazy*, or *non-strict*, evaluation arises in the dynamic semantics of function, product, sum, and recursive types. This distinction is of particular importance in the context of $\mathcal{L}\{\mu\rightharpoonup\}$, which permits the formation of divergent expressions. Quite often eager and lazy evaluation is taken to be a *language design distinction*, but we argue that it is better viewed as a *type distinction*.

43.1 Eager and Lazy Dynamics

According to the methodology outlined in Chapter 11, language features are identified with types. The constructs of the language arise as the introductory and eliminatory forms associated with a type. The static semantics specifies how these may be combined with each other and with other language constructs in a well-formed program. The dynamic semantics specifies how these constructs are to be executed, subject to the requirement of type safety. Safety is assured by the conservation principle, which states that the introduction forms are the values of the type, and the elimination forms are inverse to the introduction forms.

Within these broad guidelines there is often considerable leeway in the choice of dynamic semantics for a language construct. For example, consider the dynamic semantics of function types given in Chapter 14. There we specified the λ-abstractions are values, and that applications are evaluated according to the following rules:

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \tag{43.1a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{e_1\,(e_2) \mapsto e_1\,(e_2')} \tag{43.1b}$$

$$\frac{e_2\ \mathsf{val}}{\lambda\,(x\!:\!\tau.\,e)\,(e_2) \mapsto [e_2/x]e} \tag{43.1c}$$

The first of these states that to evaluate an application $e_1\,(e_2)$ we must first of all evaluate e_1 to determine what function is being applied. The third of these states that application is inverse to abstraction, but is subject to the requirement that the argument be a value. For this to be tenable, we must also include the second rule, which states that to apply a function, we must first evaluate its argument. This is called the *call-by-value*, or *strict*, or *eager*, evaluation order for functions.

Regarding a λ-abstraction as a value is inevitable so long as we retain the principle that only closed expressions (complete programs) can be executed. Similarly, it is natural to demand that the function part of an application be evaluated before the function can be called. On the other hand it is somewhat arbitrary to insist that the argument be evaluated before the call, since nothing seems to oblige us to do so. This suggests an alternative evaluation order, called *call-by-name*,[1] or *lazy*, which states that arguments are to be passed unevaluated to functions. Consequently, function parameters stand for computations, not values, since the argument is passed in unevaluated form. The following rules define the call-by-name evaluation order:

$$\frac{e_1 \mapsto e_1'}{e_1\,(e_2) \mapsto e_1'\,(e_2)} \tag{43.2a}$$

$$\frac{}{\lambda\,(x\!:\!\tau.\,e)\,(e_2) \mapsto [e_2/x]e} \tag{43.2b}$$

We omit the requirement that the argument to an application be a value.

This example illustrates some general principles governing the dynamic semantics of a language:

1. The conservation principle demands that the elimination forms be inverse to the introduction forms. The elimination forms associated with a type have a distinguished *principal argument*, which is of the type under consideration, to which the elimination form is inverse.

2. The principal argument of an elimination form is necessarily evaluated to an introduction form, thereby exposing an opportunity for cancellation according to the conservation principle.

[1] For obscure historical reasons.

3. It is more or less arbitrary whether the non-principal arguments to an elimination form are evaluated prior to cancellation.

4. Values of the type have introductory form, but may also be chosen to satisfy further requirements such as insisting that certain sub-expressions also be values.

Let us apply these principles to the product type. First, the sole argument to the elimination forms is, of course, principal, and hence must be evaluated. Second, if the argument is a value, it must be a pair (the only introductory form), and the projections extract the appropriate component of the pair.

$$\frac{\langle e_1, e_2 \rangle\ \mathsf{val}}{\mathtt{fst}(\langle e_1, e_2 \rangle) \mapsto e_1} \tag{43.3}$$

$$\frac{\langle e_1, e_2 \rangle\ \mathsf{val}}{\mathtt{snd}(\langle e_1, e_2 \rangle) \mapsto e_1} \tag{43.4}$$

$$\frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')} \tag{43.5}$$

$$\frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')} \tag{43.6}$$

Since there is only one introductory form for the product type, a value of product type must be a pair. But this leaves open whether the components of a pair value must themselves be values or not. The *eager* (or *strict*) semantics, which we gave in Chapter 17, evaluates the components of a pair before deeming it to be a value: specified by the following additional rules:

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\langle e_1, e_2 \rangle\ \mathsf{val}} \tag{43.7}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \tag{43.8}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2 \rangle \mapsto \langle e_1, e_2' \rangle} \tag{43.9}$$

The *lazy* (or *non-strict*) semantics, on the other hand, deems any pair to be a value, regardless of whether its components are values:

$$\frac{}{\langle e_1, e_2 \rangle\ \mathsf{val}} \tag{43.10}$$

There are similar alternatives for sum and recursive types, differing according to whether or not the argument of an injection, or to the introductory half of an isomorphism, is evaluated. There is no choice, however, regarding evaluation of the branches of a case analysis, since each branch binds a variable to the injected value for each case. Incidentally, this explains the apparent restriction on the evaluation of the conditional expression, $\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2$, arising from the definition of `bool` to be the sum type `unit + unit` as described in Chapter 18 — the "then" and the "else" branches lie within the scope of an (implicit) bound variable, and hence are not eligible for evaluation!

43.2 Eager and Lazy Types

Rather than specify a blanket policy for the eagerness or laziness of the various language constructs, it is more expressive to put this decision into the hands of the programmer by a *type distinction*. That is, we can distinguish types of by-value and by-name functions, and of eager and lazy versions of products, sums, and recursive types.

We may give eager and lazy variants of product, sum, function, and recursive types according to the following chart:

	Eager	Lazy
Unit	$\mathbf{1}$	$\top$
Product	$\tau_1 \otimes \tau_2$	$\tau_1 \times \tau_2$
Void	$\bot$	$\mathbf{0}$
Sum	$\tau_1 + \tau_2$	$\tau_1 \oplus \tau_2$
Function	$\tau_1 \circ\!\!\rightarrow \tau_2$	$\tau_1 \rightarrow \tau_2$

We leave it to the reader to formulate the static and dynamic semantics of these constructs using the following grammar of introduction and elimination forms for the unfamiliar type constructors in the foregoing chart:

	Introduction	Elimination
$\mathbf{1}$	$\bullet$	*(none)*
$\tau_1 \otimes \tau_2$	$e_1 \otimes e_2$	$\texttt{let}\,x_1 \otimes x_2\,\texttt{be}\,e\,\texttt{in}\,e'$
$\mathbf{0}$	*(none)*	$\texttt{abort}_\tau\texttt{(}e\texttt{)}$
$\tau_1 \oplus \tau_2$	$\texttt{lft}_\tau\texttt{(}e\texttt{)}$, $\texttt{rht}_\tau\texttt{(}e\texttt{)}$	$\texttt{choose}\,e\,\{\texttt{lft(}x_1\texttt{)}\Rightarrow e_1 \mid \texttt{rht(}x_2\texttt{)}\Rightarrow e_2\}$
$\tau_1 \circ\!\!\rightarrow \tau_2$	$\lambda^\circ\texttt{(}x\texttt{:}\tau_1\texttt{.}e_2\texttt{)}$	$\texttt{ap}^\circ\texttt{(}e_1\texttt{;}e_2\texttt{)}$

The elimination form for the eager product type uses pattern-matching to recover both components of the pair at the same time. The elimination form for the lazy empty sum performs a case analysis among zero choices, and is therefore tantamount to aborting the computation. Finally, the circle adorning the eager function abstraction and application is intended to suggest a correspondence to the eager product and function types.

The notation for eager and lazy types is chosen to emphasize a duality between the eager and lazy interpretations of the type constructors. We use familiar notation to emphasize that the construct has a *standard*, or *strong*, semantics, and unfamiliar notation to emphasize that the construct has a *non-standard*, or *weak*, semantics. Thus the lazy interpretation features standard products and function types but non-standard sum types. Dually, the eager interpretation features standard sums, but non-standard products and functions. In a sense that we cannot make fully precise here, the standard types enjoy a full range of properties that are valid only in limited forms for the non-standard types. For example, lazy products are standard in that the expression $\mathtt{fst}(\langle e_1, e_2\rangle)$ is interchangeable with e_1, regardless of whether or not e_2 terminates, and similarly $\mathtt{snd}(\langle e_1, e_2\rangle)$ is interchangeable with e_2, independently of whether e_1 terminates. But these conditions fail for strict products, unless both e_1 and e_2 are values. A dual, but harder to state, situation obtains for sums, with eager sums being standard, and lazy sums being non-standard. Lazy function types are standard in that $\lambda(x{:}\tau_1.\,e_2)(e_1)$ is always interchangeable with $[e_1/x]e_2$, whereas the corresponding property fails for strict function types in the case that e_2 does not terminate.

43.3 Self-Reference

We have seen in Chapter 16 that we may use general recursion at the expression level to define recursive functions. In the presence of laziness we may also define other forms of self-referential expression. For example, consider the so-called lazy natural numbers, which are defined by the recursive type $\mathtt{lnat} = \mu t\,.\,\top \oplus t$. The successor operation for the lazy natural numbers is defined by the equation $\mathtt{lsucc}(e) = \mathtt{fold}(\mathtt{rht}(e))$. Using general recursion we may form the lazy natural number

$$\omega = \mathtt{fix}\,x{:}\mathtt{lnat}\,\mathtt{is}\,\mathtt{lsucc}(x),$$

which consists of an infinite stack of successors!

Of course, one could argue (correctly) that ω is not a natural number at all, and hence should not be regarded as one. So long as we can distinguish the type lnat from the type nat, there is no difficulty—ω is the infinite *lazy* natural number, but it is not an *eager* natural number. But if the distinction is not available, then serious difficulties arise. For example, lazy languages provide only lazy product and sum types, and hence are only capable of defining the lazy natural numbers as a recursive types. In such languages ω is said to be a "natural number", but only for a non-standard use of the term; the true natural numbers are simply unavailable.

It is a significant weakness of lazy languages is that they provide only a paucity of types. One might expect that, dually, eager languages are similarly disadvantaged in providing only eager, but not lazy types. However, in the presence of function types (the common case), we may encode the lazy types as instances of the corresponding eager types, as we describe in the next section.

43.4 Suspension Type

The essence of lazy evaluation is the suspension of evaluation of certain expressions. For example, the lazy product type suspends evaluation of the components of a pair until they are needed, and the lazy sum type suspends evaluation of the injected value until it is required. To encode lazy types as eager types, then, requires only that we have a type whose *values* are *unevaluated computations* of a specified type. Such unevaluated computations are called *suspensions*, or *thunks*.[2] Moreover, since general recursion requires laziness in order to be useful, it makes sense to confine general recursion to suspension types. To model this we consider *self-referential* unevaluated computations as values of suspension type.

The abstract syntax of suspensions is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$
Expr	e	::=	$\mathtt{susp}[\tau](x.e)$	$\mathtt{susp}\,x:\tau\,\mathtt{is}\,e$
		|	$\mathtt{force}(e)$	$\mathtt{force}(e)$

The introduction form binds a variable that stands for the suspension itself. The elimination form evaluates e_1 to a suspension, then evaluates that suspension, binding its value to x for use within e_2. As a notational convenience, we sometimes write $\mathtt{susp}(e)$ for $\mathtt{susp}[\tau](x.e)$, where $x \# e$ and e is of type τ.

[2] The etymology of this term is uncertain, but its usage persists.

The static semantics of suspensions is given by the following typing rules:

$$\frac{\Gamma, x : \texttt{susp}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{susp[}\tau\texttt{]}(x\texttt{.}e) : \texttt{susp}(\tau)} \tag{43.11a}$$

$$\frac{\Gamma \vdash e : \texttt{susp}(\tau)}{\Gamma \vdash \texttt{force}(e) : \tau} \tag{43.11b}$$

In Rule (43.11a) the variable x, which refers to the suspension itself, is assumed to have type $\texttt{susp}(\tau)$ while checking that the suspended computation, e, has type τ.

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

$$\frac{}{\texttt{susp[}\tau\texttt{]}(x\texttt{.}e)\ \mathsf{val}} \tag{43.12a}$$

$$\frac{e \mapsto e'}{\texttt{force}(e) \mapsto \texttt{force}(e')} \tag{43.12b}$$

$$\frac{}{\texttt{force}(\texttt{susp[}\tau\texttt{]}(x\texttt{.}e)) \mapsto [\texttt{susp[}\tau\texttt{]}(x\texttt{.}e)/x]e} \tag{43.12c}$$

Rule (43.12c) implements recursive self-reference by replacing x by the suspension itself before substituting it into the body of the `let`.

It is straightforward to formulate and prove type safety for self-referential suspensions. We leave the proof as an exercise for the reader.

Theorem 43.1 (Safety). *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \mapsto e'$.*

We may use suspensions to encode the lazy type constructors as instances of the corresponding eager type constructors as follows:

$$\top = \mathbf{1} \tag{43.13a}$$

$$\langle\rangle = \bullet \tag{43.13b}$$

$$\tau_1 \times \tau_2 = \tau_1\,\texttt{susp} \otimes \tau_2\,\texttt{susp} \tag{43.14a}$$

$$\langle e_1, e_2\rangle = \texttt{susp}(e_1) \otimes \texttt{susp}(e_2) \tag{43.14b}$$

$$\texttt{fst}(e) = \texttt{let}\,x \otimes \_\,\texttt{be}\,e\,\texttt{in}\,\texttt{force}(x) \tag{43.14c}$$

$$\texttt{snd}(e) = \texttt{let}\,\_ \otimes y\,\texttt{be}\,e\,\texttt{in}\,\texttt{force}(y) \tag{43.14d}$$

$$\mathbf{0} = \bot \tag{43.15a}$$

$$\texttt{abort}_\tau(e) = \texttt{abort}_\tau\, e \tag{43.15b}$$

$$\tau_1 \oplus \tau_2 = \tau_1\,\mathtt{susp} + \tau_2\,\mathtt{susp} \tag{43.16a}$$

$$\mathtt{lft}(e) = \mathtt{in[l]}(\mathtt{susp}(e)) \tag{43.16b}$$

$$\mathtt{rht}(e) = \mathtt{in[r]}(\mathtt{susp}(e)) \tag{43.16c}$$

$$\begin{aligned}&\mathtt{choose}\,e\,\{\mathtt{lft}(x_1){\Rightarrow}e_1 \mid \mathtt{rht}(x_2){\Rightarrow}e_2\}\\ &= \mathtt{case}\,e\,\{\mathtt{in[l]}(y_1) \Rightarrow [\mathtt{force}(y_1)/x_1]e_1 \mid \mathtt{in[r]}(y_2) \Rightarrow [\mathtt{force}(y_2)/x_2]e_2\}\end{aligned} \tag{43.16d}$$

$$\tau_1 \rightarrow \tau_2 = \tau_1\,\mathtt{susp} \mathrel{\circ\!\!\rightarrow} \tau_2 \tag{43.17a}$$

$$\lambda(x{:}\tau_1.\,e_2) = \lambda^{\circ}(x{:}\tau_1\,\mathtt{susp}.\,[\mathtt{force}(x)/x]e_2) \tag{43.17b}$$

$$e_1(e_2) = \mathtt{ap}^{\circ}(e_1;\mathtt{susp}(e_2)) \tag{43.17c}$$

In the case of lazy case analysis and call-by-name functions we replace occurrences of the bound variable, x, with $\mathtt{force}(x)$ to recover the value of the suspension bound to x whenever it is required. Note that x may occur in a lazy context, in which case $\mathtt{force}(x)$ is delayed. In particular, expressions of the form $\mathtt{susp}(\mathtt{force}(x))$ may be safely replaced by x, since forcing the former computation simply forces x.

43.5 Exercises

Chapter 44

Lazy Evaluation

Lazy evaluation refers to a variety of concepts that seek to avoid evaluation of an expression unless its value is needed, and to share the results of evaluation of an expression among all uses of its, so that no expression need be evaluated more than once. Within this broad mandate, various forms of laziness are considered.

One is the *call-by-need* evaluation strategy for functions. This is a refinement of the *call-by-name* semantics described in Chapter 43 in which arguments are passed unevaluated to functions so that it is only evaluated if needed, and, if so, the value is shared among all occurrences of the argument in the body of the function.

Another is the *lazy* evaluation strategy for data structures, including formation of pairs, injections into summands, and recursive folding. The decisions of whether to evaluate the components of a pair, or the argument to an injection or fold, are independent of one another, and of the decision whether to pass arguments to functions in unevaluated form.

A third aspect of laziness is the ability to form *recursive values*, including as a special case recursive functions. Using general recursion we can create self-referential expressions, but these are only useful if the self-referential expression can be evaluated without needing its own values. Function abstractions provide one such mechanism, but so do lazy data constructors.

These aspects of laziness are often consolidated into a programming language with call-by-need function evaluation, lazy data structures, and unrestricted uses of recursion. Such languages are called *lazy languages*, because they impose the lazy evaluation strategy throughout. These are to be contrasted with *strict languages*, which impose an eager evaluation strategy throughout. This leads to a sense of opposition between two incompatible points of view, but, as we discussed in Chapter 43, experience has shown that this apparent conflict is neither necessary nor desirable. Rather than accept these as consequences of language design, it is preferable to put the distinction in the hands of the programmer by introducing a type of suspended computations whose evaluation is memoized so that they are only ever evaluated once. The ambient evaluation strategy remains eager, but we now have a *value* representing an *unevaluated* expression. Moreover, we may confine self-reference to suspensions to avoid the pathologies of laziness while permitting self-referential data structures to be programmed.

44.1 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.

The call-by-need dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is given by a transition system whose states have the form $e \mathbin{\texttt{@}} \mu$, where μ is a finite function mapping variables to expressions (not necessarily values!), and e is an expression whose free variables lie within the domain of μ. (We use the same notation for finite functions as in Chapter 37.)

The rules defining the call-by-need dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ are as follows:

$$\frac{}{\texttt{z}\ \mathsf{val}} \tag{44.1a}$$

$$\frac{}{\texttt{s}(x)\ \mathsf{val}} \tag{44.1b}$$

$$\frac{}{\texttt{lam}[\tau](x.e)\ \mathsf{val}} \tag{44.1c}$$

$$\frac{}{x \mathbin{\texttt{@}} \langle x : e\rangle\ \mathsf{initial}} \tag{44.1d}$$

$$\frac{e\ \mathsf{val}}{e \mathbin{\texttt{@}} \mu\ \mathsf{final}} \tag{44.1e}$$

$$\frac{e\ \mathsf{val}}{x \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle \mapsto e \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle} \tag{44.1f}$$

$$\frac{e \mathbin{\texttt{@}} \mu \otimes \langle x : \bullet \rangle \mapsto e' \mathbin{\texttt{@}} \mu' \otimes \langle x : \bullet \rangle}{x \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle \mapsto x \mathbin{\texttt{@}} \mu' \otimes \langle x : e' \rangle} \tag{44.1g}$$

$$\frac{}{\texttt{s}(e) \mathbin{\texttt{@}} \mu \mapsto \texttt{s}(x) \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle} \tag{44.1h}$$

$$\frac{e \mathbin{\texttt{@}} \mu \mapsto e' \mathbin{\texttt{@}} \mu'}{\texttt{ifz}(e;e_0;x.e_1) \mathbin{\texttt{@}} \mu \mapsto \texttt{ifz}(e';e_0;x.e_1) \mathbin{\texttt{@}} \mu'} \tag{44.1i}$$

$$\frac{}{\texttt{ifz}(\texttt{z};e_0;x.e_1) \mathbin{\texttt{@}} \mu \mapsto e_0 \mathbin{\texttt{@}} \mu} \tag{44.1j}$$

$$\frac{x \notin dom(\mu)}{\texttt{ifz}(\texttt{s}(x);e_0;x.e_1) \mathbin{\texttt{@}} \mu \mapsto e_1 \mathbin{\texttt{@}} \mu} \tag{44.1k}$$

$$\frac{e_1 \mathbin{\texttt{@}} \mu \mapsto e_1' \mathbin{\texttt{@}} \mu'}{e_1(e_2) \mathbin{\texttt{@}} \mu \mapsto e_1'(e_2) \mathbin{\texttt{@}} \mu'} \tag{44.1l}$$

$$\frac{x \notin dom(\mu)}{\lambda(x{:}\tau.\,e)(e_2) \mathbin{\texttt{@}} \mu \mapsto e \mathbin{\texttt{@}} \mu \otimes \langle x : e_2 \rangle} \tag{44.1m}$$

$$\frac{x \notin dom(\mu)}{\texttt{fix}[\tau](x.e) \mathbin{\texttt{@}} \mu \mapsto x \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle} \tag{44.1n}$$

Rules (44.1a) through (44.1c) specify that z is a value, any expression of the form $\texttt{s}(x)$, where x is a variable, is a value, and any λ-abstraction, possibly containing free variables, is a value. Importantly, variables themselves are not values, since they may be bound by the memory to an unevaluated expression.

Rule (44.1d) specifies that an initial state consists of a binding for a closed expression, e, in memory, together with a demand for its binding. Rule (44.1e) specifies that a final state has the form $e \mathbin{\texttt{@}} \mu$, where e is a value.

Rule (44.1h) specifies that evaluation of $\texttt{s}(e)$ yields the value $\texttt{s}(x)$, where x is bound in the memory to e in unevaluated form. This reflects a lazy semantics for the successor, in which the predecessor is not evaluated until it is required by a conditional branch. Rule (44.1k), which governs a conditional branch on a sucessor, makes use of α-equivalence to choose the bound variable, x, for the predecessor to be the variable to which the predecessor was already bound by the successor operation. Evaluation of the successor branch of the conditional may make a demand on x, which would then cause the predecessor to be evaluated, as discussed above.

Rule (44.1l) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (44.1m) specifies that to evaluate an application of a λ-abstraction we create a fresh binding of its parameter to its *unevaluated* argument, and continue by evaluating its body. The freshness condition may always be met by implicitly renaming the bound variable of the λ-abstraction to be a variable not otherwise bound in the memory. Thus, each call results in a fresh binding of the parameter to the argument at the call.

The rules for variables are crucial, since they implement memoization. Rule (44.1f) governs a variable whose binding is a value, which is returned as the value of that variable. Rule (44.1g) specifies that if the binding of a variable is required and that binding is not yet a value, then its value must be determined before further progress can be made. This is achieved by switching the "focus" of evaluation to the binding, while at the same time replacing the binding by a *black hole*, which represents the absence of a value for that variable (since it has not yet been determined). Evaluation of a variable whose binding is a black hole is "stuck", since it indicates a circular dependency of the value of a variable on the variable itself.

Rule (44.1n) implements general recursion. Recall from Chapter 16 that the expression $\texttt{fix[}\tau\texttt{]}(x.e)$ stands for the solution of the recursion equation $x = e$, where x may occur within e. Rule (44.1n) obtains the solution directly by equating x to e in the memory, and returning x. The role of the black hole becomes evident when evaluating an expression such as $\texttt{fix}\,x{:}\tau\,\texttt{is}\,x$. Evaluation of this expression binds the variable x to itself in the memory, and then returns x, creating a demand for its binding. Applying Rule (44.1g), we see that this immediately leads to a stuck state in which we require the value of x in a memory in which it is bound to the black hole. This captures the inherent circularity in the purported definition of x, and amounts to catching a potential infinite loop before it happens. Observe that, by contrast, an expression such as $\texttt{fix}\,f{:}\sigma \rightarrow \tau\,\texttt{is}\,\lambda(x{:}\sigma.\,e)$ does not get stuck, because the occurrence of the recursively defined variable, f, lies within the λ-expression. Evaluation of a λ-abstraction, being a value, creates no demand for f, so the black hole is not encountered. Rule (44.1g) backpatches the binding of f to be the λ-abstraction itself, so that subsequent uses of f evaluate to it, as would be expected. Thus recursion is automatically implemented by the backpatching technique described in Chapter 37.

