Practical Foundations for Programming Languages

Robert Harper
Carnegie Mellon University

Spring, 2009

[Draft of January 12, 2009 at 3:15pm.]

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

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

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 will use the meta-variable P to stand for an unspecified judgement form, and the meta-variables a, b, and c to stand for unspecified objects. We write $a\ P$ for the judgement asserting that P 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 intentionally 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. In particular, we shall make frequent use of the construction of composite objects of the form $o(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n$ are objects and o is an n-argument operator. This construction includes a special case the formation of n-tuples, $(a_1, \ldots, a_n)$, in which the tupling operator is left implicit. (In Chapters 8 and 6 we will formalize these and richer forms of objects, called abstract syntax trees.)

1.2 Inference Rules

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

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

in which J and $J_1, \ldots, J_k$ are all judgements of the form being defined. The judgements 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 called an *axiom*; otherwise it is called a *proper rule*.

An inference rule may be read as 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 `nat`:

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

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

These rules specify that a `nat` holds whenever either a is `zero`, or a is `succ(`b`)` where b `nat`. Taking these rules to be exhaustive, it follows that a `nat` iff a is a natural number written in unary.

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

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

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

These rules specify that a tree holds if either a is `empty`, or a is $\texttt{node}(a_1;a_2)$, 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:

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

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

In each of the preceding examples we have made use of a notational convention for specifying an infinite family of rules by a finite number of patterns, or *rule schemes*. For example, Rule (1.2b) is a rule scheme that determines one rule, called an *instance* of the rule scheme, for each choice of object a in the rule. We will rely on context to determine whether a rule is stated for a *specific* object, a, or is instead intended as a rule scheme specifying a rule for *each choice* of objects in the rule. (In Chapter 3 we will remove this amiguity by introducing parameterization of rules by objects.)

A collection of rules is considered to define the *strongest* judgement that is *closed under*, or *respects*, those rules. To be closed under the rules simply means that the rules are *sufficient* to show the validity of a judgement: J holds *if* there is a way to obtain it using the given rules. To be the *strongest* judgement closed under the rules means that the rules are also *necessary*: J holds *only if* there is a way to obtain it by applying the rules. The sufficiency of the rules means that we may show that J holds by *deriving* it by composing rules. Their necessity means that we may reason about it using *rule 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. We sometimes say that a derivation of J is *evidence* for the validity of 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

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

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

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

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}}\,. \tag{1.6}$$

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

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

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, and continue. If there is more than one such rule, this process must be repeated, with the same starting queue, for each candidate rule. The process terminates whenever the queue is empty, all goals having been achieved; any pending consideration of candidate rules along the way may be discarded. 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, we may prove properties of such judgements by *rule induction*, or *induction on derivations*. The principle of rule induction states that to show that a property $\mathcal{P}$ holds of a judgement J whenever J is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining J. Writing $\mathcal{P}(J)$ to mean that $\mathcal{P}$ holds of J, we say that $\mathcal{P}$ respects the rule

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

if $\mathcal{P}(J)$ holds whenever $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$. The assumptions $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$ are the *inductive hypotheses*, and $\mathcal{P}(J)$ is the *inductive conclusion*, corresponding to that rule.

The principle of rule induction is simply the expression of the definition of an inductively defined judgement form as the *strongest* judgement form closed under the rules comprising the definition. This means that the judgement form is both (a) closed under those rules, and (b) sufficient for any other property also closed under those rules. The former property means that a derivation is evidence for the validity of a judgement; the latter means that we may reason about an inductively defined judgement form by rule induction.

If $\mathcal{P}(J)$ is closed under a set of rules defining a judgement form, then so is the conjunction of $\mathcal{P}$ with the judgement itself. This means that when showing $\mathcal{P}$ to be closed under a rule, we may inductively assume not only that $\mathcal{P}(J_i)$ holds for each of the premises J_i, but also that J_i itself holds as well. We shall generally take advantage of this without explicit mentioning that we are doing so.

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. $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$.
2. $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$, 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. $\mathcal{P}(\mathtt{empty}\ \mathsf{tree})$.
2. $\mathcal{P}(\mathtt{node}(a_1; a_2)\ \mathsf{tree})$, 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 $\mathtt{zero} = \mathtt{zero}\ \mathsf{nat}$.

Rule (1.2b) Assume that $a = a\ \mathsf{nat}$. It follows that $\mathtt{succ}(a) = \mathtt{succ}(a)\ \mathsf{nat}$ 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* $\mathtt{succ}(a)$ 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 $\mathtt{succ}(a)$ for some a, then a nat. We proceed by rule induction on Rules (1.2).

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

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

□

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

Lemma 1.3. *If* $\mathtt{succ}(a_1) = \mathtt{succ}(a_2)$ 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 $\mathtt{succ}(a_1)$ and b_2 is $\mathtt{succ}(a_2)$, then $a_1 = a_2$ nat. We proceed by rule induction on Rules (1.4):

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

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 $\mathtt{succ}(b_1)$ is $\mathtt{succ}(a_1)$ and $\mathtt{succ}(b_2)$ is $\mathtt{succ}(a_2)$, 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. In an iterated inductive definition the premises of a rule

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

may be instances of either a previously defined judgement form, or the judgement form being defined. For example, the following rules, define the judgement a list stating that a is a list of natural numbers.

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

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

The first premise of Rule (1.8b) is an instance of the judgement form a nat, which was defined previously, whereas the premise b list is an instance of the judgement form being defined by these rules.

Frequently two or more judgements are defined at once by a *simultaneous inductive definition*. A simultaneous inductive definition consists of a set of rules for deriving instances of several different judgement forms, any of which may appear as the premise of any rule. Since the rules defining each judgement form may involve any of the others, none of the judgement forms may be taken to be defined prior to the others. Instead one must understand that all of the judgement forms are being defined at once by the entire collection of rules. The judgement forms defined by these rules are, as before, the strongest judgement forms that are closed under the rules. Therefore the principle of proof by rule induction continues to apply, albeit in a form that allows us to prove a property of each of the defined judgement forms simultaneously.

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:

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

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

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

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

As a simple example, we may use simultaneous rule induction to prove that (1) if a even, then a nat, and (2) if a odd, then a nat. That is, we define the property $\mathcal{P}$ by (1) $\mathcal{P}(a\ \mathsf{even})$ iff a nat, and (2) $\mathcal{P}(a\ \mathsf{odd})$ iff a nat. The principle of rule induction for Rules (1.9) states that it is sufficient to show the following facts:

1. $\mathtt{zero}\ \mathsf{nat}$, which is derivable by Rule (1.2a).
2. If a nat, then $\mathtt{succ}(a)\ \mathsf{nat}$, which is derivable by Rule (1.2b).
3. If a nat, then $\mathtt{succ}(a)\ \mathsf{nat}$, which is also derivable by Rule (1.2b).

1.6 Defining Functions by Rules

A common use of inductive definitions is to define a function by giving an inductive definition of its *graph* relating inputs to outputs, and then showing that the relation uniquely determines the outputs for given inputs. For example, we may define the addition function on natural numbers as the relation $\mathsf{sum}(a;b;c)$, with the intended meaning that c is the sum of a and b, as follows:

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

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

The rules define a ternary (three-place) relation, $\mathsf{sum}(a;b;c)$, among natural numbers a, b, and c. We may show that c is determined by a and b in this relation.

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 $a = \mathtt{succ}(a')$ and $c_1 = \mathtt{succ}(c_1')$, 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 $c_2 = \mathtt{succ}(c_2')$ 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 Modes

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. For example, we have shown that every two natural numbers have a sum according to Rules (1.10). This fact may be restated as a mode specification by saying that the judgement $\mathsf{sum}(a;b;c)$ has *mode* $(\forall, \forall, \exists)$. The notation arises from the form of the proposition it expresses: *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 $\mathsf{sum}(a;b;c)$ 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* $\mathsf{sum}(a;b;c)$. 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* $\mathsf{sum}(a;b;c)$.

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 our choice of notation. 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 (given by Rules (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 notation, writing writing $a + b$ for the unique c such that $a + b = c$ nat.

1.8 Foundations

An inductively defined judgement form, such as a 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 `nat` isolates those objects of the form `succ(...succ(zero)...)`. 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*. For example, the object `succ(succ(zero))` may be considered to be 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 $a + b = c$ `nat`, may be handled by demanding that the universe also be closed under formation of finite tuples $(a_1, \ldots, a_n)$ of objects. It is also possible to 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 (that is, 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 without recourse to encoding them as sets. Moreover, there is a significant disadvantage to working with sets. 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

[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.

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 $\mathtt{nat}$, b $\mathtt{nat}$, and c $\mathtt{nat}$, 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 $\mathsf{hgt}(a; b)$.

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

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

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

3. Give an inductive definition of the judgement "∇ is a derivation of J" for an inductively defined judgement J of your choice.

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* expresses an *entailment* between one or more *hypotheses* and a *conclusion*. We will consider two notions of entailment, called *derivability* and *admissibility*. Derivability expresses the stronger of the two forms of entailment, namely that the conclusion may be deduced directly from the hypotheses by composing rules. Admissibility expresses the weaker form, that the conclusion is derivable from the rules whenever the hypotheses are also derivable. Both forms of entailment share a common set of *structural* properties that characterize conditional reasoning. One consequence of these properties is that derivability is stronger than admissibility (but the converse fails, in general). We then generalize the concept of an inductive definition to admit rules that have not only categorical, but also hypothetical, judgements as premises. Using these we may enrich the rule set with new axioms that are available for use within a specified premise of a rule.

2.1 Derivability

For a given set, $\mathcal{R}$, of rules, we define the *derivability* judgement, written $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$, where each J_i and K are categorical, to mean that we may derive K using rules $\mathcal{R} \cup \{ J_1, \ldots, J_k \}$.[1] That is, we treat the *hypotheses*, or *antecedents*, of the judgement, $J_1, \ldots, J_n$ as *temporary axioms*, and derive the *conclusion*, or *consequent*, by composing rules in $\mathcal{R}$. We often use capital

[1] Here we are treating the judgements J_i as axioms, or rules with no premises.

Greek letters, frequently Γ or Δ, to stand for a finite set of categorical judgements, writing $\Gamma \vdash_{\mathcal{R}} K$ to mean that K is derivable from rules $\mathcal{R} \cup \Gamma$ (that is, regarding the hypotheses as axioms). We sometimes write $\vdash_{\mathcal{R}} \Gamma$, where Γ is the finite set $\{ J_1, \ldots, J_k \}$, to mean that $\vdash_{\mathcal{R}} J_i$ for each $1 \leq i \leq k$.

The validity of the derivability judgement $J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is sometimes expressed by saying that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{2.1}$$

is *derivable* from rules $\mathcal{R}$. Consequently, evidence for the derivability of a rule consists, naturally enough, of a derivation of its conclusion from presumed derivations of its premises.

For example, the derivability judgement

$$a \ \mathtt{nat} \vdash_{(1.2)} \mathtt{succ(succ}(a)\mathtt{)\ nat} \tag{2.2}$$

is valid relative to Rules (1.2). Equivalently, the rule

$$\frac{a \ \mathtt{nat}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}} \tag{2.3}$$

is derivable from Rules (1.2). Evidence for this is provided by the derivation

$$\frac{\dfrac{a \ \mathtt{nat}}{\mathtt{succ}(a)\ \mathtt{nat}}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}}\,, \tag{2.4}$$

which composes Rules (1.2), starting with a nat as an axiom, and ending with succ(succ(a)) nat.

It follows directly from the definition of derivability that it is stable under extension with new rules.

Theorem 2.1 (Uniformity)**.** *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \vdash_{\mathcal{R} \cup \mathcal{R}'} J$*.*

Proof. Any derivation of J from $\mathcal{R} \cup \Gamma$ is also a derivation from $\mathcal{R} \cup \mathcal{R}' \cup \Gamma$, since the presence of additional rules does not influence the validity of the derivation. □

Derivability enjoys a number of *structural properties* that follow from its definition, independently of the rule set, $\mathcal{R}$, in question.

Reflexivity Every judgement is a consequence of itself: $\Gamma, J \vdash J$. Each hypothesis justifies itself as conclusion.

Weakening If $\Gamma \vdash J$, then $\Gamma, K \vdash J$. Entailment is not influenced by unexercised options.

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 $\Gamma, J \vdash K$. We may use a hypothesis as many times as we like in a derivation.

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 is a derivation of its consequent without that hypothesis.

These properties may be summarized by saying that derivability is *structural*.

Theorem 2.2. *For any rule set $\mathcal{R}$, the derivability judgement $\Gamma \vdash_{\mathcal{R}} J$ is structural.*

Proof. Reflexivity follows directly from the meaning of derivability. Weakening follows directly from uniformity. Exchange and contraction are inherent in treating rules as sets. Transitivity is proved by rule induction on the first premise. □

2.2 Admissibility

Admissibility, written $\Gamma \models_{\mathcal{R}} J$, is a weaker form of hypothetical judgement than derivability stating that if $\vdash_{\mathcal{R}} \Gamma$, then $\vdash_{\mathcal{R}} J$. That is, J is derivable from rules $\mathcal{R}$ whenever Γ is derivable from rules $\mathcal{R}$. Evidence for admissibility consists of a mathematical function transforming derivations of the hypotheses into derivations of the conclusion of the judgement. This means, in particular, that if any of the hypotheses are *not* derivable, then the admissibility judgement is vacuously true. This is in sharp contrast to derivability, for which there is no concept of vacuous validity. The the admissibility judgement $J_1, \ldots, J_n \models_{\mathcal{R}} J$ is sometimes expressed by stating that the corresponding rule,

$$\frac{J_1 \quad \ldots \quad J_n}{J}, \tag{2.5}$$

is *admissible* relative to rules $\mathcal{R}$.

For example, the admissibility judgement

$$\mathsf{succ}(a)\ \mathsf{nat} \models_{(1.2)} a\ \mathsf{nat} \tag{2.6}$$

is valid, because any derivation of $\mathtt{succ}(a)\ \mathtt{nat}$ from Rules (1.2) must contain a sub-derivation of $a\ \mathtt{nat}$ from the same rules, which justifies the conclusion. This may equivalently be expressed by saying that the rule

$$\frac{\mathtt{succ}(a)\ \mathtt{nat}}{a\ \mathtt{nat}} \tag{2.7}$$

is admissible relative to Rules (1.2).

In contrast to derivability the admissibility judgement is *not* stable under extension to the rules. For example, if we enrich Rules (1.2) with the axiom

$$\frac{}{\mathtt{succ(junk)\ nat}} \tag{2.8}$$

(where `junk` is some object for which `junk nat` is not derivable), then the admissibility (2.6) is *invalid*. This is because Rule (2.8) has no premises, and there is no composition of rules deriving `junk nat`.

This example shows that admissibility is sensitive to which rules are *absent* from, as well as to which rules are *present* in, an inductive definition.

The structural properties of derivability given by Theorem 2.2 on the preceding page ensure that derivability is stronger than admissibility.

Theorem 2.3. *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \models_{\mathcal{R}} J$*.*

Proof. Repeated application of the transitivity of derivability implies that if $\Gamma \vdash_{\mathcal{R}} J$ and $\vdash_{\mathcal{R}} \Gamma$, then $\vdash_{\mathcal{R}} J$. □

To see that the converse fails, observe that there is no composition of rules such that $\mathtt{succ(junk)\ nat} \vdash_{(1.2)} \mathtt{junk\ nat}$, yet $\mathtt{succ(junk)\ nat} \models_{(1.2)} \mathtt{junk\ nat}$ holds vacuously.

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

Theorem 2.4. *The admissibility judgement $\Gamma \mid\!=_{\mathcal{R}} J$ is structural.*

Proof. Follows immediately from the definition of admissibility as stating that if the hypotheses are derivable relative to $\mathcal{R}$, then so is the conclusion. □

2.3 Conditional Inductive Definitions

It is useful to enrich the concept of an inductive definition to permit rules with derivability judgements as premises and conclusions. Doing so permits us to introduce *local hypotheses* that apply only in the derivation of a particular premise, and also allows us to constrain inferences based on the *global hypotheses* in effect at the point where the rule is applied.

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

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\ . \tag{2.9}$$

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:

$$\frac{\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma_n \vdash J_n}{J}\ . \tag{2.10}$$

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 a specified *side condition* is satisfied. Such rules are said to be *impure*. Impure rules generally have the form

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n \quad \Psi}{\Gamma \vdash J}\,, \tag{2.11}$$

where the condition, Ψ, limits the applicability of this rule to situations in which it is true. For example, Ψ may restrict the global context of the inference to be empty, so that no instances involving global hypotheses are permissible.

The use of the derivability judgement in the premises of a conditional rule demands careful consideration to avoid an apparent circularity. What does it mean to use the derivability consequence relation for a set of rules in one of the rules of that very set? One way to justify this is to consider a conditional inductive definition to be an ordinary inductive definition of a *formal derivability judgement*, $\Gamma \vdash J$, that is forced to behave like derivability by tacitly including the following *structural rules*:

$$\frac{}{\Gamma, J \vdash J} \tag{2.12a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{2.12b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{2.12c}$$

In the common case that all other rules in a conditional inductive definition are pure, Rules (2.12b) and (2.12c) are admissible, and hence need not be taken as primitive rules. (This is readily proved by rule induction, applying the inductive hypothesis to each premise, and re-applying the same rule to obtain the desired conclusion.) This leaves Rule (2.12a) to be included in any conditional inductive definition.

We say that a property, $\mathcal{P}$, of formal entailments $\Gamma \vdash J$ is *closed under* a conditional rule

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

if, and only if,

$$\mathcal{P}(\Gamma\,\Gamma_1 \vdash J_1),\ \ldots,\ \mathcal{P}(\Gamma\,\Gamma_n \vdash J_n) \textit{ imply } \mathcal{P}(\Gamma \vdash J).$$

For example, $\mathcal{P}$ is closed under reflexivity iff $\mathcal{P}(\Gamma, J \vdash J)$.

The judgement $\Gamma \vdash_{\mathcal{R}} J$ is the strongest formal entailment closed under the rules in $\mathcal{R}$. It follows immediately that, for each conditional rule of the form (2.9) in $\mathcal{R}$, if $\Gamma\,\Gamma_1 \vdash_{\mathcal{R}} J_1, \ldots, \Gamma\,\Gamma_n \vdash_{\mathcal{R}} J_n$, then $\Gamma \vdash_{\mathcal{R}} J$. Moreover, the following principle of *conditional rule induction* is valid: to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that for each rule of the form (2.9) (including the structural rules), if $\mathcal{P}(\Gamma, \Gamma_1 \vdash J_1), \ldots, \mathcal{P}(\Gamma, \Gamma_n \vdash J_n)$, then $\mathcal{P}(\Gamma \vdash J)$.

2.4 Exercises

1. Prove that if all rules in a conditional inductive definition are pure, then the structural rules of weakening (Rule (2.12b)) and transitivity (Rule (2.12c)) are admissible.

2. 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 $\Gamma' \vdash J$. *Hint*: from left to right, appeal to transitivity of entailment; from right to left, consider the case of $\Gamma' = \Gamma$.

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

Generic Judgements

Just as hypothetical judgements express reasoning under hypotheses, so *generic judgements* express reasoning *generically* with respect to unspecified objects. There are two forms of generic judgement, the *parametric*, or *uniform*, and the *non-parametric*, or *non-uniform*. The parametric form captures reasoning that is completely independent of one or more objects, which are represented by *parameters*, much as the derivability judgement expresses reasoning with respect to unjustified axioms. The non-parametric form captures reasoning that depends on the specific choice of objects, much as the admissibility judgement expresses reasoning based on the possible forms of derivation of the hypotheses. The concept of a conditional inductive definition may be generalized to admit either form of generic judgement as premises, resulting in the concept of a *generic inductive definition*, which we shall use heavily throughout the book.

3.1 Objects and Parameters

We will enrich the class of objects that we consider to include *parameters*, or *indeterminates*, that may be replaced by other objects, themselves perhaps involving parameters, by a process known as *substitution*, or *instantiation*. For the purposes of this chapter we need not be specific about the exact nature of objects, parameters, or substitution, but can instead rely only on a specification of a few operations and relations on them.

We assume given an infinite class of objects, called *parameters*, that are distinct from all other objects (so that we know a parameter when we see one, and cannot confuse parameters with other forms of object). We generally use the variables, x, y, and z to stand for parameters, and we use the

variables $\mathcal{X}$ and $\mathcal{Y}$ to stand for finite sets of parameters. We write $\mathcal{X}, x$ for $\mathcal{X} \cup \{ x \}$, and $\mathcal{X}\,\mathcal{Y}$ for $\mathcal{X} \cup \mathcal{Y}$.

We take as given a judgement $\mathcal{X} \vdash a$ obj specifying that a is an object all of whose parameters lie within the set $\mathcal{X}$. Observe that if $\mathcal{X} \vdash a$ obj and $x \notin \mathcal{X}$, then the parameter x cannot occur within the object a. If $\emptyset \vdash a$ obj, then we say that a is a *closed* object; otherwise, a is said to be an *open* object. Observe that the judgement $\mathcal{X} \vdash a$ obj is closed under expansion of the set of parameters, and that $\mathcal{X}, x \vdash x$ obj.

We further assume that parameters may be *renamed* at will, provided that no two parameters are confused in the process. If $\mathcal{X}, x \vdash a_x$ obj is an object involving parameter x, and $y \notin \mathcal{X}$, then $\mathcal{X}, y \vdash a_y$ obj as well.

Finally, we assume given a function, called *substitution*, or *instantiation*, and written $[a/x]b$, with the property that if $\mathcal{X} \vdash a$ obj and $\mathcal{X}, x \vdash b$ obj, then $\mathcal{X} \vdash [a/x]b$ obj. More generally, we make use of *simultaneous substitution*, written $[a_1, \ldots, a_n/x_1, \ldots, x_n]a$, with the evident meaning. Finally, we assume that substitution is associative in the sense that

$$[a/x][b/y]c = [[a/x]b/y][a/x]c.$$

In practice it is usually necessary to distinguish multiple categories of objects and parameters, and to restrict renaming and substitution to respect the classes of the objects and parameters involved. It is straightforward to generalize the material in this chapter to account for multiple categories of objects and parameters.

3.2 Rule Schemes

We begin by making precise the informal concepts of rules and rule schemes discussed in Chapter 1. Recall that a rule scheme is a rule that involves meta-variables standing for unspecified objects. An instance of a rule scheme is obtained by replacing these meta-variables with specific objects. We may now make these informal concepts precise using the mechanisms of parameterization and substitution.

A *rule scheme* is a configuration of the form

$$x_1, \ldots, x_n \;\Big|\; \frac{J_1 \quad \ldots \quad J_k}{J} \tag{3.1}$$

consisting of a rule prefixed by a finite set of parameters containing *all* of the parameters that may occur in the premises or conclusion of the rule

scheme. No other parameters may occur in the rule other than those specified in its prefix. An *instance* of the rule scheme (3.1) is obtained by substituting objects $a_1, \ldots, a_n$ for the parameters $x_1, \ldots, x_n$. For example, Rule (1.2b) may be presented as a rule scheme by writing it in the form

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

Its instances, obtained by replacing the parameter, x, by an object, a, are rules of the form

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

A rule scheme stands for the totality of its instances. Consequently, we say that a property, $\mathcal{P}$, is *closed under* a rule scheme iff it is closed under all of its instances. More precisely, $\mathcal{P}$ is closed under a rule scheme of the form (3.1) iff for every choice of objects $a_1, \ldots, a_n$, if

$$\mathcal{P}([a_1, \ldots, a_n/x_1, \ldots, x_n]J_1) \textit{ and } \ldots \textit{ and } \mathcal{P}([a_1, \ldots, a_n/x_1, \ldots, x_n]J_k),$$

then

$$\mathcal{P}([a_1, \ldots, a_n/x_1, \ldots, x_n]J).$$

The judgement form defined by a set, $\mathcal{R}$, of rule schemes is defined to be the strongest judgement form closed under the rule schemes in $\mathcal{R}$.

3.3 Uniform Genericity

The *parametric*, or *uniform*, *derivability* judgement, written $\mathcal{X} \mid \Gamma \vdash_{\mathcal{R}} J$, states that the categorical judgement J is derivable from rules $\mathcal{R}$ and hypotheses Γ uniformly in the parameters $\mathcal{X}$. (When Γ is empty, we abbreviate the uniform derivability judgement to $\mathcal{X} \mid_{\mathcal{R}} J$.) By "uniformly" we mean that the parameters $\mathcal{X}$ are to be regarded as *fresh* objects, distinct from all others, in the derivation of J from Γ according to the rule schemes, $\mathcal{R}$. Evidence for $\mathcal{X} \mid \Gamma \vdash_{\mathcal{R}} J$ consists of a *derivation scheme*, ∇, involving the parameters in $\mathcal{X}$ that composes rules in $\mathcal{R}$ to obtain J from the assumptions in Γ. An *instance* of a derivation scheme is obtained by replacing the parameters of the scheme with chosen objects to obtain a derivation.

Just as the hypotheses Γ are to be considered "local" to ∇, so also are the parameters $\mathcal{X}$. We do not distinguish two derivation schemes that differ only in the choice of parameter names used to represent the fresh objects of the derivation. The parameters serve only as a kind of "pronoun", referring to some unspecified concrete object to be determined later; the exact

"word" we use for the pronoun is not relevant, all that matters is that the referent be unambiguous.

For example, evidence for the uniform derivability judgement

$$x \mid x\ \mathsf{nat} \vdash_{(1.2)} \mathtt{succ(succ}(x)\mathtt{)}\ \mathsf{nat} \tag{3.2}$$

consists of the derivation scheme, ∇_x, given as follows:

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

We could just as well have used y throughout as a parameter of the scheme without affecting its meaning.

By choosing an object, a, we obtain an instance

$$a\ \mathsf{nat} \vdash \mathtt{succ(succ}(a)\mathtt{)}\ \mathsf{nat}, \tag{3.4}$$

of the uniform derivability judgement. Similarly, the derivation scheme ∇_x may be specialized to a to obtain the derivation

$$\dfrac{\dfrac{\overline{a\ \mathsf{nat}}}{\mathtt{succ}(a)\ \mathsf{nat}}}{\mathtt{succ(succ}(a)\mathtt{)}\ \mathsf{nat}}\ . \tag{3.5}$$

The resulting derivation is evidence for (3.4).

The parametric derivability judgement enjoys a collection of *structural properties* that follow directly from its meaning:

Renaming If $\mathcal{X}, x \mid \Gamma \vdash_{\mathcal{R}} J_x$, then $\mathcal{X}, y \mid \Gamma \vdash_{\mathcal{R}} J_y$, provided that $y \notin \mathcal{X}$.

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

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

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

Instantiation If $\mathcal{X}, x \mid \Gamma \vdash_{\mathcal{R}} J$, and $\mathcal{X} \vdash a\ \mathsf{obj}$, then $\mathcal{X} \mid [a/x]\Gamma \vdash_{\mathcal{R}} [a/x]J$.

Renaming states that a uniform derivability judgement is invariant under renaming of parameters. With respect to the structural properties of the hypothetical judgement, proliferation corresponds to weakening, swapping corresponds to exchange, duplication corresponds to contraction, and instantiation corresponds to transitivity.

Theorem 3.1. *The uniform derivability judgement is structural.*

Proof. Renaming follows from the identification of two parametric judgements that differ only in the names of bound variables. Proliferation is a direct consequence of uniformity. Swapping and duplication are inherent in considering the parameters of the judgement to be a set. Instantiation may be proved by rule induction on the judgement $\mathcal{X}, x \mid \Gamma \vdash_{\mathcal{R}} J$, where $x \notin \mathcal{X}$. Consider an arbitrary rule of the form (3.1), and suppose that we have evidence for each of the premises

$$\mathcal{X}, x \mid \Gamma \vdash_{\mathcal{R}} [a_1, \ldots, a_n / x_1, \ldots, x_n] J_i,$$

where $1 \leq i \leq k$. By the inductive hypothesis we have evidence for

$$\mathcal{X} \mid [a/x]\Gamma \vdash_{\mathcal{R}} [a/x][a_1, \ldots, a_n / x_1, \ldots, x_n] J_i$$

for each $1 \leq i \leq k$. By suitable renaming the parameter x may be chosen apart from $x_1, \ldots, x_n$, and hence does not occur in any premise of the rule. It follows that we have evidence for the judgement

$$\mathcal{X} \mid [a/x]\Gamma \vdash_{\mathcal{R}} [[a/x]a_1, \ldots, [a/x]a_n / x_1, \ldots, x_n] J_i.$$

We may then re-instantiate the same rule scheme to obtain the conclusion

$$\mathcal{X} \mid [a/x]\Gamma \vdash_{\mathcal{R}} [[a/x]a_1, \ldots, [a/x]a_n / x_1, \ldots, x_n] J,$$

which is to say

$$\mathcal{X} \mid [a/x]\Gamma \vdash_{\mathcal{R}} [a/x][a_1, \ldots, a_n / x_1, \ldots, x_n] J.$$

□

3.4 Non-Uniform Genericity

The *non-parametric*, or *non-uniform*, *derivability* judgement, $\mathcal{X} \parallel \Gamma \vdash_{\mathcal{R}} J$, where $\mathcal{X} = x_1, \ldots, x_n$, states that for every *closed* instance $a_1, \ldots, a_n$ of the parameters, the derivability judgement

$$[a_1, \ldots, a_n / x_1, \ldots, x_n]\Gamma \vdash [a_1, \ldots, a_n / x_1, \ldots, x_n] J$$

is valid. Since only closed instances are considered, evidence for non-uniform derivability may assign a different derivation for each instance. This is analogous to the distinction between admissibility and derivability

discussed in Chapter 2. Whereas derivability is uniform (does not depend on the evidence for the hypotheses, if any), admissibility is non-uniform (depends on the evidence for the hypotheses to determine evidence for the conclusion).

Theorem 3.2. *The non-uniform derivability judgement is structural.*

Proof. This is an immediate consequence of the definition, using familiar properties of universal quantification and implication. □

Consequently, uniform derivability is stronger than non-uniform derivability.

Theorem 3.3. *If* $\mathcal{X} \mid \Gamma \vdash_{\mathcal{R}} J$*, then* $\mathcal{X} \parallel \Gamma \vdash_{\mathcal{R}} J$*.*

Proof. Follows directly from Theorem 3.1 on the previous page. □

3.5 Generic Inductive Definitions

A *generic inductive definition* is a generalization of a conditional inductive definition in which we permit expansion not only of the rules, but also of the set of active parameters, in each premise of a rule. A *generic conditional rule* has the form

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

The set, $\mathcal{X}$, is the set of *global parameters* of the inference, and, for each $1 \leq i \leq n$, the set $\mathcal{X}_i$ is the set of *fresh local parameters* of the ith premise. The local parameters are *fresh* in the sense that, by suitable renaming, they may be chosen to be disjoint from the global parameters of the inference. The pair $\mathcal{X} \mid \Gamma$ is called the *global context* of the rule, and each pair $\mathcal{X}_i \mid \Gamma_i$ is called the *local context* of the ith premise of the rule.

A generic rule is *pure* if it is stated for all choices of global context, subject only to the freshness requirement on local parameters. Such a rule may be written in *implicit form* as follows:

$$\frac{\mathcal{X}_1 \mid \Gamma_1 \vdash J_1 \quad \ldots \quad \mathcal{X}_n \mid \Gamma_n \vdash J_n}{J}\ . \tag{3.7}$$

This form of the rule stands for all rules of the form Rule (3.6) obtained by specifying the global context $\mathcal{X} \mid \Gamma$.

As with conditional inductive definitions, we regard a generic inductive definition as defining a *formal generic entailment*, written $\mathcal{X} \mid \Gamma \vdash J$, that is expresses uniform derivability with respect to the rules themselves. To ensure that the formal uniform derivability judgement is well-behaved, the following *structural rules* are implicitly included in any generic inductive definition:

$$\frac{\mathcal{X} \mid \Gamma \vdash J}{\mathcal{X}, x \mid \Gamma \vdash J} \tag{3.8a}$$

$$\frac{\mathcal{X}, x \mid \Gamma \vdash J \quad \mathcal{X} \vdash a \ \mathsf{obj}}{\mathcal{X} \mid [a/x]\Gamma \vdash [a/x]J} \tag{3.8b}$$

If all of the rules in a generic inductive definition are pure, then these rules are admissible, otherwise they must be included as primitive rules.

The principle of rule induction for a generic inductive definition states that to show $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash J)$ whenever $\mathcal{X} \mid \Gamma \vdash J$ is derivable, it is enough to show that $\mathcal{P}$ is closed under the rules comprising the definition. Specifically, for each rule of the form (3.6), we must show that

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

Observe that, by our identification of two generic judgements that differ only in the names of their parameters, the property, $\mathcal{P}$, is not permitted to distinguish between any two such judgements. This limits the class of properties that we may consider to those that are well-defined with respect to this identification, but experience shows that all natural properties (including those of interest in this book) respect this equivalence. This means that the proof of $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash J)$ need only be carried out for one particular choice of parameters, and that this choice may be tacitly assumed to satisfy any finite freshness requirements we may wish to impose.

3.6 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 $s \downarrow$ 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 $s_1 \stackrel{i}{\mapsto}_1 s_1'$, 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 $s_1 \, S \, s_2$:

1. If $s_1 \mapsto_1 s_1'$, then $s_2 \mapsto_2^* s_2'$ for some s_2' such that $s_1' \, S \, s_2'$.

2. If $s_1 \stackrel{i}{\mapsto}_1 s_1'$, then $s_2 \mapsto_2^* \stackrel{i}{\mapsto}_2 \mapsto_2^* s_2'$ for some s_2' such that $s_1' \, S \, s_2'$.

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 $\mapsto$ iff S is a strong simulation for $\mapsto^*$.

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 sym to assert that x is a symbol, and we assume that there are infinitely many symbols at our disposal. The judgement $x \# y$, where x sym and y 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 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 $\epsilon\, abcd$ or $abcd\, \epsilon$, 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.

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

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

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

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:

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

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 $\mathcal{P}(\mathcal{X} \vdash a_1\ \mathsf{ast})$, $\ldots$, $\mathcal{P}(\mathcal{X} \vdash a_n\ \mathsf{ast})$, 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 $\mathcal{X} \vdash [a/x]b_1 = c_1$, $\ldots$, $\mathcal{X} \vdash [a/x]b_n = c_n$. 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 hypothetical judgement of the form

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

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

We use the meta-variable $\mathcal{A}$ to range over finite sets of assumptions of the form $x\ \mathsf{abt}^0$, where x is a parameter. We write $x\ \#\ \mathcal{A}$ to mean that there is no assumption of the form $x\ \mathsf{abt}^0$ in $\mathcal{A}$.

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

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

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

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

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 $x' \# \mathcal{A}$ 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.

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 $\mathcal{P}(\mathcal{A} \vdash a\ \mathsf{abt}^n)$ whenever $\mathcal{A} \vdash a\ \mathsf{abt}^n$, it suffices to show that $\mathcal{P}$ is closed under Rules (6.1). Specifically, we must show:

1. $\mathcal{P}(\mathcal{A}, x\ \mathsf{abt}^0 \vdash x\ \mathsf{abt}^0)$.
2. If o has arity $(m_1, \ldots, m_k)$ and $\mathcal{P}(\mathcal{A} \vdash a_1\ \mathsf{abt}^{m_1}), \ldots, \mathcal{P}(\mathcal{A} \vdash a_k\ \mathsf{abt}^{m_k})$, then $\mathcal{P}(\mathcal{A} \vdash o(a_1, \ldots, a_k)\ \mathsf{abt}^0)$.
3. If $\mathcal{P}(\mathcal{A}, x'\ \mathsf{abt}^0 \vdash [x' \leftrightarrow x]\, a\ \mathsf{abt}^n)$ for every $x' \# \mathcal{A}$, then $\mathcal{P}(\mathcal{A} \vdash x\,.\,a\ \mathsf{abt}^{n+1})$.

The condition on abstractors ensures that the name of a bound variable does not matter, and permits us to consider that it is chosen to be any fresh variable we like.

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 hypothetical judgement

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

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

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

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

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

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* $x_1\ \mathsf{abt}^0, \ldots, x_k\ \mathsf{abt}^0 \vdash a\ \mathsf{abt}^n$*, then there exists a unique* $s\ \mathsf{nat}$ *such that*

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

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 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:

$$\frac{}{\mathcal{A}, x\ \mathsf{abt}^0 \vdash x =_\alpha x\ \mathsf{abt}^0} \tag{6.3a}$$

$$\frac{\mathcal{A} \vdash a_1 =_\alpha b_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{A} \vdash a_k =_\alpha b_k\ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash o(a_1, \ldots, a_k) =_\alpha o(b_1, \ldots, b_k)\ \mathsf{abt}^0} \tag{6.3b}$$

$$\frac{\mathcal{A}, z\ \mathsf{abt}^0 \vdash [z \leftrightarrow x]\, a =_\alpha [z \leftrightarrow y]\, b\ \mathsf{abt}^n}{\mathcal{A} \vdash x\,.\,a =_\alpha y\,.\,b\ \mathsf{abt}^{n+1}} \tag{6.3c}$$

In Rule (6.3c) 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:*

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

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 (6.3c). 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 $u, v \# \mathcal{A}$. Let $w \# \mathcal{A}$ be an arbitrary name. By choosing u and v to be w, we obtain the desired result by an application of the inductive hypothesis. □

6.1.3 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:

$$\frac{}{\mathcal{A} \vdash [a/x]x = a\ \mathsf{abt}^0} \tag{6.4a}$$

$$\frac{x \# y}{\mathcal{A} \vdash [a/x]y = y\ \mathsf{abt}^0} \tag{6.4b}$$

$$\frac{\mathcal{A} \vdash [a/x]b_1 = c_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{A} \vdash [a/x]b_k = c_k\ \mathsf{abt}^{n_k}}{\mathcal{A} \vdash [a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k)\ \mathsf{abt}^0} \tag{6.4c}$$

$$\frac{\mathcal{A}, y'\ \mathsf{abt}^0 \vdash [a/x]([y' \leftrightarrow y]\, b) = b'\ \mathsf{abt}^n \quad y' \# \mathcal{A} \quad y' \neq x}{\mathcal{A} \vdash [a/x]y.\, b = y'.\, b'\ \mathsf{abt}^n} \tag{6.4d}$$

In Rule (6.4d) 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 (6.4d), we have $y.\,[y \leftrightarrow y']\, b' =_\alpha y'.\, b'$, provided that $y \# \mathcal{A}$, by Lemma 6.2 on the facing 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 $\mathcal{A}, x \ \mathsf{abt}^0 \vdash b =_\alpha b' \ \mathsf{abt}^n$. □

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.

$$\texttt{let}(x, x.x) =_\alpha \texttt{let}(x, y.y) \tag{6.5a}$$
$$\texttt{let}(y, x.x) =_\alpha \texttt{let}(y, y.y) \tag{6.5b}$$
$$\texttt{let}(x, x.x) =_\alpha \texttt{let}(y, y.y) \tag{6.5c}$$
$$\texttt{let}(x, x.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, z.\texttt{plus}(z, y)) \tag{6.5d}$$
$$\texttt{let}(x, x.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, y.\texttt{plus}(y, y)) \tag{6.5e}$$

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 $\mathcal{L}\{\texttt{num str}\}$ is specified as follows:

Item	itm	::=	kwd | id | num | lit | spl
Keyword	kwd	::=	$\texttt{l}\cdot\texttt{e}\cdot\texttt{t}\cdot\epsilon \mid \texttt{b}\cdot\texttt{e}\cdot\texttt{e}\cdot\epsilon \mid \texttt{i}\cdot\texttt{n}\cdot\epsilon$
Identifier	id	::=	ltr (ltr | dig)*
Numeral	num	::=	dig dig*
Literal	lit	::=	qum (ltr | dig)*qum
Special	spl	::=	+ | * | ^ | (|) | |
Letter	ltr	::=	a | b | ...
Digit	dig	::=	0 | 1 | ...
Quote	qum	::=	"

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:

$$\frac{s\ \textsf{str}}{\texttt{ID}[s]\ \textsf{tok}} \tag{7.1a}$$

$$\frac{n\ \textsf{nat}}{\texttt{NUM}[n]\ \textsf{tok}} \tag{7.1b}$$

$$\frac{s\ \textsf{str}}{\texttt{LIT}[s]\ \textsf{tok}} \tag{7.1c}$$

$$\frac{}{\texttt{LET}\ \textsf{tok}} \tag{7.1d}$$

$$\frac{}{\texttt{BE}\ \textsf{tok}} \tag{7.1e}$$

$$\frac{}{\texttt{IN}\ \textsf{tok}} \tag{7.1f}$$

$$\frac{}{\texttt{ADD}\ \textsf{tok}} \tag{7.1g}$$

$$\frac{}{\texttt{MUL}\ \textsf{tok}} \tag{7.1h}$$

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

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

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

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

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 whs states that the string s consists only of "white space", and s lord states that s is either an alphabetic letter or a digit, and so forth.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\mathcal{L}\{\texttt{num str}\}$ may be specified by a context-free grammar over the tokens defined in Section 7.1 on page 51. The grammar has

only one syntactic class, exp, which is defined by the following compound production:

Expression	exp	::=	num | lit | id | LP exp RP | exp ADD exp | exp MUL exp | exp CAT exp | VB exp VB | LET id BE exp IN exp
Number	num	::=	NUM[n] (n nat)
String	lit	::=	LIT[s] (s str)
Identifier	id	::=	ID[s] (s str)

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

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

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

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

and

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

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 55 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

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

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 $\mathcal{L}\{\texttt{num str}\}$ given in Section 7.3 on page 55 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.