The type safety of the by-need semantics for lazy $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ is proved using methods similar to those developed in Chapter 37 for references. To do so we define the judgement $e \mathbin{\texttt{@}} \mu\ \mathsf{ok}$ to hold iff there exists a set of typing assumptions Γ governing the variables in the domain of the memory, μ, such that

1. if $\Gamma = \Gamma', x : \tau_x$ and $\mu(x) = e \neq \bullet$, then $\Gamma \vdash e : \tau_x$.
2. there exists a type τ such that $\Gamma \vdash e : \tau$.

As a notational convenience, we will sometimes write $\mu : \Gamma \vdash e : \tau$ for the conjunction of these two conditions.

Theorem 44.1 (Preservation). *If $e \mathbin{\texttt{@}} \mu \mapsto e' \mathbin{\texttt{@}} \mu'$ and $e \mathbin{\texttt{@}} \mu$ ok, then $e' \mathbin{\texttt{@}} \mu'$ ok.*

Proof. The proof is by rule induction on Rules (44.1). For the induction we prove the stronger result that if $\mu : \Gamma$ and $\Gamma \vdash e : \tau$, then there exists Γ' such that $\mu' : \Gamma\,\Gamma' \vdash e' : \tau$. We will consider two illustrative cases of the proof.

Consider Rule (44.1l), for which $e = e_1(e_2)$. Suppose that $\mu : \Gamma$ and $\Gamma \vdash e : \tau$. Then by inversion of typing $\Gamma \vdash e_1 : \tau_2 \to \tau$ for some type τ_2 such that $\Gamma \vdash e_2 : \tau_2$. So by induction there exists Γ' such that $\mu' : \Gamma\,\Gamma' \vdash e_1' : \tau_2 \to \tau$. By weakening $\Gamma\,\Gamma' \vdash e_2 : \tau_2$, and hence $\mu' : \Gamma\,\Gamma' \vdash e_1'(e_2) : \tau$. We have only to notice that $e' = e_1'(e_2)$ to complete this case.

Consider Rule (44.1g), for which we have $e = e' = x$, $\mu = \mu_0 \otimes \langle x : e_0 \rangle$, and $\mu' = \mu_0' \otimes \langle x : e_0' \rangle$, where $e_0 \mathbin{\texttt{@}} \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0' \mathbin{\texttt{@}} \mu_0' \otimes \langle x : \bullet \rangle$. Assume that $\mu : \Gamma \vdash e : \tau$; we are to show that there exists Γ' such that $\mu' : \Gamma\,\Gamma' \vdash e_0' : \tau$. Since $\mu : \Gamma$ and e is the variable x, we have that $\Gamma = \Gamma'', x : \tau$ and $\Gamma \vdash e_0 : \tau$. Therefore $\mu_0 \otimes \langle x : \bullet \rangle : \Gamma$, so by induction there exists Γ' such that $\mu_0' \otimes \langle x : \bullet \rangle : \Gamma\,\Gamma' \vdash e_0' : \tau$. But then $\mu_0' \otimes \langle x : e_0' \rangle : \Gamma\,\Gamma' \vdash x : \tau$, as required. □

The progress theorem must be stated so as to account for accessing a variable that is bound to a black hole, which is tantamount to a detectable form of looping. Since the type system does not rule this out, we define the judgement $e \mathbin{\texttt{@}} \mu\ \mathsf{loops}$ by the following rules:

$$\frac{}{x \mathbin{\texttt{@}} \mu \otimes \langle x : \bullet \rangle\ \mathsf{loops}} \tag{44.2a}$$

$$\frac{e \mathbin{\texttt{@}} \mu \otimes \langle x : \bullet \rangle\ \mathsf{loops}}{x \mathbin{\texttt{@}} \mu \otimes \langle x : e \rangle\ \mathsf{loops}} \tag{44.2b}$$

$$\frac{e \mathbin{\texttt{@}} \mu\ \mathsf{loops}}{\texttt{ifz}(e; e_0; x.e_1) \mathbin{\texttt{@}} \mu\ \mathsf{loops}} \tag{44.2c}$$

$$\frac{e_1 \mathtt{@}\ \mu\ \mathsf{loops}}{\mathtt{ap}(e_1;e_2)\ \mathtt{@}\ \mu\ \mathsf{loops}} \tag{44.2d}$$

In general looping is propagated through the principal argument of every eliminatory construct, since this argument position must always be evaluated in any transition sequence involving it.

The progress theorem is weakened to account for detectable looping.

Theorem 44.2 (Progress). *If $e\ \mathtt{@}\ \mu$ ok, then either $e\ \mathtt{@}\ \mu$ final, or $e\ \mathtt{@}\ \mu$ loops, or there exists μ' and e' such that $e\ \mathtt{@}\ \mu \mapsto e'\ \mathtt{@}\ \mu'$.*

Proof. We prove by rule induction on the static semantics that if $\mu : \Gamma \vdash e : \tau$, then either e val, or $e\ \mathtt{@}\ \mu$ loops, or $e\ \mathtt{@}\ \mu \mapsto e'\ \mathtt{@}\ \mu'$ for some μ' and e'. The proof is by lexicographic induction on the measure (m, n), where $n \geq 0$ is the size of e and $m \geq 0$ is the sum of the sizes of the non-black-hole bindings of each variable in the domain of μ. This means that we may appeal to the inductive hypothesis for sub-expressions of e, since they have smaller size, provided that the size of the memory remains fixed. Since the size of $\mu \otimes \langle x : \bullet \rangle$ is strictly smaller than the size of $\mu \otimes \langle x : e_x \rangle$ for *any* expression e_x, we may also appeal to the inductive hypothesis for expressions larger than e, provided we do so relative to a smaller memory.

As an example of the former case, consider the case of Rule (16.1f), for which $e = \mathtt{ap}(e_1;e_2)$, where $\mu : \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau)$ and $\mu : \Gamma \vdash e_2 : \tau_2$. By the induction hypothesis applied to e_1, we have that either e_1 val or $e_1\ \mathtt{@}\ \mu$ loops or $e_1\ \mathtt{@}\ \mu \mapsto e_1'\ \mathtt{@}\ \mu'$.

In the first case it may be shown that $e_1 = \mathtt{lam}[\tau_2](x.e)$, and hence that $\mathtt{ap}(e_1;e_2)\ \mathtt{@}\ \mu \mapsto e\ \mathtt{@}\ \mu' \otimes \langle x : e_2 \rangle$ by Rule (44.1m), where x is chosen by α-equivalence to lie outside of the domain of μ'. In the second case we have by Rule (44.2d) that $\mathtt{ap}(e_1;e_2)\ \mathtt{@}\ \mu$ loops. In the third case we have by Rule (44.1l) that $\mathtt{ap}(e_1;e_2)\ \mathtt{@}\ \mu \mapsto \mathtt{ap}(e_1';e_2)\ \mathtt{@}\ \mu'$.

Now consider Rule (16.1a), for which we have $\Gamma \vdash x : \tau$ with $\Gamma = \Gamma', x : \tau$. For any μ such that $\mu : \Gamma$, we have that $\mu = \mu_0 \otimes \langle x : e_0 \rangle$ with $\mu_0 \otimes \langle x : \bullet \rangle : \Gamma \vdash e_0 : \tau$. Since the memory $\mu_0 \otimes \langle x : \bullet \rangle$ is smaller than the memory μ, we have by induction that either e_0 val or $e_0\ \mathtt{@}\ \mu_0 \otimes \langle x : \bullet \rangle$ loops, or $e_0\ \mathtt{@}\ \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0'\ \mathtt{@}\ \mu_0' \otimes \langle x : \bullet \rangle$.

If e_0 val, then $x\ \mathtt{@}\ \mu_0 \otimes \langle x : e_0 \rangle \mapsto e_0\ \mathtt{@}\ \mu_0 \otimes \langle x : e_0 \rangle$ by Rule (44.1f). If $e_0\ \mathtt{@}\ \mu_0 \otimes \langle x : \bullet \rangle$ loops, then $x\ \mathtt{@}\ \mu_0 \otimes \langle x : e_0 \rangle$ loops by Rule (44.2b). Finally, if $e_0\ \mathtt{@}\ \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0'\ \mathtt{@}\ \mu_0' \otimes \langle x : \bullet \rangle$, then $x\ \mathtt{@}\ \mu_0 \otimes \langle x : e_0 \rangle \mapsto x\ \mathtt{@}\ \mu_0' \otimes \langle x : e_0' \rangle$ by Rule (44.1g). □

44.2 Lazy Data Structures

Call-by-need evaluation addresses only one aspect of laziness, namely deferring evaluation of function arguments until they are needed, and sharing the value among all other uses of it. Other aspects of laziness pertain to product, sum, and recursive types, whose introductory forms may be given a lazy interpretation, with memoization of unevaluated sub-expressions to avoid needless recomputation.

The "by need" dynamic semantics of product types is given by the following set of rules:

$$\frac{}{\texttt{pair}(x_1;x_2)\ \mathsf{val}} \tag{44.3a}$$

$$\frac{}{\texttt{pair}(e_1;e_2)\ \texttt{@}\ \mu \mapsto \texttt{pair}(x_1;x_2)\ \texttt{@}\ \mu \otimes \langle x_1 : e_1 \rangle \otimes \langle x_2 : e_2 \rangle} \tag{44.3b}$$

$$\frac{e\ \texttt{@}\ \mu \mapsto e'\ \texttt{@}\ \mu'}{\texttt{fst}(e)\ \texttt{@}\ \mu \mapsto \texttt{fst}(e')\ \texttt{@}\ \mu'} \tag{44.3c}$$

$$\frac{}{\texttt{fst}(\texttt{pair}(x_1;x_2))\ \texttt{@}\ \mu \mapsto x_1\ \texttt{@}\ \mu} \tag{44.3d}$$

$$\frac{e\ \texttt{@}\ \mu\ \mathsf{loops}}{\texttt{fst}(e)\ \texttt{@}\ \mu\ \mathsf{loops}} \tag{44.3e}$$

$$\frac{e\ \texttt{@}\ \mu \mapsto e'\ \texttt{@}\ \mu'}{\texttt{snd}(e)\ \texttt{@}\ \mu \mapsto \texttt{snd}(e')\ \texttt{@}\ \mu'} \tag{44.3f}$$

$$\frac{}{\texttt{snd}(\texttt{pair}(x_1;x_2))\ \texttt{@}\ \mu \mapsto x_2\ \texttt{@}\ \mu} \tag{44.3g}$$

$$\frac{e\ \texttt{@}\ \mu\ \mathsf{loops}}{\texttt{snd}(e)\ \texttt{@}\ \mu\ \mathsf{loops}} \tag{44.3h}$$

A pair is considered a value only if its arguments are variables (Rule (44.3a)), which are introduced when the pair is created (Rule (44.3b)). The first and second projections evaluate to one or the other variable in the pair, inducing a demand for the value of that component. This ensures that another occurrence of the same projection of the same pair will yield the same value without having to recompute it.

The by-need semantics of sums and recursive types follow a similar pattern, and are left as an exercise for the reader.

44.3 Suspensions By Need

In Chapter 43 it is suggested that laziness be confined to a type of self-referential suspensions. To avoid needless recomputation it is essential to give a by-need semantics to suspensions, following along similar lines to the by-need semantics of a lazy language. The chief difference is that variables are regarded as *values*, rather than as *computations* to be evaluated. To force evaluation of a memoized computation, we must explicitly use the elimination form for suspension types, rather than simply refer to it via the variable to which it is bound.

The by-need semantics of suspensions is given by the following rules:

$$\frac{}{x\ \mathsf{val}} \tag{44.4a}$$

$$\frac{}{\texttt{susp[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \texttt{@}\ \mu \mapsto x\ \texttt{@}\ \mu \otimes \langle x : e\rangle} \tag{44.4b}$$

$$\frac{e\ \texttt{@}\ \mu \mapsto e'\ \texttt{@}\ \mu'}{\texttt{force(}e\texttt{)}\ \texttt{@}\ \mu \mapsto \texttt{force(}e'\texttt{)}\ \texttt{@}\ \mu'} \tag{44.4c}$$

$$\frac{e\ \mathsf{val}}{\texttt{force(}x\texttt{)}\ \texttt{@}\ \mu \otimes \langle x : e\rangle \mapsto e\ \texttt{@}\ \mu \otimes \langle x : e\rangle} \tag{44.4d}$$

$$\frac{e\ \texttt{@}\ \mu \otimes \langle x : \bullet\rangle \mapsto e'\ \texttt{@}\ \mu' \otimes \langle x : \bullet\rangle}{\texttt{force(}x\texttt{)}\ \texttt{@}\ \mu \otimes \langle x : e\rangle \mapsto \texttt{force(}x\texttt{)}\ \texttt{@}\ \mu' \otimes \langle x : e'\rangle} \tag{44.4e}$$

It is straightforward to adapt the type safety proof given in Section 44.1 on page 344 to the special case of suspension types.

44.4 Exercises

Part XV

Parallelism

Chapter 45

Speculative Parallelism

The semantics of call-by-need given in Chapter 44 suggests opportunities for *speculative parallelism*. Evaluation of a delayed binding is initiated as soon as the binding is created, executing simultaneously with the evaluation of the body. Should the variable ever be needed, evaluation of the body synchronizes with the concurrent evaluation of the binding, and proceeds only once the value is available. This form of parallelism is called speculative, because the value of the binding may never be needed, in which case the resources required for its evaluation are wasted. However, in some situations there are available computing resources that would otherwise be wasted, and which can be usefully employed for speculative evaluation.

There is also a speculative version of suspensions, called *futures*, which behave in the same manner, except that the synchronization points are explicit in the form of calls to force the suspension. The suspended computation can be executed in parallel on the hypothesis that its value will eventually be needed to proceed.

45.1 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 44.1 on page 344 by the following general rule:

$$\frac{e \mathbin{@} \mu \otimes \langle y : \bullet \rangle \mapsto e' \mathbin{@} \mu' \otimes \langle y : \bullet \rangle}{e_0 \mathbin{@} \mu \otimes \langle y : e \rangle \mapsto e_0 \mathbin{@} \mu' \otimes \langle y : e' \rangle} \tag{45.1}$$

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 semantics is said to be *non-deterministic* because the transition relation is no longer a partial function on states. That is, for a given state $e \mathbin{@} \mu$, there may be many different states $e' \mathbin{@} \mu'$ such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$, precisely because the foregoing rule permits us to shift attention to any location in memory at any time. The rules abstract away from the specifics of how such "context switches" might be scheduled, permitting them to occur at any time so as to be consistent with any scheduling strategy. In this sense non-determinism models parallel execution by permitting the individual steps of a complete computation to be interleaved in an arbitrary manner.

The non-deterministic semantics is said to be *speculative*, because it permits evaluation of any suspended expression at any time, without regard to whether its value is needed to determine the overall result of the computation. In this sense it is contrary to the spirit of call-by-need, since it may perform work that is not strictly necessary. The benefit of speculation is that it leads to a form of parallel computation, called *speculative parallelism*, which seeks to exploit computing resources that would otherwise be left idle. Ideally one should only use processors to compute results that are needed, but in some situations it is difficult to make full use of available resources without resorting to speculation.

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" while computation along the main thread proceeds. When a suspension is forced, evaluation 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. The semantics of futures is a straightforward modification to the semantics of suspensions given in Chapter 44 to permit arbitrary context switches that evaluate suspended computations.

45.2 Speculative Parallelism

The non-deterministic semantics given in Section 45.1 on the preceding page captures the idea of speculative execution, but addresses parallelism only indirectly, by avoiding specification of when the focus of evaluation may shift from one suspended expression to another. The semantics is specified from the point of view of an omniscient observer who sequentializes the parallel execution into a sequence of atomic steps. No particular sequentialization is enforced; rather, all possible sequentializations are derivable from the rules.

A more accurate model is one that makes explicit the parallel speculative evaluation of some number of suspended computations. We model this using a judgement of the form $\mu \mapsto \mu'$, which specifies the simultaneous execution of a computation step on each of $k > 0$ suspended computations.

$$\frac{\left\{\begin{array}{c} e_i \mathbin{@} \mu \otimes \langle x_1 : \bullet\rangle \otimes \cdots \otimes \langle x_k : \bullet\rangle \\ \mapsto \\ e_i' \mathbin{@} \mu \otimes \langle x_1 : \bullet\rangle \otimes \cdots \otimes \langle x_k : \bullet\rangle \otimes \mu_i \end{array}\right\} \quad (\forall 1 \leq i \leq k)}{\left\{\begin{array}{c} \mu \otimes \langle x_1 : e_1\rangle \otimes \cdots \otimes \langle x_k : e_k\rangle \\ \mapsto \\ \mu \otimes \langle x_1 : e_1'\rangle \otimes \cdots \otimes \langle x_k : e_k'\rangle \otimes \mu_1 \otimes \cdots \otimes \mu_k \end{array}\right\}} \tag{45.2}$$

This rule may be seen as a generalization of Rule (44.1g), except that it applies independently of whether there is a demand for any of the variables involved. The transition consists of choosing $k > 0$ suspended computations on which to make progress, and simultaneously taking a step on each, and restoring the results to the memory. The choice of k is left unspecified, but is fixed for all inferences; in practice it would be related to the number of available processors.

The speculative parallel semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is defined by replacing Rule (44.1g) by the following rule:

$$\frac{\mu \mapsto \mu'}{e \mathbin{@} \mu \mapsto e \mathbin{@} \mu'} \tag{45.3}$$

This rules specifies that, at any moment, we may make progress by executing a step of evaluation on some number of suspended computations. Since Rule (44.1g) has been omitted, this rule must be applied sufficiently often to ensure that the binding of any required variable is fully evaluated before its value is required. The goal of speculative execution is to ensure that this is always the case, but in practice a computation must sometimes be suspended to await completion of evaluation of the binding of some variable.

There is a technical complication with Rule (45.2), however, that lies at the heart of any parallel programming language. When executing computations in parallel, it is possible that two or more of them choose the *same* variable to represent a new suspended computation. Formally, this occurs when the domain of μ_i intersects the domain of μ_j for some $i \neq j$ in the premise of Rule (45.2). In practice this corresponds to two threads attempting to allocate memory at the same time: some synchronization is required to resolve the contention. In a formal model we may leave the means of achieving this abstract, and simply demand as a side condition that the memories $\mu_1, \dots, \mu_k$ have disjoint domains. This may always be achieved by choosing variable names independently for each thread. In an implementation some method is required to support memory allocation in parallel, using one of several well-known synchronization methods (such as an atomic fetch-and-add instruction).

45.3 Exercises

Chapter 46

Work-Efficient Parallelism

In this chapter we study the concept of *work-efficient parallelism*, which exploits opportunities for parallelism without increasing the workload compared to a sequential execution. This is in contrast to speculative parallelism (see Chapter 45), which exposes parallelism, but potentially at the cost of doing more work than would be done in the sequential case. In a speculative semantics we may evaluate suspended computations even though their value is never required for the ultimate result. The work expended in computing the value of the suspension is wasted; it keeps the processor warm, but could just as well have been omitted. In contrast work-efficient parallelism never wastes effort; it only performs computations whose results are required for the final outcome.

To make these ideas precise we make use of a *cost semantics*, which determines not only the value of an expression, but a measure of the cost of evaluating it. The costs are chosen so as to expose both opportunities for and obstructions to parallelism. If one computation depends on the result of another, then there is a sequential dependency between them that precludes their execution in parallel. If, on the other hand, two computations are independent of one another, then they can be executed in parallel. Functional languages without state provide ample opportunities for parallelism, and will be the focus of our work in this chapter.

46.1 A Simple Parallel Language

We begin with a very simple parallel language whose sole source of parallelism arises from the evaluation of two variable bindings simultaneously. This is modelled by a construct of the form $\texttt{let}\,x_1 = e_1\,\texttt{and}\,x_2 = e_2\,\texttt{in}\,e$, in which we bind two variables, x_1 and x_2, to two expressions, e_1 and e_2, respectively, for use within a single expression, e. This represents a simple fork-join primitive in which e_1 and e_2 may be evaluated independently of one another, with their results combined by the expression e. Some other forms of parallelism may be defined in terms of this primitive. For example, a *parallel pair* construct might be defined as the expression

$$\texttt{let}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ \langle x_1, x_2 \rangle,$$

which evaluates the components of the pair in parallel, then constructs the pair itself from these values.

The abstract syntax of the parallel binding construct is given by the abstract binding tree

$$\texttt{let}(e_1;e_2;x_1.x_2.e),$$

which makes clear that the variables x_1 and x_2 are bound *only* within e, and not within their bindings. This ensures that evaluation of e_1 is independent of evaluation of e_2, and *vice versa*. The typing rule for an expression of this form is given as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{let}(e_1;e_2;x_1.x_2.e) : \tau} \tag{46.1}$$

Although we emphasize the case of binary parallelism, it should be clear that this construct easily generalizes to n-way parallelism for any *static* value of n. One may also define an n-way parallel `let` construct from the binary parallel `let` by cascading binary splits. (For a treatment of n-way parallelism for a *dynamic* value of n, see Section 46.4 on page 367.)

We will give both a *sequential* and a *parallel* dynamic semantics of the parallel `let` construct. The definition of the sequential dynamics as a transition judgement of the form $e \mapsto_{seq} e'$ is entirely straightforward:

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{seq} \texttt{let}(e_1';e_2;x_1.x_2.e)} \tag{46.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{seq} \texttt{let}(e_1;e_2';x_1.x_2.e)} \tag{46.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{seq} [e_1, e_2/x_1, x_2]e} \tag{46.2c}$$

The parallel dynamics is given by a transition judgement of the form $e \mapsto_{par} e'$, defined as follows:

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\texttt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} \texttt{let}(e_1';e_2';x_1.x_2.e)} \tag{46.3a}$$

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \ \mathsf{val}}{\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} \mathtt{let}(e_1';e_2;x_1.x_2.e)} \tag{46.3b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto_{par} e_2'}{\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} \mathtt{let}(e_1;e_2';x_1.x_2.e)} \tag{46.3c}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} [e_1,e_2/x_1,x_2]e} \tag{46.3d}$$

The parallel semantics is idealized in that it abstracts away from any limitations on parallelism that would necessarily be imposed in practice by the availability of computing resources. (We will return to this point in Section 46.3 on page 364.)

An important advantage of the present approach is captured by the *implicit parallelism theorem*, which states that the sequential and the parallel semantics coincide. This means that one need never be concerned with the *semantics* of a parallel program (its meaning is determined by the sequential dynamics), but only with its *performance*. Put in other terms, this language exhibits *deterministic parallelism*, which does not effect the correctness of programs, in contrast to languages such as those to be considered in Chapter 47, which exhibit *non-deterministic parallelism*, or *concurrency*.

Lemma 46.1. *If $\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par}^* v$ with v val, then there exists v_1 val and v_2 val such that $e_1 \mapsto_{par}^* v_1$, $e_2 \mapsto_{par}^* v_2$, and $[v_1,v_2/x_1,x_2]e \mapsto_{par}^* v$.*

Proof. Since v val, the given derivation must consist of one or more steps. We proceed by induction on the derivation of the first step, $\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} e'$. For Rule (46.3d), we have e_1 val and e_2 val, and $e' = [e_1,e_2/x_1,x_2]e$, so we may take $v_1 = e_1$ and $v_2 = e_2$ to complete the proof. The other cases follow easily by induction. □

Lemma 46.2. *If $\mathtt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{seq}^* v$ with v val, then there exists v_1 val and v_2 val such that $e_1 \mapsto_{seq}^* v_1$, $e_2 \mapsto_{seq}^* v_2$, and $[v_1,v_2/x_1,x_2]e \mapsto_{seq}^* v$.*

Proof. Similar to the proof of Lemma 46.2. □

Theorem 46.3. *The sequential and parallel dynamics coincide: for all v val, $e \mapsto_{seq}^* v$ iff $e \mapsto_{par}^* v$.*

Proof. From left to right it is enough to prove that if $e \mapsto_{seq} e' \mapsto_{par}^* v$ with v val, then $e \mapsto_{par}^* v$. This may be shown by induction on the derivation of $e \mapsto_{seq} e'$. If $e \mapsto_{seq} e'$ by Rule (46.2c), then by Rule (46.3d) we have $e \mapsto_{par} e'$, and hence $e \mapsto_{par}^* v$. If $e \mapsto_{seq} e'$ by Rule (46.2a), then we have $e = \texttt{let}(e_1;e_2;x_1.x_2.e)$, $e' = \texttt{let}(e_1';e_2;x_1.x_2.e)$, and $e_1 \mapsto_{seq} e_1'$. By Lemma 46.1 on the preceding page there exists v_1 val and v_2 val such that $e_1' \mapsto_{par}^{*} v_1$, $e_2 \mapsto_{par}^{*} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto_{par}^{*} v$. By induction we have $e_1 \mapsto_{par}^{*} v_1$, and hence $e \mapsto_{par}^{*} v$. The other cases are handled similarly.

From right to left, it is enough to prove that if $e \mapsto_{par} e' \mapsto_{seq}^{*} v$ with v val, then $e \mapsto_{seq}^{*} v$. We proceed by induction on the derivation of $e \mapsto_{par} e'$. Rule (46.3d) carries over directly to the sequential case by Rule (46.2c). Consider Rule (46.3a). We have $\texttt{let}(e_1;e_2;x_1.x_2.e) \mapsto_{par} \texttt{let}(e_1';e_2';x_1.x_2.e)$, $e_1 \mapsto_{par} e_1'$, and $e_2 \mapsto_{par} e_2'$. By Lemma 46.2 on the previous page we have that there exists v_1 val and v_2 val such that $e_1' \mapsto_{seq}^{*} v_1$, $e_2' \mapsto_{seq}^{*} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto_{seq}^{*} v$. By induction we have $e_1 \mapsto_{seq}^{*} v_1$ and $e_2 \mapsto_{seq}^{*} v_2$, and hence $e \mapsto_{seq}^{*} v$, as required. The other cases are handled similarly. □

Theorem 46.3 on the preceding page is the basis for saying that the model of parallelism discussed in this chapter is *work-efficient*—the computations performed in any execution, sequential or parallel, are precisely those that must be performed acording to the sequential semantics. This is in contrast to speculative parallelism, as discussed in Chapter 45, in which we may schedule a task for execution whose outcome is not needed to determine the overall result of the computation. This theorem may also be read as saying that we have achieved *implicit parallelism* in that the use of parallelism in evaluation has no effect on the overall end-to-end behavior of a program. An expression has a value according to the sequential semantics iff it does so according to the parallel semantics. In other words one never need worry about correctness when programming in an implicitly parallel language, but instead only about asymptotic efficiency.

46.2 Cost Semantics

In this section we define a *parallel cost semantics* that assigns a *cost graph* to the evaluation of an expression. Cost graphs are defined by the following grammar:

Cost	c	$::=$	$\mathbf{0}$	zero cost
		$\mid$	$\mathbf{1}$	unit cost
		$\mid$	$c_1 \otimes c_2$	parallel combination
		$\mid$	$c_1 \oplus c_2$	sequential combination

A cost graph is a form of *series-parallel* directed acyclic graph, with a designated *source* node and *sink* node. For $\mathbf{0}$ the graph consists of one node and no edges, with the source and sink both being the node itself. For **1** the graph consists of two nodes and one edge directed from the source to the sink. For $c_1 \otimes c_2$, if g_1 and g_2 are the graphs of c_1 and c_2, respectively, then the graph has two additional nodes, a source node with two edges to the source nodes of g_1 and g_2, and a sink node, with edges from the sink nodes of g_1 and g_2 to it. Finally, for $c_1 \oplus c_2$, where g_1 and g_2 are the graphs of c_1 and c_2, the graph has as source node the source of g_1, as sink node the sink of g_2, and an edge from the sink of g_1 to the source of g_2.

The intuition behind a cost graph is that nodes represent subcomputations of an overall computation, and edges represent *sequentiality constraints* stating that one computation depends on the result of another, and hence cannot be started before the one on which it depends completes. The product of two graphs represents *parallelism opportunities* in which there are no sequentiality constraints between the two computations. The assignment of source and sink nodes reflects the overhead of *forking* two parallel computations and *joining* them after they have both completed.

We associate with each cost graph two numeric measures, the *work*, $wk(c)$, and the *depth*, $dp(c)$. The work is defined by the following equations:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{46.4}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{46.5}$$

Informally, the work of a cost graph determines the total number of computation steps represented by the cost graph, and thus corresponds to the *sequential complexity* of the computation. The depth of the cost graph determines the *critical path length*, the length of the longest dependency chain within the computation, which imposes a lower bound on the idealized *parallel complexity* of a computation. It is idealized in that it may be achieved only by taking full advantage of all available parallelism opportunities in the cost graph, which can only be achieved with unbounded computing resources.

In Chapter 12 we introduced the concept of a *cost semantics* as a means of assigning a step complexity to evaluation. The proof of Theorem 12.6 on page 89 shows that $e \Downarrow^{k} v$ iff $e \mapsto^{k} v$. That is, the step complexity of an evaluation of e to a value v is just the number of transitions required to derive $e \mapsto^{*} v$. Here we use cost graphs as the measure of complexity, then relate these cost graphs to the transition semantics given in Section 46.1 on page 357.

The judgement $e \Downarrow^{c} v$, where e is a closed expression, v is a closed value, and c is a cost graph specifies the cost semantics. By definition we arrange that $e \Downarrow^{\mathbf{0}} e$ when e val. The cost assignmetn for `let` is given by the following rule:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2 / x_1, x_2]e \Downarrow^{c} v}{\texttt{let}(e_1; e_2; x_1 . x_2 . e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{46.6}$$

The cost assignment specifies that, under ideal conditions, e_1 and e_2 are to be evaluated in parallel, and that their results are to be propagated to e. The cost of fork and join is implicit in the parallel combination of costs, and assign unit cost to the substitution because we expect it to be implemented in practice by a constant-time mechanism for updating an environment. The cost semantics of other language constructs is specified in a similar manner, using only sequential combination so as to isolate the source of parallelism to the `let` construct.

The link between the cost semantics and the transition semantics given in the preceding section is established by the following theorem, which states that the work cost is the sequential complexity, and the depth cost is the parallel complexity, of the computation.

Theorem 46.4. *If $e \Downarrow^{c} v$, then $e \mapsto^{w}_{seq} v$ and $e \mapsto^{d}_{par} v$, where $w = wk(c)$ and $d = dp(c)$. Conversely, if $e \mapsto^{w}_{seq} v$ with v val, then there exists a cost graph c such that $wk(c) = w$ and $e \Downarrow^{c} v$, and, moreover, if $e \mapsto^{d}_{par} v$ with v val, then $dp(c) = d$.*

Proof. The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being Rule (46.6). By induction we have $e_1 \mapsto^{w_1}_{seq} v_1$, $e_2 \mapsto^{w_2}_{seq} v_2$, and $[v_1, v_2 / x_1, x_2]e \mapsto^{w}_{seq} v$, where $w_1 = wk(c_1)$, $w_2 = wk(c_2)$, and $w = wk(c)$. By pasting together derivations we obtain a derivation

$$\begin{aligned}
\texttt{let}(e_1;e_2;x_1.x_2.e) &\mapsto_{seq}^{w_1} \texttt{let}(v_1;e_2;x_1.x_2.e) && (46.7)\\
&\mapsto_{seq}^{w_2} \texttt{let}(v_1;v_2;x_1.x_2.e) && (46.8)\\
&\mapsto_{seq} [v_1,v_2/x_1,x_2]e && (46.9)\\
&\mapsto_{seq}^{w} v. && (46.10)
\end{aligned}$$

Noting that $wk((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \mapsto_{par}^{d_1} v_1$, $e_2 \mapsto_{par}^{d_2} v_2$, and $e \mapsto_{par}^{d} v$, where $d_1 = dp(c_1)$, $d_2 = dp(c_2)$, and $d = dp(c)$. Assume, without loss of generality, that $d_1 \leq d_2$ (otherwise simply swap the roles of d_1 and d_2 in what follows). We may paste together derivations as follows:

$$\begin{aligned}
\texttt{let}(e_1;e_2;x_1.x_2.e) &\mapsto_{par}^{d_1} \texttt{let}(v_1;e_2';x_1.x_2.e) && (46.11)\\
&\mapsto_{par}^{d_2-d_1} \texttt{let}(v_1;v_2;x_1.x_2.e) && (46.12)\\
&\mapsto_{par} [v_1,v_2/x_1,x_2]e && (46.13)\\
&\mapsto_{par}^{d} v. && (46.14)
\end{aligned}$$

Calculating $dp((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1,d_2) + 1 + d$ completes the proof.

The second part is proved by induction on w (respectively, d) to obtain the required cost derivation. If $w = 0$, then $e = v$ and hence $e \Downarrow^{\mathbf{0}} v$. If $w = w' + 1$, then it is enough to show that if $e \mapsto_{seq} e'$ and $e' \Downarrow^{c'} v$ with $wk(c') = w'$, then $e \Downarrow^{c} v$ for some c such that $wk(c) = w$. We proceed by induction on the derivation of $e \mapsto_{seq} e'$. Consider Rule (46.2c). We have $e = \texttt{let}(e_1;e_2;x_1.x_2.e_0)$ with e_1 val and e_2 val, and $e' = [e_1,e_2/x_1,x_2]e_0$. By definition $e_1 \Downarrow^{\mathbf{0}} e_1$ and $e_2 \Downarrow^{\mathbf{0}} e_2$, since e_1 and e_2 are values. It follows that $e \Downarrow^{(\mathbf{0}\otimes\mathbf{0})\oplus\mathbf{1}\oplus c'} v$ by Rule (46.6). But $wk((\mathbf{0} \otimes \mathbf{0}) \oplus \mathbf{1} \oplus c') = 1 + wk(c') = 1 + w' = w$, as required. The remaining cases for sequential derivations follow a similar pattern. Turning to the parallel derivations, consider Rule (46.3a), in which we have $e = \texttt{let}(e_1;e_2;x_1.x_2.e_0) \mapsto_{par} \texttt{let}(e_1';e_2';x_1.x_2.e_0) = e'$, with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. We have by the outer inductive assumption that $e' \Downarrow^{c'} v$ for some c' such that $dp(c') = d'$, and we are to show that $e \Downarrow^{c} v$ for some c such that $dp(c) = 1 + d' = d$. It follows from the form of e' and the determinacy of evaluation that $c' = (c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0$, where $e_1' \Downarrow^{c_1'} v_1$, $e_2' \Downarrow^{c_2'} v_2$, and $[v_1,v_2/x_1,x_2]e_0 \Downarrow^{c_0} v$. It follows by the inner induction that $e_1 \Downarrow^{c_1} v_1$ for some c_1 such that $dp(c_1) = dp(c_1') + 1$, and that $e_2 \Downarrow^{c_2} v_2$ for some c_2 such that $dp(c_2) = dp(c_2') + 1$. But then $e \Downarrow^{c} v$, where

$c = (c_1 \otimes c_2) \oplus \mathbf{1} \oplus c_0$. Calculating, we obtain

$$\begin{aligned}
dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) && (46.15)\\
&= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) && (46.16)\\
&= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 && (46.17)\\
&= dp(c') + 1 && (46.18)\\
&= d' + 1 && (46.19)\\
&= d, && (46.20)
\end{aligned}$$

which completes the proof. □

46.3 Provable Implementations

Theorem 46.4 on page 362 states that the cost semantics accurately models the dynamics of the parallel `let` construct, whether executed sequentially or with maximal parallelism. This validates the cost model from the point of view of the language definition, and permits us to draw conclusions about the asymptotic complexity of a parallel program that abstracts away from the limitations imposed by a concrete implementation. Chief among these is the limitation to some fixed number, $p > 0$, of processors on which to multiplex the workload. In addition to limiting the available parallelism this also imposes some scheduling and synchronization overhead that must be accounted for in order to make accurate predictions of run-time behavior on a concrete parallel platform. A *provable implementation* is one for which we may establish an asymptotic bound on the actual execution time once these overheads are taken into account.

For the purposes of chapter, we define a *symmetric multiprocessor*, or *SMP*, to be a shared-memory multiprocessor with an interconnection network that implements a synchronization construct equivalent to a parallel-fetch-and-add instruction in which any number of processors may simultaneously add a value to a shared memory location, retrieving the previous contents, while ensuring that each processor obtains the result it would obtain in some sequential ordering of their execution. Most multiprocessors implement an instruction of expressive power equivalent to the fetch-and-add to provide a foundation for parallel programming. In the following analysis we assume that the fetch-and-add instruction takes constant time, but the result can be adjusted (as noted below) to account for the overhead of implementing it under more relaxed assumptions about the processor model.

The main result relating the abtract cost to its concrete realization on a p-processor SMP is an application of Brent's Principle, which describes how to implement arithmetic expressions on a parallel processor.

Theorem 46.5. *If $e \Downarrow^c v$ with $wk(c) = w$ and $dp(c) = d$, then e may be evaluated on a p-processor SMP in time $O(\max(w/p, d))$, given a constant-time stable fetch-and-add facility.*

Since the work always dominates the depth, if $p = 1$, then the theorem reduces to the statement that e may be evaluated in time $O(w)$, the sequential complexity of the expression. That is, the work cost is asymptotically realizable on a single processor machine. For the general case the theorem tells us that we can never evaluate e in fewer steps than its depth cost, since this is the critical path length, and, for computations with shallow depth, we can achieve the best-possible result of dividing up the work evenly among the p processors.

Theorem 46.5 suggests a characterization of those problems for which having a great degree of parallelism (more processing elements) improves the running time. For a computation of depth d and work w, we can make good use of parallelism whenever $w/p > d$, which occurs when the *parallelizability ratio*, w/d, is at least p. For a highly sequential program, the work is proportional to the depth, and so the parallelizability is constant, which means that increasing p does not speed up the computation. On the other hand, a highly parallelizable computation is one with constant depth, or depth d proportional to $\lg w$. Such programs have a high parallelizability ratio, and hence are amenable to speedup by increasing the number of available processors. It is worth stressing that *it is not known* whether all problems admit a parallelizable solution or not. The best we can say, on present knowledge, is that there are algorithms for some problems that have a high degree of parallelizability, and there are problems for which no such algorithm is known. It is an important open problem in complexity theory to characterize which problems are parallelizable, and which are not. As a stop-gap measure, if one is faced with a problem for which no parallelizable solution is known, it may make sense to exploit available parallelism that would otherwise be wasted by employing speculative methods in addition to the work-efficient methods discussed here.

The proof of Theorem 46.5 amounts to a design for the implementation of a parallel language. A critical ingredient is scheduling the workload onto the p processors so as to maximize their utilization. This is achieved by maintaining a shared worklist of tasks that have been created by evaluation of a parallel `let` construct, all of which must be completed to determine the final outcome of the computation. (Here we make use of shared memory so that all processors have access to the central worklist.) Execution is divided into rounds. At the end of each round a processor may complete execution, in which case further work can be scheduled onto it; it may continue execution into the next round; or it may fork two additional tasks to be scheduled for execution, blocking until they complete. To start the next round the processors must collectively assign work to themselves so that if sufficient work is available, then all p processors will be assigned work. Assume that we have at least p units of work remaining to be done at any given time (otherwise just consider all remaining work in what follows). Each step of execution on each processor consists of executing an instruction of our computing model. After this step a task may either be complete, or may continue with further execution, or may fork two new tasks as a result of executing a parallel `let` instruction, or it may join two completed tasks into one. The synchronization required for a join may be done in constant time on an SMP using standard methods. The main difficulty is to re-schedule tasks to processors after each round. This may be achieved in constant time by computing partial sums across the processors to determine the assignment of tasks to processors on the next round. Tasks that complete contribute zero, continuing tasks contribute one, and forking tasks contribute two to the sum. The partial sums then determine the assignment of tasks to processors for the next round of execution: processor i executes the task at index determined by the ith partial sum just calculated.

Theorem 46.5 on the preceding page assumes a constant-time fetch-and-add instruction for synchronization of p processors. In practice this assumption is not always realistic, but in most cases we may implement it from more basic primitives in $O(\lg p)$ time. In that case Theorem 46.5 on the previous page must be weakened to an upper bound of $O(w/p, d \lg p)$ time. Accounting for the execution time of fetch-and-add imposes a factor of $\lg p$ overhead on the work and depth of the computation. This factor appears in the weakened bound on the depth, but it is interesting that it can be hidden for highly parallelizable computations (*i.e.*, those for which $w/d \gg p \lg p$). This is achieved by assigning $\lg p$ units of sequential work to each processor on each scheduling round, and observing that $(w \lg p)/(p \lg p) = w/p$, indicating that we may, in this case, achieve the same execution bound even when the cost of fetch-and-add is considered.

46.4 Vector Parallelism

So far we have confined attention to binary fork/join parallelism induced by the parallel `let` construct. While technically sufficient for many purposes, a more natural programming model admit an unbounded number of parallel tasks to be spawned simultaneously, rather than forcing them to be created by a cascade of binary forks and corresponding joins. Such a model, often called *data parallelism*, ties the source of parallelism to a data structure of unbounded size. The principal example of such a data structure is a *vector* of values of a specified type. The primitive operations on vectors provide a natural source of unbounded parallelism. For example, one may consider a parallel map construct that applies a given function to every element of a vector simultaneously, forming a vector of the results. Similarly, for an associative binary operation with a unit element, one may consider a parallel reduce primitive that computes the result of performing the binary operation across all elements of the vector, starting with the unit element. Intuitively, this can be computed in logarithmic time in parallel by successively splitting the vector in half, and using the operation to combine intermediate results. This operation is itself easily derived from a scan primitive that computes all partial results, from left to right, of applying the binary operation to successive elements of the vector, starting with the unit element. (The result of the reduce operation is simply the last element of the vector obtained by the corresponding scan primitive.)

We will consider here a very simple language of vector operations to illustrate the main ideas. The upshot of this is that we may recapitulate the main results of this chapter in the general case of vector parallelism, including a version of Theorem 46.5 on page 365 for vector operations that tells us how to implement the language on an SMP. Our purpose here is not to give a detailed development, but to concentrate instead on the main features of parallel vector computation.

The following grammar specifies the syntax for a minimal language of vector computations:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{vec}(\tau)$	$\tau\,\mathtt{vec}$
Expr	e	$::=$	$\mathtt{vec}(e_0,\ldots,e_{n-1})$	$[e_0,\ldots,e_{n-1}]$
		$\mid$	$\mathtt{sub}(e_1;e_2)$	$e_1\,[e_2]$
		$\mid$	$\mathtt{siz}(e)$	$\mathtt{size}\,(e)$
		$\mid$	$\mathtt{idx}(e)$	$\mathtt{index}(e)$
		$\mid$	$\mathtt{map}(e_1;x\,.\,e_2)$	$\texttt{<}e_2 \mid x \in e_1\texttt{>}$

The expression $\mathtt{vec(}e_0\mathtt{,}\ldots\mathtt{,}e_{n-1}\mathtt{)}$ evaluates to an n-vector whose elements are given by the expressions $e_0, \ldots, e_{n-1}$. The operation $\mathtt{sub(}e_1\mathtt{;}e_2\mathtt{)}$ retrieves the element of the vector given by e_1 at the index given by e_2. The operation $\mathtt{siz(}e\mathtt{)}$ returns the number of elements in the vector given by e. The operation $\mathtt{idx(}e\mathtt{)}$ creates a vector of length n (given by e) whose elements are $0, \ldots, n-1$. The operation $\mathtt{map(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)}$ computes the vector whose ith element is the result of evaluating e_2 with x bound to the ith element of the vector given by e_1.

The static semantics of these operations is given by the following typing rules:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \mathtt{vec(}e_0\mathtt{,}\ldots\mathtt{,}e_{n-1}\mathtt{)} : \mathtt{vec(}\tau\mathtt{)}} \tag{46.21a}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{vec(}\tau\mathtt{)} \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{sub(}e_1\mathtt{;}e_2\mathtt{)} : \tau} \tag{46.21b}$$

$$\frac{\Gamma \vdash e : \mathtt{vec(}\tau\mathtt{)}}{\Gamma \vdash \mathtt{siz(}e\mathtt{)} : \mathtt{nat}} \tag{46.21c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{idx(}e\mathtt{)} : \mathtt{vec(nat)}} \tag{46.21d}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{vec(}\tau\mathtt{)} \quad \Gamma, x : \tau \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{map(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} : \mathtt{vec(}\tau'\mathtt{)}} \tag{46.21e}$$

We will not bother to give the sequential and parallel semantics of these primitives, but will instead simply give a cost semantics for them. It is a good exercise to formulate a sequential and parallel transition semantics and to relate these to the cost semantics in the manner of Theorem 46.3 on page 359. The cost semantics of these primitives is given by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\mathtt{vec(}e_0\mathtt{,}\ldots\mathtt{,}e_{n-1}\mathtt{)} \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \mathtt{vec(}v_0\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{)}} \tag{46.22a}$$

$$\frac{e_1 \Downarrow^{c_1} \mathtt{vec(}v_0\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{)} \quad e_2 \Downarrow^{c_2} \mathtt{num[}i\mathtt{]} \quad (0 \leq i < n)}{\mathtt{sub(}e_1\mathtt{;}e_2\mathtt{)} \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{46.22b}$$

$$\frac{e \Downarrow^{c} \mathtt{vec(}v_0\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{)}}{\mathtt{siz(}e\mathtt{)} \Downarrow^{c \oplus \mathbf{1}} \mathtt{num[}n\mathtt{]}} \tag{46.22c}$$

$$\frac{e \Downarrow^{c} \mathtt{num[}n\mathtt{]}}{\mathtt{idx(}e\mathtt{)} \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} \mathbf{1}} \mathtt{vec(}0\mathtt{,}\ldots\mathtt{,}n-1\mathtt{)}} \tag{46.22d}$$

$$\frac{\begin{array}{c} e_1 \Downarrow^{c_1} \mathtt{vec(}v_0\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{)} \\ {[v_0/x]e_2 \Downarrow^{c_0} v_0'} \quad \ldots \quad {[v_{n-1}/x]e_2 \Downarrow^{c_{n-1}} v_{n-1}'} \end{array}}{\mathtt{map(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \Downarrow^{c_1 \oplus (c_0 \otimes \ldots \otimes c_{n-1})} \mathtt{vec(}v_0'\mathtt{,}\ldots\mathtt{,}v_{n-1}'\mathtt{)}} \tag{46.22e}$$

The cost semantics for vectors may be validated against the sequential and parallel dynamics in a manner similar to Theorem 46.4 on page 362 so that the work cost and depth cost are, respectively, the sequential and parallel execution times measured as steps in the transition systems defining the dynamics of the language. We may also validate the cost semantics in terms of its implementation by an extension of Theorem 46.5 on page 365 to handle vectors.