Factor	fct	::=	num | lit | id | `LP` prg `RP`
Term	trm	::=	fct | fct `MUL` trm | `VB` fct `VB`
Expression	exp	::=	trm | trm `ADD` exp | trm `CAT` exp
Program	prg	::=	exp | `LET` id `BE` exp `IN` prg

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:

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

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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$:

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

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

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

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

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

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

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

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

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 $\mathcal{L}\{\texttt{num str}\}$ follow the unambiguous grammar given in Chapter 7:

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

These judgements are inductively defined simultaneously by the following rules:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 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 prg $\longleftrightarrow$ a ast*, and similarly for the other parsing judgements. That is, the parsing judgements have mode* $(\forall, \exists!)$ *over 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 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 60 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:

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

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 61. These rules take the form of a generic inductive definition (see Chapter 2) in which the premises and conclusions of the rules involve hypothetical judgments of the form

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

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

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

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:

$\sigma\ \mathsf{symtab}$	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 $\mathcal{L}\{\texttt{num str}\}$ presented in the tabular style that we shall use

throughout the book:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`num`	`num`
		|	`str`	`str`
Expr	e	::=	x	x
		|	`num[`n`]`	n
		|	`str[`s`]`	`"`s`"`
		|	`plus(`e_1`;`e_2`)`	e_1 `+` e_2
		|	`times(`e_1`;`e_2`)`	e_1 `*` e_2
		|	`cat(`e_1`;`e_2`)`	e_1 `^` e_2
		|	`len(`e`)`	`|`e`|`
		|	`let(`e_1`;`x`.`e_2`)`	`let` x `be` e_1 `in` e_2

This specification is to be understood as defining two judgments, τ type and τ exp, which specify two syntactic categories, 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 category. 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, `num[`n`]` and `str[`s`]`, 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

Most programming languages exhibit a *phase distinction* between the *static* and *dynamic* phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be *safe* exactly when well-formed programs are well-behaved when executed.

The static phase is specified by a *static semantics* comprising a collection of rules for deriving *typing judgements* stating that an expression is well-formed of a certain type. Types mediate the interaction between the constituent parts of a program by "predicting" the execution behavior of the parts so that we may ensure they fit together properly at run-time. Type safety tells us that these predictions are accurate; if not, the static semantics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

In this chapter we present the static semantics of the language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ as an illustration of the methodology that we shall employ throughout this book.

9.1 Type System

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`num`	`num`
		|	`str`	`str`
Expr	e	::=	x	x
		|	`num[`n`]`	n
		|	`str[`s`]`	`"`s`"`
		|	`plus(`e_1`;`e_2`)`	e_1 `+` e_2
		|	`times(`e_1`;`e_2`)`	e_1 `*` e_2
		|	`cat(`e_1`;`e_2`)`	e_1 `^` e_2
		|	`len(`e`)`	`|`e`|`
		|	`let(`e_1`;`x`.`e_2`)`	`let` x `be` e_1 `in` e_2

According to the conventions discussed in Chapter 8, this grammar defines two judgements, τ type defining the category of types, and e exp defining the category of expressions.

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 `plus(`x`;num[`n`])` is sensible depends on whether or not the variable x is declared to have type `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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of an inductive definition of generic 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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ are as follows:

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

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

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

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

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

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

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

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

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.

Rules (9.1) illustrate an important organizational principle, called the *principle of introduction and elimination*, for a type system. The constructs of the language may be classified into one of two forms associated with each type. The *introductory* forms of a type are the means by which values of that type are created, or introduced. In the case of $\mathcal{L}\{\texttt{num str}\}$, the introductory forms for the type `num` are the numerals, `num[`n`]`, and for the type `str` are the literals, `str[`s`]`. The *eliminatory* forms of a type are the means by which we may compute with values of that type to obtain values of some (possibly different) type. In the present case the eliminatory forms for the type `num` are addition and subtraction, and for the type `str` are concatenation and length. Each eliminatory form has one or more *principal* arguments of associated type, and zero or more *non-principal* arguments. In the present case all arguments for each of the eliminatory forms is principal, but we shall later see examples in which there are also non-principal arguments for eliminatory forms.

It is easy to check that every expression has at most one type.

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 it is easy to give necessary conditions for typing an expression that invert the sufficient conditions expressed by the corresponding typing rule.

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 generic judgements. We will focus our attention here on two key properties, the combination of proliferation (Rule (3.8a)) and weakening (Rule (2.12b)), and substitution, which generalizes transitivity (Rule (2.12c)).

Lemma 9.3 (Weakening)**.** *If* $\Gamma \vdash e' : \tau'$*, then* $\Gamma, x : \tau \vdash e' : \tau'$ *for any* $x \# \Gamma$ *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 $e' = \texttt{let}(e_1; z.e_2)$, where by the conventions on parameters we may assume z is chosen such that $z \# \Gamma$ and $z \# x$. 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, $e' = \texttt{let}(e_1; z.e_2)$, where z may be chosen so that $z \# x$, $z \# \Gamma$, and $z \# e$. 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 $z \# e$, we have

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

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

From a programming point of view, Lemma 9.3 on the facing page allows us to use an expression in any context that binds its free variables: if e is well-typed in a context Γ, then we may "import" it into any context that includes the assumptions Γ. In other words the introduction of new variables beyond those required by an expression, e, does not invalidate e itself; it remains well-formed, with the same type.[1] More significantly, Lemma 9.4 on the preceding page expresses the concepts of *modularity* and *linking*. We may think of the expressions e and e' as two *components* of a larger system in which the component e' is to be thought of as a *client* of the *implementation* e. The client declares a variable specifying the type of the implementation, and is type checked knowing only this information. The implementation must be of the specified type in order to satisfy the assumptions of the client. If so, then we may link them to form the composite system, $[e/x]e'$. This may itself be the client of another component, represented by a variable, y, that is replaced by that component during linking. When all such variables have been implemented, the result is a *closed expression* that is ready for execution (evaluation).

The converse of Lemma 9.4 on the facing page is called *decomposition*. It states that any (large) expression may be decomposed into a client and implementor by introducing a variable to mediate their interaction.

Lemma 9.5 (Decomposition)**.** *If* $\Gamma \vdash [e/x]e' : \tau'$*, then for every type* τ *such that* $\Gamma \vdash e : \tau$*, we have* $\Gamma, x : \tau \vdash e' : \tau'$*.*

Proof. The typing of $[e/x]e'$ depends only on the type of e wherever it occurs, if at all. □

This lemma tells us that any sub-expression may be isolated as a separate module of a larger system. This is especially useful in when the variable x occurs more than once in e', because then one copy of e suffices for all occurrences of x in 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 $e : \tau$.

[1] This may seem so obvious as to be not worthy of mention, but, suprisingly, there are useful type systems that lack this property. Since they do not validate the structural principle of weakening, they are called *sub-structural* type systems.

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

A structural semantics for $\mathcal{L}\{\texttt{num str}\}$ 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.

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

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

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

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

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

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

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

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.

Rules (10.2) exhibit structure arising from the principle of introduction and elimination discussed in Chapter 9. The instruction transitions express the *inversion principle*, which states that *eliminatory forms are inverse to introductory forms*. For example, Rule (10.2a) extracts the natural number from the introductory forms of its arguments, adds these two numbers, and yields the corresponding numeral as result. The search transitions specify that the principal arguments of each eliminatory form are to be evaluated. (When non-principal arguments are present, which is not the case here, there is discretion about whether to evaluate them or not.) This is essential, because it prepares for the instruction transitions, which expect their principal arguments to be introductory forms.

Rule (10.3a) specifies a *by-name* interpretation, in which the bound variable stands for the expression e_1 itself.[1] If x does not occur in e_2, the expression e_1 is never evaluated. If, on the other hand, it occurs more than once, then e_1 will be re-evaluated at each occurence. To avoid repeated work in the latter case, we may instead specify a *by-value* interpretation of binding by the following rules:

$$\frac{e_1 \text{ val}}{\texttt{let}(e_1;x\texttt{.}e_2) \mapsto [e_1/x]e_2} \tag{10.3a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x\texttt{.}e_2) \mapsto \texttt{let}(e_1';x\texttt{.}e_2)} \tag{10.3b}$$

[1] The justification for the phrase "by name" is obscure, but the terminology is well-established.

Rule (10.3b) is an additional search rule specifying that we may evaluate e_1 before e_2. Rule (10.3a) ensures that e_2 is not evaluated until evaluation of e_1 is complete.

A derivation sequence in a structural semantics 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 "height." For example, consider the following evaluation sequence.

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

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:

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

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

The principle of rule induction for the structural semantics of $\mathcal{L}\{\texttt{num str}\}$ states that to show $\mathcal{P}(e \mapsto e')$ whenever $e \mapsto e'$, it is sufficient to show that $\mathcal{P}$ is closed under Rules (10.2). For example, we may show by rule induction that structural semantics of $\mathcal{L}\{\texttt{num str}\}$ is *determinate*.

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

Proof. By rule induction on the premises $e \mapsto e'$ and $e \mapsto e''$, carried out either simultaneously or in either order. Since only one rule applies to each form of expression, e, the result follows directly in each case. □

10.2 Contextual Semantics

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, $e_1 \rightsquigarrow e_2$, 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.

$$\frac{m + n = p \text{ nat}}{\texttt{plus(num[}m\texttt{];num[}n\texttt{])} \rightsquigarrow \texttt{num[}p\texttt{]}} \tag{10.4a}$$

$$\frac{s \hat{} t = u \text{ str}}{\texttt{cat(str[}s\texttt{];str[}t\texttt{])} \rightsquigarrow \texttt{str[}u\texttt{]}} \tag{10.4b}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \rightsquigarrow [e_1/x]e_2} \tag{10.4c}$$

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.)

$$\frac{}{\circ \text{ ectxt}} \tag{10.5a}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\texttt{plus(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \text{ ectxt}} \tag{10.5b}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\texttt{plus(}e_1\texttt{;}\mathcal{E}_2\texttt{)} \text{ ectxt}} \tag{10.5c}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\texttt{cat(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \text{ ectxt}} \tag{10.5d}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\texttt{cat(}e_1\texttt{;}\mathcal{E}_2\texttt{)} \text{ ectxt}} \tag{10.5e}$$

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 (10.5c) states that in an expression $\texttt{plus(}e_1\texttt{;}e_2\texttt{)}$, 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:

$$\frac{}{e = \circ\{e\}} \tag{10.6a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathtt{plus}(e_1;e_2) = \mathtt{plus}(\mathcal{E}_1;e_2)\{e\}} \tag{10.6b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\mathtt{plus}(e_1;e_2) = \mathtt{plus}(e_1;\mathcal{E}_2)\{e\}} \tag{10.6c}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathtt{cat}(e_1;e_2) = \mathtt{cat}(\mathcal{E}_1;e_2)\{e\}} \tag{10.6d}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\mathtt{cat}(e_1;e_2) = \mathtt{cat}(e_1;\mathcal{E}_2)\{e\}} \tag{10.6e}$$

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 $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is defined using contextual semantics by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{10.7}$$

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 (10.7)).

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 $e_0 \rightsquigarrow e_0'$. For example, for Rule (10.2a), take $\mathcal{E} = \circ$, and observe that $e \rightsquigarrow e'$. 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 $e_0 \rightsquigarrow e_0'$. Take $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1;e_2)$, and observe that $e = \mathtt{plus}(\mathcal{E}_1;e_2)\{e_0\}$ and $e' = \mathtt{plus}(\mathcal{E}_1;e_2)\{e_0'\}$ with $e_0 \rightsquigarrow e_0'$.

From right to left, observe that if $e \mapsto_{\mathtt{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. We prove by induction on Rules (10.6) that $e \mapsto_{\mathtt{s}} e'$. For example, for Rule (10.6a), e_0 is e, e_0' is e', and $e \rightsquigarrow e'$. Hence $e \mapsto_{\mathtt{s}} e'$. For Rule (10.6b), we have that $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathtt{s}} e_1'$. Therefore e is $\mathtt{plus}(e_1; e_2)$, e' is $\mathtt{plus}(e_1'; e_2)$, and therefore by Rule (10.2b), $e \mapsto_{\mathtt{s}} e'$.

□

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 (10.7) in the simpler form

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}\ . \tag{10.8}$$

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 15, 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 Equational Semantics

Another formulation of the dynamic semantics of a language is based on regarding computation as a form of equational deduction, much in the style of elementary algebra. For example, in algebra we may show that the polynomials $x^2 + 2\,x + 1$ and $(x + 1)^2$ are equivalent by a simple process of calculation and re-organization using the familiar laws of addition and multiplication. The same laws are sufficient to determine the value of any polynomial, given the values of its variables. So, for example, we may plug in 2 for x in the polynomial $x^2 + 2\,x + 1$ and calculate that $2^2 + 2\,2 + 1 = 9$, which is indeed $(2 + 1)^2$. This gives rise to a model of computation in which we may determine the value of a polynomial for a given value of its variable by substituting the given value for the variable and proving that the resulting expression is equal to its value.

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where Γ consists of one assumption of the form $x : \tau$ for each

$x \in \mathcal{X}$. We only consider definitional equality of well-typed expressions, so that when considering the judgement $\Gamma \vdash e \equiv e' : \tau$, we tacitly assume that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Here, as usual, we omit explicit mention of the parameters, $\mathcal{X}$, when they can be determined from the forms of the assumptions Γ.

Definitional equivalence of expressons in $\mathcal{L}\{\texttt{num str}\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{10.9a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{10.9b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{10.9c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) \equiv \texttt{plus}(e_1'; e_2') : \texttt{num}} \tag{10.9d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) \equiv \texttt{cat}(e_1'; e_2') : \texttt{str}} \tag{10.9e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv \texttt{let}(e_1'; x.e_2') : \tau_2} \tag{10.9f}$$

$$\frac{n_1 + n_2 = n \text{ nat}}{\Gamma \vdash \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \equiv \texttt{num}[n] : \texttt{num}} \tag{10.9g}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\Gamma \vdash \texttt{cat}(\texttt{str}[s_1]; \texttt{str}[s_2]) \equiv \texttt{str}[s] : \texttt{str}} \tag{10.9h}$$

$$\frac{}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{10.9i}$$

Rules (10.9a) through (10.9c) state that definitional equivalence is an *equivalence relation*. Rules (10.9d) through (10.9f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (10.9g) through (10.9i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num str}\}$. For the sake of concision, Rules (10.9) may be characterized as defining the *strongest congruence* closed under Rules (10.9g), (10.9h), and (10.9i).

Rules (10.9) are sufficient to allow us to calculate the value of an expression by an equational deduction similar to that used in high school algebra. For example, we may derive the equation

$$\texttt{let}\ x\ \texttt{be}\ 1 + 2\ \texttt{in}\ x + 3 + 4 \equiv 10 : \texttt{num}$$

by applying Rules (10.9). Here, as in general, there may be many different ways to derive the same equation, but we need find only one derivation in order to carry out an evaluation.

Definitional equivalence is rather weak in that many equivalences that one might intuitively think are true are not derivable from Rules (10.9). A prototypical example is the putative equivalence

$$x : \mathtt{num}, y : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{10.10}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from Rules (10.9). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{10.11}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (10.10), and all of its instances, given by Equation (10.11), may be filled by enriching the notion of equivalence to include a principal of proof by mathematical induction. Such a notion of equivalence is sometimes called *semantic*, or *observational*, *equivalence*, since it expresses relationships that hold by virtue of the semantics of the expressions involved.[2] Semantic equivalence is a *synthetic judgement*, one that requires proof. It is to be distinguished from definitional equivalence, which expresses an *analytic judgement*, one that is self-evident based solely on the dynamic semantics of the operations involved. As such definitional equivalence may be thought of as *symbolic evaluation*, which permits simplification according to the evaluation rules of a language, but which does not permit reasoning by induction.

Definitional equivalence is adequate for evaluation in that it permits the calculation of the value of any closed expression.

Theorem 10.3. *$e \equiv e' : \tau$ iff there exists e_0 val such that $e \mapsto^* e_0$ and $e' \mapsto^* e_0$.*

Proof. The proof from right to left is direct, since every transition step is a valid equation. The converse follows from the following, more general, proposition. If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then whenever $e_1 : \tau_1, \ldots, e_n : \tau_n$, if

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \equiv [e_1, \ldots, e_n / x_1, \ldots, x_n]e' : \tau,$$

then there exists e_0 val such that

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \mapsto^* e_0$$

[2] This rather vague concept of equivalence is developed rigorously in Chapter 50.

and

$$[e_1, \ldots, e_n / x_1, \ldots, x_n] e' \mapsto^* e_0.$$

This is proved by rule induction on Rules (10.9). □

The formulation of definitional equivalence for the by-value semantics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (10.9i) to express the requirement that e_1 be a value? As a first cut one might consider simply adding an additional premise to the rule:

$$\frac{e_1 \ \mathsf{val}}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{10.12}$$

This is almost correct, except that the judgement e val is defined only for *closed* expressions, whereas e_1 might well involve free variables in Γ. What is required is to extend the judgement e val to the hypothetical judgement

$$x_1 \ \mathsf{val}, \ldots, x_n \ \mathsf{val} \vdash e \ \mathsf{val}$$

in which the hypotheses express the assumption that variables are only ever bound to values, and hence can be regarded as values. To maintain this invariant, we must maintain a set, Ξ, of such hypotheses as part of definitional equivalence, writing $\Xi\,\Gamma \vdash e \equiv e' : \tau$, and modifying Rule (10.9f) as follows:

$$\frac{\Xi\,\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Xi, x \ \mathsf{val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{10.13}$$

The other rules are correspondingly modified to simply carry along Ξ is an additional set of hypotheses of the inference.

10.4 Exercises

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

2. Formulate a variation of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ 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 72, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 9.4 on page 72 $[e_1/x]e_2 : \tau_2$, as desired. □

The proof of preservation is naturally structured as an induction on the transition judgement, since the argument hinges on examining all possible transitions from a given expression. In some cases one may manage to carry out a proof by structural induction on e, or by an induction on typing, but experience shows that this often leads to awkward arguments, or, in some cases, cannot be made to work at all.

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 $e\ \mathsf{val}$ and $e : \tau$, then*

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

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

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 $\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}$, 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 $\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}$, as required. In the former, we have, by the Canonical Forms Lemma 11.3 on the preceding page, $e_1 = \texttt{num[}n_1\texttt{]}$ and $e_2 = \texttt{num[}n_2\texttt{]}$, and hence

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

□

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 $\mathcal{L}\{\texttt{num str}\}$ 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 `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 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:

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

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

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

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`:

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

This rule destroys the unicity of typing property (Lemma 9.1 on page 71). 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.

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

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

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

There are similar error propagation rules for the other constructs of the language. By defining e err to hold exactly when $e = \mathtt{error}$, 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.

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

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

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

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

$$\frac{e \Downarrow \texttt{str}[s] \quad |s| = n \text{ str}}{\texttt{len}(e) \Downarrow \texttt{num}[n]} \tag{12.1e}$$

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

The value of a `let` expression is determined by substitution of the binding into the body. The rules are therefore not syntax-directed, since the premise of Rule (12.1f) is not a sub-expression of the expression in the conclusion of that rule.

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 $\mathcal{P}(\texttt{plus}(e_1;e_2) \Downarrow \texttt{num[}n\texttt{]})$, 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 $\mathcal{P}(\texttt{cat}(e_1;e_2) \Downarrow \texttt{str[}s\texttt{]})$, 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 $\mathcal{P}(\texttt{let}(e_1;x.e_2) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e 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 (12.1f) 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 $\mathcal{L}\{\texttt{num str}\}$. 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 $\texttt{plus}(e_1;e_2) \Downarrow \texttt{num}[n]$ by the rule for evaluating additions. By induction we know that $e_1 \mapsto^* \texttt{num}[n_1]$ and $e_2 \mapsto^* \texttt{num}[n_2]$. We reason as follows:

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

Therefore $\texttt{plus}(e_1;e_2) \mapsto^* \texttt{num}[n_1 + n_2]$, 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 $\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)$, where $e_1 \mapsto e_1'$. Suppose further that $\texttt{plus}(e_1';e_2) \Downarrow v$, so that $e_1' \Downarrow \texttt{num}[n_1]$, $e_2 \Downarrow \texttt{num}[n_2]$, $n_1 + n_2 = n$ nat, and v is $\texttt{num}[n]$. By induction $e_1 \Downarrow \texttt{num}[n_1]$, and hence $\texttt{plus}(e_1;e_2) \Downarrow \texttt{num}[n]$, as required. □

12.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 11.1 on page 85) 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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ were extended to admit operations that may result in an error (as discussed in Section 11.3 on page 88), 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:

$$\frac{}{\texttt{plus(str[}s\texttt{];}e_2\texttt{)} \Uparrow} \tag{12.2a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus(}e_1\texttt{;str[}s\texttt{])} \Uparrow} \tag{12.2b}$$

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

Theorem 12.5. *If $e \Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (12.2). For example, for Rule (12.2a), we observe that $\texttt{str[}s\texttt{]} : \texttt{str}$, and hence $\texttt{plus(str[}s\texttt{];}e_2\texttt{)}$ is ill-typed. □

Corollary 12.6. *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 12.5 on the preceding page 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.

12.4 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.

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

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

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

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

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

Theorem 12.7. *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 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 a data structure where they may be retrieved on demand. In this section we show how this can be expressed for a by-value interpretation of binding using hypothetical judgements. It is also possible to formulate an environment semantics for the by-name interpretation, at the cost of some additional complexity (see Chapter 42 for a full discussion of the issues involved).

The basic idea is to consider hypotheses of the form $x \Downarrow v$, where x is a variable and v is a closed value, such that no two hypotheses govern the same variable. Let Θ range over finite sets of such hypotheses, which we call an *environment*. We will consider judgements of the form $\Theta \vdash e \Downarrow v$, where Θ is an environment governing some finite set of variables.

$$\frac{}{\Theta, x \Downarrow v \vdash x \Downarrow v} \tag{12.4a}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{num}[n_1] \quad \Theta \vdash e_2 \Downarrow \mathtt{num}[n_2]}{\Theta \vdash \mathtt{plus}(e_1; e_2) \Downarrow \mathtt{num}[n_1 + n_2]} \tag{12.4b}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{str}[s_1] \quad \Theta \vdash e_2 \Downarrow \mathtt{str}[s_2]}{\Theta \vdash \mathtt{cat}(e_1; e_2) \Downarrow \mathtt{str}[s_1 \hat{} s_2]} \tag{12.4c}$$

$$\frac{\Theta \vdash e_1 \Downarrow v_1 \quad \Theta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\Theta \vdash \mathtt{let}(e_1; x.e_2) \Downarrow v_2} \tag{12.4d}$$

Rule (12.4a) is an instance of the general reflexivity rule for hypothetical judgements. The `let` rule augments the environment with a new assumption governing the bound variable, which may be chosen to be distinct from all other variables in Θ to avoid multiple assumptions for the same variable.

The environment semantics implements evaluation by deferred substitution.

Theorem 12.8. $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ *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. □

12.6 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?

Part IV

Function Types

Chapter 13

Function Definitions and Values

In $\mathcal{L}\{\texttt{num str}\}$ it is possible to express doubling of any given expression of type `num`, but it is not possible to express the general concept of doubling so that it may be defined once and used many times in a computation. A *function* captures a "pattern" of computation by using *variables* to stand for unknowns. An *instantiation*, or *application*, of a function specifies the bindings of the unknowns to obtain an instance of the general pattern. A *function definition* gives a name to the pattern so that it may be instantiated more than once in a computation.

Having introduce function definitions, the question arises as to how these relate to expression definitions introduced by `let`. The key difference is that *functions are not expressions*, but rather *expression patterns*, and hence cannot be seen as uses of the `let` construct. By introducing a function as a form of expression we may consolidate the two mechanisms into one. To do so we must introduce the *function type*, which classifies expression patterns by specifying the types of their parameters (ther *domains*) and of their results (their *ranges*).

One consequence of introducing a function type is that functions are *first-class values*, which means that they may be manipulated like any other value in an expression. In particular, they may be passed as arguments to other functions, and returned as results from them. A function type whose domain or range is itself a function type is called a *higher-order* function type to distinguish it from a *first-order* function type, whose domain and range consists only of atomic data such as numbers and strings. A language with higher-order function types is called a *higher-order language*, and one that

has only first-order functions is called a *first-order language*. Higher-order languages are surprisingly powerful. Correspondingly, they raise subtle issues that are easily overlooked, and that have historically given rise to language design errors.

13.1 First-Order Functions

The language $\mathcal{L}\{\texttt{num str fun}\}$ is the extension of $\mathcal{L}\{\texttt{num str}\}$ with function definitions and function applications as described by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)}$	$\texttt{fun}\, f\texttt{(}x_1\texttt{:}\tau_1\texttt{):}\tau_2\texttt{=}e_2\,\texttt{in}\,e$
		$\mid$	$\texttt{call[}f\texttt{](}e\texttt{)}$	$f\texttt{(}e\texttt{)}$

The variable f ranges over a distinguished class of variables, called *function names*, which are assumed to be disjoint from ordinary variables. The expression $\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)}$ binds the function name f within e to the pattern $x_1\texttt{.}e_2$, which has parameter x_1 and definition e_2. The domain and range of the function are, respectively, the types τ_1 and τ_2. The expression $\texttt{call[}f\texttt{](}e\texttt{)}$ instantiates the abstractor bound to f with the argument e.

The static semantics of $\mathcal{L}\{\texttt{num str fun}\}$ consists of judgements of the form $\Gamma \vdash e : \tau$, where Γ consists of hypotheses of one of two forms:

1. $x : \tau$, declaring the type of a variable x to be τ;
2. $f\texttt{(}\tau_1\texttt{)} : \tau_2$, declaring that f is a function name with domain τ_1 and range τ_2.

The second form of assumption is sometimes called a *function header*, since it resembles the concrete syntax of the first part of a function definition. The static semantics is defined in terms of these hypotheses by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f\texttt{(}\tau_1\texttt{)} : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)} : \tau} \tag{13.1a}$$

$$\frac{\Gamma, f\texttt{(}\tau_1\texttt{)} : \tau_2 \vdash e : \tau_1}{\Gamma, f\texttt{(}\tau_1\texttt{)} : \tau_2 \vdash \texttt{call[}f\texttt{](}e\texttt{)} : \tau_2} \tag{13.1b}$$

The structural property of substitution takes an unusual form that matches the form of the hypotheses governing function names. The operation of *function substitution*, written $[\![x\texttt{.}e \,/\, f]\!]e'$, is inductively defined similarly to

ordinary substitution, but bearing in mind that the function name, f, may only occur within e' as part of a function call. The rule governing such occurrences is given as follows:

$$\overline{[\![x\,.\,e\;/\;f]\!]\texttt{call}[f](e') = \texttt{let}(e';x\,.\,e)} \tag{13.2}$$

That is, at call sites to f, we bind x to e' within e to instantiate the pattern substituted for f.

Lemma 13.1. *If* $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1\,.\,e_2\;/\;f]\!]e : \tau$.

Proof. By induction on the structure of e'. □

The dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ is easily defined using function substitution:

$$\overline{\texttt{fun}[\tau_1;\tau_2](x_1\,.\,e_2;f\,.\,e) \mapsto [\![x_1\,.\,e_2\;/\;f]\!]e} \tag{13.3}$$

Observe that the use of function substitution eliminates all applications of f within e, so that no rule is required for evaluating them. This rule imposes either a *call-by-name* or a *call-by-value* application discipline according to whether the `let` binding is given a by-name or a by-value interpretation.

We leave it as an exercise to state and prove the safety of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$.

13.2 Higher-Order Functions

The syntactic and semantic similarity between variable definitions and function definitions in $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ is striking. This suggests that it may be possible to consolidate the two concepts into a single definition mechanism. The gap that must be bridged is the segregation of functions from expressions. A function name f is bound to an abstractor $x\,.\,e$ specifying a pattern that is instantiated when f is applied. To consolidate function definitions with expression definitions it is sufficient to *reify* the abstractor into a form of expression, called a *λ-abstraction*, written $\texttt{lam}[\tau_1](x\,.\,e)$. Corresponingly, we must generalize application to have the form $\texttt{ap}(e_1;e_2)$, where e_1 is any expression, and not just a function name. These are, respectively, the introduction and elimination forms for the *function type*, $\texttt{arr}(\tau_1;\tau_2)$, whose elements are functions with domain τ_1 and range τ_2.

The language $\mathcal{L}\{\texttt{num str} \to\}$ is the enrichment of $\mathcal{L}\{\texttt{num str}\}$ with function types, as specified by the following grammar:

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

The static semantics of $\mathcal{L}\{\texttt{num str} \to\}$ is given by extending Rules (9.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{13.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{13.4b}$$

Lemma 13.2 (Inversion)**.** *Suppose that* $\Gamma \vdash e : \tau$.

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

Lemma 13.3 (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. □

The dynamic semantics of $\mathcal{L}\{\texttt{num str} \to\}$ extends that of $\mathcal{L}\{\texttt{num str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \texttt{val}} \tag{13.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{13.5b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1} \tag{13.5c}$$

These rules specify a call-by-name discipline for function application. It is a good exercise to formulate a call-by-value discipline as well.

Theorem 13.4 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. The proof is by induction on rules (13.5), which define the dynamic semantics of the language.

Consider rule (13.5c),

$$\frac{}{\texttt{ap(lam[}\tau_2\texttt{]}\,(x\texttt{.}\,e_1)\texttt{;}e_2\texttt{)} \mapsto [e_2/x]e_1}\ .$$

Suppose that $\texttt{ap(lam[}\tau_2\texttt{]}\,(x\texttt{.}\,e_1)\texttt{;}e_2\texttt{)} : \tau_1$. By Lemma 13.2 on the facing page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 13.3 on the preceding page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 13.5 (Canonical Forms). *If e val and $e : \texttt{arr}(\tau_1;\tau_2)$, then $e = \texttt{lam[}\tau_1\texttt{]}\,(x\texttt{.}\,e_2)$ 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 13.6 (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 (13.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (13.4b). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}$. In the former case, we have by Lemma 13.5 that $e_1 = \texttt{lam[}\tau_2\texttt{]}\,(x\texttt{.}\,e)$ for some x and e. But then $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto [e_2/x]e$. □

13.3 Evaluation Semantics and Definitional Equivalence

An inductive definition of the evaluation judgement $e \Downarrow v$ for $\mathcal{L}\{\texttt{num str} \to\}$ is given by the following rules:

$$\frac{}{\texttt{lam[}\tau\texttt{]}\,(x\texttt{.}\,e) \Downarrow \texttt{lam[}\tau\texttt{]}\,(x\texttt{.}\,e)} \tag{13.6a}$$

$$\frac{e_1 \Downarrow \texttt{lam[}\tau\texttt{]}\,(x\texttt{.}\,e) \quad [e_2/x]e \Downarrow v}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \tag{13.6b}$$

It is easy to check that if $e \Downarrow v$, then v val, and that if e val, then $e \Downarrow e$.

Theorem 13.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (13.6). The proof makes use of a *pasting lemma* stating that, for example, if $e_1 \mapsto^* e_1'$, then $\mathtt{ap}(e_1; e_2) \mapsto^* \mathtt{ap}(e_1'; e_2)$, 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 *converse evaluation lemma*, which states that if $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. This is proved by rule induction on Rules (13.5). □

Definitional equivalence for the call-by-name semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\to\}$ is defined by a straightforward extension to Rules (10.9).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{13.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2') : \tau} \tag{13.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{13.7c}$$

Definitional equivalence for the call-by-value semantics requires that we maintain hypotheses, Ξ, governing variables. The crucial rules are as follows:

$$\frac{}{\Xi \vdash \mathtt{lam}[\tau_1](x.e_2)\ \mathsf{val}} \tag{13.8a}$$

$$\frac{\Xi, x\ \mathsf{val}\, \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\, \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{13.8b}$$

$$\frac{\Xi \vdash e_1\ \mathsf{val}}{\Xi\, \Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau}\,. \tag{13.8c}$$

13.4 Static and Dynamic Binding

It is surprisingly difficult, and the source of a classic mistake in language design, to give an environment semantics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\to\}$. Recall that the by-value environment semantics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ given in Chapter 12 consists of judgements of the form $\Theta \vdash e \Downarrow v$, where Θ is a finite set of hypotheses $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n$ specifying the values of the variables that may occur in e. Let us naïvely extend this semantics to higher-order functions:

$$\frac{}{\Theta \vdash \mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)} \tag{13.9a}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad \Theta \vdash e_2 \Downarrow v_2 \quad \Theta, x \Downarrow v_2 \vdash e \Downarrow v}{\Theta \vdash \mathtt{ap}(e_1; e_2) \Downarrow v} \tag{13.9b}$$

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 (13.9b) that the variable x lie apart from Θ; this condition may always be met by choosing a suitable representative of the α-equivalence class of the function.

This semantics may seem reasonable at first glance, but it is *incorrect* in that it does not agree with the dynamic semantics given by Rules (13.5). As we shall see shortly, an environment semantics for a higher-order language must distinguish between the *scope* and the *extent* of a variable. Recall from Chapter 6 that the scope of a variable is its *static* range of significance during which the variable has meaning as a reference to a binding site. The *extent* of a variable, on the other hand, is its *dynamic* range of significance during which the binding of the variable is relevant to the progress of a computation. Bear in mind, however, that the concept of extent has no meaning for the structural or evaluation semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\,\rightarrow\}$, and hence cannot be regarded as an intrinsic property of the language.

To see what is wrong with the proposed environment semantics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\,\rightarrow\}$, consider the following expression, e_1:

$$\lambda\,(x\!:\!\mathtt{num}.\,\lambda\,(y\!:\!\mathtt{num}.\,x))\,(3).$$

According to Rules (13.6) (or, for that matter, Rules (13.5)), this expression evaluates to

$$\lambda\,(y\!:\!\mathtt{num}.\,3).$$

Consequently, the application $e_1(4)$ evaluates to 3. Moreover, this outcome is not influenced by the presence (or absence) of any surrounding binding for the variable x. In particular, the expression, e_2, given by

$$\mathtt{let}\,x\!:\!\mathtt{num}\,\mathtt{be}\,5\,\mathtt{in}\,e(4)$$

also evaluates to 3.

If we evaluate e_1 using Rules (13.9), then we may derive the nonsensical judgment $e \Downarrow \lambda\,(y\!:\!\mathtt{num}.\,x)$, which involves a value containing a free variable, x, that is not governed by any assumption about its value. This leads immediately to trouble. Indeed, we need only evaluate the expression e_2 using Rules (13.9) to see what goes wrong. Doing so leads to the evaluation of the application $e_1(3)$ under the assumption $x \Downarrow 5$. This requires evaluation of e_1, resulting in $\lambda\,(y\!:\!\mathtt{num}.\,x)$, and thence to evaluation of x under the assumption $y \Downarrow 4$. Since the assumption $x \Downarrow 5$ is in force, the upshot is the erroneous assertion $e_2 \Downarrow 5$.

The source of the difficulty is that a function value may contain a free variable that is no longer in scope, yet its binding in the environment is still relevant to the progress of the computation. The naïve environment semantics employs a stack-like discipline for the environment, dropping bindings when the scope of the variable that they govern is exited. But if the value of the expression evaluated under the influence of this binding contains a free variable, this stack-like management of the environment is inappropriate—the binding of the variable must be maintained even after evaluation leaves its scope.

One way to ensure this is to substitute the binding of a variable in the returned value whenever the scope of that variable is exited. This is expressed by the following rule:

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{lam[\tau]}(x.e) \quad \Theta \vdash e_2 \Downarrow v_2 \quad \Theta, x \Downarrow v_2 \vdash e \Downarrow v}{\Theta \vdash \mathtt{ap}(e_1; e_2) \Downarrow [v_2/x]v}\ . \tag{13.10}$$

The use of substitution is, however, self-defeating in that the entire purpose of an environment semantics is to avoid substitution! In practice the substitution is not actually performed, but is rather deferred until the value is needed by preserving the bindings in a data structure, called a *closure*. The closure is dynamically allocated, and requires storage at least proportional to the number of free variables in the returned value, which can be significant.

Another way to resolve this discrepancy is to deem the bug to be a feature, called *dynamic binding*. The idea is to allow the binding of a variable to be determined by whatever assumption, if any, is in force at the moment the variable is needed. The main difficulty with this concept is that it violates the fundamental precept that the names of bound variables do not matter. Consider the expression, e_2', given by

$$\mathtt{let}\ x\mathtt{:num\ be}\ 5\ \mathtt{in}\ e_1'(4),$$

where e_1' differs from e_1 only in that the name of the bound variable, x, has been changed to x', obtaining

$$\lambda(x'\mathtt{:num}.\,\lambda(y\mathtt{:num}.\,x'))(3).$$

As we have already seen, $e_2 \Downarrow 5$. In contrast, however, the expression e_2' aborts, because the variable x' is unbound at the point at which its value is required.

A less obvious consequence is that dynamic binding is not even type safe! Consider the expression, e_3, given by

$$\texttt{let}\ x\texttt{:str be "}abc\texttt{" in}\ e_1\texttt{(4)}.$$

The expression e_3 differs from e_2 only in that the type of the `let`-bound variable, x, is `str`, rather than `num`. According to Rules (13.4), the type of e_3 is `num`, yet it evaluates to "*abc*".

Despite these shortcomings, dynamic binding nevertheless has its proponents. Advocates of dynamic binding argue that it is a convenient method for specializing the behavior of higher-order functions without having to pass additional arguments to it. The idea is that if a function has a free variable in it, then by binding that variable to a value one fixes the meaning of the function for the lifetime of that binding. Rather than disrupt the concepts of binding and scope, a better approach is to simply distinguish two different uses of identifiers, as statically scoped variables and as dynamically bound symbols. This possibility is the subject of Chapter 34.

13.5 Exercises

Chapter 14

Gödel's System T

The language $\mathcal{L}\{\mathtt{nat} \to\}$, better known as *Gödel's System* $\boldsymbol{T}$, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\mathtt{nat} \to\}$ 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 $\mathcal{L}\{\mathtt{nat} \to\}$ is that it permits the definition only of *total* functions, those that assign a value in the range type to every element of the domain type. This means that programs written in $\mathcal{L}\{\mathtt{nat} \to\}$ 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 $\mathcal{L}\{\mathtt{nat} \to\}$.

14.1 Statics

The syntax of $\mathcal{L}\{\mathtt{nat} \to\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\mathtt{nat}$	$\mathtt{nat}$
		|	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$
Expr	e	::=	x	x
		|	$\mathtt{z}$	$\mathtt{z}$
		|	$\mathtt{s}(e)$	$\mathtt{s}(e)$
		|	$\mathtt{rec}[\tau](e;e_0;x.y.e_1)$	$\mathtt{rec}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}$
		|	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		|	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$

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 16) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The static semantics of $\mathcal{L}\{\mathtt{nat} \to\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}} \tag{14.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\tau](e;e_0;x.y.e_1) : \tau} \tag{14.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \mathtt{lam}[\sigma](x.e) : \mathtt{arr}(\sigma;\tau)} \tag{14.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{14.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

Lemma 14.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

14.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ adopts a call-by-name interpretation of function application, and requires that the successor operation evaluate its argument (so that values of type `nat` are numerals).

The closed values of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ are determined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{14.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{14.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{14.2c}$$

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is given by the following rules:

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{14.3a}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{14.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{14.3c}$$

$$\frac{e \mapsto e'}{\mathtt{rec}[\tau](e;e_0;x.y.e_1) \mapsto \mathtt{rec}[\tau](e';e_0;x.y.e_1)} \tag{14.3d}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{z};e_0;x.y.e_1) \mapsto e_0} \tag{14.3e}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{s}(e);e_0;x.y.e_1) \mapsto [e,\mathtt{rec}[\tau](e;e_0;x.y.e_1)/x,y]e_1} \tag{14.3f}$$

Rules (14.3e) and (14.3f) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case the recursor evaluates e_0, and in the latter case the variable x is bound to the predecessor, e, and y is bound to the (unevaluated) recursion on e. If the value of y is not required in the rest of the computation, the recursive call will not be evaluated.

Lemma 14.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{nat}$, then $e = \mathtt{s(s(\ldots z))}$ for some number $n \geq 0$ occurrences of the successor starting with zero.*
2. *If $\tau = \tau_1 \to \tau_2$, then $e = \lambda\,(x{:}\tau_1.\, e_2)$ for some e_2.*

Theorem 14.3 (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'*

The foregoing dynamics gives rise to the concept of definitional equivalence, written $\Gamma \vdash e \equiv e' : \tau$. This judgement is defined to be the strongest congruence containing the following axioms:

$$\overline{\Gamma \vdash \mathtt{ap(lam[\tau](x.e_2);e_1)} \equiv [e_1/x]e_2 : \tau} \tag{14.4a}$$

$$\overline{\Gamma \vdash \mathtt{rec[\tau](z;e_0;x.y.e_1)} \equiv e_0 : \tau} \tag{14.4b}$$

$$\overline{\Gamma \vdash \mathtt{rec[\tau](s(e);e_0;x.y.e_1)} \equiv [e,\mathtt{rec[\tau](e;e_0;x.y.e_1)}/x,y]e_1 : \tau} \tag{14.4c}$$

It may be shown that if $e : \mathtt{nat}$, then $e \equiv \overline{n} : \mathtt{nat}$ iff $e \mapsto^* \overline{n}$.

14.3 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 $n \in \mathbb{N}$,

$$e_f(\overline{n}) \equiv \overline{f(n)} : \mathtt{nat}. \tag{14.5}$$

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

$$e_d = \lambda(x{:}\mathtt{nat}.\,\mathtt{rec}\,x\,\{\mathtt{z} \Rightarrow \mathtt{z} \mid \mathtt{s}(u)\ \mathtt{with}\ v \Rightarrow \mathtt{s(s}(v))\}).$$

To see this, observe that $e_d(\overline{0}) \equiv \overline{0} : \mathtt{nat}$, and, assuming that $e_d(\overline{n}) \equiv \overline{d(n)} : \mathtt{nat}$, that

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) \\ &\equiv \mathtt{s}(\mathtt{s}(\overline{2 \times n})) \\ &= \overline{2 \times (n+1)} \\ &= \overline{d(n+1)}. \end{aligned}$$

As another example, consider the following function, called *Ackermann's function*, defined by the following equations:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)). \end{aligned}$$

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. 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

$$\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the λ-abstraction

$$\lambda(f\!:\!\mathtt{nat} \to \mathtt{nat}.\, \lambda(n\!:\!\mathtt{nat}.\, \mathtt{rec}\, n\, \{\mathtt{z} \Rightarrow \mathtt{id} \mid \mathtt{s}(\_)\ \mathtt{with}\ g \Rightarrow f \circ g\})),$$

where $\mathtt{id} = \lambda(x\!:\!\mathtt{nat}.\, x)$ is the identity, and $f \circ g = \lambda(x\!:\!\mathtt{nat}.\, f(g(x)))$ is the composition of f and g. It is easy to check that

$$\mathtt{it}(f)(\overline{n})(\overline{m}) \equiv f^{(n)}(\overline{m}) : \mathtt{nat},$$

where the latter expression is the n-fold composition of f starting with $\overline{m}$. We may then define the Ackermann function

$$e_a : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the expression

$$\lambda(m{:}\mathtt{nat}.\,\mathtt{rec}\,m\,\{\mathtt{z} \Rightarrow \mathtt{succ} \mid \mathtt{s}(\_)\ \mathtt{with}\ f \Rightarrow \lambda(n{:}\mathtt{nat}.\,\mathtt{it}(f)(n)(f(\overline{1})))\}).$$

It is instructive to check that the following equivalences are valid:

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{14.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{14.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{14.8}$$

That is, the Ackermann function is definable in $\mathcal{L}\{\mathtt{nat} \to\}$.

14.4 Non-Definability

It is impossible to define an infinite loop in $\mathcal{L}\{\mathtt{nat} \to\}$.

Theorem 14.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 50.9 on page 424. □

Consequently, values of function type in $\mathcal{L}\{\mathtt{nat} \to\}$ behave like mathematical functions: if $f : \sigma \to \tau$ and $e : \sigma$, then $f(e)$ 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 $\mathcal{L}\{\mathtt{nat} \to\}$. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of $\mathcal{L}\{\mathtt{nat} \to\}$. This allows us to manipulate expressions as data values in $\mathcal{L}\{\mathtt{nat} \to\}$, and hence permits $\mathcal{L}\{\mathtt{nat} \to\}$ to compute with its own programs.[1]

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

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The non-definability of certain functions on the natural numbers within $\mathcal{L}\{\mathtt{nat} \to\}$ may be seen as a form of incompleteness similar to that considered by Gödel.

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 $p_0 = 2$, $p_1 = 3$, and so on), 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 a (mathematical) function $E : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that, for any $e : \mathtt{nat} \to \mathtt{nat}$, $E(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{nat}$.[2] The determinacy of the dynamic semantics, together with Theorem 14.4 on the facing page, ensure that E is a well-defined function. Using this we may define another mathematical function, $F : \mathbb{N} \to \mathbb{N}$, by the equation $F(m) = E(m)(m)$, so that $F(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$.

The function F is not definable in $\mathcal{L}\{\mathtt{nat} \to\}$. Suppose that F were defined by the expression e_F, so that we have

$$e_F(\overline{\ulcorner e \urcorner}) \equiv e(\overline{\ulcorner e \urcorner}) : \mathtt{nat}.$$

Let e_D be the expression

$$\lambda(x{:}\mathtt{nat}.\,\mathtt{s}(e_F(x))).$$

We then have

$$\begin{aligned} e_D(\overline{\ulcorner e_D \urcorner}) &\equiv \mathtt{s}(e_F(\overline{\ulcorner e_D \urcorner})) \\ &\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})), \end{aligned}$$

which is impossible, since the self-application is (by termination) equivalent to some number, $\overline{n}$, which must then be equivalent to its own successor.

[2] The value of $E(k)(m)$ may be chosen arbitrarily to be zero when k is not the code of any expression e.

14.5 Exercises

1. Explore variant dynamic semantics for $\mathcal{L}\{\mathtt{nat} \to\}$, both separately and in combination, in which the successor does not evaluate its argument, and in which functions are called by value.

Chapter 15

Plotkin's PCF

The language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, also known as *Plotkin's PCF*, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to $\mathcal{L}\{\mathtt{nat} \to\}$ expressions in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ 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 $\mathcal{L}\{\mathtt{nat} \to\}$, the language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ 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 $\bot$: $\mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\bot\ (n)$ 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 $\bot$ as the initial approximation to f, then pass to the limit

$$\lim_{i \geq 0} F^{(i)}(\bot),$$

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—all of our loops terminate.

15.1 Statics

The abstract binding syntax of PCF is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$
		$\mid$	$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
Expr	e	$::=$	x	x
		$\mid$	$\mathtt{z}$	$\mathtt{z}$
		$\mid$	$\mathtt{s}(e)$	$\mathtt{s}(e)$
		$\mid$	$\mathtt{ifz}(e;e_0;x.e_1)$	$\mathtt{ifz}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$
		$\mid$	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		$\mid$	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$
		$\mid$	$\mathtt{fix}[\tau](x.e)$	$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e$

The expression $\mathtt{fix}[\tau](x.e)$ is called *general recursion*; it is discussed in more detail below. The expression $\mathtt{ifz}(e;e_0;x.e_1)$ branches according to whether e evaluates to $\mathtt{z}$ or not, binding the predecessor to x in the case that it is not.

The static semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{15.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{15.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{15.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1) : \tau} \tag{15.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{parr}(\tau_1;\tau_2)} \tag{15.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{15.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau} \tag{15.1g}$$

Rule (15.1g) reflects the self-referential nature of general recursion. To show that $\mathtt{fix}[\tau](x.e)$ has type τ, we *assume* that it is the case by assigning that type to the variable, x, which stands for the recursive expression itself, and checking that the body, e, has type τ under this very assumption.

The structural rules, including in particular substitution, are admissible for the static semantics.

Lemma 15.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

15.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the judgements e val, specifying the closed values, and $e \mapsto e'$, specifying the steps of evaluation. We will consider a call-by-name dynamics for function application, and require that the successor evaluate its argument.

The judgement e val is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{15.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{15.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{15.2c}$$

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

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{15.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{15.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{15.3c}$$

$$\frac{}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{15.3d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{15.3e}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{15.3f}$$

$$\overline{\texttt{fix[}\tau\texttt{]}(x\texttt{.}e) \mapsto [\texttt{fix[}\tau\texttt{]}(x\texttt{.}e)/x]e} \tag{15.3g}$$

Rule (15.3g) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 15.2 (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 (15.3g). Suppose that $\texttt{fix[}\tau\texttt{]}(x\texttt{.}e) : \tau$. By inversion of typing we have $\texttt{fix[}\tau\texttt{]}(x\texttt{.}e) : \tau \vdash [\texttt{fix[}\tau\texttt{]}(x\texttt{.}e)/x]e : \tau$, 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 (15.1g) the result follows immediately since we may make progress by unwinding the recursion. □

Definitional equivalence for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is defined to be the strongest congruence containing the following axioms:

$$\overline{\Gamma \vdash \texttt{ifz(}\tau\texttt{;z;}e_0\texttt{.}x\texttt{)}e_1 \equiv e_0 : \tau} \tag{15.4a}$$

$$\overline{\Gamma \vdash \texttt{ifz(}\tau\texttt{;s(}e\texttt{);}e_0\texttt{.}x\texttt{)}e_1 \equiv [e/x]e_1 : \tau} \tag{15.4b}$$

$$\overline{\Gamma \vdash \texttt{fix[}\tau\texttt{]}(x\texttt{.}e) \equiv [\texttt{fix[}\tau\texttt{]}(x\texttt{.}e)/x]e : \tau} \tag{15.4c}$$

$$\overline{\Gamma \vdash \texttt{ap(lam[}\tau\texttt{]}(x\texttt{.}e_2\texttt{);}e_1\texttt{)} \equiv [e_1/x]e_2 : \tau} \tag{15.4d}$$

These rules are sufficient to calculate the value of any closed expression of type nat: if $e : \texttt{nat}$, then $e \equiv \overline{n} : \texttt{nat}$ iff $e \mapsto^* \overline{n}$.

15.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. 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.

General recursive functions are definable from general recursion and non-recursive functions. Specifically, we may take $\mathtt{fun}\, x(y:\tau_1):\tau_2\, \mathtt{is}\, e$ to stand for the compound expression

$$\mathtt{fix}\, x:\tau_1 \rightharpoonup \tau_2\, \mathtt{is}\, \lambda(y:\tau_1.\, e).$$

The static and dynamic semantics of recursive functions are derivable from this definition.

Every primitive recursive function is definable in PCF by taking

$$\mathtt{rec}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}$$

to stand for the application, $e'(e)$, where e' is the general recursive function

$$\mathtt{fun}\, f(u:\mathtt{nat}):\tau\, \mathtt{is}\, \mathtt{ifz}\, u\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow [f(x)/y]e_1\}.$$

The static and dynamic semantics of primitive recursion are derivable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using this expansion.

In general, functions definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ iff there is an expression $e_\phi : \mathtt{nat} \rightharpoonup \mathtt{nat}$ such that $\phi(m) = n$ iff $e_\phi(\overline{m}) \equiv \overline{n} : \mathtt{nat}$. So, for example, if ϕ is the totally undefined function, then e_ϕ is any function that loops without returning whenever it is called.

It is informative to classify those partial functions ϕ that are definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. These are the so-called *partial recursive functions*, which are defined to be the primitive recursive functions augmented by the *minimization* operation: 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 15.3. *A partial function ϕ on the natural numbers is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ iff it is partial recursive.*

Proof sketch. Minimization is readily definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ 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.

Let ϕ_{univ} be the partial function on the natural numbers such that

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \equiv \overline{n} : \texttt{nat}.$$

By Church's Law this function is definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. It is, in essence, an interpreter that, given the code $\overline{\ulcorner e \urcorner}$ of a closed expression of type $\texttt{nat} \rightharpoonup \texttt{nat}$, simulates the dynamic semantics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. In contrast, we proved in Chapter 14 that the analogous function is *not* definable in $\mathcal{L}\{\texttt{nat} \rightarrow\}$ using the technique of diagonalization. It is instructive to examine why the analogous diagonal argument does not apply to the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$.

15.4 Variations

The choice of evaluation strategy for the successor operation is motivated by the desire to retain a standard intepretation of the type `nat` as the type of natural numbers. It is easy to verify that if $e : \texttt{nat}$ and e val, then $e = \overline{n}$ for some $n \geq 0$. An alternative is to consider a lazy interpretation of the successor, given by defining $\texttt{s}(e)$ val to hold regardless of the form of e, and, correspondingly, omitting Rule (15.3a). The major consequence of this change is that the type `nat` may no longer be thought of as the type of natural numbers. In particular, the expression $\omega = \texttt{fix[nat]}(x.\texttt{s}(x))$, which has type `nat`, evaluates to the value $\texttt{s}(\omega)$.[2] The value ω may be thought of as an infinite stack of successors, and hence it is larger than (in the sense of being at least the successor of) any (finite) natural number, and hence may be informally thought of as "infinity." Since there is no largest natural number, the lazy dynamics gives rise to a new type, called the *lazy natural numbers*, written `lnat`, which is distinct from the type `nat` of the *eager natural numbers*, written `nat`.[3]

[1] See Chapter 21 for further discussion of Church's Law.

[2] This expression diverges if successor is interpreted eagerly, so that ω has no value.

[3] It is perfectly sensible to admit both `nat` and `lnat` in the same language; see Chapter 41 for further development of this point of view.

Another variation on $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is to consider a call-by-value interpretation of function application, in which the argument to an application is fully evaluated prior to being passed as argument to the function. It is a straightforward exercise to modify the dynamics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ to implement a call-by-value semantics. One may also formulate definitional equivalence for the call-by-value fragment using methods similar to those discussed in Chapter 10 to record that λ-bound variables are only ever bound to values. This is in contrast to variables used to effect self-reference in general recursion, which is not a value. With this in mind the crucial rules of definitional equivalence for a call-by-value variant of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ are as follows:

$$\frac{\Xi, x \;\mathsf{val}\, \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\, \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{15.5a}$$

$$\frac{\Xi\, \Gamma, x : \tau_1 \vdash e \equiv e' : \tau}{\Xi\, \Gamma \vdash \mathtt{fix}[\tau](x.e) \equiv \mathtt{fix}[\tau](x.e') : \tau} \tag{15.5b}$$

$$\frac{\Xi \vdash e_1 \;\mathsf{val}}{\Xi\, \Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau_1](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{15.5c}$$

In Rule (15.5a) we extend Ξ with the hypothesis x val to record that under a call-by-value interpretation the parameter, x, of the function is only ever bound to a value. Correspondingly, in Rule (15.5c) we demand that the argument of an application be a value, which is then substituted for the parameter in the right-hand side of the equivalence In contrast Rule (15.5b) does not treat the variable x as a value, since it stands for the recursive expression itself, which is not a value.

15.5 Exercises

Part V

Finite Data Types

Chapter 16

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. The product type admits both a *lazy* and an *eager* dynamics. According to the lazy dynamics, a pair is a value without regard to whether its components are values; they are not evaluated until (if ever) they are accessed and used in another computation. According to the eager dynamics, a pair is a value only if its components are values; they are evaluated when the pair is created.

More generally, we may consider the *finite product*, $\prod_{i \in I} \tau_i$, indexed by a finite set of *indices*, I. The elements of the finite product type are *I-indexed tuples* whose ith component is an element of the type τ_i. The components are accessed by *I-indexed projection* operations, generalizing the binary case. Special cases of the finite product include *n-tuples*, indexed by sets of the form $I = \{0, \ldots, n-1\}$, and *labelled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

16.1 Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`unit`	`unit`
		|	`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$
Expr	e	::=	`triv`	$\langle\rangle$
		|	`pair`$(e_1;e_2)$	$\langle e_1, e_2\rangle$
		|	`fst`(e)	`fst`(e)
		|	`snd`(e)	`snd`(e)

The type `prod`$(\tau_1;\tau_2)$ 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.

$$\frac{}{\Gamma \vdash \texttt{triv} : \texttt{unit}} \tag{16.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1;e_2) : \texttt{prod}(\tau_1;\tau_2)} \tag{16.1b}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \tag{16.1c}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2} \tag{16.1d}$$

The dynamic semantics of product types is specified by the following rules:

$$\frac{}{\texttt{triv}\ \mathsf{val}} \tag{16.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{pair}(e_1;e_2)\ \mathsf{val}} \tag{16.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\texttt{pair}(e_1;e_2) \mapsto \texttt{pair}(e_1';e_2)}\right\} \tag{16.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{pair}(e_1;e_2) \mapsto \texttt{pair}(e_1;e_2')}\right\} \tag{16.2d}$$

$$\frac{e \mapsto e'}{\texttt{fst}(e) \mapsto \texttt{fst}(e')} \tag{16.2e}$$

$$\frac{e \mapsto e'}{\texttt{snd}(e) \mapsto \texttt{snd}(e')} \tag{16.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{fst}(\texttt{pair}(e_1;e_2)) \mapsto e_1} \tag{16.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{snd}(\texttt{pair}(e_1;e_2)) \mapsto e_2} \tag{16.2h}$$

The bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics of pairing.

The safety theorem applies to both the eager and the lazy dynamics, with the proof proceeding along similar lines in each case.

Theorem 16.1 (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 $e \mapsto e'$.

Proof. Preservation is proved by induction on transition defined by Rules (16.2). Progress is proved by induction on typing defined by Rules (16.1). □

16.2 Finite Products

Finite product types are a natural generalization of nullary and binary products to the product of a *finite family of types*. A finite family of types is a finite function, $i \in I \mapsto \tau_i$, assigning a type τ_i to each element $i \in I$ of a finite index set, I. We often use the displayed form, $\langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$, for the finite family $i \in I \mapsto \tau_i$, where $I = \{ i_0, \ldots, i_{n-1} \}$. Similarly, a *finite family of expressions* is a finite function $i \in I \mapsto e_i$, which we often write in the form $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$.

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{prod}[I](i \mapsto \tau_i)$	$\prod_{i \in I} \tau_i$
Expr	e	::=	$\texttt{tuple}[I](i \mapsto e_i)$	$\langle e_i \rangle_{i \in I}$
		|	$\texttt{proj}[I][i](e)$	$e \cdot i$

Using the displayed form for finite families, the finite product type $\prod_{i \in I} \tau_1$ is written $\prod \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$, and the finite tuple $\langle e_i \rangle_{i \in I}$ is written $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$.

The static semantics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathtt{tuple}[I]\,(i \mapsto e_i) : \mathtt{prod}[I]\,(i \mapsto \tau_i)} \tag{16.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}[I]\,(i \mapsto e_i) \quad j \in I}{\Gamma \vdash \mathtt{proj}[I]\,[j]\,(e) : \tau_j} \tag{16.3b}$$

In Rule (16.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (16.3a), the index i ranges over the index set I.

The dynamic semantics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}} \tag{16.4a}$$

$$\frac{e_j \mapsto e_j' \quad (\forall i \neq j)\ e_j' = e_j}{\mathtt{tuple}[I]\,(i \mapsto e_i) \mapsto \mathtt{tuple}[I]\,(i \mapsto e_i')} \tag{16.4b}$$

$$\frac{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}}{\mathtt{proj}[I]\,[j]\,(\mathtt{tuple}[I]\,(i \mapsto e_i)) \mapsto e_j} \tag{16.4c}$$

Rule (16.4b) 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 16.2 (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 16.1 on page 130. □

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 $l \in L$.

16.3 Mutual Recursion

An important application of product types is to support *mutual recursion*. In Chapter 15 we used general recursion to define recursive functions, those that may "call themselves" when called. Product types support a natural generalization in which we may simultaneously define two or more functions, each of which may call the others, or even itself.

Consider the following recursion equations defining two mathematical functions on the natural numbers:

$$\begin{aligned} E(0) &= 1 \\ O(0) &= 0 \\ E(n+1) &= O(n) \\ O(n+1) &= E(n) \end{aligned}$$

Intuitively, $E(n)$ is non-zero iff n is even, and $O(n)$ is non-zero iff n is odd. If we wish to define these functions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, we immediately face the problem of how to define two functions simultaneously. There is a trick available in this special case that takes advantage of the fact that E and O have the same type: simply define `eo` of type $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$ so that $\texttt{eo}(\overline{0})$ represents E and $\texttt{eo}(\overline{1})$ represents O. (We leave the details as an exercise for the reader.)

A more general solution is to recognize that the definition of two mutually recursive functions may be thought of as the recursive definition of a pair of functions. In the case of the even and odd functions we will define the labelled tuple, e_{EO}, of type, τ_{EO}, given by

$$\prod \langle \texttt{even} : \texttt{nat} \to \texttt{nat}, \texttt{odd} : \texttt{nat} \to \texttt{nat} \rangle.$$

From this we will obtain the required mutually recursive functions as the projections $e_{EO} \cdot \texttt{even}$ and $e_{EO} \cdot \texttt{odd}$.

To effect the mutual recursion the expression e_{EO} is defined to be

$$\texttt{fix}\, \texttt{this} : \tau_{EO}\, \texttt{is}\, \langle \texttt{even} : e_E, \texttt{odd} : e_O \rangle,$$

where e_E is the expression

$$\lambda(\texttt{x:nat.ifz x}\{\texttt{z} \Rightarrow \texttt{z} \mid \texttt{s(y)} \Rightarrow \texttt{this} \cdot \texttt{odd(y)}\}),$$

and e_O is the expression

$$\lambda(\texttt{x:nat.ifz x}\{\texttt{z} \Rightarrow \texttt{s(z)} \mid \texttt{s(y)} \Rightarrow \texttt{this} \cdot \texttt{even(y)}\}).$$

The functions e_E and e_O refer to each other by projecting the appropriate component from the variable `this` standing for the object itself. The choice of variable name with which to effect the self-reference is, of course, immaterial, but it is common to use `this` or `self` to emphasize its role.

In the context of so-called *object-oriented* languages, labelled tuples of mutually recursive functions defined in this manner are called *objects*, and their component functions are called *methods*. Component projection is called *message passing*, viewing the component name as a "message" sent to the object to invoke the method by that name in the object. Internally to the object the methods refer to one another by sending a "message" to `this`, the canonical name for the object itself.

16.4 Exercises

Chapter 17

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. As with products, sums come in both eager and lazy variants, differing in how values of sum type are defined.

17.1 Binary and Nullary Sums

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`void`	`void`
		|	`sum`$(\tau_1;\tau_2)$	$\tau_1 + \tau_2$
Expr	e	::=	`abort[`τ`](`e`)`	`abort`$_\tau\, e$
		|	`in[l][`τ`](`e`)`	`in[l](`e`)`
		|	`in[r][`τ`](`e`)`	`in[r](`e`)`
		|	`case(`$e; x_1.e_1; x_2.e_2$`)`	`case` $e\,\{$`in[l](`x_1`)` $\Rightarrow e_1$ | `in[r](`x_2`)` $\Rightarrow e_2\}$

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

$\tau = \texttt{sum}(\tau_1;\tau_2)$ 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 $\texttt{in[l]}[\tau](e)$ or $\texttt{in[r]}[\tau](e)$. 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.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}[\tau](e) : \tau} \tag{17.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[l]}[\tau](e) : \tau} \tag{17.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[r]}[\tau](e) : \tau} \tag{17.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum}(\tau_1;\tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e; x_1.e_1; x_2.e_2) : \tau} \tag{17.1d}$$

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:

$$\frac{e \mapsto e'}{\texttt{abort}[\tau](e) \mapsto \texttt{abort}[\tau](e')} \tag{17.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[l]}[\tau](e)\ \mathsf{val}} \tag{17.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[r]}[\tau](e)\ \mathsf{val}} \tag{17.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[l]}[\tau](e) \mapsto \texttt{in[l]}[\tau](e')}\right\} \tag{17.2d}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[r]}[\tau](e) \mapsto \texttt{in[r]}[\tau](e')}\right\} \tag{17.2e}$$

$$\frac{e \mapsto e'}{\texttt{case}(e; x_1.e_1; x_2.e_2) \mapsto \texttt{case}(e'; x_1.e_1; x_2.e_2)} \tag{17.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case}(\texttt{in[l]}[\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1} \tag{17.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case}(\texttt{in[r]}[\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2} \tag{17.2h}$$

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 17.1 (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 (17.2) for preservation, and by induction on Rules (17.1) for progress. □

17.2 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	::=	$\mathtt{sum}[I]\,(i \mapsto \tau_i)$	$\sum_{i \in I} \tau_i$
Expr	e	::=	$\mathtt{inj}[I]\,[j]\,(e)$	$\mathtt{in}[j]\,(e)$
		|	$\mathtt{case}[I]\,(e; i \mapsto x_i\,.\,e_i)$	$\mathtt{case}\,e\,\{\mathtt{in}[i]\,(x_i) \Rightarrow e_i\}_{i \in I}$

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:

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \mathtt{inj}[I]\,[j]\,(e) : \mathtt{sum}[I]\,(i \mapsto \tau_i)} \tag{17.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}[I]\,(i \mapsto \tau_i) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}[I]\,(e; i \mapsto x_i\,.\,e_i) : \tau} \tag{17.3b}$$

These rules generalize to the finite case the static semantics for nullary and binary sums given in Section 17.1 on page 135.

The dynamic semantics of finite sums is defined by the following rules:

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{inj}[I]\,[j]\,(e)\ \mathsf{val}} \tag{17.4a}$$

$$\left\{ \frac{e \mapsto e'}{\mathtt{inj}[I]\,[j]\,(e) \mapsto \mathtt{inj}[I]\,[j]\,(e')} \right\} \tag{17.4b}$$

$$\frac{e \mapsto e'}{\texttt{case}[I]\,(e; i \mapsto x_i\,.\,e_i) \mapsto \texttt{case}[I]\,(e'; i \mapsto x_i\,.\,e_i)} \tag{17.4c}$$

$$\frac{\texttt{inj}[I]\,[j]\,(e)\ \mathsf{val}}{\texttt{case}[I]\,(\texttt{inj}[I]\,[j]\,(e); i \mapsto x_i\,.\,e_i) \mapsto [e/x_j]e_j} \tag{17.4d}$$

These again generalize the dynamic semantics of binary sums given in Section 17.1 on page 135.

Theorem 17.2 (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 17.1 on page 135. □

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 `abort(`e`)` may corresponding be defined as the empty case analysis, `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 `in[l](`e`)` and `in[r](`e`)` are defined to be their counterparts, `in[l](`e`)` and `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 17.1 on page 135 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.

17.3 Uses for 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.

17.3.1 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!

17.3.2 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`
		|	`ff`	`ff`
		|	`if(`e`;`e_1`;`e_2`)`	`if` e `then` e_1 `else` e_2

The values of type `bool` are `tt` and `ff`. The expression `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:

$$\texttt{bool} = \texttt{sum(unit;unit)} \tag{17.5a}$$
$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{17.5b}$$
$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{17.5c}$$
$$\texttt{if(}e\texttt{;}e_1\texttt{;}e_2\texttt{)} = \texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \tag{17.5d}$$

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.

17.3.3 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 ♣, ♢, ♡, and ♠, has as elimination form the case analysis

$$\texttt{case}\, e\, \{\clubsuit \Rightarrow e_0 \mid \diamondsuit \Rightarrow e_1 \mid \heartsuit \Rightarrow e_2 \mid \spadesuit \Rightarrow e_3\},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = \sum_{\_ \in I} \texttt{unit}$, 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.

17.3.4 Options

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{opt}(\tau)$	$\tau\,\texttt{opt}$
Expr	e	::=	`null`	`null`
		|	$\texttt{just}(e)$	$\texttt{just}(e)$
		|	$\texttt{ifnull}[\tau](e; e_1; x.e_2)$	$\texttt{check}\, e \{\texttt{null} \Rightarrow e_1 \mid \texttt{just}(x) \Rightarrow e_2\}$

The type $\texttt{opt}(\tau)$ represents the type of "optional" values of type τ. The introductory forms are `null`, corresponding to "no value", and $\texttt{just}(e)$, corresponding to a specified value of type τ. The elimination form discriminates between the two possibilities.

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

$$\texttt{opt}(\tau) = \texttt{sum}(\texttt{unit}; \tau) \tag{17.6a}$$

$$\texttt{null} = \texttt{in}[\texttt{l}][\texttt{opt}(\tau)](\texttt{triv}) \tag{17.6b}$$

$$\texttt{just}(e) = \texttt{in}[\texttt{r}][\texttt{opt}(\tau)](e) \tag{17.6c}$$

$$\texttt{ifnull}[\tau](e; e_1; x_2.e_2) = \texttt{case}(e; \_.e_1; x_2.e_2) \tag{17.6d}$$

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 $\texttt{null} : \tau \to \texttt{bool}$, 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

$$\texttt{if null(}e\texttt{) then}\ldots \textit{error} \ldots \texttt{else}\ldots \textit{proceed} \ldots. \tag{17.7}$$

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 $\texttt{opt}(\tau)$. 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,

$$\texttt{ifnull[}\tau\texttt{]}(e; e_{error}; x\,.\,e_{ok}) \tag{17.8}$$

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 (17.7); the advantage of option types is precisely that it does so.

17.4 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 18

Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

$$\texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ \texttt{fst}(x) + \texttt{snd}(x)$$

to add the components of a pair, e, of natural numbers, we may instead write

$$\texttt{match}\ e\ \{x, y\,.\,\langle x, y\rangle \Rightarrow x + y\},$$

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 `in[l](`x`)` and `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 `(unit + unit)` $\times$ `nat`, either doubles or squares its sec-

ond component depending on the form of its first component:

$$\texttt{match}\, e\, \{x.\langle \texttt{in[l]}(\langle\rangle), x\rangle \Rightarrow x + x \mid y.\langle \texttt{in[r]}(\langle\rangle), y\rangle \Rightarrow y * y\}. \quad (18.1)$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 16 and 17.

In this chapter we study a simple language, $\mathcal{L}\{pat\}$, of pattern matching over eager product and sum types.

18.1 A Pattern Language

The main challenge in formalizing $\mathcal{L}\{pat\}$ is to manage properly 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 (18.1) is given by

$$\texttt{match}\, e\, \{r_1; r_2\},$$

where r_1 is the rule

$$x.\langle \texttt{in[l]}(\langle\rangle), x\rangle \Rightarrow x + x$$

and r_2 is the rule

$$y.\langle \texttt{in[r]}(\langle\rangle), y\rangle \Rightarrow y * y.$$

The salient point is that Each rule binds its own variables, in both the pattern and the expression.

The abstract syntax of $\mathcal{L}\{pat\}$ is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{match}(e; rs)$	$\texttt{match}\, e\, \{rs\}$
Rules	rs	::=	$\texttt{rules[}n\texttt{]}(r_1; \ldots; r_n)$	$r_1 \mid \ldots \mid r_n$
Rule	r	::=	$x_1, \ldots, x_k.\texttt{rule}(p; e)$	$x_1, \ldots, x_n.\, p \Rightarrow e$
Pattern	p	::=	$\texttt{wild}$	$\_$
		|	x	x
		|	$\texttt{triv}$	$\langle\rangle$
		|	$\texttt{pair}(p_1; p_2)$	$\langle p_1, p_2\rangle$
		|	$\texttt{in[l]}(p)$	$\texttt{in[l]}(p)$
		|	$\texttt{in[r]}(p)$	$\texttt{in[r]}(p)$

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 generic 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:

$$\frac{}{\mathcal{X}, x \mid x : \tau > \langle x : \tau \rangle} \tag{18.2a}$$

$$\frac{}{\mathcal{X} \mid \_ : \tau > \emptyset} \tag{18.2b}$$

$$\frac{}{\mathcal{X} \mid \langle\rangle : \mathtt{unit} > \emptyset} \tag{18.2c}$$

$$\frac{\mathcal{X}_1 \mid p_1 : \tau_1 > \gamma_1 \quad \mathcal{X}_2 \mid p_2 : \tau_2 > \gamma_2 \quad dom(\gamma_1) \cap dom(\gamma_2) = \emptyset}{\mathcal{X}_1\, \mathcal{X}_2 \mid \langle p_1, p_2 \rangle : \tau_1 \times \tau_2 > \gamma_1 \otimes \gamma_2} \tag{18.2d}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_1 > \gamma_1}{\mathcal{X}_1 \mid \texttt{in[l]}(p) : \tau_1 + \tau_2 > \gamma_1} \tag{18.2e}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_2 > \gamma_2}{\mathcal{X}_2 \mid \texttt{in[r]}(p) : \tau_1 + \tau_2 > \gamma_2} \tag{18.2f}$$

The most interesting rule is Rule (18.2d), 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 judgement $\mathcal{X} \mid \Gamma \vdash \gamma > e : \tau$ is defined by the following rules:

$$\frac{\mathcal{X} \mid \Gamma \vdash e' : \tau'}{\mathcal{X} \mid \Gamma \vdash \emptyset > e' : \tau'} \tag{18.3a}$$

$$\frac{\mathcal{X}, x \mid \Gamma, x : \tau \vdash \gamma > e' : \tau'}{\mathcal{X}, x \mid \Gamma \vdash \langle x : \tau \rangle \otimes \gamma > e' : \tau'} \tag{18.3b}$$

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:

$$\frac{x_1, \ldots, x_n \mid p : \tau > \gamma \quad \mathcal{X}\, x_1, \ldots, x_n \mid \Gamma \vdash \gamma > e : \tau'}{\mathcal{X} \mid \Gamma \vdash x_1, \ldots, x_k \,.\, p \Rightarrow e : \tau > \tau'} \tag{18.4a}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \mathcal{X} \mid \Gamma \vdash r_n : \tau > \tau'}{\mathcal{X} \mid \Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'} \tag{18.4b}$$

In Rule (18.4a) 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 (18.4b), 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:

$$\frac{\mathcal{X} \mid \Gamma \vdash e : \tau \quad \mathcal{X} \mid \Gamma \vdash rs : \tau > \tau'}{\mathcal{X} \mid \Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{18.5}$$

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'.

18.2 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.

$$\frac{e \mapsto e'}{\texttt{match}\, e\, \{rs\} \mapsto \texttt{match}\, e'\, \{rs\}} \tag{18.6a}$$

$$\frac{}{\texttt{match}\, e\, \{\}\ \mathsf{err}} \tag{18.6b}$$

$$\frac{e\ \mathsf{val} \quad [e_1, \ldots, e_k/x_1, \ldots, x_k]p_0 = e \quad [e_1, \ldots, e_k/x_1, \ldots, x_k]e_0 = e'}{\texttt{match}\, e\, \{x_1, \ldots, x_k \,.\, p_0 \Rightarrow e_0; rs\} \mapsto e'} \tag{18.6c}$$

$$\frac{\neg \exists e_1, \ldots, e_k . [e_1, \ldots, e_k/x_1, \ldots, x_k]p_0 = e \quad e\ \mathsf{val} \quad \texttt{match}\, e\, \{rs\} \mapsto e'}{\texttt{match}\, e\, \{x_1, \ldots, x_k \,.\, p_0 \Rightarrow e_0; rs\} \mapsto e'} \tag{18.6d}$$

Rule (18.6b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (18.6c) 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 18.1 (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. □

18.3 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $\mathcal{L}\{pat\}$ as defined in Section 18.1 on page 144, it would not have much force, because the static semantics does not rule out pattern matching failure. What is missing is enforcement of the *exhaustiveness* of a sequence of rules, which ensures that every value of the domain type of a sequence of rules must match some rule in the sequence. In addition it would be useful to rule out

redundancy of rules, which means that a rule only matches values that are matched by some preceding rule. Since pattern matching considers rules in the order in which they are written, such a rule can never be executed, and hence can be safely eliminated.

The static semantics of rules given in Section 18.1 on page 144 does not ensure exhaustiveness or irredundancy of rules. To do so we introduce a language of *match conditions* that identify a subset of the closed values of a type. With each rule we associate a match condition that classifies the values that are matched by that rule. A sequence of rules is *exhaustive* if every value of the domain type of the rule satisfies the match condition of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match condition also satisfies the match condition of some preceding rule.

The language of match conditions is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Cond	ξ	::=	$\texttt{any}[\tau]$	$\top_\tau$
		|	$\texttt{in[l]}[\texttt{sum}(\tau_1;\tau_2)](\xi_1)$	$\texttt{in[l]}(\xi_1)$
		|	$\texttt{in[r]}[\texttt{sum}(\tau_1;\tau_2)](\xi_2)$	$\texttt{in[r]}(\xi_2)$
		|	$\texttt{triv}$	$\langle\rangle$
		|	$\texttt{pair}(\xi_1;\xi_2)$	$\langle\xi_1,\xi_2\rangle$
		|	$\texttt{nil}[\tau]$	$\bot_\tau$
		|	$\texttt{alt}(\xi_1;\xi_2)$	$\xi_1 \vee \xi_2$

The judgement $\xi : \tau$ is defined by the following rules:

$$\frac{}{\top_\tau : \tau} \tag{18.7a}$$

$$\frac{\xi_1 : \tau_1}{\texttt{in[l]}(\xi_1) : \tau_1 + \tau_2} \tag{18.7b}$$

$$\frac{\xi_1 : \tau_2}{\texttt{in[r]}(\xi_1) : \tau_1 + \tau_2} \tag{18.7c}$$

$$\frac{\xi_1 : \tau_1 \quad \xi_2 : \tau_2}{\langle\xi_1,\xi_2\rangle : \tau_1 \times \tau_2} \tag{18.7d}$$

$$\frac{}{\bot_\tau : \tau} \tag{18.7e}$$

$$\frac{\xi_1 : \tau \quad \xi_2 : \tau}{\xi_1 \vee \xi_2 : \tau} \tag{18.7f}$$

Informally, $\xi : \tau$ means that ξ constrains values of type τ.