To get an idea of what is involved in this extension, let us consider how to implement the operation `idx(`e`)` on a p-processor SMP. We wish to show, consistently with Theorem 46.5 on page 365, that this operation may be implemented in time $O(\max(n/p, 1))$, where e evaluates to `num[`n`]`. This may be achieved as follows. First, reserve, in constant time, an uninitialized region of n words of memory for the vector to be created by this operation. To initialize this memory, we assign responsibility for a segment of size n/p to each of the p processors, which then execute in parallel to fill in the required values. To do this we must assign to processor i the starting point, n_i, of the ith segment, which it will then initialize to $n_i, n_i + 1, \ldots, (n_i + n)/(p - 1)$. This is achieved by constructing, in constant time using fetch-and-add, the vector consisting of the values $0, \ldots, p - 1$, and then multiplying each element by the number n/p to obtain the vector $n_0, \ldots, n_{p-1}$ of starting points. Each processor then initializes its segment, requiring $O(n/p)$ steps each, executed in parallel, which achieves the required bound.

This example illustrates an important point of methodology. The cost semantics of the vector primitives is chosen so that, when combined Theorem 46.5 on page 365 the predicted asymptotic bound is actually achievable in practice. If we were unable to find an algorithm that achieves this bound, then the cost semantics of the operation would have to be adjusted to reflect the reality. Alternatively, given that we have an appropriate algorithm, any implementation that fails to achieve the predicted bound may be considered faulty, and must be corrected to bring it in line with the cost semantics.

46.5 Exercises

Part XVI

Concurrency

Chapter 47

Process Calculus

So far we have mainly studied the static and dynamic semantics of programs in isolation, without regard to their interaction with the world. But to extend this analysis to even the most rudimentary forms of input and output requires that we consider external agents that interact with the program. After all, the whole purpose of a computer is to interact with a person!

To extend our investigations to interactive systems, we begin with the study of *process calculi*, which are abstract formalisms that capture the essence of interaction among independent agents. There are many forms of process calculi, differing in technical details and in emphasis. We will consider the best-known formalism, which is called the *π-calculus*. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to the synchronous and asynchronous variants of the π-calculus itself.

Our presentation of the π-calculus differs from that in the literature in several respects. Most significantly, we maintain a distinction between *processes* and *events*. The basic form of process is one that awaits a choice of events. Other forms of process include parallel composition, the introduction of a communication channel, and, in the asychronous case, a send on a channel. The basic form of event is the ability to read (and, in the synchronous case, write) on a channel. Events are combined by a non-deterministic choice operator. Even the choice operator can be eliminated in favor of a protocol for treating a parallel composition of events as a non-deterministic choice among them.

47.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an *event*, which specifies the set of *actions* that a process is prepared to undertake in concert with another process. Two processes interact by undertaking two complementary actions, which may be thought of as a *read* and a *write* on a common channel. The processes synchronize on these complementary actions, after which they may proceed independently to interact with other processes.

To begin with we will focus on sequential processes, which simply await the arrival of one of several possible actions, known as an event.

Category	*Item*		*Abstract*	*Concrete*
Process	P	$::=$	$\mathtt{await}(E)$	$\$\,E$
Event	E	$::=$	$\mathtt{null}$	$\mathbf{0}$
		$\mid$	$\mathtt{choice}(E_1;E_2)$	$E_1 + E_2$
		$\mid$	$\mathtt{rcv}[a](P)$	$\mathtt{?}a\,\mathtt{.}\,P$
		$\mid$	$\mathtt{snd}[a](P)$	$\mathtt{!}a\,\mathtt{.}\,P$
Action	α	$::=$	$\mathtt{rcv}(a)$	$\mathtt{?}a$
		$\mid$	$\mathtt{snd}(a)$	$\mathtt{!}a$
		$\mid$	$\mathtt{sil}$	ϵ

The variables a, b, and c range over *channels*, which serve as conduits for synchronization. The *receive action*, $\mathtt{?}a$, and the *send action*, $\mathtt{!}a$, are *complementary*. The *silent action*, ϵ, is included as a technical convenience to serve as a label for the silent transition (as described in Chapter 4).

We will handle events modulo *structural congruence*, written $P_1 \equiv P_2$ and $E_1 \equiv E_2$, respectively, which is the strongest equivalence relation closed under the following rules:

$$\frac{E \equiv E'}{\$\,E \equiv \$\,E'} \tag{47.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{47.1b}$$

$$\frac{P \equiv P'}{\mathtt{?}a\,\mathtt{.}\,P \equiv \mathtt{?}a\,\mathtt{.}\,P'} \tag{47.1c}$$

$$\frac{P \equiv P'}{\mathtt{!}a\,\mathtt{.}\,P \equiv \mathtt{!}a\,\mathtt{.}\,P'} \tag{47.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{47.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{47.1f}$$

$$\frac{}{E_1 \mathtt{+} \mathtt{(} E_2 \mathtt{+} E_3 \mathtt{)} \equiv \mathtt{(} E_1 \mathtt{+} E_2 \mathtt{)} \mathtt{+} E_3} \tag{47.1g}$$

The importance of imposing structural congruence on sequential processes is that it enables us to think of an event as having the form of a finite sum of send or receive events, with the sum of zero events being the null event, **0**.

An illustrative example of Robin Milner's is a simple vending machine that may take in a 2p coin, then optionally either permit selection of a cup of tea, or take another 2p coin, then permit selection of a cup of coffee.

$$V = \mathtt{\$(?2p.\$(!tea.}V\mathtt{+?2p.\$(!cof.}V\mathtt{)))}$$

As the example indicates, we tacitly permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs.

Because we have suppressed the internal computation occurring within a process, sequential processes have no dynamic semantics on their own—their dynamics arises only as a result of interaction with another process. For the vending machine to operate there must be another process (you!) who initiates the events expected by the machine, causing both your state (the coins in your pocket) and its state (as just described) to change as a result.

47.2 Concurrent Interaction

We enrich the language of processes with concurrent composition.

Category	*Item*		*Abstract*	*Concrete*
Process	P	$::=$	$\mathtt{await(}E\mathtt{)}$	$\mathtt{\$}\,E$
		$\mid$	$\mathtt{stop}$	$\mathbf{1}$
		$\mid$	$\mathtt{par(}P_1\mathtt{;}P_2\mathtt{)}$	$P_1 \parallel P_2$

The process **1** represents the inert process, and the process $P_1 \parallel P_2$ represents the concurrent composition of P_1 and P_2. One may identify **1** with $\mathtt{\$}\,\mathbf{0}$, the process that awaits the event that will never occur, but we prefer to treat the inert process as a primitive concept.

Structural congruence for processes is enriched by the following rules governing the inert process and concurrent composition of processes:

$$\frac{}{P \parallel \mathbf{1} \equiv P} \tag{47.2a}$$

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{47.2b}$$

$$\frac{}{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{47.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{47.2d}$$

Up to structural equivalence every process has the form

$$\$\, E_1 \parallel \ldots \parallel \$\, E_n$$

for some $n \geq 0$, it being understood that when $n = 0$ this is the process **1**.

The dynamic semantics of concurrent interaction is defined by an action-indexed family of transition judgements, $P \overset{\alpha}{\longmapsto} P'$, where the silent action indexes the silent transition $P \longmapsto P'$. The action label on a transition specifies the effect of an execution step on the environment in which it occurs. Here the effect is to "announce" the action of sending or receiving on a specified channel. Two concurrent processes may interact by announcing complementary actions, resulting in a silent transition.

$$\frac{P_1 \equiv P_2 \quad P_2 \overset{\alpha}{\longmapsto} P_2' \quad P_2' \equiv P_2}{P_1 \overset{\alpha}{\longmapsto} P_2} \tag{47.3a}$$

$$\frac{}{\$\,(\,!\,a\,.\,P + E\,) \overset{!\,a}{\longmapsto} P} \tag{47.3b}$$

$$\frac{}{\$\,(\,?\,a\,.\,P + E\,) \overset{?\,a}{\longmapsto} P} \tag{47.3c}$$

$$\frac{P_1 \parallel P_2 \overset{\alpha}{\longmapsto} P_1' \parallel P_2}{P_1 \overset{\alpha}{\longmapsto} P_1'} \tag{47.3d}$$

$$\frac{P_1 \overset{!\,a}{\longmapsto} P_1' \quad P_2 \overset{?\,a}{\longmapsto} P_2'}{P_1 \parallel P_2 \longmapsto P_1' \parallel P_2'} \tag{47.3e}$$

Rules (47.3b) and (47.3c) specify that any of the events on which a process is synchronizing may occur. Rule (47.3e) synchronizes two processes that take complementary actions.

As an example, let us consider the interaction of the vending machine, V, with the user process, U, defined as follows:

$$U = \$\,!\mathtt{2p}.\$\,!\mathtt{2p}.\$\,?\mathtt{cof}.\mathbf{1}.$$

Here is a trace of the interaction between V and U:

$$\begin{aligned} V \parallel U &\mapsto \$\,\mathtt{!tea}.V + \mathtt{?2p}.\$\,\mathtt{!cof}.V \parallel \$\,\mathtt{!2p}.\$\,\mathtt{?cof}.\mathbf{1} \\ &\mapsto \$\,\mathtt{!cof}.V \parallel \$\,\mathtt{?cof}.\mathbf{1} \\ &\mapsto V \end{aligned}$$

These steps are justified, respectively, by the following pairs of labelled transitions:

$$\begin{aligned} U &\xmapsto{\mathtt{!2p}} U' = \$\,\mathtt{!2p}.\$\,\mathtt{?cof}.\mathbf{1} \\ V &\xmapsto{\mathtt{?2p}} V' = \$\,(\mathtt{!tea}.V + \mathtt{?2p}.\$\,\mathtt{!cof}.V) \end{aligned}$$

$$\begin{aligned} U' &\xmapsto{\mathtt{!2p}} U'' = \$\,\mathtt{?cof}.\mathbf{1} \\ V' &\xmapsto{\mathtt{?2p}} V'' = \$\,\mathtt{!cof}.V \end{aligned}$$

$$\begin{aligned} U'' &\xmapsto{\mathtt{?cof}} \mathbf{1} \\ V'' &\xmapsto{\mathtt{!cof}} V \end{aligned}$$

We have suppressed uses of structural congruence in the above derivations to avoid clutter, but it is important to see its role in managing the non-deterministic choice of events by a process.

47.3 Replication

Some presentations of process calculus forego reliance on defining equations for processes in favor of a *replication* construct, which we write $*\,P$. This process stands for as many concurrently executing copies of P as one may require, which may be modeled by the structural congruence

$$*\,P \equiv P \parallel *\,P.$$

Taking this as a principle of structural congruence hides the overhead of process creation, and gives no hint as to how often it can or should be applied. One could alternatively build replication into the dynamic semantics to model the details of replication more closely:

$$*\,P \mapsto P \parallel *\,P.$$

Since the application of this rule is unconstrained, it may be applied at any time to effect a new copy of the replicated process P.

So far we have been using recursive process definitions to define processes that interact repeatedly according to some protocol. Rather than take recursive definition as a primitive notion, we may instead use replication to model repetition. This may be achieved by introducing an "activator" process that is contacted to effect the replication. Consider the recursive definition $X = P(X)$, where P is a process expression involving occurrences of the process variable, X, to refer to itself. This may be simulated by defining the activator process

$$A = \mathtt{*\$\,(?}a\,\mathtt{.}\,P(\mathtt{\$\,(!}a\,\mathtt{.}\mathbf{1}\mathtt{)})\mathtt{)},$$

in which we have replaced occurrences of X within P by an initiator process that signals the event a to the activator. Observe that the activator, A, is structurally congruent to the process $A' \parallel A$, where A' is the process

$$\mathtt{\$\,(?}a\,\mathtt{.}\,P(\mathtt{\$\,(!}a\,\mathtt{.}\mathbf{1}\mathtt{)})\mathtt{)}.$$

To start process P we concurrently compose the activator, A, with an initiator process, $\mathtt{\$\,(!}a\,\mathtt{.}\mathbf{1}\mathtt{)}$. Observe that

$$A \parallel \mathtt{\$\,(!}a\,\mathtt{.}\mathbf{1}\mathtt{)} \mapsto A \parallel P(\mathtt{!}a\,\mathtt{.}\mathbf{1}),$$

which starts the process P while maintaining a running copy of the activator, A.

As an example, let us consider Milner's vending machine written using replication, rather than using recursive process definition:

$$V_1 = \mathtt{*\$\,(?}v\,\mathtt{.}\,V_2\mathtt{)} \tag{47.4}$$

$$V_2 = \mathtt{\$\,(?2p.\$\,(!tea.}V_0\,\mathtt{+}\,\mathtt{?2p.\$\,(!cof.}V_0\mathtt{)))} \tag{47.5}$$

$$V_0 = \mathtt{\$\,(!}v\,\mathtt{.}\mathbf{1}\mathtt{)} \tag{47.6}$$

The process V_1 is a replicated server that awaits a signal on channel v to create another instance of the vending machine. The recursive calls are replaced by signals along v to re-start the machine. The original machine, V, is simulated by the concurrent composition $V_0 \parallel V_1$.

47.4 Private Channels

It is often desirable to isolate interactions among a group of concurrent processes from those among another group of processes. This can be achieved by creating a private channel that is shared among those in the group, and which is inaccessible from all other processes. This may be modeled by enriching the language of processes with a construct for creating a new channel:

Category	*Item*		*Abstract*	*Concrete*
Process	P	$::=$	$\texttt{new}(a.P)$	$\nu(a.P)$

As the syntax suggests, this is a binding operator in which the channel a is bound within P.

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{47.7a}$$

$$\frac{P \equiv P'}{\nu(a.P) \equiv \nu(a.P')} \tag{47.7b}$$

$$\frac{a \mathrel{\#} P_2}{\nu(a.P_1) \parallel P_2 \equiv \nu(a.P_1 \parallel P_2)} \tag{47.7c}$$

The last rule, called *scope extrusion*, will be important for the treatment of communication in the next section.

The dynamic semantics is extended with one additional rule permitting steps to take place within the scope of a binder.

$$\frac{P \overset{\alpha}{\mapsto} P' \quad a \mathrel{\#} \alpha}{\nu(a.P) \overset{\alpha}{\mapsto} \nu(a.P')} \tag{47.8}$$

No process may interact with $\nu(a.P)$ along the newly-allocated channel, for to do so would require knowledge of the private channel, a, which is chosen, by the magic of α-equivalence, to be distinct from all other channels in the system.

As an example, let us consider again the non-recursive definition of the vending machine. The channel, v, used to initialize the machine should be considered private to the machine itself, and not be made available to a user process. This is naturally expressed by the process expression $\nu(v.V_0 \parallel V_1)$, where V_0 and V_1 are as defined above using the designated channel, v. This process correctly simulates the original machine, V, because it precludes interaction with a user process on channel V. If U is a user process, the interaction begins as follows:

$$\nu(v.V_0 \parallel V_1) \parallel U \mapsto \nu(v.V_2) \parallel U \equiv \nu(v.V_2 \parallel U)$$

The interaction continues as before, albeit within the scope of the binder, provided that v has been chosen (by structural congruence) to be apart from U, ensuring that it is private to the internal workings of the machine.

47.5 Synchronous Communication

The concurrent process calculus presented in the preceding section models synchronization based on the willingness of two processes to undertake complementary actions. A natural extension of this model is to permit data to be passed from one process to another as part of synchronization. Since we are abstracting away from the computation occurring within a process, it would not make much sense to consider, say, passing an integer during synchronization. A more interesting possibility is to permit passing *channels*, so that new patterns of connectivity can be established as a consequence of inter-process synchronization. This is the core idea of the π-calculus.

The syntax of events is changed to account for communication by generalizing send and receive events as specified in the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Event	E	$::=$	$\texttt{rcv}[a](x.P)$	$\texttt{?}a\texttt{(}x\texttt{).}P$
		$\mid$	$\texttt{snd}[a;b](P)$	$\texttt{!}a\texttt{(}b\texttt{).}P$

The event $\texttt{?}a\texttt{(}x\texttt{).}P$ binds the variable x within the process expression P. The rest of the syntax remains as described earlier in this chapter.

The syntax of actions is generalized along similar lines, with both the send and receive actions specifying the data communicated by the action.

Category	*Item*		*Abstract*	*Concrete*
Action	α	$::=$	$\texttt{rcv}[a](b)$	$\texttt{?}a\texttt{(}b\texttt{)}$
		$\mid$	$\texttt{snd}[a](b)$	$\texttt{!}a\texttt{(}b\texttt{)}$

The action $\texttt{!}a\texttt{(}b\texttt{)}$ represents a write, or send, of a channel, b, along a channel, a. The action $\texttt{?}a\texttt{(}b\texttt{)}$ represents a read, or receive, along channel, a, of another channel, b.

Interaction in the π-calculus consists of synchronization on the concurrent availability of complementary actions on a channel, passing a channel from the sender to the receiver.

$$\frac{}{\texttt{\$}\,\texttt{(}\texttt{!}a\texttt{(}b\texttt{).}P + E\texttt{)} \xmapsto{\texttt{!}a\texttt{(}b\texttt{)}} P} \tag{47.9a}$$

$$\frac{}{\texttt{\$}\,\texttt{(}\texttt{?}a\texttt{(}x\texttt{).}P + E\texttt{)} \xmapsto{\texttt{?}a\texttt{(}b\texttt{)}} [b/x]P} \tag{47.9b}$$

$$\frac{P_1 \xmapsto{\texttt{!}a\texttt{(}b\texttt{)}} P_1' \quad P_2 \xmapsto{\texttt{?}a\texttt{(}b\texttt{)}} P_2'}{P_1 \parallel P_2 \mapsto P_1' \parallel P_2'} \tag{47.9c}$$

In contrast to pure synchronization the message-passing form of interaction is fundamentally asymmetric — the receiver continues with the channel passed by the sender substituted for the bound variable of the action. Rule (47.9b) may be seen as "guessing" that the received data will be b, which is substituted into the resulting process.

47.6 Polyadic Communication

So far communication is limited to sending and receiving a single channel along another channel. It is often useful to consider more flexible forms of communcation in which zero or more channels are communicated by a single interaction. Transmitting no data corresponds to a pure *signal* on a channel in which the mere fact of the communication is all that is transmitted between the sender and the receiver. Transmitting more than one channel corresponds to a *packet* in which a single interaction communicates a finite number of channels from sender to receiver.

The *polyadic* π-calculus is the generalization of the π-calculus to admit communication of multiple channels between sender and receiver in a single interaction. The syntax of the polyadic π-calculus is a simple extension of the monadic π-calculus in which send and receive events, and their corresponding actions, are generalized as follows:

Category	*Item*		*Abstract*	*Concrete*
Event	E	$::=$	$\mathtt{rcv[}a\mathtt{](}x_1\mathtt{,}\ldots\mathtt{,}x_k\mathtt{.}P\mathtt{)}$	$\mathtt{?}a\mathtt{(}x_1\mathtt{,}\ldots\mathtt{,}x_k\mathtt{)}\mathtt{.}P$
		$\mid$	$\mathtt{snd[}a\mathtt{;}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{](}P\mathtt{)}$	$\mathtt{!}a\mathtt{(}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{)}\mathtt{.}P$
Action	α	$::=$	$\mathtt{rcv[}a\mathtt{](}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{)}$	$\mathtt{?}a\mathtt{(}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{)}$
		$\mid$	$\mathtt{snd[}a\mathtt{](}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{)}$	$\mathtt{!}a\mathtt{(}b_1\mathtt{,}\ldots\mathtt{,}b_k\mathtt{)}$

The index k ranges over natural numbers. When k is zero, the events model pure signals, and when $k > 1$, the events model communication of packets along a channel. There arises the possibility of sending more or fewer values along a channel than are expected by the receiver. To remedy this one may associate with each channel a unique *arity* $k \geq 0$, which represents the size of any packet that it may carry. The syntax of the polyadic π-calculus should then be restricted to respect the arity of the channel. We leave the specification of this refinement as an exercise for the reader.

The rules for structural congruence and interaction generalize in the evident manner to the polyadic case.

47.7 Mutable Cells as Processes

Let us consider a reference cell server that, when given an initial value, creates a cell that listens on two dedicated channels, one to get the current value of the cell, the other to set it to a new designated value. This may be defined using recursion equations as follows:

$$C(x,g,s) = \mathtt{\$}\,(S(g,s) + G(x,g,s)) \tag{47.10}$$

$$S(g,s) = \mathtt{?}s(y)\,\mathtt{.}\,C(y,g,s) \tag{47.11}$$

$$G(x,g,s) = \mathtt{!}g(x)\,\mathtt{.}\,C(x,g,s) \tag{47.12}$$

The cell is parameterized by its current value and two channels on which to contact it to get and set its value. Each message causes a new cell to be created, reflecting any update to its value.

To avoid the recursion implicit in the equations we may instead define a server that creates fresh cells whenever contacted on a specified channel, c, specifying an initial value, x, for that cell and two channels, g and s, on which to contact it to get and set its value.

$$R(c) = *\,\mathtt{\$}\,(\mathtt{?}c(x,g,s)\,\mathtt{.}\,C'(c,x,g,s)) \tag{47.13}$$

$$C'(c,x,g,s) = \mathtt{\$}\,(S'(c,g,s) + G'(c,x,g,s)) \tag{47.14}$$

$$S'(c,g,s) = \mathtt{?}s(y)\,\mathtt{.}\,\mathtt{\$}\,(\mathtt{!}c(y,g,s)\,\mathtt{.}\,\mathbf{1}) \tag{47.15}$$

$$G'(c,x,g,s) = \mathtt{!}g(x)\,\mathtt{.}\,\mathtt{\$}\,(\mathtt{!}c(x,g,s)\,\mathtt{.}\,\mathbf{1}) \tag{47.16}$$

The reference cell server repeatedly awaits receipt of a creation message on channel r, and creates a new cell with the specified initial value and channels on which to contact it. The cell awaits contact, then behaves appropriately, but this time contacting the server to create a new cell with updated value after each message.

To use reference cells in a process P, we concurrently compose P with an instance, $R(c)$, of the cell server, which is contacted via channel c. For example, the process

$$\nu(c\,\mathtt{.}\,R(c) \parallel \nu(g\,\mathtt{.}\,\nu(s\,\mathtt{.}\,\mathtt{\$}\,\mathtt{!}c(0,g,s)\,\mathtt{.}\,\mathtt{\$}\,\mathtt{!}s(1)\,\mathtt{.}\,\mathtt{\$}\,\mathtt{?}g(x)\,\mathtt{.}\ldots)))$$

allocates a channel for communication with the reference cell server, then allocates two channels for a new cell, initializes it to `0`, sets it to `1`, then retrieves its value, and so forth.

This example illustrates the importance of scope extrusion in the π-calculus. Initially, the process $R(c)$ is run concurrently with a process that allocates two new channels, g and s, and then sends these channels, along with the initial value, 0, along c. Tracing out the steps, this results in a process offering to send along g and to receive along s, which represents the new reference cell, running concurrently with the subsequent process that manipulates this newly allocated cell. For this to make sense, the scope of g and s must be enlarged to encompass the body of $R(c)$ after receipt of 0, g, and s along c. Structural congruence ensures that we may "lift" the allocation of g and s to encompass $R(c)$, since g and s may be chosen, by α-equivalence, to be distinct from any channels already occurring in $R(c)$. This enables communication of the cell server with the cell client along the channels g and s.