For $\xi : \tau$, $e : \tau$, and e val, we define the *satisfaction* judgement $e \triangleright \xi$ as follows:

$$\frac{}{e \triangleright \top_\tau} \tag{18.8a}$$

$$\frac{e_1 \triangleright \xi_1}{\mathtt{in[l]}(e_1) \triangleright \mathtt{in[l]}(\xi_1)} \tag{18.8b}$$

$$\frac{e_2 \triangleright \xi_2}{\mathtt{in[r]}(e_2) \triangleright \mathtt{in[r]}(\xi_2)} \tag{18.8c}$$

$$\frac{}{\langle\rangle \triangleright \langle\rangle} \tag{18.8d}$$

$$\frac{e_1 \triangleright \xi_1 \quad e_2 \triangleright \xi_2}{\langle e_1, e_2\rangle \triangleright \langle \xi_1, \xi_2\rangle} \tag{18.8e}$$

$$\frac{e \triangleright \xi_1}{e \triangleright \xi_1 \vee \xi_2} \tag{18.8f}$$

$$\frac{e \triangleright \xi_2}{e \triangleright \xi_1 \vee \xi_2} \tag{18.8g}$$

The *entailment* judgement $\xi_1 \triangleright \xi_2$, where $\xi_1 : \tau$ and $\xi_2 : \tau$, is defined to hold iff $e \triangleright \xi_1$ implies $e \triangleright \xi_2$.

Finally, we instrument the static semantics of patterns and rules to associate a match condition that specifies the values that may be matched by that pattern or rule. This allows us to ensure that rules are both exhaustive and irredundant.

The judgement $\mathcal{X} \mid p : \tau > \gamma\ [\xi]$ augments the judgement $p : \tau > \gamma$ with a match constraint characterizing the set of values of type τ matched by the pattern p. It is inductively defined by the following rules:

$$\frac{}{\mathcal{X}, x \mid x : \tau > \langle x : \tau\rangle\ [\top_\tau]} \tag{18.9a}$$

$$\frac{}{\mathcal{X} \mid \_ : \tau > \emptyset\ [\top_\tau]} \tag{18.9b}$$

$$\frac{}{\mathcal{X} \mid \langle\rangle : \mathtt{unit} > \emptyset\ [\langle\rangle]} \tag{18.9c}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_1 > \gamma_1\ [\xi_1]}{\mathcal{X}_1 \mid \mathtt{in[l]}(p) : \tau_1 + \tau_2 > \gamma_1\ [\mathtt{in[l]}(\xi_1)]} \tag{18.9d}$$

$$\frac{\mathcal{X}_2 \mid p : \tau_2 > \gamma_2\ [\xi_2]}{\mathcal{X}_2 \mid \mathtt{in[r]}(p) : \tau_1 + \tau_2 > \gamma_2\ [\mathtt{in[r]}(\xi_2)]} \tag{18.9e}$$

$$\frac{\mathcal{X}_1 \mid p_1 : \tau_1 > \gamma_1\ [\xi_1] \quad \mathcal{X}_2 \mid p_2 : \tau_2 > \gamma_2\ [\xi_2] \quad dom(\gamma_1) \cap dom(\gamma_2) = \emptyset}{\mathcal{X}_1\,\mathcal{X}_2 \mid \langle p_1, p_2\rangle : \tau_1 \times \tau_2 > \gamma_1 \otimes \gamma_2\ [\langle \xi_1, \xi_2\rangle]} \tag{18.9f}$$

Rules (18.9a) to (18.9b) specify that all values of the pattern type are matched. Rule (18.9c) specifies that the only value of type `unit` is matched by the pattern. Rules (18.9d) to (18.9e) specify that the pattern matches only those values with the specified injection tag and whose argument is matched by the specified pattern. Rule (18.9f) specifies that the pattern matches only pairs whose components match the specified patterns.

The judgement $\mathcal{X} \mid \Gamma \vdash r : \tau > \tau' \ [\xi]$ augments the formation judgement for a rule with a match constraint characterizing the pattern component of the rule. The judgement $\mathcal{X} \mid \Gamma \vdash rs : \tau > \tau' \ [\xi]$ augments the formation judgement for a sequence of rules with a match constraint characterizing the values matched by some rule in the given rule sequence.

$$\frac{x_1, \ldots, x_n \mid p : \tau > \gamma \ [\xi] \quad \mathcal{X}\, x_1, \ldots, x_n \mid \Gamma \vdash \gamma > e : \tau'}{\mathcal{X} \mid \Gamma \vdash x_1, \ldots, x_k \,.\, p \Rightarrow e : \tau > \tau' \ [\xi]} \tag{18.10a}$$

$$\frac{\begin{array}{c}\mathcal{X} \mid \Gamma \vdash r_1 : \tau > \tau' \ [\xi_1] \qquad \ldots \qquad \mathcal{X} \mid \Gamma \vdash r_n : \tau > \tau' \ [\xi_n] \\ (\forall 1 \leq i \leq n)\ \neg(\xi_i \rhd \xi_1 \vee \ldots \vee \xi_{i-1})\end{array}}{\mathcal{X} \mid \Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau' \ [\xi_1 \vee \ldots \vee \xi_n]} \tag{18.10b}$$

Rule (18.10b) ensures that each successive rule is irredundant relative to the preceding rules in that it demands that it *not* be the case that *every* value satisfying ξ_i satisfies some preceding ξ_j. That is, it requires that there be *some* value satisfying ξ_i that does *not* satisfy some preceding ξ_j.

Finally, the typing rule for `match` expressions requires exhaustiveness:

$$\frac{\mathcal{X} \mid \Gamma \vdash e : \tau \quad \mathcal{X} \mid \Gamma \vdash rs : \tau > \tau' \ [\xi] \quad \top_\tau \rhd \xi}{\mathcal{X} \mid \Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'} \tag{18.11}$$

The third premise ensures that *every* value of type τ satisfies the constraint ξ representing the values matched by *some* rule in the given rule sequence.

The additional constraints on the static semantics are sufficient to ensure progress, because no well-formed `match` expression can fail to match a value of the specified type. If a given sequence of rules is inexhaustive, this can always be rectified by including a "default" rule of the form $x \,.\, x \Rightarrow e_x$, where e_x handles the unmatched value x gracefully, perhaps by raising an exception (see Chapter 28 for a discussion of exceptions).

18.4 Exercises

Part VI

Infinite Data Types

Chapter 19

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, the elements of an inductive type are those that may be obtained by a finite composition of its introductory forms. Consequently, if we specify the behavior of a function on each of the introductory forms of an inductive type, then its behavior is determined for all values of that type. Such a function is called an *iterator*, or *catamorphism*. Dually, the elements of a coinductive type are those that behave properly in response to a finite composition of its elimination forms. Consequently, if we specify the behavior of an element on each elimination form, then we have fully specified that element as a value of that type. Such an element is called an *generator*, or *anamorphism*.

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 the type of regular trees, which includes nodes whose descendants are also ancestors, and the type of lazy natural numbers, which includes a "point at infinity" consisting of an infinite stack of successors.

19.1 Static Semantics

We will consider the language $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\to}{\times}{+}\}$, which extends $\mathcal{L}\{{\to}{\times}{+}\}$ with inductive and co-inductive types.

19.1.1 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
		|	$\mathtt{ind}(t.\tau)$	$\mu_{\mathsf{i}}(t.\tau)$
		|	$\mathtt{coi}(t.\tau)$	$\mu_{\mathsf{f}}(t.\tau)$

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 $t\ \mathsf{type}$, where $t\ \mathtt{name}$ is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \mid t\ \mathsf{type}} \tag{19.1a}$$

$$\frac{}{\Delta \mid \mathtt{unit}\ \mathsf{type}} \tag{19.1b}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid \tau_2\ \mathsf{type}}{\Delta \mid \mathtt{prod}(\tau_1;\tau_2)\ \mathsf{type}} \tag{19.1c}$$

$$\frac{}{\Delta \mid \mathtt{void}\ \mathsf{type}} \tag{19.1d}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \texttt{sum}(\tau_1;\tau_2) \text{ type}} \tag{19.1e}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \texttt{arr}(\tau_1;\tau_2) \text{ type}} \tag{19.1f}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type} \quad \Delta \mid t.\tau \text{ pos}}{\Delta \mid \texttt{ind}(t.\tau) \text{ type}} \tag{19.1g}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type} \quad \Delta \mid t.\tau \text{ pos}}{\Delta \mid \texttt{coi}(t.\tau) \text{ type}} \tag{19.2}$$

The premises on Rules (19.1g) and (19.2) involve a judgement of the form $t.\tau$ pos, which will be explained in Section 19.2 on the following page.

A *type operator* is an abstractor of the form $t.\tau$ such that t type $\mid$ τ 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 $t.\texttt{unit} + t$, which will be used in the definition of the natural numbers as an inductive type. Other examples include $t.\texttt{unit} + (\texttt{nat} \times t)$, 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.

19.1.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{in}[t.\tau](e)$	$\texttt{in}(e)$
		$\mid$	$\texttt{rec}[t.\tau](x.e;e')$	$\texttt{rec}(x.e;e')$
		$\mid$	$\texttt{out}[t.\tau](e)$	$\texttt{out}(e)$
		$\mid$	$\texttt{gen}[t.\tau](x.e;e')$	$\texttt{gen}(x.e;e')$

The expression $\texttt{rec}(x.e;e')$ is called an *iterator*, and the expression $\texttt{gen}(x.e;e')$ is called a *co-iterator*, or *generator*. The expression $\texttt{in}(e)$ is called a *fold* operation, or *constructor*, and $\texttt{out}(e)$ is called an *unfold* operation, or *destructor*.

The static semantics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\texttt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{in}[t.\tau](e) : \texttt{ind}(t.\tau)} \tag{19.3a}$$

$$\frac{\Gamma \vdash e' : \mathtt{ind}(t.\tau) \quad \Gamma, x : [\rho/t]\tau \vdash e : \rho}{\Gamma \vdash \mathtt{rec}[t.\tau](x.e; e') : \rho} \tag{19.3b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{out}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{19.3c}$$

$$\frac{\Gamma \vdash e' : \rho \quad \Gamma, x : \rho \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau](x.e; e') : \mathtt{coi}(t.\tau)} \tag{19.3d}$$

The dynamic semantics of these constructs is given in terms of the action of a positive type operator, which we now define.

19.2 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. For example, the type operator $t.\mathtt{nat} \to t$ is positive, as is $t.u \to t$, where u type is some type variable other than t. On the other hand, the type operator $t.t \to t$ is not positive, because t occurs in the domain of a function type.

The judgement $\Delta \mid t.\tau$ pos, where Δ, t type $\mid \tau$ type, is inductively defined by the following rules:

$$\frac{}{\Delta \mid t.t \text{ pos}} \tag{19.4a}$$

$$\frac{u \neq t}{\Delta \mid t.u \text{ pos}} \tag{19.4b}$$

$$\frac{}{\Delta \mid t.\mathtt{unit} \text{ pos}} \tag{19.4c}$$

$$\frac{\Delta \mid t.\tau_1 \text{ pos} \quad \Delta \mid t.\tau_2 \text{ pos}}{\Delta \mid t.\tau_1 \times \tau_2 \text{ pos}} \tag{19.4d}$$

$$\frac{}{\Delta \mid t.\mathtt{void} \text{ pos}} \tag{19.4e}$$

$$\frac{\Delta \mid t.\tau_1 \text{ pos} \quad \Delta \mid t.\tau_2 \text{ pos}}{\Delta \mid t.\tau_1 + \tau_2 \text{ pos}} \tag{19.4f}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid t.\tau_2 \text{ pos}}{\Delta \mid t.\tau_1 \to \tau_2 \text{ pos}} \tag{19.4g}$$

[1] A more permissive notion of *positive type operator* is sometimes considered, but we shall make use only of the strict form.

Notice that in Rule (19.4g), 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 19.1. *If* $t\,.\,\sigma$ pos *and* $u\,.\,\tau$ pos, *then* $t\,.\,[\sigma/u]\tau$ pos.

Proof. By rule induction on Rules (19.4). □

Strictly positive type operators admit a *covariant action*, or *map* operation, that transforms types and expressions in tandem. Specifically, if $t\,.\,\tau$ pos, then

1. If σ type, then $\mathtt{Map}[t\,.\,\tau](\sigma)$ type.
2. If $x : \sigma_1 \vdash e : \sigma_2$ and $\mathtt{map}[t\,.\,\tau](x\,.\,e) = x'\,.\,e'$, then $x' : \mathtt{Map}[t\,.\,\tau](\sigma_1) \vdash e' : \mathtt{Map}[t\,.\,\tau](\sigma_2)$.

The action on types is given by substitution:

$$\mathtt{Map}[t\,.\,\tau](\sigma) := [\sigma/t]\tau.$$

The action of a type operator on an expression is an example of *generic programming* in which the type of a computation determines its behavior. Specifically, the action of the type operator $t\,.\,\tau$ on an abstraction $x\,.\,e$ transforms an element e_1 of type $\mathtt{Map}[t\,.\,\tau](\sigma_1)$ into an element of e_2 of type $\mathtt{Map}[t\,.\,\tau](\sigma_2)$. This is achieved by replacing each sub-expression, d, of e_1 corresponding to an occurrence of t in τ by the expression $[d/x]e_1$. (This is well-defined provided that $t\,.\,\tau$ is a positive type operator.)

For example, consider the type operator

$$t\,.\,\tau = t\,.\,\mathtt{unit} + (\mathtt{nat} \times t).$$

The action of this operator on $x\,.\,e$ such that

$$x : \sigma_1 \vdash e : \sigma_2$$

is the abstractor $x'\,.\,e'$ with type

$$x' : \mathtt{unit} + (\mathtt{nat} \times \sigma_1) \vdash e' : \mathtt{unit} + (\mathtt{nat} \times \sigma_2).$$

The expression e' is such that if we instantiate x' by $\mathtt{in}[\mathtt{l}](\langle\rangle)$, then e' evaluates to $\mathtt{in}[\mathtt{l}](\langle\rangle)$, and if we instantiate x' by $\mathtt{in}[\mathtt{r}](\langle d_1, d_2\rangle)$, it evaluates to $\mathtt{in}[\mathtt{r}](\langle d_1, [d_2/x]e\rangle)$. Note that this action is independent of the choice of σ_1 and σ_2. Even if σ_1 happens to be the type $\mathtt{nat}$, the action in the second

case above remains the same. In particular, the first component, d_1, of the pair is passed through untouched, whereas d_2 is replaced by $[d_2/x]e$, even though it, too, has type `nat`. This is because the action is guided by the operator $t.\tau$, and not by $[\sigma_1/t]\tau$.

The action of a strictly positive type operator on an abstraction is given by the judgement

$$\mathtt{map}[t.\tau](x.e) = x'.e',$$

which is inductively defined by the following rules:

$$\frac{}{\mathtt{map}[t.t](x.e) = x.e} \tag{19.5a}$$

$$\frac{}{\mathtt{map}[t.\tau](x.e) = x.x} \tag{19.5b}$$

$$\frac{}{\mathtt{map}[t.\mathtt{unit}](x.e) = x'.\langle\rangle} \tag{19.5c}$$

$$\frac{\begin{array}{c}\mathtt{map}[t.\tau_1](x.\mathtt{fst}(e)) = x'.e_1' \\ \mathtt{map}[t.\tau_2](x.\mathtt{snd}(e)) = x'.e_2'\end{array}}{\mathtt{map}[t.\tau_1 \times \tau_2](x.e) = x'.\mathtt{pair}(e_1';e_2')} \tag{19.5d}$$

$$\frac{}{\mathtt{map}[t.\mathtt{void}](x'.\mathtt{abort}(x'))} \tag{19.5e}$$

$$\frac{\begin{array}{c}\mathtt{map}[t.\tau_1](x_1.[\mathtt{in}[\mathtt{l}](x_1)/x]e) = x_1'.e_1' \\ \mathtt{map}[t.\tau_2](x_2.[\mathtt{in}[\mathtt{r}](x_2)/x]e) = x_1'.e_2'\end{array}}{\mathtt{map}[t.\tau_1 + \tau_2](x.e) = x'.\mathtt{case}(x';x_1'.e_1';x_2'.e_2')} \tag{19.5f}$$

$$\frac{\mathtt{map}[t.\tau_2](x.e) = x_2'.e_2'}{\mathtt{map}[t.\tau_1 \to \tau_2](x.e) = x'.\lambda(x_1':[\sigma_2/t]\tau_1.[x'(x_1')/x_2']e_2')} \tag{19.5g}$$

Lemma 19.2. *If $x : \sigma \vdash e : \sigma'$, and $\mathtt{map}[t.\tau](x.e) = x'.e'$, then $x' : \mathtt{Map}[t.\tau](\sigma) \vdash e' : \mathtt{Map}[t.\tau](\sigma')$.*

Proof. By rule induction on Rules (19.5). □

19.3 Dynamic Semantics

The dynamic semantics of inductive and coinductive types is given in terms of the covariant action of the associated type operator. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\to\}$:

$$\frac{}{\mathtt{in}(e)\ \mathsf{val}} \tag{19.6a}$$

$$\frac{e' \mapsto e''}{\texttt{rec}(x.e; e') \mapsto \texttt{rec}(x.e; e'')} \tag{19.6b}$$

$$\frac{\texttt{map}[t.\tau](x'.\texttt{rec}(x.e; x')) = x''.e''}{\texttt{rec}(x.e; \texttt{in}(e')) \mapsto [[e'/x'']e''/x]e} \tag{19.6c}$$

$$\frac{}{\texttt{gen}(x.e; e')\ \mathsf{val}} \tag{19.6d}$$

$$\frac{e \mapsto e'}{\texttt{out}(e) \mapsto \texttt{out}(e')} \tag{19.6e}$$

$$\frac{\texttt{map}[t.\tau](x'.\texttt{gen}(x.e; x')) = x''.e''}{\texttt{out}(\texttt{gen}(x.e; e')) \mapsto [[e'/x]e/x'']e''} \tag{19.6f}$$

Rule (19.6c) states that to evaluate the iterator on a value of recursive type, we inductively apply the iterator as guided by the type operator to the value, and then perform the inductive step on the result. Rule (19.6f) is simply the dual of this rule for coinductive types.

Lemma 19.3. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on Rules (19.6). □

Lemma 19.4. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on Rules (19.3). □

Although we shall not give the proof here, the language $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\to}\}$ is terminating, and all functions defined within it are total.

Theorem 19.5. *If $e : \tau$ in $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\to}\}$, then there exists e' val such that $e \mapsto^* e'$.*

The judgement $\Gamma \vdash e_1 \equiv e_2 : \tau$ of *definitional equivalence* (or *symbolic evaluation*) is defined to be the strongest congruence containing the extension of the dynamic semantics to open expressions. In particular the following two rules are admissible as principles of definitional equivalence:

$$\frac{\texttt{map}[t.\tau](x'.\texttt{rec}(x.e; x')) = x''.e''}{\Gamma \vdash \texttt{rec}(x.e; \texttt{in}(e')) \equiv [[e'/x'']e''/x]e : \rho} \tag{19.7a}$$

$$\frac{\texttt{map}[t.\tau](x'.\texttt{gen}(x.e; x')) = x''.e''}{\Gamma \vdash \texttt{out}(\texttt{gen}(x.e; e')) \equiv [[e'/x]e/x'']e'' : [\texttt{coi}(t.\tau)/t]\tau} \tag{19.7b}$$

In addition to these rules we also have rules specifying that definitional equivalence is an equivalence relation, and that it is a congruence with respect to all expression-forming operators of the language. These rules license the replacement of any sub-expression of an expression by a definitionally equivalent one to obtain a definitionally equivalent result.

19.4 Fixed Point Properties

Inductive and coinductive types enjoy an important property that will play a prominent role in Chapter 20, 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:

$$x.\mathtt{in}^{-1}_{t.\tau}(x) = x.\mathtt{rec}_{t.\tau}(\mathtt{map}[t.\tau](y.\mathtt{in}(y));x) \tag{19.8}$$

$$x.\mathtt{out}^{-1}_{t.\tau}(x) = x.\mathtt{gen}_{t.\tau}(\mathtt{map}[t.\tau](y.\mathtt{out}(y));x) \tag{19.9}$$

Rule (19.7a) of definitional equivalence specifies that $x.\mathtt{in}^{-1}_{t.\tau}(x)$ is post-inverse to $y.\mathtt{in}(y)$, and Rule (19.7b) of definitional equivalence specifies that $x.\mathtt{out}^{-1}_{t.\tau}(x)$ is pre-inverse to $y.\mathtt{out}(y)$. This is to say that these properties are consequences solely of the dynamic semantics of the operators involved.

It is natural to ask whether these pairs of abstractors are, in fact, two-sided inverses of each other. This is the case, but only up to *observational equivalence*, which is defined to be the coarsest consistent congruence on expressions. This relation equates as many expressions as possible subject to the conditions that it be a congruence (to permit replacing equals by equals anywhere in an expression) and that it be consistent (not equate all expressions). It is difficult, in general, to show that two expressions are observationally equivalent. In most cases some form of inductive proof is required, rather than being simply a matter of direct calculation. (Please see Chapter 50 for further discussion of observational equivalence for $\mathcal{L}\{\mathtt{nat}\to\}$, a special case of $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}\to\}$.)

One consequence of these inverse relationships (up to observational equivalence) is that both the inductive and the coinductive type are two solutions to the type isomorphism

$$X \cong \mathtt{Map}[t.\tau](X) = [X/t]\tau.$$

This is to say that we have two isomorphisms,

$$\mu_{\mathsf{i}}(t.\tau) \cong [\mu_{\mathsf{i}}(t.\tau)/t]\tau$$

and

$$\mu_{\mathsf{f}}(t.\tau) \cong [\mu_{\mathsf{f}}(t.\tau)/t]\tau,$$

witnessed by the two pairs of mutually inverse abstractors given above. What distinguishes the two solutions is that the inductive type is the *initial* solution, whereas the coinductive type is the *final* solution to the isomorphism equation. Initiality means that the iterator is a general means of defining functions that act on values of inductive type; finality means that the generator is a general means of creating values of coinductive types.

To understand better what is happening here, let us consider a specific example. 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 defined by the expression

$$\mathtt{in}(\mathtt{in}[\mathtt{l}](\langle\rangle)),$$

and, if e is of type $\mathtt{nat}_{\mathsf{i}}$, its successor, which is defined by the expression

$$\mathtt{in}(\mathtt{in}[\mathtt{r}](e)).$$

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 defined by $\mathtt{in}[\mathtt{l}](\langle\rangle)$, or to the successor of some expression $e' : \mathtt{nat}_{\mathsf{f}}$, which is defined 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. In particular, 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. Intuitively, this is because ω is an infinite stack of successors, and hence is large than any finite stack of successors, which is to say that it is larger than any finite natural number. 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, in contradiction to the preceding remark. (To make all this precise requires that we specify what we mean by an embedding, and to argue formally that no such embedding exists.)

19.5 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

$$x.\mathtt{gen}(y.\mathtt{in}_{t\cdot\tau}^{-1}(y);x).$$

   Characterize the behavior of this term when x is replaced by an element of the inductive type.

Chapter 20

Recursive Types

Inductive and coinductive types may be seen as initial and final solutions to certain forms of recursive type equations. Both the inductive type, $\mu_{\mathsf{i}}(t.\tau)$, and the coinductive type, $\mu_{\mathsf{f}}(t.\tau)$, are fixed points of the type operator $t.\tau$. Thus both are solutions to the recursion equation $t \cong \tau$ "up to isomorphism" in that both

$$\mu_{\mathsf{i}}(t.\tau) \cong [\mu_{\mathsf{i}}(t.\tau)/t]\tau$$

and

$$\mu_{\mathsf{f}}(t.\tau) \cong [\mu_{\mathsf{f}}(t.\tau)/t]\tau.$$

The restriction to positive type operators is sufficient for many purposes, but there are situations in which this restriction must be relaxed. For example, to model self-referential data structures such as circular lists or recursive functions we must consider recursion equations such as $t \cong t \rightharpoonup \sigma$, where the argument of the function is the object itself.

In this chapter we consider the problem of finding fixed points of general type operators, with no positivity restrictions. Equivalently, we seek solutions to a general class of recursion equations of the form $t \cong \tau$, where no requirements are placed on the occurrences of t in τ. As motivation let us recall from Chapter 19 that the initial and final fixed points of a positive type operator are given by pairs of expressions witnessing the isomorphisms stated above. In the case of inductive types we have the expressions

$$x : [\mu_{\mathsf{i}}(t.\tau)/t]\tau \vdash \mathtt{in}(x) : \mu_{\mathsf{i}}(t.\tau)$$

and

$$x : \mu_{\mathsf{i}}(t.\tau) \vdash \mathtt{in}^{-1}_{t.\tau}(x) : [\mu_{\mathsf{i}}(t.\tau)/t]\tau.$$

In the case of coinductive types we have the expressions

$$x : \mu_{\mathtt{f}}(t.\tau) \vdash \mathtt{out}(x) : [\mu_{\mathtt{f}}(t.\tau)/t]\tau,$$

and

$$x : [\mu_{\mathtt{f}}(t.\tau)/t]\tau \vdash \mathtt{out}^{-1}_{t.\tau}(x) : \mu_{\mathtt{f}}(t.\tau).$$

In both cases these pairs of expressions are observationally mutually inverse to each other, and as such constitute an isomorphism.

Rather than derive the expressions that give rise to an isomorphism between the two sides of a recursion equation, we shall instead the isomporphism itself as the defining characteristic of the solution of such an equation. That is, we shall consider the *recursive type*, $\mu t.\tau$, to be the solution to the isomorphism

$$\mu t.\tau \cong [\mu t.\tau/t]\tau$$

witnessed by the primitives

$$x : \mu t.\tau \vdash \mathtt{unfold}(x) : [\mu t.\tau/t]\tau$$

and

$$x : [\mu t.\tau/t]\tau \vdash \mathtt{fold}(x) : \mu t.\tau,$$

without positivity restrictions on τ. When this restriction is relaxed, the separation between the initial and final solutions collapses, so that there is at most one solution to any such recursion equation.

But why is there a solution at all? There is reason to worry, for suppose we wish to solve the equation $t \cong t \to \mathtt{bool}$. This should raise suspicion, since it seems to specify a type that is isomorphic to its own "power type," and indeed we can carry out a version of Cantor's diagonal argument to show that this equation has no solution. However, if we limit ourselves to *partial*, rather than *total*, function types, then a solution to the equation $t \cong t \rightharpoonup \mathtt{bool}$ does indeed exist. More generally, as long as we accept the possibility of undefinedness, we may demand the solution of a very wide class of recursion equations—including such celebrated equations as $t \cong t \rightharpoonup t$, whose solution is the subject of Chapter 21.

20.1 Recursive 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 `fold(`e`)` and `unfold(`e`)` 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
		$\mid$	`rec(`$t.\tau$`)`	$\mu t.\tau$
Expr	e	::=	`fold[`$t.\tau$`](`e`)`	`fold(`e`)`
		$\mid$	`unfold(`e`)`	`unfold(`e`)`

The meta-variable t ranges over a class of *type names*, which serve as names for types. The *unfolding* of `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, `fold[`$t.\tau$`](`e`)`, introduces a value of recursive type in terms of an element of its unfolding, and the elimination form, `unfold(`e`)`, evaluates to a value of the unfolding from an element of the recursive type. In implementation terms the operation `fold[`$t.\tau$`](`e`)` may be thought of as an abstract "pointer" to a value of the unfolded type, and the operation `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

$$\Delta \mid \tau \text{ type},$$

where Δ is a finite set of assumptions of the form t_i type for some type variable t_i.

$$\frac{}{\Delta, t \text{ type} \mid t \text{ type}} \tag{20.1a}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \texttt{arr}(\tau_1;\tau_2) \text{ type}} \tag{20.1b}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type}}{\Delta \mid \texttt{rec}(t.\tau) \text{ type}} \tag{20.1c}$$

Note that, in contrast to Chapter 19, 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:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \tag{20.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{20.2b}$$

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:

$$\frac{e \text{ val}}{\mathtt{fold}[t.\tau](e) \text{ val}} \tag{20.3a}$$

$$\frac{e \mapsto e'}{\mathtt{fold}[t.\tau](e) \mapsto \mathtt{fold}[t.\tau](e')} \tag{20.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{20.3c}$$

$$\frac{e \text{ val}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \mapsto e} \tag{20.3d}$$

We have chosen to give an eager dynamics to the recursive injection, but one could as well consider a lazy dynamics.

It is straightforward to prove the safety of $\mathcal{L}\{\mu\rightharpoonup\}$.

Theorem 20.1 (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 $e \mapsto e'$.

20.2 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 15. 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

$$\mu t.\,[\mathtt{z} : \mathtt{unit}, \mathtt{s} : t], \tag{20.4}$$

which specifies that

$$\texttt{nat} \cong \texttt{[z:unit,s:nat]}.$$

The zero and successor operations are correspondingly defined by the following equations:

$$\begin{aligned}\texttt{z} &= \texttt{fold(in[z]}(\langle\rangle)\texttt{)}\\ \texttt{s}(e) &= \texttt{fold(in[s]}(e)\texttt{)}.\end{aligned}$$

The conditional branch on zero is defined by the following equation:

$$\begin{aligned}&\texttt{ifz}\,e\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\} =\\ &\qquad\texttt{case unfold}(e)\,\{\texttt{in[z]}(\_) \Rightarrow e_0 \mid \texttt{in[s]}(x) \Rightarrow e_1\},\end{aligned}$$

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 15.

As another example, the type `nat list` of lists of natural numbers may be represented by the recursive type

$$\mu t.\,\texttt{[n:unit,c:nat} \times t\texttt{]}$$

so that we have the isomorphism

$$\texttt{nat list} \cong \texttt{[n:unit,c:nat} \times \texttt{nat list]}.$$

The list formation operations are represented by the following equations:

$$\begin{aligned}\texttt{nil} &= \texttt{fold(in[n]}(\langle\rangle)\texttt{)}\\ \texttt{cons}(e_1;e_2) &= \texttt{fold(in[c]}(\langle e_1,e_2\rangle)\texttt{)}.\end{aligned}$$

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned}&\texttt{listcase}\,e\,\{\texttt{nil} \Rightarrow e_0 \mid \texttt{cons}(x;y) \Rightarrow e_1\} =\\ &\qquad\texttt{case unfold}(e)\,\{\texttt{in[n]}(\_) \Rightarrow e_0, \mid \texttt{in[c]}(\langle x,y\rangle) \Rightarrow e_1\},\end{aligned}$$

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.

20.3 Self-Reference

In Chapters 15 and 16 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. 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

$$\tau\ \mathtt{self} = \mu t\,.\,t \rightharpoonup \tau,$$

is, by definition, a solution to this type equation. It establishes the isomorphism

$$\tau\ \mathtt{self} \cong \tau\ \mathtt{self} \rightharpoonup \tau,$$

capturing the self-referential natural of values of this type.

The introductory form for the type τ `self` is the self-referential expression `self` y `is` e, where $y : \tau\ \mathtt{self} \vdash e : \tau$. It is defined to be the expression

$$\mathtt{fold}(\lambda(y{:}\tau\ \mathtt{self}.e)),$$

which we note is a value, regardless of whether `fold` is eager or lazy. According to this definition, the following typing rule is derivable:

$$\frac{\Gamma, y : \tau\ \mathtt{self} \vdash e : \tau}{\Gamma \vdash \mathtt{self}\ y\ \mathtt{is}\ e : \tau\ \mathtt{self}}\ .$$

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.

It is a small step from having a type of self-referential computations to defining general recursion in the sense of Chapter 15. To be specific, the recursive expression $\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e$ may be defined as

$$\texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)}.$$

If $x : \tau \vdash e : \tau$, then this expression also has type τ. Moreover, we have

$$\begin{aligned}\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e &= \texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)}\\ &\equiv [\texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll(}y\texttt{)}/x]e\texttt{)}/x]e\\ &= [\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e/x]e.\end{aligned}$$

(In the second line we have made use of the evident notion of definitional equivalence induced by the dynamic semantics given earlier.)

20.4 Exercises

Part VII

Dynamic Types

Chapter 21

The Untyped λ-Calculus

Types are the central organizing principle in the study of programming languages. Yet many languages of practical interest are said to be *untyped*. Have we missed something important? The answer is *no*! The supposed opposition between typed and untyped languages turns out to be illusory. In fact, untyped languages are special cases of typed languages with a single, pre-determined recursive type. Far from being *untyped*, such languages are instead *uni-typed*.[1]

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) λ-calculus*. This formalism was introduced by Church in the 1930's as a universal language of computable functions. It is distinctive for its austere elegance. The λ-calculus has but one "feature", the higher-order function, with which to compute. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a well-worn phrase, in the λ-calculus it's functions all the way down!

21.1 The λ-Calculus

The abstract syntax of $\mathcal{L}\{\lambda\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Term	u	$::=$	x	x
		$\mid$	$\lambda(x.u)$	$\lambda x.u$
		$\mid$	$\mathtt{ap}(u_1;u_2)$	$u_1(u_2)$

[1] An apt description suggested by Dana Scott.

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 general hypothetical judgements of the form $x_1, \ldots, x_n \mid x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok} \vdash u \ \mathsf{ok}$, stating that u is a well-formed expression involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the parameters when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x \ \mathsf{ok} \vdash x \ \mathsf{ok}} \tag{21.1a}$$

$$\frac{\Gamma \vdash u_1 \ \mathsf{ok} \quad \Gamma \vdash u_2 \ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(u_1; u_2) \ \mathsf{ok}} \tag{21.1b}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \ \mathsf{ok}}{\Gamma \vdash \lambda(x.u) \ \mathsf{ok}} \tag{21.1c}$$

The dynamic semantics is given by the following transition rules:

$$\frac{}{\mathtt{ap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{21.2a}$$

$$\frac{u_1 \mapsto u_1'}{\mathtt{ap}(u_1; u_2) \mapsto \mathtt{ap}(u_1'; u_2)} \tag{21.2b}$$

In the λ-calculus literature this judgement is called *weak head reduction*. The first rule is called *β-reduction*; it defines the meaning of function application as substitution of argument for parameter.

21.2 Definitional Equivalence

Definitional equivalence for $\mathcal{L}\{\lambda\}$ is a judgement of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok}$ for some $n \geq 0$, and e and e' are terms having at most the variables $x_1, \ldots, x_n$ free. This judgement is inductively defined by the following rules:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{21.3a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{21.3b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{21.3c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2')} \tag{21.3d}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \tag{21.3e}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\lambda(x.e_2); e_1) \equiv [e_1/x]e_2} \tag{21.3f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

21.3 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. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration due to the gravitational field of a massive object.[2]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 15. 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,

[2]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.

called the *Church numerals*.

$$\overline{0} = \lambda b.\, \lambda s.\, b \tag{21.4a}$$

$$\overline{n+1} = \lambda b.\, \lambda s.\, s(\overline{n}(b)(s)) \tag{21.4b}$$

It follows that

$$\overline{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

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 defined by the following untyped λ-terms:

$$\mathtt{succ} = \lambda x.\, \lambda b.\, \lambda s.\, s(x(b)(s)) \tag{21.5}$$

$$\mathtt{plus} = \lambda x.\, \lambda y.\, y(x)(\mathtt{succ}) \tag{21.6}$$

$$\mathtt{times} = \lambda x.\, \lambda y.\, y(\overline{0})((\mathtt{plus}(x))) \tag{21.7}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

We may readily define $\mathtt{ifz}(u; u_0; u_1)$ to be the application $u(u_0)((\lambda x.\, u_1))$, where x is chosen arbitrarily such that $x \mathbin{\#} u_1$. We can use this to define $\mathtt{ifz}(u; u_0; x.\, u_1)$, provided that we can compute the predecessor of a natural number. Doing so requires a bit of ingenuity. We wish to find a term $\mathtt{pred}$ such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{21.8}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{21.9}$$

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.

$$\langle u_1, u_2 \rangle = \lambda f.\, f(u_1)(u_2) \tag{21.10}$$
$$\mathtt{fst}(u) = u((\lambda x.\, \lambda y.\, x)) \tag{21.11}$$
$$\mathtt{snd}(u) = u((\lambda x.\, \lambda y.\, y)) \tag{21.12}$$

It is easy to check that under this encoding $\mathtt{fst}(\langle u_1, u_2 \rangle) \equiv u_1$, and similarly for the second projection. We may now define the required term u representing the predecessor:

$$u'_p = \lambda x.\, x(\langle \overline{0}, \overline{0} \rangle)(\lambda y.\, \langle \mathtt{snd}(y), \mathtt{s}(\mathtt{snd}(y)) \rangle) \tag{21.13}$$
$$u_p = \lambda x.\, \mathtt{fst}(u(x)) \tag{21.14}$$

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*:

$$\mathbf{Y} = \lambda F.\, (\lambda f.\, F(f(f)))(\lambda f.\, F(f(f))). \tag{21.15}$$

Observe that

$$\mathbf{Y}(F) \equiv F(\mathbf{Y}(F)).$$

Using the **Y** combinator, we may define general recursion by writing $\mathbf{Y}(\lambda x.\, u)$, where x stands for the recursive expression itself.

21.4 Undecidability of Definitional Equivalence

Definitional equivalence for the untyped λ-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equivalent. The proof of this result is based on two key lemmas:

1. For any untyped λ-term u, we may find an untyped term v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of v, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 14 for a discussion of Gödel-numbering.)

2. Any two non-vacuous properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms that respect definitional equivalence are *computably inseparable*. This means that there is no *decidable* property $\mathcal{B}$ of untyped terms such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ and $\mathcal{A}_1\ u$ implies that it is *not* the case that $\mathcal{B}\ u$. In particular, neither $\mathcal{A}_0$ nor $\mathcal{A}_1$ are decidable.

For a property $\mathcal{B}$ of untyped terms to respect definitional equivalence means that if $\mathcal{B}\ u$ and $u \equiv u'$, then $\mathcal{B}\ u'$.