47.8 Asynchronous Communication

This form of interaction is called *synchronous*, because both the sender and the receiver are blocked from further interaction until synchronization has occurred. On the receiving side this is inevitable, because the receiver cannot continue execution until the channel which it receives has been determined, much as the body of a function cannot be executed until its argument has been provided. On the sending side, however, there is no fundamental reason why notification is required; the sender could simply send the message along a channel without specifying how to continue once that message has been received. This "fire and forget" semantics is called *asynchronous* communication, in constrast to the *synchronous* form just described.

The *asynchronous π-calculus* is obtained by removing the synchronous send *event*, $\texttt{!}a\texttt{(}b\texttt{)}.P$, and adding a new form of process, the asynchronous send *process*, written $\texttt{!}a\texttt{(}b\texttt{)}$, which has no continuation after the send. The syntax of the asynchronous π-calculus is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Process	P	::=	$\texttt{snd[}a\texttt{](}b\texttt{)}$	$\texttt{!}a\texttt{(}b\texttt{)}$
		|	$\texttt{await(}E\texttt{)}$	$\texttt{\$}E$
		|	$\texttt{par(}P_1\texttt{;}P_2\texttt{)}$	$P_1 \parallel P_2$
		|	$\texttt{new(}a.P\texttt{)}$	$\nu\texttt{(}a.P\texttt{)}$
Event	E	::=	$\texttt{null}$	$\mathbf{0}$
		|	$\texttt{rcv[}a\texttt{](}x.P\texttt{)}$	$\texttt{?}a\texttt{(}x\texttt{)}.P$
		|	$\texttt{choice(}E_1\texttt{;}E_2\texttt{)}$	$E_1 + E_2$

Up to structural congruence, an event is just a choice of zero or more reads along any number of channels.

The dynamic semantics for the asynchronous π-calculus is defined by omitting Rule (47.9a), and adding the following rule for the asynchronous send process:

$$\frac{}{\mathtt{!}\,a\,(b) \xmapsto{\mathtt{!}\,a\,(b)} \mathbf{1}} \tag{47.17}$$

One may regard the pending asynchronous write as a kind of buffer in which the message is held until a receiver is chosen.

In a sense the synchronous π-calculus is more fundamental than the asynchronous variant, because we may always mimic the asynchronous send by a process of the form $\$\;\mathtt{!}\,a\,(b)\,.\,\mathbf{1}$, which performs the send, and then becomes the inert process $\mathbf{1}$. In another sense, however, the asynchronous π-calculus is more fundamental, because we may encode a synchronous send by introducing a notification channel on which the receiver sends a message to notify the sender of the successful receipt of its message. This exposes the implicit communication required to implement synchronous send, and avoids it in cases where it is not needed (in particular, when the resumed process is just the inert process, as just illustrated).

To get an idea of what is involved in the encoding of the synchronous π-calculus in the asynchronous π-calculus, we sketch the implementation of an acknowledgement protocol that only requires (polyadic) asynchronous communication. A synchronous process of the form

$$\nu\,(a\,.\,\$\,((\mathtt{!}\,a\,(b)\,.\,P) + E) \parallel \$\,((\mathtt{?}\,a\,(x)\,.\,Q) + F))$$

is represented by the asynchronous process

$$\nu\,(a\,.\,\nu\,(a_0\,.\,P' \parallel Q')),$$

where $a_0 \# P$, $a_0 \# Q$, and we define

$$P' = \mathtt{!}\,a\,(b) \parallel \$\,(\mathtt{?}\,a_0\,()\,.\,P + E)$$

and

$$Q' = \$\,(\mathtt{?}\,a\,(x,x_0)\,.\,(\mathtt{!}\,a_0\,() \parallel Q) + F).$$

The process that is awaiting the outcome of a send event along channel a instead sends the argument, b, along with a newly allocated acknowledgement channel, a_0, along the channel a, then awaits receipt of a signal in the form of a null message along a_0, then acts as the process P. Correspondingly, the process that is awaiting a receive event along channel a must be prepared to receive, in addition, the acknowledgement channel, x_0, on which it sends an asychronous signal back to the sender, and proceeds to act as the process Q. It is easy to check that the synchronous interaction of the original process is simulated by several steps of execution of the translation into asynchronous form.

47.9 Exercises

Chapter 48

Concurrency

In this chapter we utilize the machinery of process calculus presented in Chapter 47 to derive a uniform treatment of several seemingly disparate concepts: mutable storage, speculative parallelism, input/output, process creation, and inter-process communication. The unifying theme is to use the methods of process calculus to give an account of *context-sensitive* execution. For example, inter-process communication necessarily involves the execution of two processes, each in a context that includes the other. The two processes synchronize, and continue execution separately after their rendezvous.

48.1 Framework

The language $\mathcal{L}\{\texttt{conc}\}$ is an extension of $\mathcal{L}\{\texttt{comm}\}$ (described in Chapter 39) with an additional level of *processes*, which represent concurrently executing agents. The syntax of $\mathcal{L}\{\texttt{conc}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{comp}(\tau)$	$\tau\,\texttt{comp}$
Expr	e	$::=$	$\texttt{comp}(m)$	$\texttt{comp}(m)$
Comm	m	$::=$	$\texttt{return}(e)$	$\texttt{return}\,e$
		$\mid$	$\texttt{letcomp}(e;x.m)$	$\texttt{let}\,\texttt{comp}(x)\,\texttt{be}\,e\,\texttt{in}\,m$
Proc	p	$::=$	$\texttt{proc}[a](m)$	$\{m\}_a$
		$\mid$	$\texttt{par}(p_1;p_2)$	$p_1 \parallel p_2$
		$\mid$	$\texttt{new}(x.p)$	$\nu(x.p)$

The basic form of process is $\texttt{proc[}a\texttt{](}m\texttt{)}$, consisting of a single command, m, labelled with a symbol, a, that serves to identify it. We may also form the parallel composition of processes, and generate a new symbol for use within a process.

As always, we identify syntactic objects up to α-equivalence, so that bound names may always be chosen so as to satisfy any finitary contraint on their occurrence. As in Chapter 47, we also identify processes up to structural congruence, which specifies that parallel composition is commutative and associative, and that new symbol generation may have its scope expanded to encompass any parallel process, subject only to avoidance of capture.

In the succeeding sections of this chapter, the language $\mathcal{L}\{\texttt{conc}\}$ will be extended to model various forms of computational phenomena. In each case we will enrich the language with new forms of command, representing primitive capabilities of the language, and new forms of process, used to model the context in which commands are executed. In this respect it is misleading to think of processes as necessarily having to do with concurrent execution and synchronization! Rather, what processes provide is a simple, uniform means of describing the context in which a command is executed. This can include concurrent interaction (synchronization) in the familiar sense, but is not limited to this case.

The static semantics of $\mathcal{L}\{\texttt{conc}\}$ extends that of $\mathcal{L}\{\texttt{comm}\}$ (see Chapter 39) to include the additional level of processes. Let Σ range over finite sets of judgements of the form $a : \tau$, where a is a symbol and τ is a type, such that no symbol is the subject of more than one such judgement in Σ. We define the judgement $\Sigma\ \Gamma \vdash p\ \mathsf{ok}$ by the following rules:

$$\frac{}{\Sigma, a : \tau\,\texttt{pid}\,\Gamma \vdash m \sim \tau} \tag{48.1a}$$

$$\frac{\Sigma\ \Gamma \vdash p_1\ \mathsf{ok} \quad \Sigma \vdash p_2\ \mathsf{ok}}{\Sigma\ \Gamma \vdash p_1 \parallel p_2\ \mathsf{ok}} \tag{48.1b}$$

$$\frac{\Sigma, a : \tau\ \Gamma \vdash p\ \mathsf{ok}}{\Sigma\ \Gamma \vdash \nu(a\,.\,p)\ \mathsf{ok}} \tag{48.1c}$$

Rule (48.1a) specifies that a process of the form $\{m\}_a$ is well-formed if m is a command yielding a value of type τ under the assumption that a is a process identifier for a process of type τ. Rule (48.1b) states that a parallel composition of processes is well-formed if both processes are well-formed. Rule (48.1c) enriches Σ with a new symbol with a type τ chosen so that p is well-formed under this assumption.

Each extension of $\mathcal{L}\{\texttt{conc}\}$ considered below may introduce new forms of process governed by new rules of process formation.

The dynamic semantics of $\mathcal{L}\{\texttt{conc}\}$ is defined by judgements of the form $p \mapsto p'$, where p and p' are processes. Contextual execution of processes includes structural normalization, may apply to any active process, may occur within the scope of a newly introduced symbol, and respects structural congruence:

$$\frac{m \mapsto m'}{\texttt{proc[}a\texttt{](}m\texttt{)} \mapsto \texttt{proc[}a\texttt{](}m'\texttt{)}} \tag{48.2a}$$

$$\frac{p_1 \mapsto p_1'}{\texttt{par(}p_1\texttt{;}p_2\texttt{)} \mapsto p_1' \parallel p_2} \tag{48.2b}$$

$$\frac{p \mapsto p'}{\texttt{new(}a\,\texttt{.}\,p\texttt{)} \mapsto \texttt{new(}a\,\texttt{.}\,p'\texttt{)}} \tag{48.2c}$$

$$\frac{p \equiv q \quad q \mapsto q' \quad q' \equiv p'}{p \mapsto p'} \tag{48.2d}$$

Other rules of contextual execution are specific to the particular computational phenomenon under consideration, but in general all rules have the form

$$\frac{}{\{m_1\,;x\,.\,m_2\}_a \parallel p \mapsto \nu(a_1,\ldots,a_k\,.\,\{m_2'\}_a \parallel p')}\,. \tag{48.3}$$

That is, the execution of a command depends on the context of processes in which it is executed, and may result in the generation of new symbols, the creation of new processes, or the perpetuation of existing processes (in any combination).

48.2 Mutable Cells

We begin with the specification of mutable storage in $\mathcal{L}\{\texttt{conc}\}$. We augment the grammar of the language with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{ref(}\tau\texttt{)}$	$\tau\,\texttt{ref}$
Expr	e	::=	$\texttt{loc[}l\texttt{]}$	l
Comm	m	::=	$\texttt{new(}e\texttt{)}$	$\texttt{new(}e\texttt{)}$
		$\mid$	$\texttt{get(}e\texttt{)}$	$\texttt{!}\,e$
		$\mid$	$\texttt{set(}e_1\texttt{;}e_2\texttt{)}$	$e_1\texttt{:=}e_2$
Proc	p	::=	$\texttt{ref[}l\texttt{](}e\texttt{)}$	$\langle l : e\rangle$

The basic machinery of mutable cells is as described in Chapter 37. In addition we have a new form of process, written $\langle l : e\rangle$, which represents a mutable cell with location l and contents e. One may think of the cell as a "server" that interacts with a command process on channel l to retrieve and alter the contents of that cell.

The static semantics of reference cells is essentially as described in Chapter 39, transposed to the setting of $\mathcal{L}\{\texttt{conc}\}$. There is one new form of process, $\langle l : e\rangle$, whose well-formation is defined by the following rule:

$$\frac{\Sigma, l : \tau\,\texttt{ref}\;\Gamma \vdash e : \tau}{\Sigma, l : \tau\,\texttt{ref}\;\Gamma \vdash \langle l : e\rangle\ \mathsf{ok}} \tag{48.4}$$

The process $\langle l : e\rangle$ is well-formed if the assumed type of l is $\tau\,\texttt{ref}$, where e is of type τ under the full set of typing assumptions for locations (and other active symbols).

The dynamic semantics of mutable storage is specified in $\mathcal{L}\{\texttt{conc}\}$ by the following rules:

$$\frac{e\ \mathsf{val}}{\{\texttt{new}(e)\texttt{;}x\texttt{.}m\}_a \mapsto \nu(l\texttt{.}\{[l/x]m\}_a \parallel \langle l : e\rangle)} \tag{48.5a}$$

$$\frac{}{\{\texttt{!}\,l\texttt{;}x\texttt{.}m\}_a \parallel \langle l : e\rangle \mapsto \{[e/x]m\}_a \parallel \langle l : e\rangle} \tag{48.5b}$$

$$\frac{e\ \mathsf{val}}{\{l\texttt{:=}e\texttt{;}x\texttt{.}m\}_a \parallel \langle l : e'\rangle \mapsto \{[e/x]m\}_a \parallel \langle l : e\rangle} \tag{48.5c}$$

Rule (48.5a) gives the semantics of new, which allocates a new location, l, and creates a new process, a mutable cell with location l and contents e. The original process continues with l bound to x in m. Rule (48.5b) specifies the semantics of fetching the contents of a mutable cell by synchronization with the process representing that cell.

It is a simple matter to transpose the type safety proof given in Chapter 37 to the present setting. The static semantics of processes requires that we "guess" the type of each location such that the contents has that type under this very hypothesis. This is sufficient for preservation. For progress, it is necessary to establish some obvious inversion principles for typing. For example, if $\Sigma \vdash \langle l : e\rangle$ ok, then $\Sigma = \Sigma', l : \tau\,\texttt{ref}$ for some type τ such that $\Sigma \vdash e : \tau$. With this is hand it is straightforward to prove progress for $\mathcal{L}\{\texttt{conc}\}$ enriched with mutable storage.

48.3 Futures

The semantics of futures (speculatively executed suspensions) is readily expressed in $\mathcal{L}\{\mathtt{conc}\}$. Whenever a future is evaluated, we create a new process to evaluate the suspended expression, and continue execution, yielding the name of the newly created process as the value of the future. Whenever that future process is forced, we synchronize with it to ensure that it has achieved a value, and, if so, return that to the forcing process.

The syntax of futures in $\mathcal{L}\{\mathtt{comm}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{fut}(\tau)$	$\tau\,\mathtt{fut}$
Expr	e	$::=$	$\mathtt{pid}[a]$	a
Comm	m	$::=$	$\mathtt{fut}(e)$	$\uparrow e$
		$\mid$	$\mathtt{syn}(e)$	$\downarrow e$

The command $\mathtt{fut}(e)$ creates a suspended computation that is evaluated concurrently with the process that created it. The command $\mathtt{syn}(e)$ synchronizes with the process given as argument to retrieve its value, once its evaluation has completed. No new forms of process are required to express the ssemantics of futures.

The static semantics of futures specifies the evident typing rules for the commands $\uparrow e$ and $\downarrow e$:

$$\frac{\Sigma\;\Gamma \vdash e : \tau}{\Sigma\;\Gamma \vdash \uparrow e \sim \tau\,\mathtt{fut}} \tag{48.6a}$$

$$\frac{\Sigma\;\Gamma \vdash e : \tau\,\mathtt{fut}}{\Sigma\;\Gamma \vdash \downarrow e \sim \tau} \tag{48.6b}$$

$$\frac{\Sigma\;\Gamma \vdash l : \tau\,\mathtt{pid}}{\Sigma\;\Gamma \vdash l : \tau\,\mathtt{fut}} \tag{48.6c}$$

In addition the treatment of futures relies on Rule (48.1a), which specifies the formation of a basic process expression. A value of type $\tau\,\mathtt{fut}$ is a process identifier of type τ, since the dynamic semantics will specify that the future is represented by a concurrently executing process.

The dynamic semantics of futures is specified by the following rules:

$$\frac{}{\{\uparrow e\,;x\,.\,m\}_a \mapsto \nu(b\,.\,\{[b/x]m\}_a \parallel \{\mathtt{return}\,e\}_b)} \tag{48.7a}$$

$$\frac{e\ \mathsf{val}}{\{\downarrow b\,;x\,.\,m\}_a \parallel \{\mathtt{return}\,e\}_b \mapsto \{[e/x]m\}_a \parallel \{\mathtt{return}\,e\}_b} \tag{48.7b}$$

Rule (48.7a) specifies that a future is executed by creating a new process that returns the value of the suspended expression e, passing the identifier of the new process to the original for later synchronization. Rule (48.7b) specifies that to synchronize on a future requires that evaluation of the speculatively executed expression have finished before the synchronizing process may continue.

48.4 Input/Output

Character input and output are readily modeled in $\mathcal{L}\{\texttt{conc}\}$ by considering input and output ports to be mutable cells containing lists of characters.

Category	*Item*		*Abstract*	*Concrete*
Comm	m	::=	`getc()`	`getc()`
		|	`putc(`e`)`	`putc(`e`)`

The static semantics assumes that we have a type `char` of characters:

$$\frac{}{\Sigma\;\Gamma \vdash \texttt{getc()} \sim \texttt{char}} \tag{48.8a}$$

$$\frac{\Sigma\;\Gamma \vdash e : \texttt{char}}{\Sigma\;\Gamma \vdash \texttt{putc(}e\texttt{)} \sim \texttt{char}} \tag{48.8b}$$

Assuming that there are two ports, `in` and `out`, the dynamic semantics of character input/output may be given by the following rules:

$$\frac{}{\{\texttt{getc()}\,\texttt{;}\,x\,\texttt{.}\,m\}_a \parallel \langle \texttt{in} : \texttt{cons(}c\texttt{;}cs\texttt{)}\rangle \mapsto \{[c/x]m\}_a \parallel \langle \texttt{in} : cs\rangle} \tag{48.9a}$$

$$\frac{}{\{\texttt{putc(}c\texttt{)}\,\texttt{;}\,x\,\texttt{.}\,m\}_a \parallel \langle \texttt{out} : cs\rangle \mapsto \{[c/x]m\}_a \parallel \langle \texttt{out} : \texttt{cons(}c\texttt{;}cs\texttt{)}\rangle} \tag{48.9b}$$

(Here we specify that `putc` returns the character that it sent to the output.)

It is a worthwhile exercise to extend these simple primitives to admit creating new output ports, opening input ports linked to some data source (such as a file), duplicating ports, closing ports, and so forth.

48.5 Concurrent Processes

Spawning a new process, and awaiting its completion, are similar to creating and synchronizing with a future. However, a process may execute an arbitrary command, not just evaluate an expression. The argument to spawn is a value of type $\tau\,\texttt{comp}$ for some τ; the newly created process executes the encapsulated command, returning its value.

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{pid}(\tau)$	$\tau\,\texttt{pid}$
Expr	e	$::=$	$\texttt{pid}[a]$	a
Comm	m	$::=$	$\texttt{spawn}(e)$	$\texttt{spawn}(e)$
		$\mid$	$\texttt{wait}(e)$	$\texttt{wait}(e)$

The static semantics is given by the following rules:

$$\frac{\Sigma\,\Gamma \vdash e : \tau\,\texttt{comp}}{\Sigma\,\Gamma \vdash \texttt{spawn}(e) \sim \tau\,\texttt{pid}} \tag{48.10a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \tau\,\texttt{pid}}{\Sigma\,\Gamma \vdash \texttt{wait}(e) \sim \tau} \tag{48.10b}$$

The dynamic semantics is given by the following rules:

$$\frac{}{\{\texttt{spawn(comp(}m_1\texttt{));}x\,.\,m_2\}_a \mapsto \nu(b\,.\,\{[b/x]m_2\}_a \parallel \{m_1\}_b)} \tag{48.11a}$$

$$\frac{e\ \mathsf{val}}{\{\texttt{wait}(b)\texttt{;}x\,.\,m\}_a \parallel \{\texttt{return}\,e\}_b \mapsto \{[e/x]m\}_a \parallel \{\texttt{return}\,e\}_b} \tag{48.11b}$$

Rule (48.11a) creates a new process, passing its identifier to the spawning process. Rule (48.11b) applies only once the process labeled b has completed, at which point its return value is passed to the waiting process.

48.6 Inter-Process Communication

(to be completed)

48.7 Excercises

Part XVII

Modularity

Chapter 49

Separate Compilation and Linking

49.1 Linking and Substitution

49.2 Exercises

Chapter 50

Basic Modules

Chapter 51

Parameterized Modules

Part XVIII

Equivalence

Chapter 52

Equational Reasoning for T

Equations are the heart and soul of mathematics. We derive equations such as

$$(x+1)^2 = x^2 + 2\,x + 1$$

to express the equivalence of two functions of the variable $x \in \mathbb{R}$. We solve equations such as

$$z^2 + 1 = 0$$

for $z \in \mathbb{C}$ for the complex number $i = \sqrt{-1}$. In elementary geometry congruence and similarity are forms of equality between geometric objects.

The beauty of functional programming is that equality of expressions in a functional language corresponds very closely to familiar patterns of mathematical reasoning. For example, in the language $\mathcal{L}\{\texttt{nat}\rightarrow\}$ of Chapter 15 in which we can express addition as the function plus, the expressions

$$\lambda(x\texttt{:nat.}\,\lambda(y\texttt{:nat.plus(}e_1\texttt{)(}e_2\texttt{)))}$$

and

$$\lambda(x\texttt{:nat.}\,\lambda(y\texttt{:nat.plus(}e_2\texttt{)(}e_1\texttt{)))}$$

are equal, regardless of what e_1 and e_2, so long as they are of type nat. In other words, the addition function *as programmed in* $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is commutative.

This may seem to be obviously true, but *why*, precisely, is it so? More importantly, what do we even *mean* when we say that two expressions of a programming language are equal? In this chapter we will develop answers to these questions for the language $\mathcal{L}\{\texttt{nat}\rightarrow\}$ introduced in Chapter 15. The development is based on a lazy dynamic semantics, but one could equally well consider an eager semantics. Since $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is a *totally* *pure* language (all expressions terminate), there is no significant distinction between the two variants in the sense that an equation is valid for one semantics iff it is valid for the other.

52.1 Observational Equivalence

When are two expressions equal? Whenever we cannot tell them apart! This may seem tautological, but it is not, because it depends on what we consider to be a means of telling expressions apart. What "experiment" are we permitted to perform on expressions in order to distinguish them? What counts as an observation that, if different for two expressions, is a sure sign that they are different?

If we permit ourselves to consider the syntactic details of the expressions, then very few expressions could be considered equal. For example, if it is deemed significant that an expression contains, say, more than one function application, or that it has an occurrence of λ-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation, and not to the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightarrow\}$ is given only for expressions without free variables, so a complete program should clearly be a *closed* expression. Second, the outcome of a computation should be *observable*, so that it is evident whether the outcome of two computations differs or not. We define a *complete program* to be a closed expression of type `nat`, and define the *observable behavior* of the program to be the outermost form of its value, either `z` or $\texttt{s}(-)$.[1]

An *experiment* on, or *observation* about, an expression is any means of using that expression within a complete program. We define an *expression context* to be an expression with a "hole" in it serving as a placeholder for another expression. The hole is permitted to occur anywhere, including within the scope of a binder. The bound variables within whose scope the hole lies are said to be *exposed (to capture)* by the expression context. These variables may be assumed, without loss of generality, to be distinct from

[1] This notion of behavior is chosen so as to be compatible with both the eager and lazy semantics.

one another. A *program context* is a closed expression context of type `nat`—that is, it is a complete program with a hole in it. The meta-variable $\mathcal{C}$ stands for any expression context.

Replacement is the process of filling a hole in an expression context, $\mathcal{C}$, with an expression, e, which is written $\mathcal{C}\{e\}$. Importantly, the free variables of e that are exposed by $\mathcal{C}$ are *captured* by replacement (which is why replacement is not a form of substitution, which is defined so as to avoid capture). If $\mathcal{C}$ is a program context, then $\mathcal{C}\{e\}$ is a complete program iff all free variables of e are captured by the replacement. For example, if $\mathcal{C} = \lambda(x\texttt{:nat.}\,\circ)$, and $e = x\texttt{+}x$, then

$$\mathcal{C}\{e\} = \lambda(x\texttt{:nat.}\,x\texttt{+}x).$$

The free occurrences of x in e are captured by the λ-abstraction as a result of the replacement of the hole in $\mathcal{C}$ by e.

We sometimes write $\mathcal{C}\{\circ\}$ to emphasize the occurrence of the hole in $\mathcal{C}$. Expression contexts are closed under *composition* in that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are expression contexts, then so is

$$\mathcal{C}\{\circ\} := \mathcal{C}_1\{\mathcal{C}_2\{\circ\}\},$$

and we have $\mathcal{C}\{e\} = \mathcal{C}_1\{\mathcal{C}_2\{e\}\}$. The *trivial*, or *identity*, expression context is the "bare hole", written $\circ$, for which $\circ\{e\} = e$.

The static semantics of expressions of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is extended to expression contexts by defining the typing judgement

$$\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$$

so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$. This judgement may be inductively defined by a collection of rules derived from the static semantics of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ (for which see Rules (15.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \tau)} \tag{52.1a}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})}{\texttt{s}(\mathcal{C}) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})} \tag{52.1b}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \texttt{nat}, y : \tau' \vdash e_1 : \tau'}{\texttt{rec}\,\mathcal{C}\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\ \texttt{with}\,y\Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{52.1c}$$

$$\frac{\Gamma' \vdash e : \texttt{nat} \quad \mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \quad \Gamma', x : \texttt{nat}, y : \tau' \vdash e_1 : \tau'}{\texttt{rec}\,e\,\{\texttt{z}\Rightarrow \mathcal{C}_0 \mid \texttt{s}(x)\ \texttt{with}\,y\Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{52.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \triangleright \tau')}{\mathtt{rec}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \Rightarrow \mathcal{C}_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{52.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \tau_1 \triangleright \tau_2)}{\lambda(x{:}\tau_1.\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1 \to \tau_2)} \tag{52.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{52.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{52.1h}$$

Lemma 52.1. *If $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, then $\Gamma' \subseteq \Gamma$, and if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$.*

Observe that the trivial context consisting only of a "hole" acts as the identity under replacement. Moreover, contexts are closed under composition in the following sense.

Lemma 52.2. *If $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, and $\mathcal{C}' : (\Gamma' \triangleright \tau') \rightsquigarrow (\Gamma'' \triangleright \tau'')$, then $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma'' \triangleright \tau'')$.*

Kleene equivalence determines when two experiments have the same observable outcome. If e and e' are complete programs, then e is *Kleene equivalent* to e', written $e \simeq e'$, provided that $e \mapsto^* \mathtt{z}$ iff $e' \mapsto^* \mathtt{z}$. It follows from Lemma 15.2 on page 116 that $e \simeq e'$ iff $e \mapsto^* \mathtt{s}(e_1)$ (for some e_1) iff $e' \mapsto^* \mathtt{s}(e_1')$ (for some e_1'). This relation is easily seen to be reflexive, symmetric, and transitive. Kleene equivalence is quite coarse in that, for example, all non-zero natural numbers are considered Kleene equivalent! This is not a concern, however, because we are only interested in Kleene equivalence as an auxiliary notion in the definition of observational equivalence. What is important, however, is that Kleene equivalence does not relate zero to any successor, which ensures consistency of observational equivalence.

Definition 52.1. *Suppose that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are two expressions of the same type. We say that e and e' are* observationally equivalent, *written $e \cong e' : \tau\ [\Gamma]$, iff $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ for every program context $\mathcal{C}$.*

In other words, for all possible experiments, the outcome of an experiment on e is the same as the outcome on e'. This is obviously an equivalence relation.

A type-indexed family of equivalence relations $e_1 \equiv e_2 : \tau\ [\Gamma]$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } e \equiv e' : \tau\ [\Gamma], \text{ then } \mathcal{C}\{e\} \equiv \mathcal{C}\{e'\} : \tau'\ [\Gamma']$$

for every expression context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is *consistent* iff $e \equiv e' : \texttt{nat}\ [\Gamma]$ implies $e \simeq e'$. Observe that the converse does not hold. For example, s(z) and s(s(z)) are Kleene equivalent, but not observationally equivalent, because there is a context that distinguishes them, namely the predecessor function, which sends the first to z but the second to s(z).

Theorem 52.3. *Observational equivalence is the coarsest consistent congruence on expressions.*

Proof. Consistency follows directly from the definition by noting that the trivial context is a program context. Observational equivalence is obviously an equivalence relation. To show that it is a congruence, we need only observe that type-correct composition of a program contex with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $e \ncong e' : \tau\ [\Gamma]$, then there is a program context $\mathcal{C}$ such that $\mathcal{C}\{e\} \not\simeq \mathcal{C}\{e'\}$, so that extending observational equivalence with this pair would be inconsistent. □

Theorem 52.3 licenses the principle of *proof by coinduction* to show that two expressions are observational equivalence: to show that $e \cong e' : \tau\ [\Gamma]$, it is enough to exhibit a consistent congruence such that $e \equiv e' : \tau\ [\Gamma]$. It can be difficult, however, to construct such a relation. In the next section we will provide a general method for doing so that will prove useful in many situations.

52.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of expressions of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that merely manipulate expressions without actually inspecting them. For example, we may pass an expression of type τ to a function that merely returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate expressions at arm's length, affording no opportunities to distinguish one from another.

This leads to the definition of *typed*, or *logical*, *equivalence*.

Definition 52.2. Logical equivalence *is a type-indexed family of relations $e \sim e' : \tau$ between closed expressions of type τ. It is defined by induction on the structure of τ as follows:*

$$\begin{array}{lll} e \sim e' : \mathtt{nat} & \textit{iff} & e \mapsto^* \mathtt{z} \textit{ and } e' \mapsto^* \mathtt{z} \textit{ or} \\ & & e \mapsto^* \mathtt{s(}e_1\mathtt{)} \textit{ and } e' \mapsto^* \mathtt{s(}e_1'\mathtt{)}, \textit{ and } e_1 \sim e_1' : \mathtt{nat}. \\ \\ e \sim e' : \tau_1 \to \tau_2 & \textit{iff} & \textit{if } e_1 \sim e_1' : \tau_1, \textit{ then } e\mathtt{(}e_1\mathtt{)} \sim e'\mathtt{(}e_1'\mathtt{)} : \tau_2 \end{array}$$

Logical equivalence at type `nat` is inductively defined to be the *strongest* relation satisfying the specified conditions.

Logical equivalence is extended to open terms by substitution. An *expression assignment*, γ, for a set of assumptions, Γ, is a finite partial function that assigns to each variable x such that $\Gamma \vdash x : \tau$ an expression $e = \gamma(x)$ such that $\Gamma \vdash e : \tau$, and that is undefined on all other variables. If γ and γ' are two expression assignments for assumptions Γ, we define $\gamma \sim \gamma' : \Gamma$ to hold iff $\gamma(x) \sim \gamma'(x) : \Gamma(x)$ for every variable, x, such that $\Gamma \vdash x : \tau$. Finally, we define $e \sim e' : \tau\ [\Gamma]$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$ whenever $\gamma \sim \gamma' : \Gamma$.

52.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

Lemma 52.4 (Substitution and Functionality). *If $e \cong e' : \tau\ [\Gamma, x : \sigma]$, and $d : \sigma$, then $[d/x]e \cong [d/x]e' : \tau\ [\Gamma]$. Furthermore, if $d \cong d' : \sigma$, then $[d/x]e \cong [d'/x]e' : \tau\ [\Gamma]$.*

Proof. Suppose that $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\triangleright \mathtt{nat})$ is a program context. We are to show that $\mathcal{C}\{[d/x]e\} \simeq \mathcal{C}\{[d/x]e'\}$. Since d and d' are closed, and since $\mathcal{C}$ is a program context, this is equivalent to showing that $[d/x]\mathcal{C}\{e\} \simeq [d/x]\mathcal{C}\{e'\}$. Let $\mathcal{D}$ be the context $\mathtt{(}\lambda\mathtt{(}x\mathtt{:}\sigma\mathtt{.}\,\mathcal{C}\{\circ\}\mathtt{))(}d\mathtt{)}$, and note that $\mathcal{D} : (\Gamma, x : \sigma \triangleright \tau) \rightsquigarrow (\triangleright \mathtt{nat})$. It follows from the assumption that $\mathcal{D}\{e\} \simeq \mathcal{D}\{e'\}$. But by construction $\mathcal{D}\{e\} \simeq [d/x]\mathcal{C}\{e\}$, and $\mathcal{D}\{e'\} \simeq [d/x]\mathcal{C}\{e'\}$. Let $\mathcal{D}'$ be the context $\mathtt{(}\lambda\mathtt{(}x\mathtt{:}\sigma\mathtt{.}\,\mathcal{C}\{\circ\}\mathtt{))(}d'\mathtt{)}$, and note that it, too, is a program context. Now if $d \cong d' : \sigma$, by congruence of observational equivalence, $\mathcal{D}\{e\} \cong \mathcal{D}'\{e\} : \mathtt{nat}$, and similarly $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\} : \mathtt{nat}$. By consistency of observational equivalence these are valid Kleene equivalences, from which the result follows. □

Lemma 52.5 (Closure Under Converse Evaluation). *Suppose that $e \sim e' : \tau$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows immediately from Definition 52.2 on the facing page. Suppose that $\tau = \tau_1 \rightarrow \tau_2$, $e \sim e' : \tau$, and $d \mapsto e$. To show that $d \sim e' : \tau$, we assume $e_1 \sim e_1' : \tau_1$ and show $d\mathtt{(}e_1\mathtt{)} \sim e'\mathtt{(}e_1'\mathtt{)} : \tau_2$. It follows from the assumption that $e\mathtt{(}e_1\mathtt{)} \sim e'\mathtt{(}e_1'\mathtt{)} : \tau_2$. Noting that $d\mathtt{(}e_1\mathtt{)} \mapsto e\mathtt{(}e_1\mathtt{)}$, the result follows by induction. □

Lemma 52.6 (Consistency). *If $e \sim e' : \mathtt{nat}$, then $e \simeq e'$.*

Proof. Immediate, from Definition 52.2 on the preceding page. □

Theorem 52.7 (Reflexivity). *If $\Gamma \vdash e : \tau$, then $e \sim e : \tau\ [\Gamma]$.*

Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim \gamma' : \Gamma$, then $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$. The proof proceeds by induction on typing derivations; we consider a few representative cases.

Consider the case of Rule (14.2b), in which $\tau = \tau_1 \rightharpoonup \tau_2$, $e = \lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e_2\mathtt{)}$ and $e' = \lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e_2'\mathtt{)}$. Since e and e' are values, we are to show that

$$\lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,\hat{\gamma}(e_2)\mathtt{)} \sim \lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,\widehat{\gamma'}(e_2')\mathtt{)} : \tau_1 \rightharpoonup \tau_2.$$

Assume that $e_1 \sim e_1' : \tau_1$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim [e_1'/x]\widehat{\gamma'}(e_2') : \tau_2$. Let $\gamma_2 = \gamma[x \mapsto e_1]$ and $\gamma_2' = \gamma'[x \mapsto e_1']$, and observe that $\gamma_2 \sim \gamma_2' : \Gamma, x : \tau_1$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim \hat{\gamma}_2'(e_2') : \tau_2$, from which the result follows directly.

Now consider the case of Rule (15.1d), for which we are to show that

$$\mathtt{rec[}\tau\mathtt{](}\hat{\gamma}(e)\mathtt{;}\hat{\gamma}(e_0)\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}(e_1)\mathtt{)} \sim \mathtt{rec[}\tau\mathtt{](}\hat{\gamma}'(e')\mathtt{;}\hat{\gamma}(e_0')\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}'(e_1')\mathtt{)} : \tau.$$

By the induction hypothesis applied to the first premise of Rule (15.1d), we have

$$\hat{\gamma}(e) \sim \hat{\gamma}'(e') : \mathtt{nat}.$$

Since logical equivalence at type nat is inductively defined, and bearing in mind Lemma 52.5, it suffices to show

$$\mathtt{rec[}\tau\mathtt{](z;}\hat{\gamma}(e_0)\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}(e_1)\mathtt{)} \sim \mathtt{rec[}\tau\mathtt{](z;}\hat{\gamma}'(e_0')\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}'(e_1')\mathtt{)} : \tau,$$

and, assuming

$$\mathtt{rec[}\tau\mathtt{](}d\mathtt{;}\hat{\gamma}(e_0)\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}(e_1)\mathtt{)} \sim \mathtt{rec[}\tau\mathtt{](}d'\mathtt{;}\hat{\gamma}'(e_0')\mathtt{;}x\mathtt{.}y\mathtt{.}\hat{\gamma}'(e_1')\mathtt{)} : \tau,$$

that

$$\texttt{rec}[\tau]\texttt{(s(}d\texttt{);}\hat{\gamma}(e_0)\texttt{;}x\texttt{.}y\texttt{.}\hat{\gamma}(e_1)\texttt{)} \sim \texttt{rec}[\tau]\texttt{(s(}d'\texttt{);}\hat{\gamma}'(e_0')\texttt{;}x\texttt{.}y\texttt{.}\hat{\gamma}'(e_1')\texttt{)} : \tau.$$

For the former case, it is enough to show by Lemma 52.5 on the previous page that $\hat{\gamma}(e_0) \sim \hat{\gamma}(e_0') : \tau$, which is assured by the outer inductive hypothesis applied to the third premise of Rule (15.1d). For the latter, define

$$\delta = \gamma[x \mapsto \hat{\gamma}(e)][y \mapsto \texttt{rec}[\tau]\texttt{(}\hat{\gamma}(e)\texttt{;}\hat{\gamma}(e_0)\texttt{;}x\texttt{.}y\texttt{.}\hat{\gamma}(e_1)\texttt{)}]$$

and

$$\delta' = \gamma'[x \mapsto \hat{\gamma}'(e')][y \mapsto \texttt{rec}[\tau]\texttt{(}\hat{\gamma}(e)\texttt{;}\hat{\gamma}(e_0)\texttt{;}x\texttt{.}y\texttt{.}\hat{\gamma}(e_1)\texttt{)}].$$

We then have that $\delta \sim \delta' : \Gamma, x : \texttt{nat}, y : \tau$, and hence the required follows from the outer inductive hypothesis applied to the third premise of Rule (15.1d). □

It follows that complete programs of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ terminate.

Corollary 52.8. *If e : $\texttt{nat}$ in $\mathcal{L}\{\texttt{nat}\rightarrow\}$, then there exists e' such that $e \mapsto^* e'$ and e' val.*

Proof. By Lemma 52.7 on the preceding page we have $e \sim e' : \texttt{nat}$, from which the result follows by Definition 52.2 on page 410. □

Symmetry and transitivity of logical equivalence are easily established by induction on types; logical equivalence is therefore an equivalence relation.

Lemma 52.9 (Congruence). *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $e \sim e' : \tau\ [\Gamma]$, then $\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0\ [\Gamma_0]$.*

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$. We consider a representative case in which $\mathcal{C}_0 = \lambda\texttt{(}x\texttt{:}\tau_1\texttt{.}\mathcal{C}_2\texttt{)}$ so that $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_1 \rightarrow \tau_2)$ and $\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $e \sim e' : \tau\ [\Gamma]$, we are to show that

$$\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_1 \rightarrow \tau_2\ [\Gamma_0],$$

which is to say

$$\lambda\texttt{(}x\texttt{:}\tau_1\texttt{.}\mathcal{C}_2\{e\}\texttt{)} \sim \lambda\texttt{(}x\texttt{:}\tau_1\texttt{.}\mathcal{C}_2\{e'\}\texttt{)} : \tau_1 \rightarrow \tau_2\ [\Gamma_0].$$

We know, by induction, that

$$\mathcal{C}_2\{e\} \sim \mathcal{C}_2\{e'\} : \tau_2\ [\Gamma_0, x : \tau_1].$$

Suppose that $\gamma_0 \sim \gamma_0' : \Gamma_0$, and that $e_1 \sim e_1' : \tau_1$. Let $\gamma_1 = \gamma_0[x \mapsto e_1]$, $\gamma_1' = \gamma_0'[x \mapsto e_1']$, and observe that $\gamma_1 \sim \gamma_1' : \Gamma_0, x : \tau_1$. By Definition 53.1 on page 418 it is enough to show that

$$\hat{\gamma}_1(\mathcal{C}_2\{e\}) \sim \hat{\gamma}_1'(\mathcal{C}_2\{e'\}) : \tau_2,$$

which follows immediately from the inductive hypothesis.

□

Theorem 52.10. *If $e \sim e' : \tau\ [\Gamma]$, then $e \cong e' : \tau\ [\Gamma]$.*

Proof. By Lemmas 52.6 on page 411 and 52.9 on the facing page, and Theorem 52.3 on page 409. □

Lemma 52.11. *If $e \cong e' : \tau$, then $e \sim e' : \tau$.*

Proof. We proceed by induction on the structure of τ. When $\tau = \texttt{nat}$, we appeal to Theorem 52.7 on page 411 and proceed by a simultaneous induction licensed by Definition 52.2 on page 410. If both $e \mapsto^* \texttt{z}$ and $e' \mapsto^* \texttt{z}$, then evidently $e \sim e' : \tau$. Otherwise, suppose that $e \mapsto^* \texttt{s}(d)$ and $e' \mapsto^* \texttt{s}(d')$ with $d \sim d : \texttt{nat}$ and $d' \sim d' : \texttt{nat}$. Since the predecessor operation is definable in $\mathcal{L}\{\texttt{nat}\rightarrow\}$, it follows from the assumption that $d \cong d' : \texttt{nat}$. Hence, by induction, we have $d \sim d' : \texttt{nat}$, from which the result follows by head expansion. The remaining cases, in which e evaluates to z and e' evalautes to $\texttt{s}(-)$, or *vice versa*, are ruled out by the assumption of observational equivalence of e and e'.

If $\tau = \tau_1 \rightarrow \tau_2$, then we are to show that whenever $e_1 \sim e_1' : \tau_1$, we have $e(e_1) \sim e'(e_1') : \tau_2$. By Theorem 52.10 we have $e_1 \cong e_1' : \tau_1$, and hence by congruence of observational equivalence it follows that $e(e_1) \cong e'(e_1') : \tau_2$, from which the result follows by induction. □

Theorem 52.12. *If $e \cong e' : \tau\ [\Gamma]$, then $e \sim e' : \tau\ [\Gamma]$.*

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$. Suppose that $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $n \geq 0$, and that $e_1 \sim e_1' : \tau_1$, ..., $e_n \sim e_n' : \tau_n$. By Theorem 52.10 we have that $e_1 \cong e_1' : \tau_1, \ldots, e_n \cong e_n' : \tau_n$, and hence by Lemma 52.4 on page 410 we have

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \cong [e_1', \ldots, e_n' / x_1, \ldots, x_n]e' : \tau.$$

Therefore by Lemma 52.11 we have

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \sim [e_1', \ldots, e_n' / x_1, \ldots, x_n]e' : \tau,$$

as required. □

Observational equivalence coincides with logical equivalence.

Corollary 52.13. *$e \cong e' : \tau\ [\Gamma]$ iff $e \sim e' : \tau\ [\Gamma]$.*

Even though the evaluation order is lazy, every closed expression of natural number type is observationally equivalent to a numeral.

Corollary 52.14. *If $e : \tau$, then $e \cong \overline{n} : \mathtt{nat}$ for some $n \geq 0$.*

Proof. By Theorem 52.7 on page 411 $e \sim e : \tau$. By a simple induction based on Definition 52.2 on page 410 we may show that $e \sim \overline{n} : \tau$ for some $n \geq 0$, and hence the result follows from Theorem 52.10 on the preceding page. □

52.4 Some Laws of Equivalence

In this section we summarize some useful principles of observational equivalence for $\mathcal{L}\{\mathtt{nat} \rightarrow\}$. For the most part these may be proved as laws of logical equivalence, and then transferred to observational equivalence by appeal to Corollary 52.13.

52.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{e \cong e : \tau\ [\Gamma]} \tag{52.2a}$$

$$\frac{e' \cong e : \tau\ [\Gamma]}{e \cong e' : \tau\ [\Gamma]} \tag{52.2b}$$

$$\frac{e \cong e' : \tau\ [\Gamma] \quad e' \cong e'' : \tau\ [\Gamma]}{e \cong e'' : \tau\ [\Gamma]} \tag{52.2c}$$

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{e \cong e' : \tau\ [\Gamma] \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'\ [\Gamma']} \tag{52.3a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad e_2 \cong e_2' : \tau' \; [\Gamma, x : \tau]}{[e/x]e_2 \cong [e/x]e_2' : \tau' \; [\Gamma]} \tag{52.4a}$$

$$\frac{e_1 \cong e_1' : \tau \; [\Gamma] \quad e_2 \cong e_2' : \tau' \; [\Gamma, x : \tau]}{[e_1/x]e_2 \cong [e_1'/x]e_2' : \tau' \; [\Gamma]} \tag{52.4b}$$

52.4.2 Symbolic Evaluation Laws

All of the instruction steps of an operational semantics are valid laws of equivalence. These are called *symbolic evaluation* laws, because they are extensions of the operational semantics to expressions with free variables that may occur anywhere within a program.

$$\frac{}{\texttt{rec}\,\texttt{z}\,\{\texttt{z}{\Rightarrow}e_0 \mid \texttt{s}(x)\,\texttt{with}\,y{\Rightarrow}e_1\} \cong e_0 : \tau \; [\Gamma]} \tag{52.5a}$$

$$\frac{e = \texttt{rec}\,\texttt{s}(e')\,\{\texttt{z}{\Rightarrow}e_0 \mid \texttt{s}(x)\,\texttt{with}\,y{\Rightarrow}e_1\}}{e \cong [e', e/x, y]e_1 : \tau \; [\Gamma]} \tag{52.5b}$$

$$\frac{}{(\lambda(x{:}\tau_1.\,e_2))(e_1) \cong [e_1/x]e_2 : \tau_2 \; [\Gamma]} \tag{52.5c}$$

52.4.3 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{e(x) \cong e'(x) : \tau_2 \; [\Gamma, x : \tau_1]}{e \cong e' : \tau_1 \to \tau_2 \; [\Gamma]} \tag{52.6}$$

Consequently, every expression of function type is equivalent to a λ-abstraction:

$$\frac{}{e \cong \lambda(x{:}\tau_1.\,e(x)) : \tau_1 \to \tau_2 \; [\Gamma]} \tag{52.7}$$

52.4.4 Induction Law

An equation involving a free variable, x, of type nat can be proved by induction on x.

$$\frac{[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \; [\Gamma] \; (\text{for every } n \in \mathbb{N})}{e \cong e' : \tau \; [\Gamma, x : \texttt{nat}]} \tag{52.8a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $[\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau\ [\Gamma]$, and
2. $[\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau\ [\Gamma]$, if $[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau\ [\Gamma]$.

52.5 Exercises

Chapter 53

Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, Plotkin's PCF, under an eager semantics. We proceed along similar lines to Chapter 52, but the development is a bit more involved in the presence of general recursion. The difficulty may be traced to the self-referential nature of general recursion—some care must be taken to avoid a circular argument. The proof relies on the compactness property of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ proved in Chapter 16 to establish the principle of *fixed point induction*, which is then used to prove reflexivity and congruence for logical equivalence.