Lemma 21.1. *For any u there exists v such that $u\,(\overline{\ulcorner v \urcorner}) \equiv v$.*

Proof Sketch. The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. $\mathbf{ap}\,(\overline{\ulcorner u_1 \urcorner})\,(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1\,(u_2) \urcorner}$.
2. $\mathbf{nm}\,(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms, and builds the representation of the application of one to the other. The second takes a numeral for n, and yields the representation of $\overline{n}$. Given these, we may find the required term v by defining $v = w(\overline{\ulcorner w \urcorner})$, where $w = \lambda x.\, u(\mathbf{ap}(x)(\mathbf{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathbf{ap}(\overline{\ulcorner w \urcorner})(\mathbf{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathbf{Y}(u)$, except that u takes as input the representation of a term, and we find a v such that, when applied to the representation of v, the term u yields v itself. □

Lemma 21.2. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-vacuous properties of untyped terms that respect definitional equivalence. Then there is no untyped term w such that*

1. *For every u either $w\,(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w\,(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*
2. *If $\mathcal{A}_0\ u$, then $w\,(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*
3. *If $\mathcal{A}_1\ u$, then $w\,(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

Proof. Suppose there is such an untyped term w. Let v be the untyped term $\lambda x.\,\mathtt{ifz}(w(x); u_1; u_0)$, where $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. By Lemma 21.1 on the preceding page there is an untyped term t such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, since $\mathcal{A}_1$ respects definitional equivalence and $\mathcal{A}_1\ u_1$. But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of w, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

Corollary 21.3. *There is no algorithm to decide whether or not $u \equiv u'$.*

Proof. For fixed u consider the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$. This is non-vacuous and respects definitional equivalence, and hence is undecidable. □

21.5 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

$$D = \mu t\,.\,t \to t.$$

A value of type D is of the form $\mathtt{fold}(e)$ where e is a value of type $D \to D$ — 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

$$D \cong D \to D.$$

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 $\mathcal{L}\{\mu\rightharpoonup\}$:

$$x^\dagger = x \tag{21.16a}$$

$$\lambda x.\, u^\dagger = \mathtt{fold}(\lambda(x\mathtt{:}D.\, u^\dagger)) \tag{21.16b}$$

$$u_1\,(u_2)^\dagger = \mathtt{unfold}(u_1^\dagger)\,(u_2^\dagger) \tag{21.16c}$$

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,

$$\begin{aligned}\lambda x.\, u_1\,(u_2)^\dagger &= \mathtt{unfold}(\mathtt{fold}(\lambda(x\mathtt{:}D.\, u_1^\dagger)))\,(u_2^\dagger)\\ &\equiv \lambda(x\mathtt{:}D.\, u_1^\dagger)\,(u_2^\dagger)\\ &\equiv [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

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.

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 typed language after all! Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the expense of incurring substantial dynamic overhead to coerce values to and from the recursive type. In Chapter 22 we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus a traditional distinction can hardly be considered an opposition, since dynamic languages are but particular forms of static language in which (undue) emphasis is placed on a single recursive type.

21.6 Exercises

Chapter 22

Dynamic Typing

We saw in Chapter 21 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 (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!

22.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{\mathit{dyn}\}$. The abstract syntax of $\mathcal{L}\{\mathit{dyn}\}$ is given by the

following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	d	::=	x	x
		|	$\mathtt{num}(\overline{n})$	$\overline{n}$
		|	$\mathtt{s}(d)$	$\mathtt{s}(d)$
		|	$\mathtt{ifz}(d; d_0; x.d_1)$	$\mathtt{ifz}\, d\, \{\mathtt{z} \Rightarrow d_0 \mid \mathtt{s}(x) \Rightarrow d_1\}$
		|	$\mathtt{fun}(\lambda(x.d))$	$\lambda x.d$
		|	$\mathtt{dap}(d_1; d_2)$	$d_1(d_2)$
		|	$\mathtt{fix}(x.d)$	$\mathtt{fix}\, x\, \mathtt{is}\, d$

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 21; 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:

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{val}} \tag{22.1a}$$

$$\frac{}{\mathtt{fun}(\lambda(x.d))\ \mathsf{val}} \tag{22.1b}$$

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.

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{22.2a}$$

$$\frac{}{\mathtt{fun}(\lambda(x.d))\ \mathsf{is\_fun}\ \lambda(x.d)} \tag{22.2b}$$

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.

$$\frac{}{\mathtt{num}(\_)\ \mathsf{isnt\_fun}} \tag{22.3a}$$

$$\frac{}{\mathtt{fun}(\_)\ \mathsf{isnt\_num}} \tag{22.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, d err, are defined simultaneously by the following rules.

$$\frac{d \mapsto d'}{\mathtt{s}(d) \mapsto \mathtt{s}(d')} \tag{22.4a}$$

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\mathtt{s}(d) \mapsto \mathtt{num}(\mathtt{s}(\overline{n}))} \tag{22.4b}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{s}(d)\ \mathsf{err}} \tag{22.4c}$$

$$\frac{d \mapsto d'}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto \mathtt{ifz}(d'; d_0; x.d_1)} \tag{22.4d}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{z}}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto d_0} \tag{22.4e}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{s}(\overline{n})}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto [\mathtt{num}(\overline{n})/x]d_1} \tag{22.4f}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}} \tag{22.4g}$$

$$\frac{d_1 \mapsto d_1'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1';d_2)} \tag{22.4h}$$

$$\frac{d_1\ \mathsf{val} \quad d_2 \mapsto d_2'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1;d_2')} \tag{22.4i}$$

$$\frac{d_1\ \mathsf{is\_fun}\ \lambda(x.d) \quad d_2\ \mathsf{val}}{\mathtt{dap}(d_1;d_2) \mapsto [d_2/x]d} \tag{22.4j}$$

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{dap}(d_1;d_2)\ \mathsf{err}} \tag{22.4k}$$

$$\frac{}{\mathtt{fix}(x.d) \mapsto [\mathtt{fix}(x.d)/x]d} \tag{22.4l}$$

Note that in Rule (22.4f) 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 22.1. *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 (22.4). 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.

22.2 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!

22.3 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 15 with the following addi-

tional constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`dyn`	`dyn`
Expr	e	::=	`new[`l`](`e`)`	l `!` e
		$\mid$	`cast[`l`](`e`)`	e `?` l
Class	l	::=	`num`	`num`
		$\mid$	`fun`	`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 $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ is the extension of that of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the following rules governing the type `dyn`.

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}(e) : \texttt{dyn}} \tag{22.5a}$$

$$\frac{\Gamma \vdash e : \texttt{parr(dyn;dyn)}}{\Gamma \vdash \texttt{new[fun]}(e) : \texttt{dyn}} \tag{22.5b}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[num]}(e) : \texttt{nat}} \tag{22.5c}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[fun]}(e) : \texttt{parr(dyn;dyn)}} \tag{22.5d}$$

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 $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ is given by the following rules:

$$\frac{e \ \mathsf{val}}{\texttt{new[}l\texttt{]}(e) \ \mathsf{val}} \tag{22.6a}$$

$$\frac{e \mapsto e'}{\texttt{new[}l\texttt{]}(e) \mapsto \texttt{new[}l\texttt{]}(e')} \tag{22.6b}$$

$$\frac{e \mapsto e'}{\texttt{cast[}l\texttt{]}(e) \mapsto \texttt{cast[}l\texttt{]}(e')} \tag{22.6c}$$

$$\frac{\texttt{new[}l\texttt{]}(e) \ \mathsf{val}}{\texttt{cast[}l\texttt{](new[}l\texttt{]}(e)\texttt{)} \mapsto e} \tag{22.6d}$$

$$\frac{\texttt{new[}l'\texttt{]}(e) \ \mathsf{val} \quad l \neq l'}{\texttt{cast[}l\texttt{](new[}l'\texttt{]}(e)\texttt{)} \ \mathsf{err}} \tag{22.6e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 22.2 (Canonical Forms)**.** *If $e : \mathtt{dyn}$ and e val, then $e = \mathtt{new}[l](e')$ for some class l and some e' val. If $l = \mathtt{num}$, then $e' : \mathtt{nat}$, and if $l = \mathtt{fun}$, then $e' : \mathtt{parr}(\mathtt{dyn}; \mathtt{dyn})$.*

Proof. By a straightforward rule induction on static semantics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn} \rightharpoonup\}$. □

Theorem 22.3 (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. □

22.4 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

$$\mathtt{fun}\,!\,\lambda(x{:}\mathtt{dyn}.\,\mathtt{fix}\,p{:}\mathtt{dyn}\,\mathtt{is}\,\mathtt{fun}\,!\,\lambda(y{:}\mathtt{dyn}.\,e_{x,p,y})),$$

where

$$x : \mathtt{dyn}, p : \mathtt{dyn}, y : \mathtt{dyn} \vdash e_{x,p,y} : \mathtt{dyn}$$

is defined to be the expression

$$\mathtt{ifz}\,(y\,?\,\mathtt{num})\,\{\mathtt{z} \Rightarrow x \mid \mathtt{s}(y') \Rightarrow \mathtt{num}\,!\,(\mathtt{s}(((p\,?\,\mathtt{fun})((\mathtt{num}\,!\,y')))\,?\,\mathtt{num}))\}.$$

This is essentially an explicit form of the dynamically typed addition function given in Section 22.2 on page 186. 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:

$$\texttt{fun}\,!\,\lambda(x\!:\!\texttt{dyn}.\,\texttt{fun}\,!\,\texttt{fix}\,p\!:\!\texttt{dyn} \rightharpoonup \texttt{dyn}\,\texttt{is}\,\lambda(y\!:\!\texttt{dyn}.\,e'_{x,p,y})),$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,?\,\texttt{num})\,\{\texttt{z} \Rightarrow x \mid \texttt{s}(y') \Rightarrow \texttt{num}\,!\,(\texttt{s}((p((\texttt{num}\,!\,y')))\,?\,\texttt{num}))\}.$$

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 $\texttt{dyn} \rightharpoonup \texttt{dyn}$, 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 $\texttt{dyn} \rightharpoonup \texttt{dyn}$ to $\texttt{nat} \rightharpoonup \texttt{dyn}$. 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 $x\,?\,\texttt{num}$, 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:

$$\texttt{fix}\,p\texttt{:nat} \rightharpoonup \texttt{nat}\,\texttt{is}\,\lambda(y\texttt{:nat.}\,\texttt{ifz}\,y\,\{\texttt{z} \Rightarrow x\,?\,\texttt{num} \mid \texttt{s}(y') \Rightarrow \texttt{s}(p(y'))\}).$$

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`:

$$\texttt{fun}\,!\,\lambda(x\texttt{:dyn.}\,\texttt{fun}\,!\,\lambda(y\texttt{:dyn.}\,\texttt{num}\,!\,(e_x''(y\,?\,\texttt{num})))).$$

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`.

22.5 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 `cons`$(d_1;d_2)$, 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 22.4 on page 189.

22.6 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]

$$\texttt{dyn} = \mu t\,.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t] \tag{22.7}$$

$$\texttt{new[num]}(e) = \texttt{fold}(\texttt{in[num]}(e)) \tag{22.8}$$

$$\texttt{new[fun]}(e) = \texttt{fold}(\texttt{in[fun]}(e)) \tag{22.9}$$

$$\texttt{cast[num]}(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in[num]}(x) \Rightarrow x \mid \texttt{in[fun]}(x) \Rightarrow \texttt{error}\} \tag{22.10}$$

$$\texttt{cast[fun]}(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in[num]}(x) \Rightarrow \texttt{error} \mid \texttt{in[fun]}(x) \Rightarrow x\} \tag{22.11}$$

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.

22.7 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 28.

Part VIII

Polymorphism

Chapter 23

Girard's System F

The languages $\mathcal{L}\{\mathtt{nat} \to\}$ 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,

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda(f:\tau_2 \to \tau_3.\, \lambda(g:\tau_1 \to \tau_2.\, \lambda(x:\tau_1.\, f(g(x))))).$$

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 *polymorphic* in the types involved. 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*.

23.1 System F

System F, or the *polymorphic λ-calculus*, or $\mathcal{L}\{\to\forall\}$, 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 gram-

mar:

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

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, τ type, stating that τ is a well-formed type, and $e : \tau$, stating that e is a well-formed expression of type τ. The definitions of these judgements make use of generic hypothetical judgements of the form

$$\mathcal{T} \mid \Delta \vdash \tau \text{ type}$$

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 cons and $\mathcal{X}$ consists of a finite set of *expression variable* declarations of the form x exp. The finite set of hypotheses Δ consists of assumptions of the form t type such that $\mathcal{T} \vdash t$ cons, and the finite set Γ consists of hypotheses of the form $x : \tau$, where $\mathcal{X} \vdash x$ exp and $\mathcal{T} \mid \Delta \vdash \tau$ 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:

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}} \tag{23.1a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ type}} \tag{23.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathsf{type}} \tag{23.1c}$$

The rules for typing expressions are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{23.2a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{23.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{23.2c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{23.2e}$$

For example, the polymorphic composition function is written as follows:

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda(f:t_2 \to t_3.\lambda(g:t_1 \to t_2.\lambda(x:t_1.f(g(x)))))))).$$

This expression has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

The static semantics validates the expected structural rules, including substitution for both type and expression variables.

Lemma 23.1 (Substitution). *1. If $\Delta, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ and $\Delta \vdash \tau\ \mathsf{type}$, then $\Delta \vdash [\tau/t]\tau'\ \mathsf{type}$.*

2. If $\Delta, t\ \mathsf{type}\ \Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau\ \mathsf{type}$, 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 $\mathcal{L}\{\rightarrow\forall\}$ is given as follows:

$$\frac{}{\texttt{lam[}\tau\texttt{]}(x.e)\ \texttt{val}} \tag{23.3a}$$

$$\frac{}{\texttt{Lam}(t.e)\ \texttt{val}} \tag{23.3b}$$

$$\frac{}{\texttt{ap}(\texttt{lam[}\tau_1\texttt{]}(x.e);e_2) \mapsto [e_2/x]e} \tag{23.3c}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{23.3d}$$

$$\frac{}{\texttt{App[}\tau\texttt{]}(\texttt{Lam}(t.e)) \mapsto [\tau/t]e} \tag{23.3e}$$

$$\frac{e \mapsto e'}{\texttt{App[}\tau\texttt{]}(e) \mapsto \texttt{App[}\tau\texttt{]}(e')} \tag{23.3f}$$

These rules endow $\mathcal{L}\{\rightarrow\forall\}$ with a call-by-name interpretation of application, but one could as well consider a call-by-value variant.

It is a simple matter to prove safety for $\mathcal{L}\{\rightarrow\forall\}$, using familiar methods.

Lemma 23.2 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \texttt{arr}(\tau_1;\tau_2)$, then $e = \texttt{lam[}\tau_1\texttt{]}(x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \texttt{all}(t.\tau')$, then $e = \texttt{Lam}(t.e')$ with t type $\vdash e' : \tau'$.*

Proof. By rule induction on the static semantics. □

Theorem 23.3 (Preservation)**.** *If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.*

Proof. By rule induction on the dynamic semantics. □

Theorem 23.4 (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. □

23.2 Polymorphic Definability

It will be proved in Chapter 52 that every well-typed expression in $\mathcal{L}\{\rightarrow\forall\}$ 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 $\mathcal{L}\{\rightarrow\forall\}$. 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 $\mathcal{L}\{\rightarrow\forall\}$. We then show that the natural numbers, under the lazy semantics, are definable as well.

23.2.1 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 $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned}\texttt{prod}(\sigma;\tau) &= \ldots\\ \texttt{pair}(e_1;e_2) &= \ldots\\ \texttt{fst}(e) &= \ldots\\ \texttt{snd}(e) &= \ldots\end{aligned}$$

The required definitions are derived from the inversion principle, 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:

$$\begin{aligned}\sigma\times\tau &= \forall(r.(\sigma\rightarrow\tau\rightarrow r)\rightarrow r)\\ \langle e_1,e_2\rangle &= \Lambda(r.\lambda(x{:}\sigma\rightarrow\tau\rightarrow r.\,x(e_1)(e_2)))\\ \texttt{fst}(e) &= e[\sigma](\lambda(x{:}\sigma.\lambda(y{:}\tau.\,x)))\\ \texttt{snd}(e) &= e[\tau](\lambda(x{:}\sigma.\lambda(y{:}\tau.\,y)))\end{aligned}$$

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 $\mathcal{L}\{\rightarrow\forall\}$.

The nullary product, or unit, type is similarly definable:

$$\begin{aligned}\mathtt{unit} &= \forall(r.r \to r)\\ \langle\rangle &= \Lambda(r.\lambda(x{:}r.x))\end{aligned}$$

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.

$$\begin{aligned}\sigma + \tau &= \forall(r.(\sigma \to r) \to (\tau \to r) \to r)\\ \mathtt{in[l]}(e) &= \Lambda(r.\lambda(x{:}\sigma \to r.\lambda(y{:}\tau \to r.x(e))))\\ \mathtt{in[r]}(e) &= \Lambda(r.\lambda(x{:}\sigma \to r.\lambda(y{:}\tau \to r.y(e))))\\ \mathtt{case}\,e\,\{\mathtt{in[l]}(x_1) \Rightarrow e_1 \mid \mathtt{in[r]}(x_2) \Rightarrow e_2\} &=\\ e[\rho](\lambda(x_1{:}\sigma.e_1))(\lambda(x_2{:}\tau.e_2))\end{aligned}$$

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:

$$\begin{aligned}\mathtt{void} &= \forall(r.r)\\ \mathtt{abort}(e) &= e[\rho]\end{aligned}$$

Once again, observe that this is formally consistent with the binary case, albeit for a sum of no types.

23.2.2 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:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}[\tau](e_0; e_1; x.e_2) : \tau}\,.$$

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

$$\mathtt{nat} \to \forall(t.t \to (t \to t) \to t).$$

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 21.4 on page 176). 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.)

23.2.3 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 14) 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 $\mathcal{L}\{\rightarrow\forall\}$, but not in $\mathcal{L}\{\mathtt{nat}\rightarrow\}$—it diagonalizes out of the restricted language, but is definable within the richer one.

Put in other terms, the language $\mathcal{L}\{\rightarrow\forall\}$ is able to prove the termination of *all* expressions in $\mathcal{L}\{\mathtt{nat}\rightarrow\}$. But it cannot do this for $\mathcal{L}\{\rightarrow\forall\}$ itself, by a diagonal argument analogous to the one given in Chapter 14. Thus we see the beginnings of a hierarchy of expressiveness, with $\mathcal{L}\{\mathtt{nat}\rightarrow\}$ at the bottom, $\mathcal{L}\{\rightarrow\forall\}$ 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.

23.3 Exercises

1. Show that primitive recursion is definable in $\mathcal{L}\{\rightarrow\forall\}$ by exploiting the definability of iteration and binary products.

2. Investigate the representation of eager products and sums in eager and lazy variants of $\mathcal{L}\{\rightarrow\forall\}$.

3. Show how to write an interpreter for $\mathcal{L}\{\mathtt{nat}\rightarrow\}$ in $\mathcal{L}\{\rightarrow\forall\}$.

Chapter 24

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!

24.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Types	τ	::=	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$
Expr	e	::=	$\texttt{pack}[t.\tau;\rho]\,(e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$
		|	$\texttt{open}[t.\tau]\,(e_1;t,x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\tau\,\texttt{in}\,e_2$

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 $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\tau\,\texttt{in}\,e_2$, 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.

24.1.1 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.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{24.1a}$$

$$\frac{\Delta\ \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}[t.\tau;\rho]\,(e) : \texttt{some}(t.\tau)} \tag{24.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}[t.\tau](e_1; t, x.e_2) : \tau_2} \tag{24.1c}$$

Rule (24.1c) 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.

24.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified as follows:

$$\frac{}{\texttt{pack}[t.\tau;\rho](e)\ \mathsf{val}} \tag{24.2a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{open}[t.\tau](e_1; t, x.e_2) \mapsto \texttt{open}[t.\tau](e_1'; t, x.e_2)} \tag{24.2b}$$

$$\frac{e\ \mathsf{val}}{\texttt{open}[t.\tau](\texttt{pack}[t.\tau;\rho](e); t, x.e_2) \mapsto [\rho, e/t, x]e_2} \tag{24.2c}$$

These rules endow $\mathcal{L}\{\rightarrow\forall\exists\}$ with a lazy semantics for packages. More importantly, these rules specify that *there are no abstract types at run time*! The representation type is exposed to the client by substitution when the package is opened. In other words, data abstraction is a *compile-time discipline* that leaves no traces of its presence at execution time.

24.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\rightarrow\forall\}$ to the new constructs.

Theorem 24.1 (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 24.2 (Canonical Forms)**.** *If e : $\texttt{some}(t.\tau)$ and e val, then $e = \texttt{pack}[t.\tau;\rho](e')$ 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 24.3 (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. □

24.2 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, $\exists(t.\sigma)$, which serves as the interface of the queue abstraction:[1]

$$\exists(t.\langle \mathtt{emp}:t, \mathtt{ins}:\tau \times t \to t, \mathtt{rem}:t \to \tau \times t\rangle).$$

The representation type, t, of queues is *abstract* — all that is specified about it is that it supports the operations `emp`, `ins`, and `rem`, with the specified types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations.

[1] For the sake of illustration, we assume that type constructors such as products, records, and lists are also available in the language.

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

$$e_i : \tau \times \tau\,\texttt{list} \to \tau\,\texttt{list} = \lambda(x{:}\tau \times \tau\,\texttt{list}.\,e_i'),$$

and

$$e_r : \tau\,\texttt{list} \to \tau \times \tau\,\texttt{list} = \lambda(x{:}\tau\,\texttt{list}.\,e_r').$$

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

$$\tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}) \to (\tau\,\texttt{list} \times \tau\,\texttt{list}),$$

and e_r has type

$$(\tau\,\texttt{list} \times \tau\,\texttt{list}) \to \tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}).$$

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

$$\langle \mathtt{emp}:t, \mathtt{ins}:\tau \times t \to t, \mathtt{rem}:t \to \tau \times t \rangle,$$

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.

24.3 Definability of Existentials

It turns out that it is not necessary to extend $\mathcal{L}\{\to\forall\}$ with existential types to model data abstraction, because they are already definable using only universal types! 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

$$\mathtt{open}\, e\, \mathtt{as}\, t\, \mathtt{with}\, x:\tau\, \mathtt{in}\, e' : \tau',$$

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:

$$\begin{aligned} \exists(t.\sigma) &= \forall(t'.\forall(t.\sigma \to t') \to t') \\ \mathtt{pack}\,\rho\,\mathtt{with}\,e\,\mathtt{as}\,\exists(t.\tau) &= \Lambda(t'.\lambda(x:\forall(t.\tau \to t').x[\rho](e))) \\ \mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,x:\tau\,\mathtt{in}\,e' &= e[\tau'](\Lambda(t.\lambda(x:\tau.e'))) \end{aligned}$$

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.

24.4 Exercises

Chapter 25

Constructors and Kinds

In Chapters 23 and 24 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 tracks the *phase distinction* between the static and dynamic phases of processing: 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 26, 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.

25.1 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	::=	`Type`	`Type`
		|	`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$
		|	`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \rightarrow \kappa_2$
Cons	c	::=	t	t
		|	`arr`	$\rightarrow$
		|	`all[`κ`]`	$\forall_\kappa$
		|	`some[`κ`]`	$\exists_\kappa$
		|	`pair`$(c_1;c_2)$	$\langle c_1, c_2\rangle$
		|	`fst`(c)	`fst`(c)
		|	`snd`(c)	`snd`(c)
		|	`lambda[`κ`]`$(t.c)$	$\lambda(t::\kappa.c)$
		|	`app`$(c_1;c_2)$	$c_1[c_2]$
Type	τ	::=	c	c
Expr	e	::=	x	x
		|	`lam[`τ`]`$(x.e)$	$\lambda(x:\tau.e)$
		|	`ap`$(e_1;e_2)$	$e_1(e_2)$
		|	`Lam[`κ`]`$(t.e)$	$\Lambda(t::\kappa.e)$
		|	`App[`c`]`(e)	$e[c]$
		|	`pack[`$\kappa;c;c'$`]`(e)	`pack` c' `with` e `as` $\exists_\kappa[c]$
		|	`open[`$\kappa;c$`]`$(e_1;t,x.e_2)$	`open` e_1 `as` $t::\kappa$ `with` $x:c[t]$ `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 25.3 on page 217 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 equivalence
$\mathcal{T} \mid \Delta \vdash \tau$ 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 $\mathcal{T} \vdash t$ cons, and the hypotheses in Γ have the form $x : \tau$ where $\mathcal{X} \vdash x$ exp and $\mathcal{T} \mid \Delta \vdash \tau$ 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 $\Delta \vdash \tau$ type is synonymous with the judgement $\Delta \vdash \tau :: \texttt{Type}$.

Definitional equivalence of constructors plays an important role, because types are particular constructors (those of kind `Type`), and because expression typing respects definitional equivalence of types.

25.1.1 Constructor Formation

The constructor formation judgement, $\Delta \vdash c :: \kappa$, is inductively defined by the following rules:

$$\frac{}{\Delta, t :: \kappa \vdash t :: \kappa} \tag{25.1a}$$

$$\frac{}{\Delta \vdash \texttt{arr} :: \texttt{Arr(Type;Arr(Type;Type))}} \tag{25.1b}$$

$$\frac{}{\Delta \vdash \texttt{all[}\kappa\texttt{]} :: \texttt{Arr(Arr(}\kappa\texttt{;Type);Type)}} \tag{25.1c}$$

$$\frac{}{\Delta \vdash \texttt{some[}\kappa\texttt{]} :: \texttt{Arr(Arr(}\kappa\texttt{;Type);Type)}} \tag{25.1d}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \texttt{pair(}c_1\texttt{;}c_2\texttt{)} :: \texttt{Prod(}\kappa_1\texttt{;}\kappa_2\texttt{)}} \tag{25.1e}$$

$$\frac{\Delta \vdash c :: \texttt{Prod(}\kappa_1\texttt{;}\kappa_2\texttt{)}}{\Delta \vdash \texttt{fst(}c\texttt{)} :: \kappa_1} \tag{25.1f}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \mathtt{snd}(c) :: \kappa_2} \tag{25.1g}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{lambda}[\kappa_1](t.c_2) :: \mathtt{Arr}(\kappa_1;\kappa_2)} \tag{25.1h}$$

$$\frac{\Delta \vdash c_1 :: \mathtt{Arr}(\kappa_2;\kappa) \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{app}(c_1;c_2) :: \kappa} \tag{25.1i}$$

There is an evident correspondence between these rules and the rules for functions and products given in Chapters 14 and 16, 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`$[\kappa]$, and `some`$[\kappa]$ 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`.

25.1.2 Definitional Equivalence of Constructors

Definitional equivalence of well-formed constructors is defined to be the least congruence closed under the following rules:

$$\frac{\Delta \vdash \mathtt{pair}(c_1;c_2) :: \mathtt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \mathtt{fst}(\mathtt{pair}(c_1;c_2)) \equiv c_1 :: \kappa_1} \tag{25.2a}$$

$$\frac{\Delta \vdash \mathtt{pair}(c_1;c_2) :: \mathtt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \mathtt{snd}(\mathtt{pair}(c_1;c_2)) \equiv c_2 :: \kappa_2} \tag{25.2b}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1;\kappa_2)}{\Delta \vdash \mathtt{pair}(\mathtt{fst}(c);\mathtt{snd}(c)) \equiv c :: \mathtt{Prod}(\kappa_1;\kappa_2)} \tag{25.2c}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{app}(\mathtt{lambda}[\kappa](t.c_1);c_2) \equiv [c_2/t]c_1 :: \kappa_2} \tag{25.2d}$$

$$\frac{\Delta \vdash c_2 :: \mathtt{Arr}(\kappa_1;\kappa_2) \quad t \# \Delta}{\Delta \vdash \mathtt{lambda}[\kappa_1](t.\mathtt{app}(c_2;t)) \equiv c_2 :: \mathtt{Arr}(\kappa_1;\kappa_2)} \tag{25.2e}$$

Rules (25.2a) and (25.2b) specify the meaning of the projection constructors when acting on a pair. Rule (25.2c) specifies that every constructor of product kind arises as a pair of projections. Rule (25.2d) specifies that the meaning of a function constructor is given by substitution of an argument into the function body. Rule (25.2e) specifies, moreover, that every constructor of function kind arises in this way.

The role of definitional equivalence is captured by the following rule of kind equivalence:

$$\frac{\Delta\ \Gamma \vdash e : \tau' \quad \Delta \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Delta\ \Gamma \vdash e : \tau} \tag{25.3}$$

In words, definitionally equivalent types (constructors of kind `Type`) classify the *same* expressions.

25.2 Expression Formation

The rules for typing expressions are a straightforward generalization of those given in Chapters 23 and 24.

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{25.4a}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}(x.e_2) : \mathtt{app(app(arr;}\tau_1\mathtt{);}\tau_2\mathtt{)}} \tag{25.4b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{app(app(arr;}\tau_2\mathtt{);}\tau\mathtt{)} \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)} : \tau} \tag{25.4c}$$

$$\frac{\Delta, t :: \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam[}\kappa\mathtt{]}(t.e) : \mathtt{app(all[}\kappa\mathtt{];lambda[}\kappa\mathtt{]}(t.\tau)\mathtt{)}} \tag{25.4d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{app(all[}\kappa\mathtt{];}c'\mathtt{)} \quad \Delta \vdash c :: \kappa}{\Delta\ \Gamma \vdash \mathtt{App[}c\mathtt{]}(e) : \mathtt{app(}c'\mathtt{;}c\mathtt{)}} \tag{25.4e}$$

$$\frac{\Delta \vdash c' :: \mathtt{Arr(}\kappa\mathtt{;Type)} \quad \Delta \vdash c :: \kappa \quad \Delta\ \Gamma \vdash e : \mathtt{app(}c'\mathtt{;}c\mathtt{)}}{\Delta\ \Gamma \vdash \mathtt{pack[}\kappa\mathtt{;}c'\mathtt{;}c\mathtt{]}(e) : \mathtt{app(some[}\kappa\mathtt{];}c'\mathtt{)}} \tag{25.4f}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{app(some[}\kappa\mathtt{];}c\mathtt{)} \quad \Delta, t :: \kappa\ \Gamma, x : \mathtt{app(}c\mathtt{;}t\mathtt{)} \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathtt{type}}{\Delta\ \Gamma \vdash \mathtt{open[}\kappa\mathtt{;}c\mathtt{]}(e_1\mathtt{;}t\mathtt{,}x.e_2\mathtt{)} : \tau_2} \tag{25.4g}$$

25.3 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
		|	`arr`	`arr`
		|	`all[`κ`]`	`all[`κ`]`
		|	`some[`κ`]`	`some[`κ`]`
		|	`pair(`c_1`;`c_2`)`	$\langle c_1, c_2 \rangle$
		|	`fst(`c`)`	`fst(`c`)`
		|	`snd(`c`)`	`snd(`c`)`
		|	`lambda[`κ`](`t`.`c`)`	$\lambda(t::\kappa.\, c)$
		|	`app(`c_1`;`c_2`)`	$c_1[c_2]$
Type	τ	::=	`typ(`c`)`	$\hat{c}$
		|	`arr(`τ_1`;`τ_2`)`	$\tau_1 \rightarrow \tau_2$
		|	`all[`κ`](`t`.`τ`)`	$\forall_\kappa[t::\kappa.\, \tau]$
		|	`some[`κ`](`t`.`τ`)`	$\exists_\kappa[t::\kappa.\, \tau]$

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 equivalence, stated with implicit contexts for brevity:

$$\frac{}{\texttt{typ(app(app(arr;}c_1\texttt{);}c_2\texttt{))} \equiv \texttt{arr(typ(}c_1\texttt{);typ(}c_2\texttt{))} :: \texttt{Type}} \tag{25.5a}$$

$$\frac{}{\texttt{typ(app(all[}\kappa\texttt{];}c\texttt{))} \equiv \texttt{all[}\kappa\texttt{](}t\texttt{.typ(app(}c\texttt{;}t\texttt{)))} :: \texttt{Type}} \tag{25.5b}$$

$$\frac{}{\texttt{typ(app(some[}\kappa\texttt{];}c\texttt{))} \equiv \texttt{some[}\kappa\texttt{](}t\texttt{.typ(app(}c\texttt{;}t\texttt{)))} :: \texttt{Type}} \tag{25.5c}$$

These equivalences specify which types are designated by which constructors. For example, Rule (25.5a) 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 $\mathcal{L}\{\mathtt{typ}, \rightarrow, \forall_\kappa, \exists_\kappa\}$ is that it sheds light on the distinction between predicative and impredicative type quantification discussed in Chapter 23. The universal and existential quantifiers range over constructors of a specified kind. In $\mathcal{L}\{\rightarrow, \forall_\kappa, \exists_\kappa\}$ the kind `Type` includes all types-as-classifiers, and hence quantification is impredicative. In $\mathcal{L}\{\mathtt{typ}, \rightarrow, \forall_\kappa, \exists_\kappa\}$ 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.

25.4 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ given in Chapter 15 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 52), 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. Such languages require a more sophisticated form of dynamic semantics to support run-time dispatch on the form of a type.

25.5 Exercises

Chapter 26

Indexed Families of Types

Part IX

Control Flow

Chapter 27

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, $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. 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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ with the structural operational semantics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

27.1 Machine Definition

A state, s, of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ 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 $k \lhd e$, 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*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{27.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f\,;k\ \mathsf{stack}} \tag{27.1b}$$

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:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{27.2a}$$

$$\frac{}{\mathtt{ifz}(-;e_1;x.e_2)\ \mathsf{frame}} \tag{27.2b}$$

$$\frac{}{\mathtt{ap}(-;e_2)\ \mathsf{frame}} \tag{27.2c}$$

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.

$$\frac{}{k \rhd \mathtt{z} \mapsto k \lhd \mathtt{z}} \tag{27.3a}$$

$$\frac{}{k \rhd \mathtt{s}(e) \mapsto \mathtt{s}(-)\,;k \rhd e} \tag{27.3b}$$

$$\frac{}{\mathtt{s}(-)\,;k \lhd e \mapsto k \lhd \mathtt{s}(e)} \tag{27.3c}$$

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.

$$\frac{}{k \rhd \mathtt{ifz}(e;e_1;x.e_2) \mapsto \mathtt{ifz}(-;e_1;x.e_2)\,;k \rhd e} \tag{27.4a}$$

$$\frac{}{\mathtt{ifz}(-;e_1;x.e_2)\,;k \lhd \mathtt{z} \mapsto k \rhd e_1} \tag{27.4b}$$

$$\overline{\mathtt{ifz}(-;e_1;x.e_2);k \triangleleft \mathtt{s}(e) \mapsto k \triangleright [e/x]e_2} \tag{27.4c}$$

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.

$$\overline{k \triangleright \mathtt{lam}[\tau](x.e) \mapsto k \triangleleft \mathtt{lam}[\tau](x.e)} \tag{27.5a}$$

$$\overline{k \triangleright \mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(-;e_2);k \triangleright e_1} \tag{27.5b}$$

$$\overline{\mathtt{ap}(-;e_2);k \triangleleft \mathtt{lam}[\tau](x.e) \mapsto k \triangleright [e_2/x]e} \tag{27.5c}$$

$$\overline{k \triangleright \mathtt{fix}[\tau](x.e) \mapsto k \triangleright [\mathtt{fix}[\tau](x.e)/x]e} \tag{27.5d}$$

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 42 for more on evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\overline{\epsilon \triangleright e \text{ initial}} \tag{27.6a}$$

$$\frac{e \text{ val}}{\epsilon \triangleleft e \text{ final}} \tag{27.6b}$$

27.2 Safety

To define and prove safety for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ 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:

$$\overline{\epsilon : \tau} \tag{27.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{f;k : \tau} \tag{27.7b}$$

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'.

$$\overline{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{27.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-;e_1;x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{27.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-;e_2) : \mathtt{arr}(\tau_2;\tau) \Rightarrow \tau} \tag{27.8c}$$

$$\frac{e_1 : \mathtt{arr}(\tau_2;\tau) \quad e_1\ \mathsf{val}}{\mathtt{ap}(e_1;-) : \tau_2 \Rightarrow \tau} \tag{27.8d}$$

The two forms of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e\ \mathsf{ok}} \tag{27.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e\ \mathsf{val}}{k \triangleleft e\ \mathsf{ok}} \tag{27.9b}$$

We leave the proof of safety of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ as an exercise.

Theorem 27.1 (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 $s \mapsto s'$.

27.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ correctly implements $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. If we evaluate a given expression, e, using $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, and *vice versa*?

Answering this question decomposes into two conditions relating $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ to $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$:

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 27.2. *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 93 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, $s \leftarrow\!\!\shortmid e$, 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 $s \leftarrow\!\!\shortmid e$. Then we show that if $s \mapsto^* s'$, where s' final, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then e' val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \leftarrow\!\!\shortmid e$ and s final, then e val.

2. If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, $s' \leftarrow\!\!\shortmid e'$, 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 27.3. *If $s \mapsto s'$, $s \not\mapsto e$, and $s' \not\mapsto e'$, then $e \mapsto^* e'$.*

Corollary 27.4. *$e \mapsto^* \overline{n}$ iff $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft \overline{n}$.*

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

27.3.1 Completeness

Proof of Lemma 27.2 on the previous page. The proof is by induction on an evaluation semantics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam[}\tau_2\mathtt{]}(x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{27.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. Applying each of the three inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1;e_2) &\mapsto \mathtt{ap}(-;e_2);k \triangleright e_1 \\ &\mapsto^* \mathtt{ap}(-;e_2);k \triangleleft \mathtt{lam[}\tau_2\mathtt{]}(x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

27.3.2 Soundness

The judgement $s \not\mapsto e'$, 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:

$$\frac{k \bowtie e = e'}{k \triangleright e \not\mapsto e'} \tag{27.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \not\mapsto e'} \tag{27.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{27.12a}$$

$$\frac{k \rhd\lhd \mathtt{s}(e) = e'}{\mathtt{s}(-);k \rhd\lhd e = e'} \tag{27.12b}$$

$$\frac{k \rhd\lhd \mathtt{ifz}(e_1; e_2; x.e_3) = e'}{\mathtt{ifz}(-; e_2; x.e_3);k \rhd\lhd e_1 = e'} \tag{27.12c}$$

$$\frac{k \rhd\lhd \mathtt{ap}(e_1; e_2) = e}{\mathtt{ap}(-; e_2);k \rhd\lhd e_1 = e} \tag{27.12d}$$

These judgements both define total functions.

Lemma 27.5. *The judgement $s \mathrel{\rightarrow\!\!\!\!\!\circ}\; e$ has mode $(\forall, \exists!)$, and the judgement $k \rhd\lhd e = e'$ 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 $k \rhd\lhd e$ for the unique e' such that $k \rhd\lhd e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

Lemma 27.6. *If $e \mapsto e'$, $k \rhd\lhd e = d$, $k \rhd\lhd e' = d'$, 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 $k \rhd\lhd e = d$ and $k \rhd\lhd e' = d'$. It follows from Rules (27.12) that $\mathtt{ap}(-; e_2);k \rhd\lhd e_1 = d$ and $\mathtt{ap}(-; e_2);k \rhd\lhd e_1' = d'$. 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 $k \rhd\lhd e = d$ and $k \rhd\lhd e' = d'$; 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 (27.12) that $k' \rhd\lhd \mathtt{ap}(e; c_2) = d$ and $k' \rhd\lhd \mathtt{ap}(e'; c_2) = d'$. 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 27.3 on the facing page.

Proof of Lemma 27.3 on the preceding page. The proof is by case analysis on the transitions of $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = \mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $\mathtt{s}(-);k \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = \mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-);k \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $\mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-);k \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}[\tau](x.e_1);e_2) = e'$. The result follows from Lemma 27.6 on the preceding page.

□

27.4 Exercises

Chapter 28

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.

28.1 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	::=	`fail[`τ`]`	`fail`
		|	`catch(`e_1`;`e_2`)`	`try` e_1 `ow` e_2

The expression `fail[`τ`]` aborts the current evaluation. The expression `catch(`e_1`;`e_2`)`

evaluates e_1. If it terminates normally, its value is returned; if it fails, its value is the value of e_2.