53.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, remain essentially as in Chapter 52, but adapted to the language of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. In particular, the syntax of contexts is given by rules similar to Rules (52.1), modified to replace primitive recursion by case analysis and general recursion. The complete rules for typing contexts in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ are as follows:

$$\frac{}{\circ : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \tau)} \tag{53.1a}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})}{\texttt{s}(\mathcal{C}) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})} \tag{53.1b}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \texttt{nat} \vdash e_1 : \tau'}{\texttt{ifz}\,\mathcal{C}\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})} \tag{53.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \quad \Gamma', x : \mathtt{nat} \vdash e_1 : \tau'}{\mathtt{ifz}\, e\, \{\mathtt{z} \Rightarrow \mathcal{C}_0 \mid \mathtt{s}(x) \Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{53.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat} \triangleright \tau')}{\mathtt{ifz}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow \mathcal{C}_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \tau')} \tag{53.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \tau_1 \triangleright \tau_2)}{\lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,\mathcal{C}_2\mathtt{)} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1 \rightarrow \tau_2)} \tag{53.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2 \rightarrow \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1\mathtt{(}e_2\mathtt{)} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{53.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \rightarrow \tau' \quad \mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2)}{e_1\mathtt{(}\mathcal{C}_2\mathtt{)} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{53.1h}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', \mathtt{fix}\, x\mathtt{:}\tau'\, \mathtt{is}\, e : \tau' \triangleright \tau')}{\mathtt{fix}\, x\mathtt{:}\tau'\, \mathtt{is}\, \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{53.1i}$$

The class of *applicative contexts*, $\mathcal{A}$, for the eager variant of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined similarly by considering only Rule (53.1a) and Rule (53.1g), but restricted to applicative sub-contexts. Such contexts do not bind any variables, so that if $\mathcal{A} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, then $\Gamma = \Gamma'$.

53.2 Logical Equivalence For Eager PCF

The definition of logical equivalence for the eager semantics of PCF must take account of two characteristic features. First, any two computations that diverge are to be related, since they are observationally equivalent. Second, because the successor is evaluated eagerly, and because functions are called with the value of their argument, we must distinguish values from computations.

Definition 53.1. Logical equivalence *is a type-indexed family of relations $e \sim e' : \tau$ between closed expressions of type τ. It is defined simultaneously with the relation $e \approx e' : \tau$ of logical equivalence between closed values by induction on the structure of τ as follows:*

$$\begin{array}{lll}
e \sim e' : \tau & \textit{iff} & \textit{if } e \mapsto^* e_1 \ \mathsf{val} \textit{ then } e' \mapsto^* e_1' \ \mathsf{val} \textit{ and } e_1 \approx e_1' : \tau \textit{ and} \\
 & & \textit{if } e' \mapsto^* e_1' \ \mathsf{val} \textit{ then } e \mapsto^* e_1 \ \mathsf{val} \textit{ and } e_1 \approx e_1' : \tau \\
 & & \\
e \approx e' : \mathtt{nat} & \textit{iff} & e = e' = \mathtt{z}, \textit{ or} \\
 & & e = \mathtt{s(}e_1\mathtt{)} \textit{ and } e' = \mathtt{s(}e_1'\mathtt{)} \textit{ and } e_1 \approx e_1' : \mathtt{nat} \\
 & & \\
e \approx e' : \tau_1 \rightarrow \tau_2 & \textit{iff} & e = \lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e_2\mathtt{)},\ e' = \lambda\mathtt{(}x\mathtt{:}\tau_1\mathtt{.}\,e_2'\mathtt{)}, \textit{ and} \\
 & & e_1 \approx e_1' : \tau_1 \textit{ implies } [e_1/x]e_2 \sim [e_1'/x]e_2' : \tau_2
\end{array}$$

In addition, two closed applicative contexts are logically related, $\mathcal{A} \sim \mathcal{A}' : \tau$ iff their respective arguments are logically related at appropriate type. That is, we define $\circ \sim \circ : \tau$, and if $\mathcal{A} \sim \mathcal{A}' : \tau_2 \rightarrow \tau$ and $e_2 \sim e_2' : \tau_2$, then $\mathcal{A}(e_2) \sim \mathcal{A}'(e_2') : \tau$.

The conditions defining logical equivalence at type nat are self-referential, which raises the question of whether the relation is properly defined. To be more precise, we define $e \approx e' : \texttt{nat}$ to be the strongest binary relation R on closed values of type nat such that the following two conditions hold:

1. $\texttt{z}\, R\, \texttt{z}$, and

2. if $e\, R\, e'$, then $\texttt{s}(e)\, R\, \texttt{s}(e')$.

If R is the strongest such relation, it is easy to see that it satisfies the conditions given in Definition 53.1 on the facing page. In particular, observe that $e \approx e' : \texttt{nat}$ iff $e = e' = \overline{n}$ for some $n \geq 0$. This may be proved by showing that the diagonal relation on closed values of type nat satisfies the foregoing conditions.

Logical equivalence is extended to open terms by substitution. A *value assignment*, γ, for a finite set of variables, $\mathcal{X}$, is a finite function assigning to each variable $x \in \mathcal{X}$ a closed expression $e = \gamma(x)$ such that e val. A value assignment γ for $\mathcal{X} = \{ x_1, \ldots, x_n \}$ extends to a simultaneous substitution, $\hat{\gamma}$, on expressions by defining

$$\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n) / x_1, \ldots, x_n]e.$$

If γ and γ' are two value assignments for $\mathcal{X}$, and Γ is a finite set of typing hypotheses either of the form $x : \tau$, where $x \in \mathcal{X}$, or of the form $\texttt{fix}\, x{:}\tau\, \texttt{is}\, e : \tau$, then we define $\gamma \approx \gamma' : \Gamma$ to hold iff

1. if $x : \tau$ is in Γ, then $\gamma(x) \approx \gamma'(x) : \tau$;

2. if $\texttt{fix}\, x{:}\tau\, \texttt{is}\, e : \tau$ is in Γ, then $\texttt{fix}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e) \sim \texttt{fix}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e) : \tau$.

We define $e \sim e' : \tau\ [\Gamma]$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$ whenever $\gamma \approx \gamma' : \Gamma$.

53.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

Lemma 53.1 (Substitution and Functionality). *If $e \cong e' : \tau\ [\Gamma, x : \sigma]$, and $d : \sigma$, then $[d/x]e \cong [d/x]e' : \tau\ [\Gamma]$. Furthermore, if $d \cong d' : \sigma$, then $[d/x]e \cong [d'/x]e' : \tau\ [\Gamma]$.*

Lemma 53.2 (Closure Under Converse Evaluation)**.** *Suppose that $e \sim e' : \tau$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

Lemma 53.3 (Closure Under Application)**.** *$e \approx e' : \tau_1 \rightarrow \tau_2$ iff whenever $e_1 \approx e_1' : \tau_1$, $e\,(e_1) \sim e'\,(e_1') : \tau_2$.*

Proof. By assumption and Definition 53.1 on page 418 we know that $e = \lambda(x{:}\tau_1.\,e_2)$ and $e' = \lambda(x{:}\tau_1.\,e_2')$, and hence that $[e_1/x]e_2 \sim [e_1'/x]e_2' : \tau_2$. But since $e(e_1) \mapsto [e_1/x]e_2$ and $e'(e_1') \mapsto [e_1'/x]e_2'$, the result follows immediately from the definition of logical equivalence of expressions. The converse follows directly from determinacy of evaluation. □

Lemma 53.4 (Consistency)**.** *If $e \sim e' : \mathtt{nat}$, then $e \simeq e'$.*

Proof. Immediate, from Definition 53.1 on page 418. □

Theorem 53.5 (Fixed Point Induction)**.** *$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e \sim \mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e' : \tau$ if, for every $k \geq 0$, $\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e \sim \mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e' : \tau$.*

Proof. We prove the stronger result that whenever $\mathcal{A} \sim \mathcal{A}' : \tau$, if

$$\text{for every } k \geq 0,\ \mathcal{A}\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e\} \sim \mathcal{A}'\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e'\} : \tau,$$

then

$$\mathcal{A}\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e\} \sim \mathcal{A}'\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e'\} : \tau.$$

We proceed by induction on the structure of τ.

If $\tau = \mathtt{nat}$, then first suppose that $\mathcal{A}\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e\} \mapsto^* \overline{n}$. By Theorem 16.7 on page 131 there exists $k \geq 0$ such that $\mathcal{A}\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e\} \mapsto^* \overline{n}$. By the assumption $\mathcal{A}'\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e'\} \mapsto^* \overline{n}$, and hence $\mathcal{A}'\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e'\} \mapsto^* \overline{n}$ by simply erasing the bounds on recursion. The second, symmetric, case is handled analogously.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Lemma 53.3, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e\}(v_1) \sim \mathcal{A}'\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e'\}(v_1') : \tau_2$$

whenever $v_1 \approx v_1' : \tau_1$. Let $\mathcal{A}_2 = \mathcal{A}(v_1)$ and $\mathcal{A}_2' = \mathcal{A}'(v_1')$. Observe that for every $k \geq 0$ we have $\mathcal{A}_2\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e\} \sim \mathcal{A}_2'\{\mathtt{fix}^k\,x{:}\tau\,\mathtt{is}\,e'\} : \tau_2$, which follows directly from the outer assumption. Consequently,

$$\mathcal{A}_2\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e\} \sim \mathcal{A}_2'\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e'\} : \tau_2,$$

from which it follows that

$$\mathcal{A}\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e\} \sim \mathcal{A}'\{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e'\} : \tau_1 \rightarrow \tau_2,$$

as required. □

Lemma 53.6 (Reflexivity)**.** *If* $\Gamma \vdash e : \tau$*, then* $e \sim e : \tau\ [\Gamma]$*.*

Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \approx \gamma' : \Gamma$, then $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$. The proof proceeds by induction on typing derivations.

Consider the case of Rule (14.2b), in which $\tau = \tau_1 \rightharpoonup \tau_2$, $e = \lambda(x{:}\tau_1.\, e_2)$ and $e' = \lambda(x{:}\tau_1.\, e_2')$. Since $\hat{\gamma}(e)$ and $\widehat{\gamma'}(e')$ are values, it is enough to show that

$$\lambda(x{:}\tau_1.\, \hat{\gamma}(e_2)) \approx \lambda(x{:}\tau_1.\, \widehat{\gamma'}(e_2')) : \tau_1 \rightharpoonup \tau_2.$$

Assume that $e_1 \approx e_1' : \tau_1$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim [e_1'/x]\widehat{\gamma'}(e_2') : \tau_2$. Let $\gamma_2 = \gamma[x \mapsto e_1]$ and $\gamma_2' = \gamma'[x \mapsto e_1']$, and observe that $\gamma_2 \approx \gamma_2' : \Gamma, x : \tau_1$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim \widehat{\gamma_2'}(e_2') : \tau_2$, from which the result follows directly.

Consider the case of Rule (16.1g). Assuming $\gamma \approx \gamma' : \Gamma$, we are to show that

$$\texttt{fix}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e) \sim \texttt{fix}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e') : \tau.$$

By Theorem 53.5 on the preceding page it is enough to show that, for every $k \geq 0$,

$$\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e') \sim \tau : .\ [\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)]$$

We proceed by an inner induction on k. When $k = 0$ the result is immediate, since both sides of the desired equivalence diverge. Assuming the result for k, and applying Lemma 53.2 on the facing page, it is enough to show that $\hat{\gamma}(e_1) \sim \widehat{\gamma'}(e_1') : \tau$, where

$$e_1 = [\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{53.2}$$

$$e_1' = [\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e')/x]\widehat{\gamma'}(e'). \tag{53.3}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e) \sim \tau : ,\ [\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)]$$

then

$$[\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e) \sim \tau : .\ [[\texttt{fix}^k\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e)]$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows. □

Symmetry and transitivity of logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Logical equivalence is therefore an equivalence relation.

Lemma 53.7 (Congruence). *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $e \sim e' : \tau\ [\Gamma]$, then $\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0\ [\Gamma_0]$.*

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$.

Suppose that $\mathcal{C}_0 = \lambda(x{:}\tau_1.\,\mathcal{C}_2)$ so that $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_1 \to \tau_2)$ and $\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $e \sim e' : \tau\ [\Gamma]$, we are to show that

$$\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_1 \to \tau_2\ [\Gamma_0],$$

which is to say

$$\lambda(x{:}\tau_1.\,\mathcal{C}_2\{e\}) \sim \lambda(x{:}\tau_1.\,\mathcal{C}_2\{e'\}) : \tau_1 \to \tau_2\ [\Gamma_0].$$

We know, by induction, that

$$\mathcal{C}_2\{e\} \sim \mathcal{C}_2\{e'\} : \tau_2\ [\Gamma_0, x : \tau_1].$$

Suppose that $\gamma_0 \approx \gamma_0' : \Gamma_0$, and that $e_1 \approx e_1' : \tau_1$. Let $\gamma_1 = \gamma_0[x \mapsto e_1]$, $\gamma_1' = \gamma_0'[x \mapsto e_1']$, and observe that $\gamma_1 \approx \gamma_1' : \Gamma_0, x : \tau_1$. By Definition 53.1 on page 418 it is enough to show that

$$\hat{\gamma_1'}(\mathcal{C}_2\{e'\}) \sim \tau_2 : ,\ [\hat{\gamma}_1(\mathcal{C}_2\{e\})]$$

which follows immediately from the inductive hypothesis.

Suppose that $\mathcal{C}_0 = \texttt{fix}\,x{:}\tau\,\texttt{is}\,\mathcal{C}_1$, so that $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_0 \triangleright \tau_0)$ and $\mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)\tau_0$. Assuming $e \sim e' : \tau\ [\Gamma]$, we are to show that

$$\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0\ [\Gamma_0],$$

which is to say

$$\texttt{fix}\,x{:}\tau_0\,\texttt{is}\,\mathcal{C}_1\{e\} \sim \texttt{fix}\,x{:}\tau_0\,\texttt{is}\,\mathcal{C}_1\{e'\} : \tau_0\ [\Gamma_0].$$

It follows from the inductive assumption that

$$[\texttt{fix}\,x{:}\tau_0\,\texttt{is}\,\mathcal{C}_1\{e\}/x]\mathcal{C}_1\{e\} \sim [\texttt{fix}\,x{:}\tau_0\,\texttt{is}\,\mathcal{C}_1\{e'\}/x]\mathcal{C}_1\{e'\} : \tau_0\ [\Gamma_0],$$

which is enough for the desired result.

□

Theorem 53.8. *If $e \sim e' : \tau\ [\Gamma]$, then $e \cong e' : \tau\ [\Gamma]$.*

Proof. By Lemmas 53.4 on page 420 and 53.7, and the maximality of observational equivalence among consistent congruences. □

Lemma 53.9. *If $e \cong e' : \tau$, then $e \sim e' : \tau$.*

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result is immediate, since the trivial expression context is a program context. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \mapsto^* d$ val and $e' \mapsto^* d'$ val. Since d and d' are closed values of function type, $d = \lambda(x{:}\tau_1.\,e_2)$ and $d' = \lambda(x{:}\tau_1.\,e_2')$ for some e_2 and e_2'. We are to show that $d \approx d' : \tau_1 \to \tau_2$. So suppose further that $d_1 \approx d_1' : \tau_1$, and show that $[d_1/x]e_2 \sim [d_1'/x]e_2' : \tau_2$. By Theorem 53.8 on the facing page $d_1 \cong d_1' : \tau_1$, and hence by Lemma 53.1 on page 419 $[d_1/x]e_2 \cong [d_1'/x]e_2' : \tau_2$, from which the result follows by induction. □

Theorem 53.10. *If $e \cong e' : \tau\ [\Gamma]$, then $e \sim e' : \tau\ [\Gamma]$.*

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$. Suppose that $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $n \geq 0$, and that $e_1 \approx e_1' : \tau_1, \ldots, e_n \approx e_n' : \tau_n$. By Theorem 53.8 on the facing page we have that $e_1 \cong e_1' : \tau_1, \ldots, e_n \cong e_n' : \tau_n$, and hence by Lemma 53.1 on page 419 we have

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \cong [e_1', \ldots, e_n' / x_1, \ldots, x_n]e' : \tau.$$

Therefore by Lemma 53.9 we have

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \sim [e_1', \ldots, e_n' / x_1, \ldots, x_n]e' : \tau,$$

as required. □

Corollary 53.11. *$e \cong e' : \tau\ [\Gamma]$ iff $e \sim e' : \tau\ [\Gamma]$.*

53.4 Some Laws of Equivalence

To state the laws of equivalence for the eager variant of PCF reflects the distinction between values and computations in the definition of logical equivalence. In particular we must consider the extension of the judgement e val to open expressions e in which we hypothesize that variables stand for values. That is, we consider typing hypotheses Γ of the form

$$x_1 : \tau_1, x_1\ \mathsf{val}, \ldots, x_n : \tau_n, x_n\ \mathsf{val},$$

in which each variable is assumed to have a type and is assumed to be a value. We write $\Gamma \vdash e$ val to indicate that e is a value under the assumptions Γ. For example, the judgement

$$x : \mathtt{nat}, x\ \mathsf{val} \vdash \mathtt{s}(x)\ \mathsf{val}$$

is valid, since a successor is a value if its argument is a value, and, by hypothesis, the variable x is a value.

53.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{e \cong e : \tau\ [\Gamma]} \tag{53.4a}$$

$$\frac{e' \cong e : \tau\ [\Gamma]}{e \cong e' : \tau\ [\Gamma]} \tag{53.4b}$$

$$\frac{e \cong e' : \tau\ [\Gamma] \quad e' \cong e'' : \tau\ [\Gamma]}{e \cong e'' : \tau\ [\Gamma]} \tag{53.4c}$$

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{e \cong e' : \tau\ [\Gamma] \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'\ [\Gamma']} \tag{53.5a}$$

Equivalence is stable under substitution of values for free variables, and substituting equivalent values in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e\ \mathsf{val} \quad e_2 \cong e_2' : \tau'\ [\Gamma, x : \tau, x\ \mathsf{val}]}{[e/x]e_2 \cong [e/x]e_2' : \tau'\ [\Gamma]} \tag{53.6a}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{val} \quad \Gamma \vdash e_2\ \mathsf{val} \quad e_1 \cong e_1' : \tau\ [\Gamma] \quad e_2 \cong e_2' : \tau'\ [\Gamma, x : \tau, x\ \mathsf{val}]}{[e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'\ [\Gamma]} \tag{53.6b}$$

53.4.2 Symbolic Evaluation Laws

All of the instruction steps of an operational semantics are valid laws of equivalence. These are called *symbolic evaluation* laws, because they are extensions of the operational semantics to expressions with free variables that may occur anywhere within a program.

$$\frac{}{\mathtt{ifz}\,\mathtt{z}\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \cong e_0 : \tau\ [\Gamma]} \tag{53.7a}$$

$$\frac{\Gamma \vdash e'\ \mathsf{val}}{\mathtt{ifz}\,\mathtt{s}(e')\ \{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \cong [e'/x]e_1 : \tau\ [\Gamma]} \tag{53.7b}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{val}}{(\lambda(x{:}\tau_1.\,e_2))(e_1) \cong [e_1/x]e_2 : \tau_2\ [\Gamma]} \tag{53.7c}$$

$$\frac{}{\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e \cong [\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e/x]e : \tau\ [\Gamma]} \tag{53.7d}$$

53.4.3 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{e(x) \cong e'(x) : \tau_2 \; [\Gamma, x : \tau_1, x \; \mathsf{val}]}{e \cong e' : \tau_1 \to \tau_2 \; [\Gamma]} \tag{53.8}$$

Consequently, every value of function type is equivalent to a λ-abstraction:

$$\frac{\Gamma \vdash e \; \mathsf{val}}{e \cong \lambda(x{:}\tau_1.\, e(x)) : \tau_1 \to \tau_2 \; [\Gamma]} \tag{53.9}$$

53.4.4 Induction Law

An equation involving a free variable, x, of type `nat` can be proved by induction on x.

$$\frac{[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \; [\Gamma] \; (\text{for every } n \in \mathbb{N})}{e \cong e' : \tau \; [\Gamma, x : \mathtt{nat}, x \; \mathsf{val}]} \tag{53.10a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $[\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau \; [\Gamma]$, and
2. $[\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau \; [\Gamma]$, if $[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \; [\Gamma]$.

53.5 Exercises

Chapter 54

Parametricity

The 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 given in Chapter 25. Then if a program *does not* depend on the choice of types, we can code it using polymorphism. Moreover, if we wish to insist that a program *can not* depend on a choice of types, we demand that it be polymorphic. Thus polymorphism can be used both to expand the class of programs we may write, and also to limit the class of programs that are permissible in a given context.

The restrictions imposed by polymorphic typing give rise to the experience that in a polymorphic functional language, if the types are correct, then the program is correct. Roughly speaking, if a function has a polymorphic type, then the strictures of type genericity vastly cut down the set of programs with that type. Thus if you have written a program with this type, it is quite likely to be the one you intended!

The technical foundation for these remarks is called *parametricity*. The goal of this chapter is to give an account of parametricity for $\mathcal{L}\{\rightarrow\forall\}$ under a call-by-name interpretation.

54.1 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 $\forall(t\,.\,t \rightarrow t)$. What function could it be? When instantiated at a type τ it should evaluate to a function g of type $\tau \rightarrow \tau$ that, when further applied to a value v of type τ returns a value v' of type τ. Since f is polymorphic, g cannot depend on v, so v'

must be v. In other words, g must be the identity function at type τ, and f must therefore be the *polymorphic identity*.

Suppose that f is a function of type $\forall(t\,.\,t)$. What function could it be? A moment's thought reveals that it cannot exist at all! For it must, when instantiated at a type τ, return a value of that type. But not every type has a value (including this one), so this is an impossible assignment. The only conclusion is that $\forall(t\,.\,t)$ is an *empty* type.

Let N be the type of polymorphic Church numerals introduced in Chapter 25, namely $\forall(t\,.\,t \to (t \to t) \to t)$. What are the values of this type? Given any type τ, and values $z : \tau$ and $s : \tau \to \tau$, 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 one-to-one correspondence with the natural numbers.

54.2 Observational Equivalence

In this section we give a precise formulation of observational equivalence for $\mathcal{L}\{\to\forall\}$.

An expression context is, as in Chapter 52, an expression with a single occurrence of a "hole" that may be filled by an open expression. The typing judgement for expression contexts,

$$\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\Delta';\Gamma' \triangleright \tau'),$$

is defined as in Chapter 52 to mean that $\mathcal{C}$ exposes the variables Γ, and is such that $\Delta';\Gamma' \vdash \mathcal{C}\{e\} : \tau'$ whenever $\Delta;\Gamma \vdash e : \tau$.

We define a program to be a closed expression of type $\mathbf{2} = \forall(t\,.\,t \to t \to t)$, the Church booleans, with closed values $\mathbf{tt} = \Lambda(t\,.\,\lambda(x{:}t.\,\lambda(y{:}t.\,x)))$ and $\mathbf{ff} = \Lambda(t\,.\,\lambda(x{:}t.\,\lambda(y{:}t.\,y)))$. Kleene equivalence is defined for programs by $e \simeq e'$ iff (1) $e \mapsto^* \mathbf{tt}$ iff $e' \mapsto^* \mathbf{tt}$, and (2) $e \mapsto^* \mathbf{ff}$ iff $e' \mapsto^* \mathbf{ff}$. This is obviously an equivalence relation. We say that a type-indexed family of relations between closed expressions of the same type is *consistent* if it does not relate **tt** and **ff** at type **2**. Kleene equivalence is therefore evidently consistent.