The static semantics of failures is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail[}\tau\texttt{]} : \tau} \tag{28.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1;e_2) : \tau} \tag{28.1b}$$

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 27, 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:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{catch(}-\texttt{;}e_2\texttt{)} \ \mathsf{frame}} \tag{28.2}$$

The transition rules of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail[}\tau\texttt{]} \mapsto k \blacktriangleleft} \tag{28.3a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1;e_2) \mapsto \texttt{catch}(-;e_2);k \triangleright e_1} \tag{28.3b}$$

$$\frac{}{\texttt{catch}(-;e_2);k \triangleleft v \mapsto k \triangleleft v} \tag{28.3c}$$

$$\frac{}{\texttt{catch}(-;e_2);k \blacktriangleleft \ \mapsto k \triangleright e_2} \tag{28.3d}$$

$$\frac{(f \neq \texttt{catch}(-;e_2))}{f;k \blacktriangleleft \ \mapsto k \blacktriangleleft} \tag{28.3e}$$

Evaluating $\texttt{fail[}\tau\texttt{]}$ propagates a failure up the stack. Evaluating $\texttt{catch}(e_1;e_2)$ 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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{28.4a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}} \tag{28.4b}$$

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 27 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 28.1 (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 $s \mapsto s'$.

28.2 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	::=	$\mathtt{raise[}\tau\mathtt{]}(e)$	$\mathtt{raise}(\tau)e$
		|	$\mathtt{handle}(e_1; x.e_2)$	$\mathtt{try}\, e_1\, \mathtt{ow}\, x \Rightarrow e_2$

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{handle}(e_1; x.e_2)$ 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 28.1 on page 233. 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:

$$\frac{}{\texttt{raise}[\tau](-)\ \mathsf{frame}} \tag{28.5a}$$

$$\frac{}{\texttt{handle}(-;x.e_2)\ \mathsf{frame}} \tag{28.5b}$$

The rules for evaluating exceptions are as follows:

$$\frac{}{k \triangleright \texttt{raise}[\tau](e) \mapsto \texttt{raise}[\tau](-);k \triangleright e} \tag{28.6a}$$

$$\frac{}{\texttt{raise}[\tau](-);k \triangleleft e \mapsto k \blacktriangleleft e} \tag{28.6b}$$

$$\frac{}{\texttt{raise}[\tau](-);k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.6c}$$

$$\frac{}{k \triangleright \texttt{handle}(e_1;x.e_2) \mapsto \texttt{handle}(-;x.e_2);k \triangleright e_1} \tag{28.6d}$$

$$\frac{}{\texttt{handle}(-;x.e_2);k \triangleleft e \mapsto k \triangleleft e} \tag{28.6e}$$

$$\frac{}{\texttt{handle}(-;x.e_2);k \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{28.6f}$$

$$\frac{(f \neq \texttt{handle}(-;x.e_2))}{f;k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.6g}$$

The static semantics of exceptions generalizes that of failures.

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau} \tag{28.7a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1;x.e_2) : \tau} \tag{28.7b}$$

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 e1 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 36.

28.3 Exercises

Chapter 29

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.

29.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type `cont(`τ`)` of continuations accepting values of type τ. The introduction form for `cont(`τ`)` is `letcc[`τ`](`x`.`e`)`, which binds the *current continuation* (that is, the current control stack) to the variable x, and evaluates the expression e. The corresponding elimination form is `throw[`τ`](`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 `nat` $\rightarrow$ `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 `letcc[`τ`](`x`.`e`)`, write `SMLofNJ.Cont.callcc (fn` x `=>` e`)`, and for `throw[`τ`](`e_1`;`e_2`)`, write `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 $\tau' \to \tau$, 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:`$\tau' \to \tau$`,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!

29.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\to\}$ expressions with these additional forms:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`cont(`τ`)`	τ `cont`
Expr	e	::=	`letcc[`τ`](`$x.e$`)`	`letcc` x `in` e
		|	`throw[`τ`](`$e_1;e_2$`)`	`throw` e_1 `to` e_2
		|	`cont(`k`)`	

The expression `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:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1; e_2) : \tau'} \tag{29.1b}$$

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:

$$\frac{k : \tau}{\Gamma \vdash \mathtt{cont}(k) : \mathtt{cont}(\tau)} \tag{29.2}$$

A continuation value $\mathtt{cont}(k)$ has type $\mathtt{cont}(\tau)$ exactly if it is a stack accepting values of type τ.

To define the dynamic semantics, we extend $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \ \mathsf{exp}}{\mathtt{throw}[\tau]\,(-; e_2) \ \mathsf{frame}} \tag{29.3a}$$

$$\frac{e_1 \ \mathsf{val}}{\mathtt{throw}[\tau]\,(e_1; -) \ \mathsf{frame}} \tag{29.3b}$$

Every reified control stack is a value:

$$\frac{k \ \mathsf{stack}}{\mathtt{cont}(k) \ \mathsf{val}} \tag{29.4}$$

The transition rules for the continuation constructs are as follows:

$$\overline{k \rhd \mathtt{letcc}[\tau]\,(x.e) \mapsto k \rhd [\mathtt{cont}(k)/x]e} \tag{29.5a}$$

$$\overline{\mathtt{throw}[\tau]\,(v; -); k \lhd \mathtt{cont}(k') \mapsto k' \lhd v} \tag{29.5b}$$

$$\overline{k \rhd \mathtt{throw}[\tau]\,(e_1; e_2) \mapsto \mathtt{throw}[\tau]\,(-; e_2); k \rhd e_1} \tag{29.5c}$$

$$\frac{e_1 \ \mathsf{val}}{\mathtt{throw}[\tau]\,(-; e_2); k \lhd e_1 \mapsto \mathtt{throw}[\tau]\,(e_1; -); k \rhd e_2} \tag{29.5d}$$

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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ given in Chapter 27.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau]\,(-; e_2) : \tau \Rightarrow \tau'} \tag{29.6a}$$

$$\frac{e_1 : \tau \quad e_1 \ \mathsf{val}}{\mathtt{throw}[\tau]\,(e_1; -) : \mathtt{cont}(\tau) \Rightarrow \tau'} \tag{29.6b}$$

The rest of the definitions remain as in Chapter 27.

Lemma 29.1 (Canonical Forms)**.** *If $e : \mathtt{cont}(\tau)$ and e val, then $e = \mathtt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 29.2 (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 $s \mapsto s'$.

29.3 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

$$\texttt{ap}(f;-);\texttt{throw}[\tau](-;k);\epsilon$$

   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 30

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.

30.1 Constructive Logic

30.1.1 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 (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 (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 $P \stackrel{?}{=} NP$ 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 ϕ prop and ϕ true are basic, or *categorical*, judgements. These are the building blocks of reason, but they are rarely of interest by themselves. Rather, we are interested in the more general case of the *hypothetical judgement*, or *consequence relation*, of the form

$$\phi_1 \text{ true}, \ldots, \phi_n \text{ true} \vdash \phi \text{ true}.$$

This judgement expresses that the proposition ϕ is true (has a proof), *under the assumptions* that each of $\phi_1, \ldots, \phi_n$ are also true (have proofs). Of course, when $n = 0$ this is just the same as the categorical judgement ϕ true. We let Γ range over finite sets of assumptions.

The hypothetical judgement satisfies the following *structural properties*, which characterize what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{30.1a}$$

$$\frac{\Gamma \vdash \phi \text{ true} \quad \Gamma, \phi \text{ true} \vdash \psi \text{ true}}{\Gamma \vdash \psi \text{ true}} \tag{30.1b}$$

$$\frac{\Gamma \vdash \psi \text{ true}}{\Gamma, \phi \text{ true} \vdash \psi \text{ true}} \tag{30.1c}$$

$$\frac{\Gamma, \phi \text{ true}, \phi \text{ true} \vdash \theta \text{ true}}{\Gamma, \phi \text{ true} \vdash \theta \text{ true}} \tag{30.1d}$$

$$\frac{\Gamma, \psi \text{ true}, \phi \text{ true}, \Gamma' \vdash \theta \text{ true}}{\Gamma, \phi \text{ true}, \psi \text{ true}, \Gamma' \vdash \theta \text{ true}} \tag{30.1e}$$

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.

30.1.2 Propositional Logic

The syntax of propositional logic is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Prop	ϕ	::=	`true`	$\top$
		|	`false`	$\bot$
		|	`and`$(\phi_1;\phi_2)$	$\phi_1 \wedge \phi_2$
		|	`or`$(\phi_1;\phi_2)$	$\phi_1 \vee \phi_2$
		|	`imp`$(\phi_1;\phi_2)$	$\phi_1 \supset \phi_2$

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.

$$\frac{}{\Gamma \vdash \top\ \mathsf{true}} \tag{30.2a}$$

(no elimination rule)

(30.2b)

Conjunction Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi\ \mathsf{true} \quad \Gamma \vdash \psi\ \mathsf{true}}{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}} \tag{30.3a}$$

$$\frac{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Gamma \vdash \phi\ \mathsf{true}} \tag{30.3b}$$

$$\frac{\Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Gamma \vdash \psi\ \mathsf{true}} \tag{30.3c}$$

Implication Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi \text{ true} \vdash \psi \text{ true}}{\Gamma \vdash \phi \supset \psi \text{ true}} \tag{30.4a}$$

$$\frac{\Gamma \vdash \phi \supset \psi \text{ true} \quad \Gamma \vdash \phi \text{ true}}{\Gamma \vdash \psi \text{ true}} \tag{30.4b}$$

Falsehood Falsehood expresses the trivially false (refutable) proposition.

(no introduction rule)

(30.5a)

$$\frac{\Gamma \vdash \bot \text{ true}}{\Gamma \vdash \phi \text{ true}} \tag{30.5b}$$

Disjunction Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi \text{ true}}{\Gamma \vdash \phi \vee \psi \text{ true}} \tag{30.6a}$$

$$\frac{\Gamma \vdash \psi \text{ true}}{\Gamma \vdash \phi \vee \psi \text{ true}} \tag{30.6b}$$

$$\frac{\Gamma \vdash \phi \vee \psi \text{ true} \quad \Gamma, \phi \text{ true} \vdash \theta \text{ true} \quad \Gamma, \psi \text{ true} \vdash \theta \text{ true}}{\Gamma \vdash \theta \text{ true}} \tag{30.6c}$$

30.1.3 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.

$$\frac{}{\Gamma \vdash \texttt{trueI} : \top} \tag{30.7a}$$

$$\frac{\Gamma \vdash p : \phi \quad \Gamma \vdash q : \psi}{\Gamma \vdash \texttt{andI}(p;q) : \phi \wedge \psi} \tag{30.7b}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{andEl}(p) : \phi} \tag{30.7c}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{andEr}(p) : \psi} \tag{30.7d}$$

$$\frac{\Gamma, x : \phi \vdash p : \psi}{\Gamma \vdash \texttt{impI}[\phi](x\,.\,p) : \phi \supset \psi} \tag{30.7e}$$

$$\frac{\Gamma \vdash p : \phi \supset \psi \quad \Gamma \vdash q : \phi}{\Gamma \vdash \texttt{impE}(p;q) : \psi} \tag{30.7f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \texttt{falseE}[\phi](p) : \phi} \tag{30.7g}$$

$$\frac{\Gamma \vdash p : \phi}{\Gamma \vdash \texttt{orIl}[\psi](p) : \phi \vee \psi} \tag{30.7h}$$

$$\frac{\Gamma \vdash p : \psi}{\Gamma \vdash \texttt{orIr}[\phi](p) : \phi \vee \psi} \tag{30.7i}$$

$$\frac{\Gamma \vdash p : \phi \vee \psi \quad \Gamma, x : \phi \vdash q : \theta \quad \Gamma, y : \psi \vdash r : \theta}{\Gamma \vdash \texttt{orE}[\phi;\psi](p;x\,.\,q;y\,.\,r) : \theta} \tag{30.7j}$$

30.2 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`
$\phi \wedge \psi$	$\phi^* \times \psi^*$
$\phi \supset \psi$	$\phi^* \to \psi^*$
$\phi \vee \psi$	$\phi^* + \psi^*$

The correspondence extends to proofs and programs as well:

Proof	*Program*
`trueI`	`triv`
`falseE[`ϕ`](`p`)`	`abort[`ϕ^*`](`p^*`)`
`andI(`p`;`q`)`	`pair(`p^*`;`q^*`)`
`andEl(`p`)`	`fst(`p^*`)`
`andEr(`p`)`	`snd(`p^*`)`
`impI[`ϕ`](`x`.`p`)`	`lam[`ϕ^*`](`x`.`p^*`)`
`impE(`p`;`q`)`	`ap(`p^*`;`q^*`)`
`orIl[`ψ`](`p`)`	`in[l][`ψ^*`](`p^*`)`
`orIr[`ϕ`](`p`)`	`in[r][`ϕ^*`](`p^*`)`
`orE[`ϕ`;`ψ`](`p`;`x`.`q`;`y`.`r`)`	`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 30.1 (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:

$$\begin{aligned}
\texttt{andEl(andI(}p\texttt{;}q\texttt{))} &\mapsto p \\
\texttt{andEr(andI(}p\texttt{;}q\texttt{))} &\mapsto q \\
\texttt{impE(impI[}\phi\texttt{](}x\texttt{.}q\texttt{);}p\texttt{)} &\mapsto [p/x]q \\
\texttt{orE[}\phi\texttt{;}\psi\texttt{](orIl[}\psi\texttt{](}p\texttt{);}x\texttt{.}q\texttt{;}y\texttt{.}r\texttt{)} &\mapsto [p/x]q \\
\texttt{orE[}\phi\texttt{;}\psi\texttt{](orIr[}\phi\texttt{](}p\texttt{);}x\texttt{.}q\texttt{;}y\texttt{.}r\texttt{)} &\mapsto [p/y]r
\end{aligned}$$

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.

30.3 Exercises

Chapter 31

Classical Proofs and Control Operators

In Chapter 30 we saw that constructive logic is a logic of positive information in that the meaning of the judgement ϕ true is that there exists a proof of ϕ. A refutation of a proposition ϕ consists of a proof of the hypothetical judgement ϕ true $\vdash \bot$ true, asserting that the assumption of ϕ leads to a proof of logical falsehood (a contradiction). Since there are propositions, ϕ, for which we possess neither a proof nor a refutation, we cannot assert, in general, $\phi \vee \neg\phi$ true.

By contrast classical logic (the one we all learned in school) maintains a complete symmetry between truth and falsehood — that which is not true is false and that which is not false is true. Obviously such an interpretation conflicts with the constructive interpretation, for lack of a proof of a proposition is not a refutation, nor is lack of a refutation a proof.[1] In this sense classical logic is a logic of perfect information, in which all mathematical problems have 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.

31.1 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

$$\phi_1 \text{ false}, \ldots, \phi_m \text{ false}; \psi_1 \text{ true}, \ldots, \psi_n \text{ true} \vdash J,$$

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:

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash k \# p} \tag{31.1a}$$

The reflexivity rules capture the meaning of hypotheses:

$$\frac{}{\Delta, u \div \phi;\Gamma \vdash u \div \phi} \tag{31.1b}$$

$$\frac{}{\Delta;\Gamma, x : \psi \vdash x : \phi} \tag{31.1c}$$

Truth and falsity are complementary:

$$\frac{\Delta, u \div \phi;\Gamma \vdash k \# p}{\Delta;\Gamma \vdash \mathtt{ccr}(u \div \phi . k \# p) : \phi} \tag{31.1d}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash k \# p}{\Delta;\Gamma \vdash \mathtt{ccp}(x : \phi . k \# p) \div \phi} \tag{31.1e}$$

In both of these rules the entire contradiction, $k \# p$, 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.

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \top} \tag{31.1f}$$

$$\frac{}{\Delta;\Gamma \vdash \mathtt{abort} \div \bot} \tag{31.1g}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash q : \psi}{\Delta;\Gamma \vdash \langle p, q\rangle : \phi \wedge \psi} \tag{31.1h}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi}{\Delta;\Gamma \vdash \mathtt{fst};k \div \phi \wedge \psi} \tag{31.1i}$$

$$\frac{\Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \mathtt{snd};k \div \phi \wedge \psi} \tag{31.1j}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash p : \psi}{\Delta;\Gamma \vdash \lambda(x : \phi . p) : \phi \supset \psi} \tag{31.1k}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \mathtt{app}(p);k \div \phi \supset \psi} \tag{31.1l}$$

$$\frac{\Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash \mathtt{inl}(p) : \phi \vee \psi} \tag{31.1m}$$

$$\frac{\Delta;\Gamma \vdash p : \psi}{\Delta;\Gamma \vdash \mathtt{inr}(p) : \phi \vee \psi} \tag{31.1n}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash l \div \psi}{\Delta;\Gamma \vdash \mathtt{case}(k;l) \div \phi \vee \psi} \tag{31.1o}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi}{\Delta;\Gamma \vdash \mathtt{not}(k) : \neg\phi} \tag{31.1p}$$

$$\frac{\Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash \mathtt{not}(p) \div \neg\phi} \tag{31.1q}$$

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.

$$\mathtt{fst};k \# \langle p, q\rangle \mapsto k \# p \tag{31.2a}$$

$$\mathtt{snd};k \# \langle p, q\rangle \mapsto k \# q \tag{31.2b}$$

$$\mathtt{case}(k;l) \# \mathtt{inl}(p) \mapsto k \# p \tag{31.2c}$$

$$\mathtt{case}(k;l) \# \mathtt{inr}(q) \mapsto l \# q \tag{31.2d}$$

$$\mathtt{app}(p);k \# \lambda(x{:}\phi.\, q) \mapsto k \# [p/x]q \tag{31.2e}$$

$$\mathtt{not}(p) \# \mathtt{not}(k) \mapsto k \# p \tag{31.2f}$$

The symmetry of the transition rule for negation is particularly elegant.

Here are the rules for the generic primitives relating truth and falsity.

$$\mathtt{ccp}(x{:}\phi.k \# p) \# q \mapsto [q/x]k \# [q/x]p \tag{31.2g}$$

$$k \# \mathtt{ccr}(u \div \phi.l \# p) \mapsto [k/u]l \# [k/u]p \tag{31.2h}$$

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 (31.2g) to (31.2h) overlap in the sense that there are two possible transitions for a state of the form

$$\texttt{ccp}(x:\phi.k \mathbin{\#} p) \mathbin{\#} \texttt{ccr}(u \div \phi.l \mathbin{\#} q).$$

This state may transition either to the state

$$[r/x]k \mathbin{\#} [r/x]p,$$

where r is $\texttt{ccr}(u \div \phi.l \mathbin{\#} q)$, or to the state

$$[m/u]l \mathbin{\#} [m/u]q,$$

where m is $\texttt{ccp}(x:\phi.k \mathbin{\#} p)$, and these are not equivalent.

There are two possible attitudes about this. One is to simply accept that classical logic has a non-deterministic dynamic semantics, and leave it at that. But this means that it is difficult to predict the outcome of a computation, since it could be radically different in the case of the overlapping state just described. The alternative is to impose an arbitrary priority ordering among the two cases, either preferring the first transition to the second, or *vice versa*. Preferring the first corresponds, very roughly, to a "lazy" semantics for proofs, because we pass the unevaluated proof, r, to the refutation on the left, which is thereby activated. Preferring the second corresponds to an "eager" semantics for proofs, in which we pass the unevaluated refutation, m, to the proof, which is thereby activated. Dually, these choices correspond to an "eager" semantics for refutations in the first case, and a "lazy" one for the second. Take your pick.

The final issue is the initial state: how is computation to be started? Or, equivalently, when is it finished? The difficulty is that we need both a proof and a refutation of the same proposition! While this can easily come up in the "middle" of a proof, it would be impossible to have a finished proof and a finished refutation of the same proposition! The solution for an eager interpretation of proofs (and, correspondingly, a lazy interpretation of refutations) is simply to postulate an initial (or final, depending on your point of view) refutation, `halt`, and to deem a state of the form $\texttt{halt} \mathbin{\#} p$ 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 $k \mathbin{\#} \texttt{halt}$ as initial, and also final, provided that k is not a "ccp" instruction.

31.2 Exercises

Part XI

Subtyping

Chapter 32

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. 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.

32.1 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:

$$\frac{}{\tau <: \tau} \tag{32.1a}$$

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \tag{32.1b}$$

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*:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{32.2}$$

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 $\sigma <: \tau$.

32.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 32.4 on page 270 we will examine some of these in more detail from the point of view of type safety.

32.2.1 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 (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 (that is, as their 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.

32.2.2 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:

$$\frac{n \leq m}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \cdot \tag{32.3}$$

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:

$$\frac{n \leq m}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \cdot \tag{32.4}$$

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:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \ . \tag{32.5}$$

This rule generalizes the preceding rules, using the represetation of tuples and records as finite products described in Chapter 16.

32.2.3 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:

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{32.6}$$

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:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j} \ . \tag{32.7}$$

Note well the reversal of the containment as compared to Rule (32.5).

32.3 Variance

In addition to basic subtyping principles such as those considered in Section 32.2 on page 264, 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.

32.3.1 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:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle \sigma_1, \ldots, \sigma_n \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \ . \tag{32.8}$$

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:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \ . \tag{32.9}$$

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:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\prod_{i \in I} \sigma_i <: \prod_{i \in I} \tau_i} \tag{32.10}$$

32.3.2 Sums

Both unlabelled and labelled sum types are covariant in each position:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{[l_1 : \sigma_1, \ldots, l_n : \sigma_n] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \ . \tag{32.11}$$

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:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\sum_{i \in I} \sigma_i <: \sum_{i \in I} \tau_i} \tag{32.12}$$

32.3.3 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 $e : \sigma \to \tau$. 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:

$$\frac{\tau <: \tau'}{\sigma \to \tau <: \sigma \to \tau'} \tag{32.13}$$

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 $\sigma' \to \tau$.

$$\frac{\sigma' <: \sigma}{\sigma \to \tau <: \sigma' \to \tau} \tag{32.14}$$

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!

32.4 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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle l_1 = e_1, \ldots, l_n = e_n \rangle : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{32.15a}$$

$$\frac{\Gamma \vdash e : \langle l : \tau \rangle}{\Gamma \vdash e \cdot l : \tau} \tag{32.15b}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{32.15c}$$

$$\frac{n \leq m}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{32.15d}$$

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{32.15e}$$

Rule (32.15b) is simplified compared to Rule (16.3b), 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 32.1 (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 32.2 (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:

$$\frac{e_1 \text{ val} \quad \ldots \quad e_n \text{ val}}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \text{ val}} \tag{32.16a}$$

$$\frac{\begin{array}{c} e_1 \text{ val} \quad e_1' = e_1 \quad \ldots \quad e_{i-1} \text{ val} \quad e_{i-1}' = e_{i-1} \\ e_i \mapsto e_i' \quad e_{i+1}' = e_{i+1} \quad \ldots \quad e_n' = e_n \end{array}}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \mapsto \langle l_1 = e_1', \ldots, l_n = e_n' \rangle} \tag{32.16b}$$

$$\frac{e \mapsto e'}{e \cdot l \mapsto e' \cdot l} \tag{32.16c}$$

$$\frac{e_1 \text{ val} \quad \ldots \quad e_n \text{ val} \quad 1 \leq i \leq n}{\langle l_1 = e_1, \ldots, l_n = e_n \rangle \cdot l_i \mapsto e_i} \tag{32.16d}$$

Theorem 32.3 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By induction on Rules (32.16). For example, consider Rule (32.16d). 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 32.4 (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 32.5 (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 (32.15b), 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 (32.16c) $e \cdot l \mapsto e' \cdot l$. 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. □

32.5 Recursive Subtyping

Consider the types of labelled binary trees with natural numbers at each node,

$$\mu t.\,[\texttt{empty} : \texttt{unit}, \texttt{binode} : \langle \texttt{data} : \texttt{nat}, \texttt{lft} : t, \texttt{rht} : t \rangle],$$

and of "bare" binary trees, without labels on the nodes,

$$\mu t.\,[\texttt{empty} : \texttt{unit}, \texttt{binode} : \langle \texttt{lft} : t, \texttt{rht} : t \rangle].$$

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:

$$\mu t.\,[\texttt{empty} : \texttt{unit}, \texttt{binode} : \langle \texttt{lft} : t, \texttt{rht} : t \rangle, \texttt{trinode} : \langle \texttt{lft} : t, \texttt{mid} : t, \texttt{rht} : t \rangle].$$

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:

$$\frac{t \mid \sigma <: \tau}{\mu t.\sigma <: \mu t.\tau} \ \textbf{??} \tag{32.17}$$

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 (32.17) 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 (32.17) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\sigma = \mu t.\langle \mathtt{a}:t \to \mathtt{nat}, \mathtt{b}:t \to \mathtt{int}\rangle$$

and

$$\tau = \mu t.\langle \mathtt{a}:t \to \mathtt{int}, \mathtt{b}:t \to \mathtt{int}\rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using Rule (32.17) 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 $q : \mathtt{nat} \to \mathtt{nat}$ is the discrete square root function. Since $\sigma <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(e) : \langle \mathtt{a}:\tau \to \mathtt{nat}, \mathtt{b}:\tau \to \mathtt{int}\rangle.$$

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 (32.17) 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:

$$\frac{\mu s.\,\sigma <: \mu t.\,\tau \vdash [\mu s.\,\sigma/s]\sigma <: [\mu t.\,\tau/t]\tau}{\mu s.\,\sigma <: \mu t.\,\tau}\ . \tag{32.18}$$

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 (32.18) makes use of parameters, rather than substitution.

$$\frac{s, t \mid s <: t \vdash \sigma <: \tau}{\mu s.\,\sigma <: \mu t.\,\tau}\ . \tag{32.19}$$

It is easy to verify that each rule is admissible in the presence of the other.

32.6 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:

$$\frac{\sigma <: \tau}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \quad \textbf{??}$$

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:

$$\frac{\tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \quad \textbf{??}$$

Combining these two observations, we see that reference types are *invariant*:

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \tag{32.20}$$

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).

32.7 Exercises

1. Consider the subtyping issues related to signed and unsigned, fixed precision integer types.

[1] Please see Chapter 35 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 33

Singleton and Dependent Kinds

The expression $\texttt{let}\, e_1 : \tau\, \texttt{be}\, x\, \texttt{in}\, e_2$ 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 $\lambda(x:\tau.\, e_2)(e_1)$, 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

$$\texttt{let}\, t\, \texttt{be}\, \texttt{nat} \times \texttt{nat}\, \texttt{in}\, \lambda(x:t.\, \texttt{s}(\texttt{fst}(x))).$$

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 $\Lambda(t.e)[\tau]$, 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 $\texttt{s}(\texttt{fst}(x))$ 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 $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$ would be taken as a primitive form of expression whose static semantics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e : \tau'} \tag{33.1}$$

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.

33.1 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 $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$, 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 $\texttt{Eqv}(\tau)$ 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 $u :: \texttt{Eqv}(\tau)$ is a variable of singleton kind, then within its scope, the variable u is synonymous with τ. Thus we may represent $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$ by

$\Lambda(t::\mathtt{Eqv}(\tau).e)[\tau]$, 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 $\mathtt{Eqv}(\tau) :<: \mathtt{Type}$, 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 $\mathtt{Eqv}(\tau)$. 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 $\mathtt{Eqv}(\tau)$ 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 $\mathtt{Eqv}(c::\kappa)$, 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 34

Fluid Binding

In Chapter 13 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 in Chapter 6, is the static range of significance of a variable. Within a specified phrase the variable serves as an (unambiguous) reference to its binding site, much as a pronoun refers to the noun to which it is associated (albeit ambiguously) in a sentence. Outside of its scope a variable has no meaning at all, just as it makes no sense (in English) to refer to "her" in a context in which no female person has been identified as its referent. Static scope is fundamental to modularity, since it strictly segregates the variables introduced in one program unit from those introduced in another. Were this not the case, integrating components would be prone to errors arising from confusing two variables in two different program units that happen to have the same name.

The *extent* of a variable is its *dynamic* range of significance, the period of execution during which the variable has meaning. In a statically scoped

language the scope and the extent of a variable coincide. When entering the scope of a variable during execution, the variable is replaced by the value to which it refers—in effect, replacing the pronoun by the noun to which it refers. Since the static scope of the variable is a particular phrase, replacement of a variable by its binding consists of a process of substitution that is defined by structural induction on the phase in which the variable may occur.

In a dynamically scoped language the scope and the extent of a variable are distinguished by separating the declaration of a variable within a scope from its association with a value (or even several different values) during execution. When entering the scope of a variable, it becomes available for use during execution. One way to use it is to retrieve its binding (if any!) to a value. If the variable is unbound, execution aborts with an error; otherwise, the value to which it is bound is returned. Another way to use such a variable is to bind it to a value within a particular expression, the *extent* of its binding. While executing that expression the variable has the specified binding; outside of it the variable reverts to its previous binding (or unbound state).

Although the terminology may suggest otherwise, the concepts of static and dynamic scope are not *opposed* to one another, but are rather two *differnt* concepts that may co-exist in a language. We will use the term *fluid binding* for the separation between the scope and the extent of a class of dynamically scoped variables, which we will call *symbols*. A new symbol is introduced for use within a scope by a *symbol generation* primitive. A binding is associated with an active symbol within a scope by a *fluid let* construct.

We will study a language fragment, called $\mathcal{L}\{\texttt{symb}\}$, with fluid binding of symbols to values. In Section 34.1 we will consider the mechanisms of fluid binding for a fixed collection of symbols. Then in Section 34.2 on page 287 we will add the mechanisms required to create new symbols during execution of a program.

34.1 Fluid Binding

The syntax of $\mathcal{L}\{\texttt{symb}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\texttt{set[}a\texttt{]}(e_1;e_2)$	$\texttt{set}\ a\ \texttt{to}\ e_1\ \texttt{in}\ e_2$
		|	$\texttt{get[}a\texttt{]}$	$\texttt{get}\ a$

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 $\mathcal{L}\{\texttt{symb}\}$ 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 generic 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 34.2 on page 287, 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 $\mathcal{L}\{\texttt{symb}\}$ are given as follows:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \texttt{get}[a] : \tau} \tag{34.1a}$$

$$\frac{\Sigma \vdash a : \tau_1 \quad \Sigma\,\Gamma \vdash e_1 : \tau_1 \quad \Sigma\,\Gamma \vdash e_2 : \tau_2}{\Sigma\,\Gamma \vdash \texttt{set}[a]\,(e_1; e_2) : \tau_2} \tag{34.1b}$$

Rule (34.1b) treats the symbol a as given by Σ, rather than introducing a "new" symbol, as would be the case for a statically scoped `let` construct: neither Σ nor Γ are extended in Rule (34.1b).

The dynamic semantics of $\mathcal{L}\{\texttt{symb}\}$ 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 $a \in \mathit{dom}(\theta)$, then we may write $\theta = \theta' \otimes \langle a : e \rangle$. If $a \notin \mathit{dom}(\theta)$, 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 the "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 $\mathcal{L}\{\texttt{symb}\}$ is given by the following rules:

$$\frac{}{\texttt{get}[a] \longmapsto_{\theta \otimes \langle a : e \rangle} e} \tag{34.2a}$$

$$\frac{e_1 \longmapsto_\theta e_1'}{\texttt{set}[a]\,(e_1; e_2) \longmapsto_\theta \texttt{set}[a]\,(e_1'; e_2)} \tag{34.2b}$$

$$\frac{e_1 \text{ val} \quad e_2 \longmapsto_{\theta \otimes \langle a : e_1 \rangle} e_2'}{\texttt{set}[a]\,(e_1; e_2) \longmapsto_{\theta \otimes \langle a : \_ \rangle} \texttt{set}[a]\,(e_1; e_2')} \tag{34.2c}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\texttt{set}[a]\,(e_1; e_2) \longmapsto_\theta e_2} \tag{34.2d}$$

Rule (34.2a) specifies that $\texttt{get}[a]$ evaluates to the current binding of a, if any. Rule (34.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (34.2c) 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 (34.2d) 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 $\texttt{get}[a]$, 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 $\texttt{get}[a] \longmapsto_{\theta \otimes \langle a : \bullet \rangle} e$ 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]

$$\frac{}{\texttt{get}[a] \ \mathsf{unbound}_{\theta \otimes \langle a : \bullet \rangle}} \tag{34.3a}$$

$$\frac{e_1 \ \mathsf{unbound}_\theta}{\texttt{set}[a]\,(e_1; e_2) \ \mathsf{unbound}_\theta} \tag{34.3b}$$

$$\frac{e_1 \text{ val} \quad e_2 \ \mathsf{unbound}_\theta}{\texttt{set}[a]\,(e_1; e_2) \ \mathsf{unbound}_\theta} \tag{34.3c}$$

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.

[1] In the presence of other constructs stuck states would have to be propagated through the evaluated arguments of a compound expression.

In preparation for proving the type safety of $\mathcal{L}\{\texttt{symb}\}$, we define the auxiliary judgement $\theta : \Sigma$ by the following rules:

$$\frac{}{\varnothing : \varnothing} \tag{34.4a}$$

$$\frac{\Sigma \vdash e : \tau \quad \theta : \Sigma}{\theta \otimes \langle a : e \rangle : \Sigma, a : \tau} \tag{34.4b}$$

$$\frac{\theta : \Sigma}{\theta \otimes \langle a : \bullet \rangle : \Sigma, a : \tau} \tag{34.4c}$$

These rules specify that if a symbol is bound to a value, then that value must be of the type associated to the symbol by Σ. No demand is made in the case that the symbol is unbound (that is, bound to the "black hole").

Theorem 34.1 (Preservation)**.** *If $e \mapsto_\theta e'$, where $\theta : \Sigma$ and $\Sigma \vdash e : \tau$, then $\Sigma \vdash e' : \tau$.*

Proof. By rule induction on Rules (34.2). Rule (34.2a) is handled by the definition of $\theta : \Sigma$. Rule (34.2b) follows immediately by induction. Rule (34.2d) is handled by inversion of Rules (34.1). Finally, Rule (34.2c) is handled by inversion of Rules (34.1) and induction. □

Theorem 34.2 (Progress)**.** *If $\Sigma \vdash e : \tau$ and $\theta : \Sigma$, then either e val, or e $\mathsf{unbound}_\theta$, or there exists e' such that $e \mapsto_\theta e'$.*

Proof. By induction on Rules (34.1). For Rule (34.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\theta : \Sigma$, we have either $\theta(a) = \bullet$ (unbound) or $\theta(a) = e$ for some e such that $\Sigma \vdash e : \tau$. In the former case we have e $\mathsf{unbound}_\theta$, and in the latter we have $\texttt{get[}a\texttt{]} \mapsto_\theta e$. For Rule (34.1b), we have by induction that either e_1 val or e_1 $\mathsf{unbound}_\theta$, or $e_1 \mapsto_\theta e_1'$. In the latter two cases we may apply Rule (34.2b) or Rule (34.3b), respectively. If e_1 val, we apply induction to obtain that either e_2 val, in which case Rule (34.2d) applies; e_2 $\mathsf{unbound}_\theta$, in which case Rule (34.3b) applies; or $e_2 \mapsto_\theta e_2'$, in which case Rule (34.2c) applies. □

34.2 Symbol Generation

The language $\mathcal{L}\{\texttt{symb gen}\}$ enriches $\mathcal{L}\{\texttt{symb}\}$ with the ability to generate new symbols during execution. The syntax of this extension is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{sym}(\sigma;\tau)$	$\langle\sigma\rangle\tau$
Expr	e	::=	$\texttt{new[}\sigma\texttt{]}(a.e)$	$\nu(a{:}\sigma.e)$
		|	$\texttt{gen}(e)$	$\texttt{gen}(e)$

The type $\mathtt{sym}(\sigma;\tau)$ consists of expressions of type τ that require a symbol of type σ for their execution. Such an expression is introduced by the symbol abstraction construct, symbol abstraction $\mathtt{new}[\sigma](a.e)$. It is eliminated by $\mathtt{gen}(e)$, which generates a "new" symbol at execution time and supplies it to e prior to executing it.

The static semantics of $\mathcal{L}\{\mathtt{symb}\,\mathtt{gen}\}$ is given by the following rules:

$$\frac{\Sigma, a:\sigma\,\Gamma \vdash e:\tau}{\Sigma\,\Gamma \vdash \mathtt{new}[\sigma](a.e) : \mathtt{sym}(\sigma;\tau)} \tag{34.5a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \mathtt{sym}(\sigma;\tau)}{\Sigma\,\Gamma \vdash \mathtt{gen}(e) : \tau} \tag{34.5b}$$

Rule (34.5a) 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 (34.5b) generates a fresh symbol to be used in place of the symbol introduced by e in accordance with Rule (34.5a).

The dynamic semantics of $\mathcal{L}\{\mathtt{symb}\,\mathtt{gen}\}$ is given by the following rules:

$$\frac{}{\mathtt{new}[\sigma](a.e)\ \mathsf{val}} \tag{34.6a}$$

$$\frac{e \longmapsto_\theta e'}{\mathtt{gen}(e) \longmapsto_\theta \mathtt{gen}(e')} \tag{34.6b}$$

$$\frac{a'\ \mathit{fresh}}{\mathtt{gen}(\mathtt{new}[\sigma](a.e)) \longmapsto_\theta [a'/a]e} \tag{34.6c}$$

Rule (34.6c) makes use of an informal convention regarding freshness of symbols. While the intention is intuitively clear (the symbol a' should be chosen so as to not otherwise occur in an evaluation), Rules (34.6) do not fully specify what this means. To make this precise, we define a transition system between 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, $e \mathbin{@} \nu \longmapsto_\theta e' \mathbin{@} \nu'$, 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 state transition are as follows:

$$\frac{a \in \nu}{\mathtt{get}[a] \mathbin{@} \nu \longmapsto_{\theta \otimes \langle a:e\rangle} e \mathbin{@} \nu} \tag{34.7a}$$

$$\frac{e_1 \mathbin{@} \nu \mapsto_\theta e_1' \mathbin{@} \nu'}{\mathtt{set}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_\theta \mathtt{set}\,[a]\,(e_1'; e_2) \mathbin{@} \nu'} \tag{34.7b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : e_1 \rangle} e_2' \mathbin{@} \nu'}{\mathtt{set}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : \_ \rangle} \mathtt{set}\,[a]\,(e_1; e_2') \mathbin{@} \nu'} \tag{34.7c}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{set}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_\theta e_2 \mathbin{@} \nu} \tag{34.7d}$$

$$\frac{e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'}{\mathtt{gen}(e) \mathbin{@} \nu \mapsto_\theta \mathtt{gen}(e') \mathbin{@} \nu'} \tag{34.7e}$$

$$\frac{a' \notin \nu}{\mathtt{gen}(\mathtt{new}\,[\sigma]\,(a.e)) \mathbin{@} \nu \mapsto_\theta [a'/a]e \mathbin{@} \nu \cup \{ a' \}} \tag{34.7f}$$

Rule (34.7f) 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 $e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'$, then $\nu' \supseteq \nu$.

To prove safety we define a state $e \mathbin{@} \nu$ 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 34.1 on page 284, 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 (34.7f) relies on the invariance of typing under renaming of parameters, as described in Chapter 3.

34.3 Subtleties of Fluid Binding

Fluid binding in the context of a first-order language is easy to understand. If the expression $\mathtt{set}\,a\,\mathtt{to}\,e_1\,\mathtt{in}\,e_2$ has a type such as `nat`, then its execution consists of the evaluation of e_2 to a number in the presence of a binding of a to the value of expression e_1. When execution is completed, the binding of a is dropped (reverted to its state in the surrounding context), and the value is returned. Since this value is a number, it cannot contain any reference to a, and so no issue of its binding arises.

But what if the type of $\mathtt{set}\,a\,\mathtt{to}\,e_1\,\mathtt{in}\,e_2$ is a function type, so that the returned value is a λ-abstraction? In that case the body of the λ may contain references to the symbol a whose binding is dropped upon return. This raises an important question about the interaction between fluid binding and higher-order functions. For example, consider the expression

$$\mathtt{set}\,a\,\mathtt{to}\,\overline{17}\,\mathtt{in}\,\lambda(x\mathtt{:nat}.\,x + \mathtt{get}\,a), \tag{34.8}$$

which has type `nat`, given that a is a symbol of the same type. Let us assume, for the sake of discussion, that a is unbound at the point at which this expression is evaluated. Doing so binds a to the number $\overline{17}$, and returns the function $\lambda(x{:}\texttt{nat}.\, x + \texttt{get}\, a)$. This function contains the symbol a, but is returned to a context in which the symbol a is not bound. This means that, for example, application of the expression (34.8) to an argument will incur an error because the symbol a is not bound.

Contrast this with the similar expression

$$\texttt{let}\, y\, \texttt{be}\, \overline{17}\, \texttt{in}\, \lambda(x{:}\texttt{nat}.\, x + y), \tag{34.9}$$

in which we have replaced the fluid-bound symbol, a, by a statically bound variable, y. This expression evaluates to $\lambda(x{:}\texttt{nat}.\, x + \overline{17})$, which adds 17 to its argument when applied. There is never any possibility of an unbound identifier arising at execution time, precisely because the identification of scope and extent ensures that the association between a variable and its binding is never violated.

It is not possible to say that either of these two behaviors is "right" or "wrong," but experience has shown that providing only one or the other of these behaviors is a mistake. Static binding is an important mechanism for encapsulation of behavior in a program; without static binding, one cannot ensure that the meaning of a variable is unchanged by the context in which it is used. Fluid binding is a useful mechanism for avoiding a surfeit of parameters to higher-order functions. For example, letting x stand for the value of expression (34.8), we may obtain instances of it by surrounding it with a binding in the context of an application. For example, the expression

$$\texttt{set}\, a\, \texttt{to}\, \overline{7}\, \texttt{in}\, e(\overline{8})$$

evaluates to $\overline{15}$, and the expression

$$\texttt{set}\, a\, \texttt{to}\, \overline{8}\, \texttt{in}\, e(\overline{8})$$

evaluates to $\overline{16}$. This is not far removed from simply adding a as an addition argument to the function, and applying it to both arguments at each call site. However, it can be convenient in some situations to use fluid binding to avoid such parameters; it is a matter of taste which is more natural in a particular case.

There is a significant drawback, however, to using fluid binding in place of additional function parameters. This is well-illustrated by expression (34.8), which has type $\texttt{nat} \to \texttt{nat}$. The problem is that the type of the function

does not reveal its dependence on a binding for the symbol a. A caller of this function is not informed of the requirement to provide a binding for a in the context of the call. Failure to do so results in a run-time (rather than compile-time) error. If, instead, a were an additional argument to the function, the type would become $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$, which expresses the demand for an additional number on which the ultimate result may depend. Using fluid binding introduces considerable opportunities for error that may otherwise be avoided.

One may argue that it is a deficiency of the type system that it does not record the dependency of functions on the bindings for a set of symbols. For example, one may consider a type $\tau_1 \to_{\mathcal{X}} \tau_2$ of functions whose evaluation may rely on the bindings of symbols in $\mathcal{X}$. To ensure safety, these symbols must be bound at every call site for a function of this type. This necessitates tracking the set of symbols bound within the context of each expression so that we may ensure that the conditions on any call to such a function are met. A type system of this sort is developed in Chapter 40.

34.4 Exercises

1. Complete the formalization of $\mathcal{L}\{\texttt{symb}\,\texttt{gen}\}$ and prove type safety for it.

Chapter 35

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 im-

mediately 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 $\texttt{nat} \times \texttt{(nat ref)}$ 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 $\texttt{nat ref} \times \texttt{nat ref}$ is a pair both of whose components are mutable cells whose contents may change. Contrast this with a value of type $\texttt{(nat} \times \texttt{nat)ref}$, 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 $\texttt{nat ref}$, 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.

35.1 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	::=	$\texttt{loc}[l]$	l
		|	$\texttt{ref}(e)$	$\texttt{ref}(e)$
		|	$\texttt{get}(e)$	$\texttt{\^{}}\,e$
		|	$\texttt{set}(e_1;e_2)$	$e_1\,\texttt{<-}\,e_2$

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 symbols reserved as names for reference cells. The expression $\texttt{ref}(e)$ 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 indexed by 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 $\mathcal{L}\{\texttt{ref}\}$ is specified by the following rules:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash \texttt{loc}[l] : \texttt{ref}(\tau)} \tag{35.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \texttt{ref}(e) : \texttt{ref}(\tau)} \tag{35.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{ref}(\tau)}{\Lambda\,\Gamma \vdash \mathtt{get}(e) : \tau} \tag{35.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \mathtt{ref}(\tau_2) \quad \Lambda\,\Gamma \vdash e_2 : \tau_2}{\Lambda\,\Gamma \vdash \mathtt{set}(e_1; e_2) : \tau_2} \tag{35.1d}$$

The type of the expression $\mathtt{loc}[l]$ 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.

A *memory* is a finite function mapping each of a finite set of locations to closed value (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:

$$\frac{}{\mathtt{loc}[l]\ \mathsf{val}} \tag{35.2a}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{ref}(e') \mathbin{@} \mu'} \tag{35.2b}$$

$$\frac{e\ \mathsf{val}}{\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{loc}[l] \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2c}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\mathtt{get}(e) \mathbin{@} \mu \mapsto \mathtt{get}(e') \mathbin{@} \mu'} \tag{35.2d}$$

$$\frac{e\ \mathsf{val}}{\mathtt{get}(\mathtt{loc}[l]) \mathbin{@} \mu \otimes \langle l : e \rangle \mapsto e \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2e}$$

$$\frac{e_1 \mathbin{@} \mu \mapsto e_1' \mathbin{@} \mu'}{\mathtt{set}(e_1; e_2) \mathbin{@} \mu \mapsto \mathtt{set}(e_1'; e_2) \mathbin{@} \mu'} \tag{35.2f}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mathbin{@} \mu \mapsto e_2' \mathbin{@} \mu'}{\mathtt{set}(e_1; e_2) \mathbin{@} \mu \mapsto \mathtt{set}(e_1; e_2') \mathbin{@} \mu'} \tag{35.2g}$$

$$\frac{e \text{ val}}{\texttt{set(loc[}l\texttt{];}e\texttt{)} \mathbin{@} \mu \otimes \langle l : e' \rangle \mapsto e \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2h}$$

In Rule (35.2c) it is tacitly assumed that l is chosen so as not to occur in the domain of μ, corresponding to the intuition that l is a "new" location in memory.

35.2 Safety

As usual, type safety is the conjunction of preservation and progress for well-formed machine states. Informally, the state $e \mathbin{@} \mu$ 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 15 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 `nat` $\rightarrow$ `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 $e \mathbin{@} \mu$ ok is defined by the following rule:

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu : \Lambda}{e \mathbin{@} \mu \text{ ok}} \tag{35.3}$$

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:

$$\frac{}{\Lambda \vdash \emptyset : \emptyset} \tag{35.4a}$$

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu' : \Lambda'}{\Lambda \vdash \mu' \otimes \langle l : e \rangle : \Lambda', l : \tau} \tag{35.4b}$$

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 35.1 (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 (35.2). 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 (35.2c). We have $\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{loc}[l] \mathbin{@} \mu'$, 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 $\Lambda' \vdash \mathtt{loc}[l] : \mathtt{ref}(\tau)$. 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 35.2 (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 (35.1). 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 (35.1c). We have $\Lambda \vdash \mathtt{get}(e) : \tau$ because $\Lambda \vdash e : \tau$. By induction either e val or there exists μ' and e' such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$. In the latter case we have $\mathtt{get}(e) \mathbin{@} \mu \mapsto \mathtt{get}(e') \mathbin{@} \mu'$ by Rule (35.2d). In the former it follows from an analysis of Rules (35.1) that $e = \mathtt{loc}[l]$ 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 (35.2e) we have $\mathtt{get}(e) \mathbin{@} \mu \mapsto e' \mathbin{@} \mu$.

The remaining cases follow a similar pattern. □

35.3 Exercises

35.4 References and Fluid Binding

Reference cells are closely related to fluidly bound symbols as described in Chapter 34. The crucial differences may be summarized as follows:

1. The binding of a reference cell is *persistent*. A reference cell is never unbound; the cell always has a contents determined when it is allocated and by subsequent assignments. Put in other terms, the bindings of reference cells are *heap-allocated*, since they persist throughout an execution.

2. The fluid binding of a symbol is *ephemeral*. It is in force only for the duration of evaluation of a particular expression, and is reverted after evaluation completes. The bindings of fluid-bound symbols are *stack-allocated*, since they are installed for the duration of evaluation of an expression, and reverted to their previous state afterwards.

To illustrate the differences, let us consider an expression analogous to Example (34.8).

$$\texttt{let}\ y\ \texttt{be}\ \texttt{ref}\,(\overline{17})\ \texttt{in}\ \lambda\,(x\texttt{:nat}.\,x + \hat{}\ y). \tag{35.5}$$

The value of this expression is the function

$$\lambda\,(x\texttt{:nat}.\,x + \hat{}\ a) \tag{35.6}$$

for some location, a, allocated in memory. When applied, it returns the sum of its argument and the contents of this location, which in this example will *always* be $\overline{17}$. If, instead, we bind the variable y to some previously allocated reference cell, then the contents of that cell can be changed by any other piece of code with access to it. However, its contents will *never* be "unbound," as can occur with fluid-bound symbols.

As this example may suggest, reference cells are strictly more expressive than fluid-bound symbols. In particular the mechanisms of fluid binding with dynamic symbol generation may be implemented using reference cells according to the following plan. Each fluid-bound symbol of type τ is represented by a reference cell of type $\tau\ \texttt{opt}\ \texttt{ref}$ that is initialized to `null`.[1] Fetching the binding of a symbol corresponds to retrieving the contents of

[1] See Chapter 17 for a discussion of the type $\tau\ \texttt{opt}$.

that cell, and performing a case analysis on the result. If the cell contains `null`, then execution aborts; if it contains `just(`e`)`, then the result is e. Finally, a fluid let of the symbol a to the value e_1 within e_2 is implemented by the following protocol: (1) bind the current contents of the cell associated to a to a variable, x; (2) assign `just(`e_1`)` to the cell; (3) bind the value of e_2 to a variable, y; (4) re-assign the cell associated to a to the value x; (5) return the value y. We leave it as an exercise for the reader to flesh out the details of this implementation and to prove that it constitutes a definition of fluid binding using reference cells.

The only weakness of the implementation of fluid binding using reference cells is that the bindings of symbols are heap-allocated, whereas, strictly speaking, it is only necessary to heap-allocate the symbol itself, and not its binding. This cannot be expressed using reference cells alone, and hence one may argue that the implementation is not quite faithful to fluid binding, which may therefore be seen as a separate notion. However, for most practical purposes the distinction is minor (there is nothing magic about being stack-allocated in this sense).

Chapter 36

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 `in[`i`](`e_i`)`, 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 can only be determined at execution time, for example when processing input from an external data source. For example, a data stream from a sensor might consist of several different types of data according to the form of a stimulus. To ensure safe processing the items in the stream are labeled with a class that determines the type of the underlying datum. The items are processed by performing a case analysis on the class, and passing the underlying datum to a handler for items of that class.

A difficulty with using sums for this purpose, however, is that the developer must specify in advance the classes of data that are to be considered. That is, sums support *static classification* of data based on a fixed collection of classes. While this works well in the vast majority of cases, there are situations where static classification is inadequate, and *dynamic classification* is required. For example, we may wish to classify data in order to keep it secret from an intermediary in a computation. By creating a fresh class at execution time, two parties engaging in a communication can arrange that they, and only they, are able to compute with a given datum; all others must merely handle it passively without examining its structure or value.

One example of this sort of interaction arises when programming with exceptions, as described in Chapter 28. 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.

One may wonder why dynamic, as opposed to static, classification is appropriate for exception values. To do otherwise—that is, to use static classification—would require a global commitment to the possible forms of exception value that may be used in a program. This creates problems for modularity, since any such global commitment must be made for the whole program, rather than for each of its components separately. Dynamic classification ensures that when any two components are integrated, the classes they introduce are disjoint from one another, avoiding integration problems while permitting separate development.

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 34.2 on page 287 of Chapter 34). 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 24), are sufficient to implement dynamic classification.

36.1 Dynamic Classification

The language $\mathcal{L}\{\texttt{classified}\}$ uses dynamically generated symbols as class identifiers.[1] We rely on the implicit identification of α-equivalent expressions to ensure that a dynamically generated symbol is distinct from any of

[1] Dynamic symbol generation was introduced in Chapter 34. However, the use of symbols for dynamic classification does not imply that there is any form of fluid binding involved!

the finitely many symbols that have been previously generated.

The syntax of $\mathcal{L}\{\texttt{classified}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`clsfd`	`clsfd`
Expr	e	::=	`in[`a`](`e`)`	`in[`a`](`e`)`
		|	`ccase(`e`;`e_0`;`r_1`,...,`r_n`)`	`ccase` e `{`r_1 `|` ... `|` r_n`}` `ow` e_0
Rule	r	::=	`in?[`a`](`x`.`e`)`	`in[`a`](`x`)` $\Rightarrow$ e

The expression `in[`a`](`e`)` classifies the value of the expression e by labelling it with the symbol a. The expression `ccase` e `{`r_1 `|` ... `|` r_n`}` `ow` e_0 analyzes the class of e according to the rule $r_1, \ldots, r_n$. Each rule has the form `in[`a_i`](`x_i`)` $\Rightarrow$ 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. A default case is required, since no static type system can, in general, circumscribe the set of possible classes of a classified value, and hence pattern matches on classified values cannot be guaranteed to be exhaustive.

The static semantics of $\mathcal{L}\{\texttt{classified}\}$ consists of a judgement of the form $\Sigma\,\Gamma \vdash e : \tau$, where Σ specifies the types of the symbols (as described in Chapter 34), 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 `in[`a`](`e`)` and yields a value of type τ. These judgements are inductively defined by the following rules:

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{in[}a\texttt{](}e\texttt{)} : \texttt{clsfd}} \tag{36.1a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \texttt{clsfd} \quad \Sigma\,\Gamma \vdash r_1 : \tau \quad \ldots \quad \Sigma\,\Gamma \vdash r_n : \tau}{\Sigma\,\Gamma \vdash \texttt{ccase(}e\texttt{;}e_0\texttt{;}r_1\texttt{,...,}r_n\texttt{)} : \tau} \tag{36.1b}$$

$$\frac{\Sigma \vdash a : \sigma \quad \Sigma\,\Gamma, x : \sigma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{in?[}a\texttt{](}x\texttt{.}e\texttt{)} : \tau} \tag{36.1c}$$

The dynamic semantics of these operations is an entirely straightforward extension of the semantics of dynamic symbol generation given in Section 34.2 on page 287.

$$\frac{e\ \mathsf{val}}{\texttt{in[}a\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{36.2a}$$

$$\frac{e\ \texttt{@}\ \nu \mapsto e_0\ \texttt{@}\ \nu'}{\texttt{in[}a\texttt{](}e\texttt{)}\ \texttt{@}\ \nu \mapsto \texttt{in[}a\texttt{](}e_0\texttt{)}\ \texttt{@}\ \nu'} \tag{36.2b}$$

$$\frac{e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'}{\texttt{ccase}(e; e_0; r_1, \ldots, r_n) \mathbin{@} \nu \mapsto \texttt{ccase}(e'; e_0; r_1, \ldots, r_n) \mathbin{@} \nu'} \tag{36.2c}$$

$$\frac{\texttt{in}[a]\,(e)\ \mathsf{val}}{\texttt{ccase}(\texttt{in}[a]\,(e); e_0; \epsilon) \mathbin{@} \nu \mapsto e_0 \mathbin{@} \nu} \tag{36.2d}$$

$$\frac{\texttt{in}[a_1]\,(e_1)\ \mathsf{val}}{\begin{array}{c}\texttt{ccase}(\texttt{in}[a_1]\,(e_1); e_0; \texttt{in?}[a_1]\,(x_1.e_1'), \ldots, \texttt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu \\ \mapsto \\ [e_1/x_1]e_1' \mathbin{@} \nu\end{array}} \tag{36.2e}$$

$$\frac{\texttt{in}[a]\,(e)\ \mathsf{val} \quad a \neq a_1 \quad n > 0}{\begin{array}{c}\texttt{ccase}(\texttt{in}[a_1]\,(e_1); e_0; \texttt{in?}[a_1]\,(x_1.e_1'), \ldots, \texttt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu \\ \mapsto \\ \texttt{ccase}(\texttt{in}[a]\,(e); e_0; \texttt{in?}[a_2]\,(x_2.e_2'), \ldots, \texttt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu\end{array}} \tag{36.2f}$$

Rule (36.2d) specifies that the default case is evaluated when all rules have been exhausted (that is, the sequence of rules is empty). Rules (36.2e) and (36.2f) 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 17, 18, and 34.

Theorem 36.1 (Preservation)**.** *Suppose that $e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'$, 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 $a' \in \nu'$.*

Lemma 36.2 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e : \texttt{clsfd}$ and e val. Then $e = \texttt{in}[a]\,(e)$ for some a such that $\Sigma \vdash a : \tau$ and some e such that e val and $\Sigma \vdash e : \tau$.*

Theorem 36.3 (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 $e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'$ for some ν' and e'.*

36.2 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 $\tau \to \texttt{clsfd}$ and $\texttt{clsfd} \to \tau$ 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 34 together with the primitive operations for the type τ `class`. The syntax of $\mathcal{L}\{\texttt{class}\}$ is specified by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`class(`τ`)`	τ `class`
Expr	e	::=	`cls[`a`]`	`cls[`a`]`
		$\mid$	`ccase[`$t.\sigma$`](`e`;`e_0`;`r_1`,...,`r_n`)`	`ccase` e $\{r_1 \mid \ldots \mid r_n\}$ `ow` e_0
Rule	r	$\mid$	`cls?[`a`](`e`)`	`cls[`a`]` $\Rightarrow$ e

The type τ `class` represents the type of classes with associated values of type τ. A value of this type has the form `cls[`a`]`, where a is a symbol labelling a class of values. The expression `ccase` e $\{r_1 \mid \ldots \mid r_n\}$ `ow` e_0 is analogous to the class case construct of $\mathcal{L}\{\texttt{classified}\}$, 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 $\mathcal{L}\{\texttt{class}\}$.

The static semantics of $\mathcal{L}\{\texttt{class}\}$ must take care to propagate type identity information gained during pattern matching. For suppose that e is an expression of type `class(`τ`)`, and that we analyze its class using a series of rules of the form `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 `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 $\mathcal{L}\{\texttt{class}\}$ consists of expression typing judgements of the form $\Sigma\,\Gamma \vdash e : \tau$, and rule typing judgements of the form $\Sigma\,\Gamma \vdash r :$ `class(`τ`)` $> \tau'$. These judgements are inductively defined by the

following rules:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \texttt{cls}[a] : \texttt{class}(\tau)} \tag{36.3a}$$

$$\frac{\begin{array}{c}\Sigma\,\Gamma \vdash e : \texttt{class}(\tau) \qquad \Sigma\,\Gamma \vdash e_0 : [\tau/t]\sigma \\ \Sigma\,\Gamma \vdash r_1 : \texttt{class}(\tau_1) > [\tau_1/t]\sigma \quad \ldots \quad \Sigma\,\Gamma \vdash r_n : \texttt{class}(\tau_n) > [\tau_n/t]\sigma\end{array}}{\Sigma\,\Gamma \vdash \texttt{ccase}[t.\sigma]\,(e; e_0; r_1, \ldots, r_n) : [\tau/t]\sigma} \tag{36.3b}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau'}{\Sigma\,\Gamma \vdash \texttt{cls?}[a]\,(e) : \texttt{class}(\tau) > \tau'} \tag{36.3c}$$

Rule (36.3a) specifies that the class $\texttt{cls}[a]$ has type $\texttt{class}(\tau)$, where τ is the type associated to the class a by Σ. Rule (36.3b) specifies the type of a case analysis on a class of type $\texttt{class}(\tau)$ 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 $e \,@\, \nu$, 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 $e \,@\, \nu \mapsto e' \,@\, \nu'$ are easily derived from Rules (36.2), and are omitted here for the sake of brevity.

Theorem 36.4 (Preservation)**.** *Suppose that $e \,@\, \nu \mapsto e' \,@\, \nu'$, 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 $e = \texttt{cls}[a]$ and $a = a_i$ for some rule $\texttt{cls}[a_i] \Rightarrow e_i$. 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 36.5 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e : \tau$ `class` and e val. Then $e =$ `cls[a]` for some a such that $\Sigma \vdash a : \tau$.*

Proof. By rule induction on Rules (36.3), taking account of the definition of values. □

Theorem 36.6 (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 $e \,@\, \nu \mapsto e' \,@\, \nu'$ for some ν' and e'.*

Proof. By rule induction on Rules (36.3). 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 36.5 that $e = \texttt{cls}[a]$ for some a. Either $a = a_i$ for some rule $\texttt{cls}[a_i] \Rightarrow e_i$ in $r_1, \ldots, r_n$, in which case we progress to e_i, or else we progress to e_0. □

36.3 From Classes to Classification

Dynamic classification is definable in a language with dynamic class types, existential types, and product types. Specifically, the type `clsfd` may be considered to stand for the existential type

$$\exists(t.t\,\texttt{class} \times t).$$

According to this identification, the classified value $\texttt{in}[a]\,(e)$ is 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. Case analysis is performed by opening the package and dispatching on its encapsulated class component. To be specific, suppose that the class case expression $\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e'$ has type ρ, where r_i is the rule $\texttt{in}[a_i]\,(x_i) \Rightarrow e_i$, with $x_i : \tau_i \vdash e_i : \rho$. This expression is defined to be

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle x, y\rangle : t\,\texttt{class} \times t\,\texttt{in}\,(e_{body}(y)),$$

where e_{body} is an expression to be defined shortly. Case analysis proceeds by opening the package, e, representing the classified value, and decomposing it into a class, x, and an underlying value, y. The body of the `open` analyzes the class x, yielding a function of type $t \to \rho$, 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 core of the case analysis, namely the expression e_{body}, analyzes the encapsulated class, x, of the package. The case analysis is parameterized by the type abstractor $u.\,u \to \rho$, where u is not free in ρ. The overall type of the case is $[t/u]u \to \rho = t \to \rho$, which ensures that the application to y to the classified value is well-typed. Each branch of the case analysis has type $\tau_i \to \rho$, as required by Rule (36.3b). In sum, the expression e_{body} is defined to be the expression

$$\texttt{ccase}\,x\,\{r'_1 \mid \ldots \mid r'_n\}\,\texttt{ow}\,\lambda(\_ : t.\,e_0),$$

where for each $1 \leq i \leq n$, the rule r'_i is defined to be

$$\texttt{cls}[a_i] \Rightarrow \lambda(x_i : \tau_i.\,e_i).$$

One may check that the static and dynamic semantics of $\mathcal{L}\{\texttt{classified}\}$ are derivable according to these definitions.

36.4 Exercises

1. Derive the Standard ML exception mechanism from the machinery developed here.

Part XIII

Modalities

Chapter 37

Computational Effects

In this chapter we begin the study of *computational effects*. So far we have concentrated on languages whose dynamics consists of rules to determine the value of an expression. In this chapter we begin the study of languages for which the dynamics consists not only of evaluation, but also may give rise to *effects*. Rougly speaking, we classify as an effect any consequence of the execution of a program other than the determination of its value. For example, one may consider run-time errors to be forms of effect, as would be input or output operations (with which a program may interact with a person or another program), or in-place modifications to a data structure such as are found in most familiar programming languages.

While it is difficult to be precise about what constitutes an effect, a rough-and-ready rule is *any behavior that constrains the order of execution beyond the flow of data within a program*. A *data dependency* in a program arises when the value of one expression depends on the value of another. Perhaps the most basic form of data dependency arises from by-value binding. In the expression $\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$, the value of e_2 depends on the value of e_1 via the variable x. Consequently, we cannot, in general, evaluate e_2 without first evaluating e_1. A *control dependency* in a program arises from restrictions on the order of evaluation above and beyond those required by data dependencies. For example, the call-by-value evaluation strategy for function applications specifies that the argument must be evaluated before the call, even if the body of the function does not rely on the argument. As we have seen, it is possible to consider instead a call-by-name evaluation strategy in which this control dependency does not arise, giving rise to a less constrained evaluation order. And in Chapter 44 we will explore further variations on the control flow in a program.

The characteristic feature of a computational effect is that it imposes control dependencies above and beyond the natural data dependencies in a program. For example, in a language with screen output, it is clearly necessary to specify the order in which output statements are executed. For example, consider the expression

$$\langle \texttt{print "hello"}, \texttt{print "goodbye"} \rangle.$$

The outcome of this computation is obviously dependent on whether we evaluate the components of a pair from left-to-right, from right-to-left, or not at all. If we wish to program with effects, then we must specify control dependencies so as to make the effect of execution predictable. Lacking this, it would be impossible to program with an effect such as output.

There are two major methods of managing effects in a language. One is to constrain the evaluation order so tightly that adding effects presents no difficulties such as the one just outlined. The downside of this approach is that the imposed control dependencies can be burdensome in the case that the program does not employ effects. If, as we shall say, both e_1 and e_2 are pure (without effects), then there is no reason to insist that the pair $\langle e_1, e_2 \rangle$ be evaluated left-to-right, right-to-left, or simultaneously. We pay for the presence of effects, even when we do not use them.

A natural question is whether we can have the best of both worlds, imposing control dependencies only where needed, and otherwise not insisting on them where they are not forced. This may be achieved by introducing a distinction, called a *modality*, between two *modes* of expression:

1. The *pure* expressions that are executed solely for their value, and that may engender no effects.

2. The *impure* expressions that are executed for their value and their effect.

Impure expressions are called *commands* because their evaluation involves execution of *imperative* operations, such as output to the screen or input from a keyboard, that must be performed at the point at which they are executed. This mode distinction gives rise to a new form of type, called the *lax modality*, whose values are *encapsulated* commands. That is, we may bundle up an un-executed command as a form of value that will be executed later when the bundle is unwrapped.

Returning to the question of control dependencies, commands are structured so as to impose a fixed order of evaluation so that all effects are performed in a predictable manner. Expressions, on the other hand, are liberated from this requirement, allowing much greater flexibility in choosing

the order of evaluation of the constituents of an expression. The syntax of commands is chosen to enforce a sequential order of execution. The most basic form of command is the *return* command, which simply returns a specified value without engendering any effects. Commands are combined using the *bind* construct, which sequentializes execution of one command before another. These two forms of command give rise to a concrete syntax that is familiar from conventional programming languages.

37.1 A Modality for Effects

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	::=	$\texttt{return}(e)$	$\texttt{return}(e)$
		$\mid$	$\texttt{letcomp}(e; x.m)$	$\texttt{let comp}(x)\ \texttt{be}\ e\ \texttt{in}\ m$

The language $\mathcal{L}\{\texttt{comm}\}$ distinguishes distinguishes two modes of expression, the pure (effect-free) *expressions*, and the impure (effect-capable) *commands*. The modal 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 modal 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:

$$\frac{\Gamma \vdash m \sim \tau}{\Gamma \vdash \texttt{comp}(m) : \texttt{comp}(\tau)} \tag{37.1a}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) \sim \tau} \tag{37.1b}$$

$$\frac{\Gamma \vdash e : \texttt{comp}(\tau) \quad \Gamma, x : \tau \vdash m \sim \tau'}{\Gamma \vdash \texttt{letcomp}(e; x.m) \sim \tau'} \tag{37.1c}$$

The dynamic semantics of an instance of $\mathcal{L}\{\texttt{comm}\}$ is specified by two transition judgements:

1. *Evaluation* of expressions, $e \mapsto e'$.
2. *Execution* of commands, $m \mapsto m'$.

The rules of expression evaluation are carried over directly from the definition of evaluation for each of the types involved in the expression. The rules for command execution, however, are specified here to adhere to the sequential ordering imposed by the modal type.

$$\frac{e \mapsto e'}{\texttt{return}(e) \mapsto \texttt{return}(e')} \tag{37.2a}$$

$$\frac{e \ \mathsf{val}}{\texttt{return}(e) \ \mathsf{final}} \tag{37.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e; x.m) \mapsto \texttt{letcomp}(e'; x.m)} \tag{37.2c}$$

$$\frac{m_1 \mapsto m_1'}{\texttt{letcomp}(\texttt{comp}(m_1); x.m_2) \mapsto \texttt{letcomp}(\texttt{comp}(m_1'); x.m_2)} \tag{37.2d}$$

$$\frac{\texttt{return}(e) \ \mathsf{final}}{\texttt{letcomp}(\texttt{comp}(\texttt{return}(e)); x.m) \mapsto [e/x]m} \tag{37.2e}$$

Rules (37.2a) and (37.2c) specify that the expression part of a `return` or `let` command is to be evaluated before execution can proceed. Rule (37.2b) specifies that a `return` command whose argument is a value is a final state of command execution. Rule (37.2d) specifies that a `let` activates an encapsulated command, and Rule (37.2e) specifies that a completed command passes its return value to the body of the `let`.

Particular effects are introduced by adding new forms of command, along with their static and dynamic semantics. 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. From that point of view, Rules (37.2) are to be regarded as silent transitions that have no influence on the context. Their role is simply to ensure that commands are executed in a specified sequential order.

37.2 Imperative Programming

The bind construct imposes a sequential evaluation order on commmands, according to which the encapsulated command is executed prior to execution of the body of the bind. This gives rise to a familiar programming idiom, called *sequential composition*, which we now derive from the lax modality.

Since there are only two constructs for forming commands, the bind and the return command, it is easy to see that a command of type τ always has the form

$$\texttt{let comp}(x_1)\ \texttt{be}\ e_1\ \texttt{in} \ldots \texttt{let comp}(x_n)\ \texttt{be}\ e_n\ \texttt{in return}(e),$$

where $e_1 : \tau_1\ \texttt{comp}, \ldots, e_n : \tau_n\ \texttt{comp}$, and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$. The dynamic semantics of $\mathcal{L}\{\texttt{comm}\}$ specifies that this is evaluated by evaluating the expression, e_1, to an encapsulated command, m_1, then executing m_1 for its value and effects, then passing this value to e_2, and so forth, until finally the value determined by the expression e is returned.

To execute m_1 and m_2 in sequence, where m_2 may refer to the value of m_1 via a variable x_1, we may write

$$\texttt{let comp}(x_1)\ \texttt{be comp}(m_1)\ \texttt{in}\ m_2.$$

This encapsulates, and then immediate activates, the command m_1, binding it value to x_1, and continuing by executing m_2. More generally, to execute a sequence of commands in order, passing the value of each to the next, we may write

$$\texttt{let comp}(x_1)\ \texttt{be comp}(m_1)\ \texttt{in} \ldots \texttt{let comp}(x_{k-1})\ \texttt{be comp}(m_{k-1})\ \texttt{in}\ m_k.$$

Notationally, this quickly gets out of hand. We therefore 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

$$\texttt{let comp}(x)\ \texttt{be comp}(m_1)\ \texttt{in}\ m_2,$$

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:

$$\texttt{do}\,\{x_1 \leftarrow m_1\,\texttt{;}\ldots\texttt{do}\,\{x_k \leftarrow m_k\,\texttt{;}\,\texttt{return}(e)\}\ldots\}.$$

This notation is remiscent of that used in many well-known programming languages. The point here is that sequential composition of commands arises from the presence of the lax modality in the language. In other words conventional imperative programming languages are implicitly structured by this type, even if the connection is not made explicit.

37.3 Integrating Effects

The modal separation of expressions from commands ensures that the semantics of expression evaluation is not compromised by the possibility of effects. One consequence of this restriction is that it is impossible to define an expression $x : \tau\,\texttt{comp} \vdash \texttt{run}\,x : \tau$ whose behavior is to unbundle the command bound to x, execute it, and return its value as the value of the entire expression. For if such an expression were to exist, expression evaluation would engender effects, ruining the very distinction we are trying to preserve

The only way for a command to occur inside of an expression is for it to be encapsulated as a value of modal type. To execute such a command it is necessary to bind it to a variable using the bind construct, which is itself a form of command. This is the essential means by which effects are confined to commands, and by which expressions are ensured to remain pure. 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, a *command*, namely

$$\texttt{let comp}(x)\,\texttt{be}\,e\,\texttt{in return}(x),$$

which executes the encapsulated command and returns its value.

Now consider the extension of $\mathcal{L}\{\texttt{comm}\}$ with function types. Recall from Chapter 13 that a function has the form $\lambda(x\texttt{:}\tau.\,e)$, where e is a (pure) expression. In the context of $\mathcal{L}\{\texttt{comm}\}$ this implies that no function may engender an effect when applied! For example, it is not possible to write a function of the form $\lambda(x\texttt{:unit.print "hello"})$ that, when applied, outputs the string `hello` to the screen!

This may seem like a serious limitation, but this apparent "bug" is actually an important "feature"! To see why, observe that the type of the

foregoing function would, in the absence of the lax modality, be something like `unit` $\rightarrow$ `unit`. Intuitvely, a function of this type is either the identity function, the constant function returning the null tuple (this is, in fact, the identity function), or a function that diverges or incurs an error when applied (in the presence of such possibilities). But, above all, it *cannot* be the function that prints `hello`.

However, let us consider the closely related type `unit` $\rightarrow$ `(unit comp)`. This is the type of functions that, when applied, yield an *encapsulated command*, of type `unit`. One such function is

$$\lambda(x\texttt{:unit.\,comp(print "hello")}).$$

This function *does not* output to the screen when applied, since no pure function can have an effect, but it *does* yield a command that, when executed, performs this output. Thus, if e is the above function, then the command

$$\texttt{let comp(\_) be } e(\langle\rangle) \texttt{ in return}(\langle\rangle) \tag{37.3}$$

executes the encapsulated command yielded by e when applied, engendering the intended effect, and returning the trivial element of unit type.

The importance of this example lies in the distinction between the type `unit` $\rightarrow$ `unit`, which can only contain uninteresting functions such as the identity, and the type `unit` $\rightarrow$ `(unit comp)`, which reveals in its type that the result of applying it is an encapsulated command that may, when executed, engender an arbitrary effect. In short, the type reveals the reliance on effects. The function type retains its meaning, and, in combination with the lax modality, provides a type of *procedures* that yield a command when applied. A *procedure call* is implemented by combining function application with the modal bind operation in the manner illustrated by expression (37.3).

37.4 Exercises

Chapter 38

Monadic Exceptions

As we saw in Chapter 37, 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.

38.1 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 com-

mand. 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 $\mathcal{L}\{\texttt{comm exc}\}$ is a modification of $\mathcal{L}\{\texttt{comm}\}$ to account for this additional outcome of execution. The following grammar specifies the characteristic features of $\mathcal{L}\{\texttt{comm exc}\}$:

Category	*Item*		*Abstract*	*Concrete*
Comm	m	$::=$	$\texttt{raise[}\tau\texttt{](}e\texttt{)}$	$\texttt{raise(}e\texttt{)}$
		$\mid$	$\texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)}$	$\texttt{let comp(}x\texttt{) be }e\texttt{ in }m_1\texttt{ ow(}y\texttt{) in }m_2$

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

$$\texttt{let comp(}x\texttt{) be }e\texttt{ in }m_1\texttt{ ow(}y\texttt{) in }m_2$$

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

$$\texttt{let comp(}x\texttt{) be }e\texttt{ in }m\texttt{ ow(}y\texttt{) in raise(}y\texttt{),}$$

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:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} \sim \tau} \tag{38.1a}$$

$$\frac{\Gamma \vdash e : \texttt{comp(}\tau\texttt{)} \quad \Gamma, x : \tau \vdash m_1 \sim \tau' \quad \Gamma, y : \tau_{exn} \vdash m_2 \sim \tau'}{\Gamma \vdash \texttt{letcomp(}e\texttt{;}x\texttt{.}m_1\texttt{;}y\texttt{.}m_2\texttt{)} \sim \tau'} \tag{38.1b}$$

The dynamic semantics of these commands consists of a transition system of the form $m \mapsto m'$ defined by the following rules:

$$\frac{e \mapsto e'}{\texttt{return(}e\texttt{)} \mapsto \texttt{return(}e'\texttt{)}} \tag{38.2a}$$

$$\frac{e \mapsto e'}{\texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{raise[}\tau\texttt{](}e'\texttt{)}} \tag{38.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e; x.m_1; y.m_2) \mapsto \texttt{letcomp}(e'; x.m_1; y.m_2)} \tag{38.2c}$$

$$\frac{m \mapsto m'}{\texttt{letcomp}(\texttt{comp}(m); x.m_1; y.m_2) \mapsto \texttt{letcomp}(\texttt{comp}(m'); x.m_1; y.m_2)} \tag{38.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp}(\texttt{comp}(\texttt{return}(e)); x.m_1; y.m_2) \mapsto [e/x]m_1} \tag{38.2e}$$

$$\frac{e\ \mathsf{val}}{\texttt{letcomp}(\texttt{comp}(\texttt{raise}[\tau](e)); x.m_1; y.m_2) \mapsto [e/y]m_2} \tag{38.2f}$$

38.2 Programming With Monadic Exceptions

The chief virtue of monadic exceptions is also its chief vice. A value of type `nat` $\rightarrow$ `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 `nat` $\rightarrow$ `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

$$\lambda(x{:}\texttt{nat}.\,\texttt{do}\,\{y \leftarrow \texttt{run}\,g(x)\ ;\ z \leftarrow \texttt{run}\,f(y)\,;\texttt{return}(z)\}).$$

Here we have used the `do` syntax introduced in Chapter 37, 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

$$\texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{nat}\,\texttt{comp}$$

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.

38.3 Exercises

Chapter 39

Monadic State

In Chapter 35 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` $\rightarrow$ `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` $\rightarrow$ `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.

39.1 Storage Effects

The language $\mathcal{L}\{\texttt{comm ref}\}$ is an extension of $\mathcal{L}\{\texttt{comm}\}$ (described in Chapter 37) with mutable references into $\mathcal{L}\{\texttt{comm}\}$. The syntax of $\mathcal{L}\{\texttt{comm ref}\}$

extends that of $\mathcal{L}\{\texttt{comm}\}$ with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{ref}(\tau)$	$\tau\,\texttt{ref}$
Expr	e	::=	l	l
Comm	m	::=	$\texttt{ref}(e)$	$\texttt{ref}(e)$
		|	$\texttt{get}(e)$	$\texttt{\^{}}\,e$
		|	$\texttt{set}(e_1;e_2)$	$e_1\,\texttt{<-}\,e_2$

Locations are pure expressions, whereas the primitives for reference cells are forms of command.

The static semantics of $\mathcal{L}\{\texttt{comm}\,\texttt{ref}\}$ extends that of $\mathcal{L}\{\texttt{comm}\}$ with the following rules:

$$\frac{}{\Lambda, l:\tau\,\Gamma \vdash l : \texttt{ref}(\tau)} \tag{39.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \texttt{ref}(e) \sim \texttt{ref}(\tau)} \tag{39.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \texttt{ref}(\tau)}{\Lambda\,\Gamma \vdash \texttt{get}(e) \sim \tau} \tag{39.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \texttt{ref}(\tau) \quad \Lambda\,\Gamma \vdash e_2 : \tau}{\Lambda\,\Gamma \vdash \texttt{set}(e_1;e_2) \sim \tau} \tag{39.1d}$$

Here we make explicit the location typing assumptions, Λ, as well as the variable typing assumptions, Γ.