Definition 54.1. *Two expressions of the same type are* observationally equivalent, *written* $e \cong e' : \tau\ [\Delta;\Gamma]$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\triangleright \mathbf{2})$.

Lemma 54.1. *Observational equivalence is the coarsest consistent congruence.*

Proof. The composition of a program context with another context is itself a program context. It is consistent by virtue of the empty context being a program context. □

Lemma 54.2 (Closed Substitution and Functionality)**.**

1. *If* $e \cong e' : \tau\ [\Delta, t;\Gamma]$ *and* ρ type, *then* $[\rho/t]e \cong [\rho/t]e' : [\rho/t]\tau\ [\Delta;[\rho/t]\Gamma]$.

2. *If* $e \cong e' : \tau\ [\emptyset;\Gamma, x : \sigma]$ *and* $d : \sigma$, *then* $[d/x]e \cong [d/x]e' : \tau\ [\emptyset;\Gamma]$. *Moreover, if* $d \cong d' : \sigma$, *then* $[d/x]e \cong [d'/x]e : \tau\ [\emptyset;\Gamma]$, *and similarly for* e'.

Proof. 1. Let $\mathcal{C} : (\Delta;[\rho/t]\Gamma \triangleright [\rho/t]\tau) \rightsquigarrow (\triangleright \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\rho/t]e\} \simeq \mathcal{C}\{[\rho/t]e'\}.$$

Since $\mathcal{C}$ and ρ are closed, this is equivalent to

$$[\rho/t]\mathcal{C}\{e\} \simeq [\rho/t]\mathcal{C}\{e'\}.$$

Let $\mathcal{C}'$ be the context $\Lambda(t\,.\,\mathcal{C}\{\circ\})\,[\rho]$, and observe that

$$\mathcal{C}' : (\Delta, t;\Gamma \triangleright \tau) \rightsquigarrow (\triangleright \mathbf{2}).$$

Therefore, from the assumption, it follows that

$$\mathcal{C}'\{e\} \simeq \mathcal{C}'\{e'\}.$$

But $\mathcal{C}'\{e\} \simeq [\rho/t]\mathcal{C}\{e\}$, and $\mathcal{C}'\{e'\} \simeq [\rho/t]\mathcal{C}\{e'\}$, from which the result follows.

2. By an argument essentially similar to that for Lemma 52.4 on page 410. □

54.3 Logical Equivalence

In this section we introduce a form of logical equivalence that captures the informal concept of parametricity, and also provides a characterization of observational equivalence. This will permit us to derive properties of observational equivalence of polymorphic programs of the kind suggested earlier.

The definition of logical equivalence for $\mathcal{L}\{\rightarrow\forall\}$ is somewhat more complex than for $\mathcal{L}\{\mathtt{nat}\rightarrow\}$. The main idea is to define logical equivalence for a polymorphic type, $\forall(t.\tau)$ to satisfy a very strong condition that captures the essence of parametricity. As a first approximation, we might say that two expressions, e and e', of this type should be logically equivalent if they are logically equivalent for "all possible" interpretations of the type t. More precisely, we might require that $e[\rho]$ be related to $e'[\rho]$ at type $[\rho/t]\tau$, for any choice of type ρ. But this runs into two problems, one technical, the other conceptual. The same device will be used to solve both problems.

The technical problem stems from impredicativity. In Chapter 52 logical equivalence is defined by induction on the structure of types. But when polymorphism is impredicative, the type $[\rho/t]\tau$ might well be larger than $\forall(t.\tau)$! At the very least we would have to justify the definition of logical equivalence on some other grounds, but no criterion appears to be available. The conceptual problem is that, even if we could make sense of the definition of logical equivalence, it would be too restrictive. For such a definition amounts to saying that the unknown type t is to be interpreted as logical equivalence at whatever type it turns out to be when instantiated. To obtain useful parametricity results, we shall ask for much more than this. What we shall do is to consider *separately* instances of e and e' by types ρ and ρ', and treat the type variable t as standing for *any relation* (of a suitable class) between ρ and ρ'. One may suspect that this is asking too much: perhaps logical equivalence is the *empty* relation! Surprisingly, this is not the case, and indeed it is this very feature of the definition that we shall exploit to derive parametricity results about the language.

To manage both of these problems we will consider a generalization of logical equivalence that is parameterized by a relational interpretation of the free type variables of its classifier. The parameters determine a separate binding for each free type variable in the classifier for each side of the equation, with the discrepancy being mediated by a specified relation between them. This permits us to consider a notion of "equivalence" between two expressions of different type—they are equivalent, *modulo* a relation between the interpretations of their free type variables.

We will restrict attention to a certain class of "admissible" binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

Definition 54.2 (Admissibility)**.** *A relation R between expressions of types ρ and ρ' is* admissible*, written $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:*

1. *Respect for observational equivalence: if $R(e, e')$ and $d \cong e : \rho$ and $d' \cong e' : \rho'$, then $R(d, d')$.*

2. *Closure under converse evaluation: if $R(e, e')$, then if $d \mapsto e$, then $R(d, e')$ and if $d' \mapsto e'$, then $R(e, d')$.*

The second of these conditions will turn out to be a consequence of the first, but we are not yet in a position to establish this fact.

The judgement $\delta : \Delta$ states that δ is a *type assignment* that assigns a closed type to each type variable $t \in \Delta$. A type assignment, δ, induces a substitution function, $\hat{\delta}$, on types given by the equation

$$\hat{\delta}(\tau) = [\delta(t_1), \ldots, \delta(t_n) / t_1, \ldots, t_n]\tau,$$

and similarly for expressions. Substitution is extended to contexts pointwise by defining $\hat{\delta}(\Gamma)(x) = \hat{\delta}(\Gamma(x))$ for each $x \in \mathit{dom}(\Gamma)$.

Let δ and δ' be two type assignments of closed types to the type variables in Δ. A *relation assignment*, η, between δ and δ' is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ for each $t \in \Delta$. The judgement $\eta : \delta \leftrightarrow \delta'$ states that η is a relation assignment between δ and δ'.

Logical equivalence is defined in terms of its generalization, called *parameterized logical equivalence*, written $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$, is defined as follows.

Definition 54.3 (Parameterized Logical Equivalence)**.** *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is defined by induction on the structure of τ by the following conditions:*

$$\begin{array}{lll}
e \sim e' : t\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e, e') \\
e \sim e' : \tau_1 \to \tau_2\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim e'_1 : \tau_1\ [\eta : \delta \leftrightarrow \delta']\ \textit{implies} \\
 & & e\,(e_1) \sim e'\,(e'_1) : \tau_2\ [\eta : \delta \leftrightarrow \delta'] \\
e \sim e' : \forall\,(t\,.\,\tau)\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every } \rho,\ \rho',\ \textit{and every } R : \rho \leftrightarrow \rho', \\
 & & e\,[\rho] \sim e'\,[\rho'] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']]
\end{array}$$

Logical equivalence is defined in terms of parameterized logical equivalence by considering all possible interpretations of its free type- and expression variables. An *expression assignment*, γ, for a context Γ, written $\gamma : \Gamma$, is an assignment of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in \mathit{dom}(\Gamma)$. An expression assignment, $\gamma : \Gamma$, induces a substitution function, $\hat{\gamma}$, defined by the equation

$$\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]e,$$

where the domain of Γ consists of the variables $x_1, \ldots, x_n$.

The relation $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$ is defined to hold iff $\mathit{dom}(\gamma) = \mathit{dom}(\gamma') = \mathit{dom}(\Gamma)$, and $\gamma(x) \sim \gamma'(x) : \Gamma(x)\ [\eta : \delta \leftrightarrow \delta']$ for every variable, x, in their common domain.

Definition 54.4 (Logical Equivalence)**.** *The expressions $\Delta;\Gamma \vdash e : \tau$ and $\Delta;\Gamma \vdash e' : \tau$ are* logically equivalent*, written $e \sim e' : \tau\ [\Delta;\Gamma]$ iff for every assigment δ and δ' of closed types to type variables in Δ, and every relation assignment $\eta : \delta \leftrightarrow \delta'$, if $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$, then $\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e')) : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

When e, e', and τ are closed, then this definition states that $e \sim e' : \tau$ iff $e \sim e' : \tau\ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

Lemma 54.3 (Closure under Converse Evaluation)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

Proof. By induction on the structure of τ. When $\tau = t$, the result holds because all relations under consideration are closed under converse evaluation. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications. □

Lemma 54.4 (Respect for Observational Equivalence)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \cong e : \hat{\delta}(\tau)$ and $d' \cong e' : \widehat{\delta'}(\tau)$, then $d \sim d' : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

Proof. By induction on the structure of τ, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall(t.\sigma)$, then we are to show that for every $R : \rho \leftrightarrow \rho'$,

$$d\,[\rho] \sim d'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']].$$

Since observational equivalence is a congruence, $d\,[\rho] \cong e\,[\rho] : [\rho/t]\hat{\delta}(\sigma)$, $d'\,[\rho] \cong e'\,[\rho] : [\rho'/t]\widehat{\delta'}(\sigma)$. From the assumption it follows that

$$e\,[\rho] \sim e'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']],$$

from which the result follows by induction. □

Corollary 54.5. *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\widehat{\delta'}(\tau)$.*

Proof. By Lemmas 54.3 on the facing page and 54.4 on the preceding page. □

Logical Equivalence respects observational equivalence.

Corollary 54.6. *If $e \sim e' : \tau\ [\Delta; \Gamma]$, and $d \cong e : \tau\ [\Delta; \Gamma]$ and $d' \cong e' : \tau\ [\Delta; \Gamma]$, then $d \sim d' : \tau\ [\Delta; \Gamma]$.*

Proof. By Lemma 54.2 on page 429 and Corollary 54.5. □

Lemma 54.7 (Compositionality). *Suppose that*

$$e \sim e' : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\delta}(\rho)] \leftrightarrow \delta'[t \mapsto \widehat{\delta'}(\rho)]],$$

where $R : \hat{\delta}(\rho) \leftrightarrow \widehat{\delta'}(\rho)$ is such that $R(d, d')$ holds iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. Then $e \sim e' : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$.

Proof. By induction on the structure of τ. When $\tau = t$, the result is immediate from the definition of the relation R. When $\tau = t' \neq t$, the result holds vacuously. When $\tau = \tau_1 \rightarrow \tau_2$ or $\tau = \forall(u\,.\,\tau)$, where without loss of generality $u \neq t$ and $u \# \rho$, the result follows by induction. □

Despite the strong conditions on polymorphic types, logical equivalence is not vacuous—in fact, expression satisfies its constraints.

Theorem 54.8 (Reynolds). *If $e : \tau$ is a closed expression, then $e \sim e : \tau$.*

Proof. By induction on derivations of the typing judgement for $\mathcal{L}\{\rightarrow\forall\}$. We consider two representative cases here.

Rule (25.2d) By induction we have that for all $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and all ρ, ρ', and $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}(\hat{\delta}(e)) \sim [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)) : \tau\ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

where $\eta_* = \eta[t \mapsto R]$, $\delta_* = \delta[t \mapsto \rho]$, and $\delta'_* = \delta'[t \mapsto \rho']$. Since

$$\Lambda(t\,.\,\hat{\gamma}(\hat{\delta}(e)))\,[\rho] \mapsto^* [\rho/t]\hat{\gamma}(\hat{\delta}(e))$$

and

$$\Lambda(t\,.\,\widehat{\gamma'}(\widehat{\delta'}(e)))\,[\rho'] \mapsto^* [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)),$$

the result follows by Lemma 54.3 on the facing page.

Rule (25.2e) By induction we have that for all $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$,

$$\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e)) : \forall(t.\tau)\ [\eta : \delta \leftrightarrow \delta']$$

Let $\hat{\rho} = \hat{\delta}(\rho)$ and $\widehat{\rho'} = \widehat{\delta'}(\rho)$. Define the relation $R : \hat{\rho} \leftrightarrow \widehat{\rho'}$ by $R(d, d')$ iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. By Corollary 54.5 on the previous page, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\rho'}] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \widehat{\rho'}]].$$

By Lemma 54.7 on the preceding page

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\rho'}] : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))[\hat{\delta}(\rho)] && (54.1) \\ &= \hat{\gamma}(\hat{\delta}(e[\rho])), && (54.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\rho'}] &= \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\delta'}(\rho)] && (54.3) \\ &= \widehat{\gamma'}(\widehat{\delta'}(e[\rho])), && (54.4) \end{aligned}$$

from which the result follows.

□

Corollary 54.9. *If $e \cong e' : \tau\ [\Delta; \Gamma]$, then $e \sim e' : \tau\ [\Delta; \Gamma]$.*

Proof. By Theorem 54.8 on the previous page $e \sim e : \tau\ [\Delta; \Gamma]$, and hence by Corollary 54.6 on the preceding page, $e \sim e' : \tau\ [\Delta; \Gamma]$. □

Lemma 54.10 (Congruence). *If $e \sim e' : \tau\ [\Delta\,\Delta'; \Gamma\,\Gamma']$ and $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau')$, then $\mathcal{C}\{e\} \sim \mathcal{C}\{e'\} : \tau\ [\Delta'; \Gamma']$.*

Proof. By induction on the structure of $\mathcal{C}$. □

Lemma 54.11 (Consistency). *Logical equivalence is consistent.*

Proof. To see that **tt** and **ff** are not logically related at type **2**, pick any type τ for which we have two expressions that are not observationally equivalent. The presumed logical equivalence of **tt** and **ff** implies that any two values of any type are observationally equivalent, from which we may derive a contradiction. For specificity, observe that **tt** and **ff** are not observationally equivalent at type **2**, since we may consider their behavior in the empty context. □

Corollary 54.12. *If $e \sim e' : \tau\ [\Delta;\Gamma]$, then $e \cong e' : \tau\ [\Delta;\Gamma]$.*

Proof. By Lemma 54.11 on the preceding page Logical equivalence is consistent, and by Lemma 54.10 on the facing page, it is a congruence, and hence is contained in observational equivalence. □

Corollary 54.13. *Logical and observational equivalence coincide.*

Proof. By Corollaries 54.9 on the preceding page and 54.12. □

If $d : \tau$ and $d \mapsto e$, then $d \sim e : \tau$, and hence by Corollary 54.12, $d \cong e : \tau$. Therefore if a relation respects observational equivalence, it must also be closed under converse evaluation. This shows that the second condition on admissibility is redundant, though it cannot be omitted at such an early stage.

54.4 Relational Parametricity

Using the Parametricity Theorem we may prove results about the inhabitants of polymorphic types. For example, if $e : \forall(t.t \to t)$, then we may show that if ρ is any type, and $d : \rho$, then $e[\rho](d) \mapsto^* d$. Let R be such that $R(e,e')$ iff $e \mapsto^* d$ and $e' \mapsto^* d$. Observe that $R : \rho \leftrightarrow \rho$, and that $R(d,d)$. It follows by Theorem 54.8 on page 433 that $R(e[\rho](d), e[\rho](d))$, which is to say that $e[\rho](d) \mapsto^* d$, as required.

(More examples to follow. . . .)

54.5 Exercises

Chapter 55

Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given in 26.1 on page 204 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 behavior of the client is independent of the choice of representation.

To say that no client can distinguish between two implementations of the same existential type is just to say that these two implementations are observationally equivalent as expressions of the existential type. Therefore representation independence for abstract types boils down to observational equivalence. But, as we have argued in Chapters 52 and 54, it can be quite difficult to reason directly about observational equivalence. A useful sufficient condition is derived from the concept of logical equivalence defined in Chapter 54 for polymorphic languages. This condition is called *bisimilarity*.

55.1 Bisimilarity of Packages

For two packages

$$e_1' = \texttt{pack}\,\rho_1\,\texttt{with}\,e_1\,\texttt{as}\,\exists(t\,.\,\tau)$$

and

$$e_2' = \texttt{pack}\,\rho_2\,\texttt{with}\,e_2\,\texttt{as}\,\exists(t\,.\,\tau)$$

of the same existential type, $\exists(t\,.\,\tau)$, to be observationally equivalent, it is sufficient to exhibit a relation $R : \rho_1 \leftrightarrow \rho_2$ between closed expressions of types ρ_1 and ρ_2, respectively, such that

$$e_1 \sim e_2 : \tau\ [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]].$$

This means that e_1 and e_2 are to be logically related as elements of type τ, under the assumption that elements of type t (which may occur free in τ) are related by the specified relation R. When this is the case, we say that R is a *bisimulation* between the two packages, and that the packages are thereby *bisimilar*.

Recall from Chapter 26 that the client, e_c, of the abstract type $\exists(t.\tau)$ is such that $t\ \mathsf{type}, x : \tau \vdash e_c : \tau_c$ for some type τ_c such that $t \mathbin{\#} \tau_c$. It follows from Theorem 54.8 on page 433 that

$$[e_1/x]e_c \sim [e_2/x]e_c : \tau_c\ [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]]$$

whenever

$$e_1 \sim e_2 : \tau\ [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]].$$

It follows that

$$\mathtt{open}\, e'_1\, \mathtt{as}\, t\, \mathtt{with}\, x{:}\tau\, \mathtt{in}\, e_c \sim \mathtt{open}\, e'_2\, \mathtt{as}\, t\, \mathtt{with}\, x{:}\tau\, \mathtt{in}\, e_c : \tau_c.$$

That is, the two implementations are indistinguishable by any client of the abstraction. This crucial property is called *representation independence* for abstract types. It is crucial that $t \mathbin{\#} \tau_c$ to ensure that the equivalence of the client under change of representation is independent of the relation R, which governs only the "private" parts of the abstraction.

Representation independence validates the following technique for proving the correctness of an ADT implementation. Suppose that we have a "clever" implementation of an abstract type $\exists(t.\tau)$ whose correctness we wish to verify. Let us call this the *candidate* implementation. To prove correctness of the candidate, we exhibit a *reference* implementation that is taken to be manifestly correct (or proved correct by a separate argument), and show that the reference and candidate implementations are bisimilar. It follows that they are observationally equivalent, and hence interchangeable in all contexts. In other words the candidate is "as correct as" the reference implementation.

55.2 Two Representations of Queues

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.

Letting $\rho_{\mathsf{ls}} = \texttt{nat list}$ and $\rho_{\mathsf{fb}} = \texttt{nat list} \times \texttt{nat list}$, define the relation $R : \rho_{\mathsf{ls}} \leftrightarrow \rho_{\mathsf{fb}}$ as follows:

$$R = \{\, (l, \langle b, f \rangle)) \mid l \cong b\,\texttt{@}\,\texttt{rev}(f) : \texttt{nat list} \,\}$$

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. $\texttt{QL.empty}\, R\, \texttt{QFB.empty}$.
2. Assuming that $m \sim n : \texttt{nat}$ and $l\, R\, \langle b, f \rangle$, show that
$$\texttt{QL.insert}(\langle m, l \rangle)\, R\, \texttt{QFB.insert}(\langle n, \langle b, f \rangle \rangle).$$
3. Assuming that $l\, R\, \langle b, f \rangle$, show that
$$\texttt{QL.remove}(l) \sim \texttt{QFB.remove}(\langle b, f \rangle) : \texttt{nat} \times t\, [[t \mapsto R] : [t \mapsto \rho_{\mathsf{ls}}] \leftrightarrow [t \mapsto \rho_{\mathsf{fb}}]].$$

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.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that $\texttt{QL.empty}\, R\, \texttt{QFB.empty}$, it suffices to show that
$$\texttt{nil}\,\texttt{@}\,\texttt{rev(nil)} \cong \texttt{nil} : \texttt{nat list},$$
which follows by symbolic execution, using the definitions of the operations involved.

2. For `insert`, we assume that $m \sim n : \texttt{nat}$ and $l \mathrel{R} \langle b, f\rangle$, and prove that
$$\texttt{QL.insert}(m, l) \mathrel{R} \texttt{QFB.insert}(n, \langle b, f\rangle).$$
By the definition of `QL.insert`, the left-hand side is observationally equivalent to $m :: l$, and by the definition of `QR.insert`, the right-hand side is observationally equivalent to $\langle n :: b, f\rangle$. It suffices to show that
$$m :: l \cong (n :: b) \mathbin{\texttt{@}} \texttt{rev}(f) : \texttt{nat list}.$$
Calculating, we obtain
$$(n :: b) \mathbin{\texttt{@}} \texttt{rev}(f) \cong n :: (b \mathbin{\texttt{@}} \texttt{rev}(f)) : \texttt{nat list}$$
and
$$n :: (b \mathbin{\texttt{@}} \texttt{rev}(f)) \cong n :: l : \texttt{nat list},$$
since $l \cong b \mathbin{\texttt{@}} \texttt{rev}(f) : \texttt{nat list}$. Since $m \sim n : \texttt{nat}$, it follows that $m = n$, which completes the proof.

3. For `remove`, we assume that l is related by R to $\langle b, f\rangle$, which is to say that $l \cong b \mathbin{\texttt{@}} \texttt{rev}(f) : \texttt{nat list}$. We are to show
$$\texttt{QL.remove}(l) \sim \texttt{QFB.remove}(\langle b, f\rangle) : \texttt{nat} \times t\,[[t \mapsto R] : [t \mapsto \rho_{\textsf{ls}}] \leftrightarrow [t \mapsto \rho_{\textsf{fb}}]].$$
Assuming that the queue is non-empty, so that removing an element is well-defined, it can be shown that $l \cong l' \mathbin{\texttt{@}} [m] : \texttt{nat list}$ for some l' and m. We proceed by cases according to whether or not f is empty. If f is non-empty, then it can be shown that $f \cong n :: f' : \texttt{nat list}$ for some n and f'. Then by the definition of `QFB.remove`,
$$\texttt{QFB.remove}(\langle b, f\rangle) \cong \langle n, \langle b, f'\rangle\rangle : \texttt{nat} \times t,$$
taking equality at type t to be the relation R. We must show that
$$\langle m, l'\rangle \sim \langle n, \langle b, f'\rangle\rangle : \texttt{nat} \times t,$$
with t equality being R. This means that we must show that $m = n$ and $l' \cong b \mathbin{\texttt{@}} \texttt{rev}(f') : \texttt{nat list}$.

Calculating from our assumptions,
$$\begin{array}{rcl} l & \cong & l' \mathbin{\texttt{@}} [m] \\ & \cong & b \mathbin{\texttt{@}} \texttt{rev}(f) \\ & \cong & b \mathbin{\texttt{@}} \texttt{rev}(n :: f') \\ & \cong & b \mathbin{\texttt{@}} (\texttt{rev}(f') \mathbin{\texttt{@}} [n]) \\ & \cong & (b \mathbin{\texttt{@}} \texttt{rev}(f')) \mathbin{\texttt{@}} [n], \end{array}$$

> from which the result follows. Finally, if f is empty, then it can be shown that $b \cong b' \texttt{ @ } [n] : \texttt{nat list}$ for some b' and n. But then
>
> $$\texttt{rev}(b) \cong n \texttt{ :: } \texttt{rev}(b') : \texttt{nat list},$$
>
> which reduces to the case for f non-empty.

This completes the proof — by representation independence the reference and candidate implementations are equivalent.

55.3 Exercises

Part XIX

Working Drafts of Chapters