The dynamic semantics of $\mathcal{L}\{\texttt{comm}\,\texttt{ref}\}$ 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.

$$\frac{}{\texttt{comp}(m)\ \mathsf{val}} \tag{39.2a}$$

$$\frac{e \mapsto e'}{\texttt{return}(e) \mathbin{@} \mu \mapsto \texttt{return}(e') \mathbin{@} \mu} \tag{39.2b}$$

$$\frac{e \mapsto e'}{\texttt{letcomp}(e; x\,.\,m) \mathbin{@} \mu \mapsto \texttt{letcomp}(e'; x\,.\,m) \mathbin{@} \mu} \tag{39.2c}$$

$$\frac{m_1 \mathbin{@} \mu \mapsto m_1' \mathbin{@} \mu'}{\mathtt{letcomp(comp}(m_1)\mathtt{;}x\mathtt{.}m_2\mathtt{)} \mathbin{@} \mu \mapsto \mathtt{letcomp(comp}(m_1')\mathtt{;}x\mathtt{.}m_2\mathtt{)} \mathbin{@} \mu'} \tag{39.2d}$$

$$\frac{e \text{ val}}{\mathtt{letcomp(comp(return}(e)\mathtt{);}x\mathtt{.}m\mathtt{)} \mathbin{@} \mu \mapsto [e/x]m \mathbin{@} \mu} \tag{39.2e}$$

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:

$$\frac{}{l \text{ val}} \tag{39.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{ref}(e') \mathbin{@} \mu} \tag{39.3b}$$

$$\frac{e \text{ val}}{\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{return}(l) \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3c}$$

$$\frac{e \mapsto e'}{\mathtt{get}(e) \mathbin{@} \mu \mapsto \mathtt{get}(e') \mathbin{@} \mu} \tag{39.3d}$$

$$\frac{e \text{ val}}{\mathtt{get}(l) \mathbin{@} \mu \otimes \langle l : e \rangle \mapsto \mathtt{return}(e) \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3e}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{set}(e_1\mathtt{;}e_2) \mathbin{@} \mu \mapsto \mathtt{set}(e_1'\mathtt{;}e_2) \mathbin{@} \mu} \tag{39.3f}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\mathtt{set}(e_1\mathtt{;}e_2) \mathbin{@} \mu \mapsto \mathtt{set}(e_1\mathtt{;}e_2') \mathbin{@} \mu} \tag{39.3g}$$

$$\frac{e \text{ val}}{\mathtt{set}(l\mathtt{;}e) \mathbin{@} \mu \otimes \langle l : e' \rangle \mapsto \mathtt{return}(e) \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{39.3h}$$

Type safety for $\mathcal{L}\{\mathtt{comm}\,\mathtt{ref}\}$ is stated and proved much as it is for $\mathcal{L}\{\mathtt{ref}\}$.

39.2 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 35.

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 τ `comp` $\to \tau$, 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!

39.3 Exercises

Chapter 40

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 (that is, 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 46 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 38 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 34). 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.

40.1 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	::=	`box[`χ`](`τ`)`	$\Box_\chi\,\tau$
Const	χ	::=	`tt`	$\top$
		|	`and(`χ_1`;`χ_2`)`	$\chi_1 \wedge \chi_2$
Expr	e	::=	`box(`e`)`	`box(`e`)`
		|	`unbox(`e`)`	`unbox(`e`)`

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 `box(`e`)` is the introduction form for the comonad, and the expression `unbox(`e`)` is the corresponding elimination form.

The judgement χ `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.

$$\frac{}{\mathtt{tt}\ \mathsf{true}} \tag{40.1a}$$

$$\frac{\chi_1\ \mathsf{true} \quad \chi_2\ \mathsf{true}}{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}} \tag{40.1b}$$

$$\frac{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_1\ \mathsf{true}} \tag{40.1c}$$

$$\frac{\mathtt{and}(\chi_1;\chi_2)\ \mathsf{true}}{\chi_2\ \mathsf{true}} \tag{40.1d}$$

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 generic 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 $\mathcal{L}\{\mathtt{comon}\}$ is defined by the following rules:

$$\frac{\chi' \vdash \chi}{\Gamma, x : \tau\,[\chi] \vdash x : \tau\,[\chi']} \tag{40.2a}$$

$$\frac{\Gamma \vdash e : \tau\,[\chi]}{\Gamma \vdash \mathtt{box}(e) : \Box_\chi\,\tau\,[\chi']} \tag{40.2b}$$

$$\frac{\Gamma \vdash e : \Box_\chi\,\tau\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \mathtt{unbox}(e) : \tau\,[\chi']} \tag{40.2c}$$

Rule (40.2b) 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 (40.2c) 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 (40.2) are formulated to ensure that the constraint on a typing judgement may be strengthened arbitrarily.

Lemma 40.1 (Constraint Strengthening)**.** *If* $\Gamma \vdash e : \tau\,[\chi]$ *and* $\chi' \vdash \chi$*, then* $\Gamma \vdash e : \tau\,[\chi']$.

Proof. By rule induction on Rules (40.2). □

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.

$$\frac{}{\texttt{box}(e)\ \texttt{val}} \tag{40.3a}$$

$$\frac{e \mapsto e'}{\texttt{unbox}(e) \mapsto \texttt{unbox}(e')} \tag{40.3b}$$

$$\frac{}{\texttt{unbox}(\texttt{box}(e)) \mapsto e} \tag{40.3c}$$

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 $\mathcal{L}\{\texttt{comon}\}$:

$$\frac{\Gamma, x : \tau_1\,[\texttt{tt}] \vdash e_2 : \tau_2\,[\texttt{tt}]}{\Gamma \vdash \texttt{lam}[\tau_1](x.e_2) : \texttt{arr}(\tau_1;\tau_2)\,[\chi]} \tag{40.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau\,[\chi] \quad \Gamma \vdash e_2 : \tau_2\,[\chi]}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau\,[\chi]} \tag{40.4b}$$

These rules are formulated so as to ensure that constraint strengthening remains admissible. Rule (40.4a) states that a λ-abstraction has type $\tau_1 \to \tau_2$ 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 $\texttt{unbox\_app}(e_1;e_2)$ be an abbreviation for $\texttt{unbox}(\texttt{ap}(e_1;e_2))$, which applies e_1 to e_2, then activates the result. The derived static semantics for this construct is given by the following rule:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \Box_\chi\,\tau\,[\chi'] \quad \Gamma \vdash e_2 : \tau_2\,[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \texttt{unbox\_app}(e_1;e_2) : \tau\,[\chi']} \tag{40.5}$$

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 $\tau_1 \to \Box_\chi \tau_2$ 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.

40.2 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.

40.2.1 Exceptions

To model exceptions we extend $\mathcal{L}\{\texttt{comon}\}$ as follows:

Category	*Item*		*Abstract*	*Concrete*
Const	χ	::=	$\uparrow$	$\uparrow$
Expr	e	::=	$\texttt{raise[}\tau\texttt{]}(e)$	$\texttt{raise}(e)$
		|	$\texttt{handle}(e_1; x.e_2)$	$\texttt{try}\, e_1\, \texttt{ow}\, x \Rightarrow e_2$

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:

$$\frac{\Gamma \vdash e : \tau_{exn}\,[\chi] \quad \chi \vdash \uparrow}{\Gamma \vdash \texttt{raise[}\tau\texttt{]}(e) : \tau\,[\chi]} \tag{40.6a}$$

$$\frac{\Gamma \vdash e_1 : \tau\,[\chi \wedge \uparrow] \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau\,[\chi]}{\Gamma \vdash \texttt{handle}(e_1; x.e_2) : \tau\,[\chi]} \tag{40.6b}$$

Rule (40.6a) 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 (40.6b) 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 28. The interesting question is to explore the additional assurances given by the comonadic type system given by Rules (40.6). 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:

$$\frac{}{\epsilon : \tau\,[\top]} \tag{40.7a}$$

$$\frac{k : \tau'\,[\chi'] \quad f : \tau\,[\chi] \Rightarrow \tau'\,[\chi']}{f\,;k : \tau\,[\chi]} \tag{40.7b}$$

Rule (40.7a) 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 (40.7b) 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,

$$\frac{x : \tau_{exn} \vdash e : \tau\,[\chi]}{\texttt{handle(}-\texttt{;}x\texttt{.}e\texttt{)} : \tau\,[\chi \wedge \uparrow] \Rightarrow \tau\,[\chi]} \tag{40.8}$$

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 27.

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi]}{k \triangleright e\ \mathsf{ok}} \tag{40.9a}$$

$$\frac{k : \tau\,[\chi] \quad e : \tau\,[\chi] \quad e\ \mathsf{val}}{k \triangleleft e\ \mathsf{ok}} \tag{40.9b}$$

Observe that a state of the form $\epsilon \triangleright \texttt{raise}(e)$, 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.

$$\frac{e \ \mathsf{val}}{\epsilon \lhd e \ \mathsf{final}} \tag{40.10}$$

In contrast to Chapter 28, we *do not* consider an uncaught exception state to be final!

Theorem 40.2 (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. □

40.2.2 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 $\mathcal{L}\{\texttt{symb}\}$ defined in Chapter 34. For this purpose we consider atomic constraints of the form $\texttt{bd}(a)$, 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.

$$\frac{\Sigma \vdash a : \tau \quad \chi \vdash \texttt{bd}(a)}{\Sigma\,\Gamma \vdash \texttt{get}[a] : \tau\,[\chi]} \tag{40.11a}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e_1 : \tau\,[\chi] \quad \Sigma\,\Gamma \vdash e_2 : \tau\,[\chi \wedge \texttt{bd}(a)]}{\Sigma\,\Gamma \vdash \texttt{set}[a]\,(e_1; e_2) : \tau\,[\chi]} \tag{40.11b}$$

Rule (40.11a) records the demand for a binding for the symbol a incurred by retrieving its value. Rule (40.11b) propagates the fact that the symbol a is bound to the body of the fluid binding.

The dynamic semantics is as specified in Chapter 34. 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 40.3 (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 $\mathcal{L}\{\mathtt{symb}\}$ may be extended to $\mathcal{L}\{\mathtt{symb}\,\mathtt{gen}\}$, 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:

$$\frac{\Sigma, a : \sigma\,\Gamma \vdash e : \tau \quad \Sigma \vdash \tau\ \mathsf{type}}{\Sigma\,\Gamma \vdash \mathtt{new}[\sigma]\,(a.e) : \langle\sigma\rangle\tau} \tag{40.12}$$

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

$$\mathtt{gen}(\nu(a{:}\mathtt{nat}.\,\mathtt{set}\,a\,\mathtt{to}\,\mathtt{z}\,\mathtt{in}\,\lambda(x{:}\mathtt{nat}.\,\mathtt{box}(\ldots\mathtt{get}\,a\ldots))))$$

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 (40.12) 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,

$$\mathtt{gen}(\nu(a{:}\mathtt{nat}.\,\lambda(x{:}\mathtt{nat}.\,\mathtt{box}(\mathtt{set}\,a\,\mathtt{to}\,\mathtt{z}\,\mathtt{in}\,\ldots\mathtt{get}\,a\ldots))),$$

then the type of the λ-abstraction may have the form $\texttt{nat} \to \Box_\chi\, \tau$, 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 (40.12) 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.

40.3 Exercises

Part XIV

Laziness

Chapter 41

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*.

41.1 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 13. There we specified the λ-abstractions are values, and that applications are evaluated according to the following rules:

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \tag{41.1a}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')} \tag{41.1b}$$

$$\frac{e_2 \text{ val}}{\lambda(x:\tau.e)(e_2) \mapsto [e_2/x]e} \tag{41.1c}$$

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:

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \tag{41.2a}$$

$$\frac{}{\lambda(x:\tau.e)(e_2) \mapsto [e_2/x]e} \tag{41.2b}$$

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.

$$\frac{\langle e_1, e_2\rangle \ \mathsf{val}}{\mathtt{fst}(\langle e_1, e_2\rangle) \mapsto e_1} \tag{41.3}$$

$$\frac{\langle e_1, e_2\rangle \ \mathsf{val}}{\mathtt{snd}(\langle e_1, e_2\rangle) \mapsto e_1} \tag{41.4}$$

$$\frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')} \tag{41.5}$$

$$\frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')} \tag{41.6}$$

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 16, evaluates the components of a pair before deeming it to be a value: specified by the following additional rules:

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\langle e_1, e_2\rangle \ \mathsf{val}} \tag{41.7}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle} \tag{41.8}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle} \tag{41.9}$$

The *lazy* (or *non-strict*) semantics, on the other hand, deems any pair to be a value, regardless of whether its components are values:

$$\frac{}{\langle e_1, e_2\rangle \ \mathsf{val}} \tag{41.10}$$

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, `if` e `then` e_1 `else` e_2, arising from the definition of `bool` to be the sum type `unit + unit` as described in Chapter 17 — the "then" and the "else" branches lie within the scope of an (implicit) bound variable, and hence are not eligible for evaluation!

41.2 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)*
$\tau_1 \otimes \tau_2$	$e_1 \otimes e_2$	`let` $x_1 \otimes x_2$ `be` e `in` e'
$\mathbf{0}$	*(none)*	$\mathtt{abort}_\tau(e)$
$\tau_1 \oplus \tau_2$	$\mathtt{lft}_\tau(e)$, $\mathtt{rht}_\tau(e)$	`choose` e $\{\mathtt{lft}(x_1) \Rightarrow e_1 \mid \mathtt{rht}(x_2) \Rightarrow e_2\}$
$\tau_1 \circ\!\!\rightarrow \tau_2$	$\lambda^\circ(x\!:\!\tau_1.\, e_2)$	$\mathtt{ap}^\circ(e_1; e_2)$

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.

41.3 Self-Reference

We have seen in Chapter 15 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.

41.4 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	::=	`susp(`τ`)`	τ `susp`
Expr	e	::=	`susp[`τ`](`x`.`e`)`	`susp` x `:` τ `is` e
		|	`force(`e`)`	`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 `susp(`e`)` for `susp[`τ`](`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:

$$\frac{\Gamma, x : \mathtt{susp}(\tau) \vdash e : \tau}{\Gamma \vdash \mathtt{susp}[\tau](x.e) : \mathtt{susp}(\tau)} \tag{41.11a}$$

$$\frac{\Gamma \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{force}(e) : \tau} \tag{41.11b}$$

In Rule (41.11a) the variable x, which refers to the suspension itself, is assumed to have type $\mathtt{susp}(\tau)$ while checking that the suspended computation, e, has type τ.

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

$$\overline{\mathtt{susp}[\tau](x.e)\ \mathsf{val}} \tag{41.12a}$$

$$\frac{e \mapsto e'}{\mathtt{force}(e) \mapsto \mathtt{force}(e')} \tag{41.12b}$$

$$\overline{\mathtt{force}(\mathtt{susp}[\tau](x.e)) \mapsto [\mathtt{susp}[\tau](x.e)/x]e} \tag{41.12c}$$

Rule (41.12c) 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 41.1 (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:

$$\top = \mathbf{1} \tag{41.13a}$$

$$\langle\rangle = \bullet \tag{41.13b}$$

$$\tau_1 \times \tau_2 = \tau_1\,\mathtt{susp} \otimes \tau_2\,\mathtt{susp} \tag{41.14a}$$

$$\langle e_1, e_2\rangle = \mathtt{susp}(e_1) \otimes \mathtt{susp}(e_2) \tag{41.14b}$$

$$\mathtt{fst}(e) = \mathtt{let}\ x \otimes \_\ \mathtt{be}\ e\ \mathtt{in}\ \mathtt{force}(x) \tag{41.14c}$$

$$\mathtt{snd}(e) = \mathtt{let}\ \_ \otimes y\ \mathtt{be}\ e\ \mathtt{in}\ \mathtt{force}(y) \tag{41.14d}$$

$$\mathbf{0} = \bot \tag{41.15a}$$

$$\mathtt{abort}_\tau(e) = \mathtt{abort}_\tau\, e \tag{41.15b}$$

$$\tau_1 \oplus \tau_2 = \tau_1\,\texttt{susp} + \tau_2\,\texttt{susp} \tag{41.16a}$$
$$\texttt{lft}(e) = \texttt{in[l]}(\texttt{susp}(e)) \tag{41.16b}$$
$$\texttt{rht}(e) = \texttt{in[r]}(\texttt{susp}(e)) \tag{41.16c}$$

$$\begin{aligned}&\texttt{choose}\,e\,\{\texttt{lft}(x_1)\Rightarrow e_1 \mid \texttt{rht}(x_2)\Rightarrow e_2\}\\ &= \texttt{case}\,e\,\{\texttt{in[l]}(y_1) \Rightarrow [\texttt{force}(y_1)/x_1]e_1 \mid \texttt{in[r]}(y_2) \Rightarrow [\texttt{force}(y_2)/x_2]e_2\}\end{aligned} \tag{41.16d}$$

$$\tau_1 \to \tau_2 = \tau_1\,\texttt{susp} \circ\!\!\to \tau_2 \tag{41.17a}$$
$$\lambda(x{:}\tau_1.\,e_2) = \lambda^{\circ}(x{:}\tau_1\,\texttt{susp}.\,[\texttt{force}(x)/x]e_2) \tag{41.17b}$$
$$e_1(e_2) = \texttt{ap}^{\circ}(e_1;\texttt{susp}(e_2)) \tag{41.17c}$$

In the case of lazy case analysis and call-by-name functions we replace occurrences of the bound variable, x, with $\texttt{force}(x)$ 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 $\texttt{force}(x)$ is delayed. In particular, expressions of the form $\texttt{susp}(\texttt{force}(x))$ may be safely replaced by x, since forcing the former computation simply forces x.

41.5 Exercises

Chapter 42

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 41 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 41, 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.

42.1 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is given by a transition system whose states have the form $e \mathbin{@} \mu$, 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 35.)

The rules defining the call-by-need dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ are as follows:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{42.1a}$$

$$\frac{}{\mathtt{s}(x)\ \mathsf{val}} \tag{42.1b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{42.1c}$$

$$\frac{}{x \mathbin{@} \langle x : e \rangle\ \mathsf{initial}} \tag{42.1d}$$

$$\frac{e\ \mathsf{val}}{e \mathbin{@} \mu\ \mathsf{final}} \tag{42.1e}$$

$$\frac{e \text{ val}}{x \mathbin{@} \mu \otimes \langle x : e \rangle \mapsto e \mathbin{@} \mu \otimes \langle x : e \rangle} \tag{42.1f}$$

$$\frac{e \mathbin{@} \mu \otimes \langle x : \bullet \rangle \mapsto e' \mathbin{@} \mu' \otimes \langle x : \bullet \rangle}{x \mathbin{@} \mu \otimes \langle x : e \rangle \mapsto x \mathbin{@} \mu' \otimes \langle x : e' \rangle} \tag{42.1g}$$

$$\overline{\mathtt{s}(e) \mathbin{@} \mu \mapsto \mathtt{s}(x) \mathbin{@} \mu \otimes \langle x : e \rangle} \tag{42.1h}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\mathtt{ifz}(e; e_0; x.e_1) \mathbin{@} \mu \mapsto \mathtt{ifz}(e'; e_0; x.e_1) \mathbin{@} \mu'} \tag{42.1i}$$

$$\overline{\mathtt{ifz}(\mathtt{z}; e_0; x.e_1) \mathbin{@} \mu \mapsto e_0 \mathbin{@} \mu} \tag{42.1j}$$

$$\frac{x \notin dom(\mu)}{\mathtt{ifz}(\mathtt{s}(x); e_0; x.e_1) \mathbin{@} \mu \mapsto e_1 \mathbin{@} \mu} \tag{42.1k}$$

$$\frac{e_1 \mathbin{@} \mu \mapsto e_1' \mathbin{@} \mu'}{e_1(e_2) \mathbin{@} \mu \mapsto e_1'(e_2) \mathbin{@} \mu'} \tag{42.1l}$$

$$\frac{x \notin dom(\mu)}{\lambda(x{:}\tau.\, e)(e_2) \mathbin{@} \mu \mapsto e \mathbin{@} \mu \otimes \langle x : e_2 \rangle} \tag{42.1m}$$

$$\frac{x \notin dom(\mu)}{\mathtt{fix}[\tau](x.e) \mathbin{@} \mu \mapsto x \mathbin{@} \mu \otimes \langle x : e \rangle} \tag{42.1n}$$

Rules (42.1a) through (42.1c) specify that $\mathtt{z}$ is a value, any expression of the form $\mathtt{s}(x)$, 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 (42.1d) specifies that an initial state consists of a binding for a closed expression, e, in memory, together with a demand for its binding. Rule (42.1e) specifies that a final state has the form $e \mathbin{@} \mu$, where e is a value.

Rule (42.1h) specifies that evaluation of $\mathtt{s}(e)$ yields the value $\mathtt{s}(x)$, 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 (42.1k), 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 (42.1l) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (42.1m) 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 (42.1f) governs a variable whose binding is a value, which is returned as the value of that variable. Rule (42.1g) 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 (42.1n) implements general recursion. Recall from Chapter 15 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 (42.1n) 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 (42.1g), 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 $\texttt{fix}\, f{:}\sigma \to \tau\, \texttt{is}\, \lambda(x{:}\sigma.e)$ 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 (42.1g) 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 35.

The type safety of the by-need semantics for lazy $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is proved using methods similar to those developed in Chapter 35 for references. To do so we define the judgement $e \mathbin{@} \mu\ \mathsf{ok}$ 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 42.1 (Preservation)**.** *If $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$ and $e \mathbin{@} \mu$ ok, then $e' \mathbin{@} \mu'$ ok.*

Proof. The proof is by rule induction on Rules (42.1). 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 (42.1l), 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 (42.1g), 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 $e_0 \mathbin{@} \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0' \mathbin{@} \mu_0' \otimes \langle x : \bullet \rangle$. 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 $e \mathbin{@} \mu\ \mathsf{loops}$ by the following rules:

$$\frac{}{x \mathbin{@} \mu \otimes \langle x : \bullet \rangle\ \mathsf{loops}} \tag{42.2a}$$

$$\frac{e \mathbin{@} \mu \otimes \langle x : \bullet \rangle\ \mathsf{loops}}{x \mathbin{@} \mu \otimes \langle x : e \rangle\ \mathsf{loops}} \tag{42.2b}$$

$$\frac{e \mathbin{@} \mu\ \mathsf{loops}}{\mathtt{ifz}(e; e_0; x.e_1) \mathbin{@} \mu\ \mathsf{loops}} \tag{42.2c}$$

$$\frac{e_1 \mathbin{@} \mu \ \mathsf{loops}}{\mathtt{ap}(e_1;e_2) \mathbin{@} \mu \ \mathsf{loops}} \tag{42.2d}$$

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 42.2 (Progress)**.** *If $e \mathbin{@} \mu$ ok, then either $e \mathbin{@} \mu$ final, or $e \mathbin{@} \mu$ loops, or there exists μ' and e' such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$.*

Proof. We prove by rule induction on the static semantics that if $\mu : \Gamma \vdash e : \tau$, then either e val, or $e \mathbin{@} \mu$ loops, or $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$ 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 (15.1f), 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 $e_1 \mathbin{@} \mu$ loops or $e_1 \mathbin{@} \mu \mapsto e_1' \mathbin{@} \mu'$.

In the first case it may be shown that $e_1 = \mathtt{lam}[\tau_2](x.e)$, and hence that $\mathtt{ap}(e_1;e_2) \mathbin{@} \mu \mapsto e \mathbin{@} \mu' \otimes \langle x : e_2 \rangle$ by Rule (42.1m), where x is chosen by α-equivalence to lie outside of the domain of μ'. In the second case we have by Rule (42.2d) that $\mathtt{ap}(e_1;e_2) \mathbin{@} \mu$ loops. In the third case we have by Rule (42.1l) that $\mathtt{ap}(e_1;e_2) \mathbin{@} \mu \mapsto \mathtt{ap}(e_1';e_2) \mathbin{@} \mu'$.

Now consider Rule (15.1a), 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 $e_0 \mathbin{@} \mu_0 \otimes \langle x : \bullet \rangle$ loops, or $e_0 \mathbin{@} \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0' \mathbin{@} \mu_0' \otimes \langle x : \bullet \rangle$.

If e_0 val, then $x \mathbin{@} \mu_0 \otimes \langle x : e_0 \rangle \mapsto e_0 \mathbin{@} \mu_0 \otimes \langle x : e_0 \rangle$ by Rule (42.1f). If $e_0 \mathbin{@} \mu_0 \otimes \langle x : \bullet \rangle$ loops, then $x \mathbin{@} \mu_0 \otimes \langle x : e_0 \rangle$ loops by Rule (42.2b). Finally, if $e_0 \mathbin{@} \mu_0 \otimes \langle x : \bullet \rangle \mapsto e_0' \mathbin{@} \mu_0' \otimes \langle x : \bullet \rangle$, then $x \mathbin{@} \mu_0 \otimes \langle x : e_0 \rangle \mapsto x \mathbin{@} \mu_0' \otimes \langle x : e_0' \rangle$ by Rule (42.1g). □

42.2 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:

$$\frac{}{\texttt{pair}(x_1;x_2)\ \mathsf{val}} \tag{42.3a}$$

$$\frac{}{\texttt{pair}(e_1;e_2)\ \texttt{@}\ \mu \mapsto \texttt{pair}(x_1;x_2)\ \texttt{@}\ \mu \otimes \langle x_1 : e_1\rangle \otimes \langle x_2 : e_2\rangle} \tag{42.3b}$$

$$\frac{e\ \texttt{@}\ \mu \mapsto e'\ \texttt{@}\ \mu'}{\texttt{fst}(e)\ \texttt{@}\ \mu \mapsto \texttt{fst}(e')\ \texttt{@}\ \mu'} \tag{42.3c}$$

$$\frac{}{\texttt{fst}(\texttt{pair}(x_1;x_2))\ \texttt{@}\ \mu \mapsto x_1\ \texttt{@}\ \mu} \tag{42.3d}$$

$$\frac{e\ \texttt{@}\ \mu\ \mathsf{loops}}{\texttt{fst}(e)\ \texttt{@}\ \mu\ \mathsf{loops}} \tag{42.3e}$$

$$\frac{e\ \texttt{@}\ \mu \mapsto e'\ \texttt{@}\ \mu'}{\texttt{snd}(e)\ \texttt{@}\ \mu \mapsto \texttt{snd}(e')\ \texttt{@}\ \mu'} \tag{42.3f}$$

$$\frac{}{\texttt{snd}(\texttt{pair}(x_1;x_2))\ \texttt{@}\ \mu \mapsto x_2\ \texttt{@}\ \mu} \tag{42.3g}$$

$$\frac{e\ \texttt{@}\ \mu\ \mathsf{loops}}{\texttt{snd}(e)\ \texttt{@}\ \mu\ \mathsf{loops}} \tag{42.3h}$$

A pair is considered a value only if its arguments are variables (Rule (42.3a)), which are introduced when the pair is created (Rule (42.3b)). 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.

42.3 Suspensions By Need

In Chapter 41 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:

$$\frac{}{x\ \mathsf{val}} \tag{42.4a}$$

$$\frac{}{\mathtt{susp[}\tau\mathtt{]}(x.e)\ \mathtt{@}\ \mu \mapsto x\ \mathtt{@}\ \mu \otimes \langle x : e\rangle} \tag{42.4b}$$

$$\frac{e\ \mathtt{@}\ \mu \mapsto e'\ \mathtt{@}\ \mu'}{\mathtt{force(}e\mathtt{)}\ \mathtt{@}\ \mu \mapsto \mathtt{force(}e'\mathtt{)}\ \mathtt{@}\ \mu'} \tag{42.4c}$$

$$\frac{e\ \mathsf{val}}{\mathtt{force(}x\mathtt{)}\ \mathtt{@}\ \mu \otimes \langle x : e\rangle \mapsto e\ \mathtt{@}\ \mu \otimes \langle x : e\rangle} \tag{42.4d}$$

$$\frac{e\ \mathtt{@}\ \mu \otimes \langle x : \bullet\rangle \mapsto e'\ \mathtt{@}\ \mu' \otimes \langle x : \bullet\rangle}{\mathtt{force(}x\mathtt{)}\ \mathtt{@}\ \mu \otimes \langle x : e\rangle \mapsto \mathtt{force(}x\mathtt{)}\ \mathtt{@}\ \mu' \otimes \langle x : e'\rangle} \tag{42.4e}$$

It is straightforward to adapt the type safety proof given in Section 42.1 on page 350 to the special case of suspension types.

42.4 Exercises

Part XV

Parallelism

Chapter 43

Speculative Parallelism

The semantics of call-by-need given in Chapter 42 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.

43.1 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 42.1 on page 350 by the following general rule:

$$\frac{e \mathbin{@} \mu \otimes \langle y : \bullet \rangle \mapsto e' \mathbin{@} \mu' \otimes \langle y : \bullet \rangle}{e_0 \mathbin{@} \mu \otimes \langle y : e \rangle \mapsto e_0 \mathbin{@} \mu' \otimes \langle y : e' \rangle} \tag{43.1}$$

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 42 to permit arbitrary context switches that evaluate suspended computations.

43.2 Speculative Parallelism

The non-deterministic semantics given in Section 43.1 on the previous 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.

$$\frac{\left\{\begin{array}{c} e_i \mathbin{@} \mu \otimes \langle x_1 : \bullet\rangle \otimes \cdots \otimes \langle x_k : \bullet\rangle \\ \mapsto \\ e_i' \mathbin{@} \mu \otimes \langle x_1 : \bullet\rangle \otimes \cdots \otimes \langle x_k : \bullet\rangle \otimes \mu_i \end{array}\right\} \quad (\forall 1 \leq i \leq k)}{\left\{\begin{array}{c} \mu \otimes \langle x_1 : e_1\rangle \otimes \cdots \otimes \langle x_k : e_k\rangle \\ \mapsto \\ \mu \otimes \langle x_1 : e_1'\rangle \otimes \cdots \otimes \langle x_k : e_k'\rangle \otimes \mu_1 \otimes \cdots \otimes \mu_k \end{array}\right\}} \tag{43.2}$$

This rule may be seen as a generalization of Rule (42.1g), 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by replacing Rule (42.1g) by the following rule:

$$\frac{\mu \mapsto \mu'}{e \mathbin{@} \mu \mapsto e \mathbin{@} \mu'} \tag{43.3}$$

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 (42.1g) 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 (43.2), 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 (43.2). 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 $\mu_1, \ldots, \mu_k$ 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).

43.3 Exercises

Chapter 44

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 43), 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.

44.1 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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{let}(e_1; e_2; x_1.x_2.e) : \tau} \tag{44.1}$$

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 44.4 on page 373.)

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:

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{seq} \texttt{let}(e_1'; e_2; x_1.x_2.e)} \tag{44.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{seq} \texttt{let}(e_1; e_2'; x_1.x_2.e)} \tag{44.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{seq} [e_1, e_2/x_1, x_2]e} \tag{44.2c}$$

The parallel dynamics is given by a transition judgement of the form $e \mapsto_{par} e'$, defined as follows:

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par} \texttt{let}(e_1'; e_2'; x_1.x_2.e)} \tag{44.3a}$$

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \text{ val}}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par} \texttt{let}(e_1'; e_2; x_1.x_2.e)} \tag{44.3b}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto_{par} e_2'}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par} \texttt{let}(e_1; e_2'; x_1.x_2.e)} \tag{44.3c}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par} [e_1, e_2/x_1, x_2]e} \tag{44.3d}$$

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 44.3 on page 370.)

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 45, which exhibit *non-deterministic parallelism*, or *concurrency*.

Lemma 44.1. *If $\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par}^* v$ 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, $\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{par} e'$. For Rule (44.3d), 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 44.2. *If $\texttt{let}(e_1; e_2; x_1.x_2.e) \mapsto_{seq}^* v$ 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 44.2. □

Theorem 44.3. *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 (44.2c), then by Rule (44.3d) we have $e \mapsto_{par} e'$, and hence $e \mapsto_{par}^* v$. If $e \mapsto_{seq} e'$ by Rule (44.2a), 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 44.1 on the previous 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 (44.3d) carries over directly to the sequential case by Rule (44.2c). Consider Rule (44.3a). 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 44.2 on the preceding 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 44.3 on the previous 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 43, 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.

44.2 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:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{44.4}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{44.5}$$

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 12.7 on page 96 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 44.1 on page 363.

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:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\texttt{let}(e_1; e_2; x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{44.6}$$

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 44.4. *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 (44.6). 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

$$\begin{aligned}
\texttt{let}(e_1; e_2; x_1\,.\,x_2\,.\,e) &\mapsto^{w_1}_{seq} \texttt{let}(v_1; e_2; x_1\,.\,x_2\,.\,e) && (44.7)\\
&\mapsto^{w_2}_{seq} \texttt{let}(v_1; v_2; x_1\,.\,x_2\,.\,e) && (44.8)\\
&\mapsto_{seq} [v_1, v_2/x_1, x_2]e && (44.9)\\
&\mapsto^{w}_{seq} v. && (44.10)
\end{aligned}$$

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 $e_1 \mapsto^{d_1}_{par} v_1$, $e_2 \mapsto^{d_2}_{par} v_2$, and $e \mapsto^{d}_{par} v$, 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:

$$\begin{aligned}
\texttt{let}(e_1; e_2; x_1\,.\,x_2\,.\,e) &\mapsto^{d_1}_{par} \texttt{let}(v_1; e_2'; x_1\,.\,x_2\,.\,e) && (44.11)\\
&\mapsto^{d_2-d_1}_{par} \texttt{let}(v_1; v_2; x_1\,.\,x_2\,.\,e) && (44.12)\\
&\mapsto_{par} [v_1, v_2/x_1, x_2]e && (44.13)\\
&\mapsto^{d}_{par} v. && (44.14)
\end{aligned}$$

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 (44.2c). 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 (44.6). 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 (44.3a), 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

$$\begin{aligned} dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) && (44.15)\\ &= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) && (44.16)\\ &= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 && (44.17)\\ &= dp(c') + 1 && (44.18)\\ &= d' + 1 && (44.19)\\ &= d, && (44.20) \end{aligned}$$

which completes the proof. □

44.3 Provable Implementations

Theorem 44.4 on page 368 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 44.5. *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 44.5 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 44.5 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 evalu-

ation 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 44.5 on the previous 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 44.5 on the preceding 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 (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.

44.4 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 44.5 on page 371 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	::=	`vec(`τ`)`	τ `vec`
Expr	e	::=	`vec(`e_0`,...,`e_{n-1}`)`	`[`e_0`,...,`e_{n-1}`]`
		|	`sub(`e_1`;`e_2`)`	e_1`[`e_2`]`
		|	`siz(`e`)`	`size (`e`)`
		|	`idx(`e`)`	`index(`e`)`
		|	`map(`e_1`;`x`.`e_2`)`	`<`e_2 `|` $x \in e_1$`>`

The expression $\texttt{vec}(e_0, \ldots, e_{n-1})$ evaluates to an n-vector whose elements are given by the expressions $e_0, \ldots, e_{n-1}$. The operation $\texttt{sub}(e_1; e_2)$ retrieves the element of the vector given by e_1 at the index given by e_2. The operation $\texttt{siz}(e)$ returns the number of elements in the vector given by e. The operation $\texttt{idx}(e)$ creates a vector of length n (given by e) whose elements are $0, \ldots, n-1$. The operation $\texttt{map}(e_1; x.e_2)$ 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:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \texttt{vec}(e_0, \ldots, e_{n-1}) : \texttt{vec}(\tau)} \tag{44.21a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{vec}(\tau) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{sub}(e_1; e_2) : \tau} \tag{44.21b}$$

$$\frac{\Gamma \vdash e : \texttt{vec}(\tau)}{\Gamma \vdash \texttt{siz}(e) : \texttt{nat}} \tag{44.21c}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{idx}(e) : \texttt{vec}(\texttt{nat})} \tag{44.21d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{vec}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \tau'}{\Gamma \vdash \texttt{map}(e_1; x.e_2) : \texttt{vec}(\tau')} \tag{44.21e}$$

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 44.3 on page 365. The cost semantics of these primitives is given by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{vec}(e_0, \ldots, e_{n-1}) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \texttt{vec}(v_0, \ldots, v_{n-1})} \tag{44.22a}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{vec}(v_0, \ldots, v_{n-1}) \quad e_2 \Downarrow^{c_2} \texttt{num}[i] \quad (0 \leq i < n)}{\texttt{sub}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{44.22b}$$

$$\frac{e \Downarrow^{c} \texttt{vec}(v_0, \ldots, v_{n-1})}{\texttt{siz}(e) \Downarrow^{c \oplus \mathbf{1}} \texttt{num}[n]} \tag{44.22c}$$

$$\frac{e \Downarrow^{c} \texttt{num}[n]}{\texttt{idx}(e) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} \mathbf{1}} \texttt{vec}(0, \ldots, n-1)} \tag{44.22d}$$

$$\frac{\begin{array}{c} e_1 \Downarrow^{c_1} \texttt{vec}(v_0, \ldots, v_{n-1}) \\ [v_0/x]e_2 \Downarrow^{c_0} v_0' \quad \ldots \quad [v_{n-1}/x]e_2 \Downarrow^{c_{n-1}} v_{n-1}' \end{array}}{\texttt{map}(e_1; x.e_2) \Downarrow^{c_1 \oplus (c_0 \otimes \ldots \otimes c_{n-1})} \texttt{vec}(v_0', \ldots, v_{n-1}')} \tag{44.22e}$$

The cost semantics for vectors may be validated against the sequential and parallel dynamics in a manner similar to Theorem 44.4 on page 368 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 44.5 on page 371 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 44.5 on page 371, 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 44.5 on page 371 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.

44.5 Exercises

Part XVI

Concurrency

Chapter 45

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 nondeterministic choice operator. Even the choice operator can be eliminated in favor of a protocol for treating a parallel composition of events as a nondeterministic choice among them.

45.1 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	::=	`await(`E`)`	`$`E
Event	E	::=	`null`	**0**
		|	`choice(`E_1`;`E_2`)`	$E_1 + E_2$
		|	`rcv[`a`](`P`)`	`?`a`.`P
		|	`snd[`a`](`P`)`	`!`a`.`P
Action	α	::=	`rcv(`a`)`	`?`a
		|	`snd(`a`)`	`!`a
		|	`sil`	ϵ

The variables a, b, and c range over *channels*, which serve as conduits for synchronization. The *receive action*, `?`a, and the *send action*, `!`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:

$$\frac{E \equiv E'}{\$\,E \equiv \$\,E'} \tag{45.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{45.1b}$$

$$\frac{P \equiv P'}{?a\,.\,P \equiv ?a\,.\,P'} \tag{45.1c}$$

$$\frac{P \equiv P'}{!a\,.\,P \equiv !a\,.\,P'} \tag{45.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{45.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{45.1f}$$

$$\overline{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3} \tag{45.1g}$$

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.

$$V = \texttt{\$(?2p.\$(!tea.}V\texttt{ + ?2p.\$(!cof.}V\texttt{)))}$$

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.

45.2 Concurrent Interaction

We enrich the language of processes with concurrent composition.

Category	*Item*		*Abstract*	*Concrete*
Process	P	::=	`await(`E`)`	`$`E
		|	`stop`	**1**
		|	`par(`P_1`;`P_2`)`	$P_1 \parallel P_2$

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 `$`**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:

$$\overline{P \parallel \mathbf{1} \equiv P} \tag{45.2a}$$

$$\overline{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{45.2b}$$

$$\overline{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{45.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{45.2d}$$

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 $\mathbf{1}$.

The dynamic semantics of concurrent interaction is defined by an action-indexed family of transition judgements, $P \overset{\alpha}{\mapsto} P'$, where the silent action indexes the silent transition $P \mapsto P'$. 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.

$$\frac{P_1 \equiv P_2 \quad P_2 \overset{\alpha}{\mapsto} P_2' \quad P_2' \equiv P_2}{P_1 \overset{\alpha}{\mapsto} P_2} \tag{45.3a}$$

$$\overline{\$\,(\mathtt{!}a\,.\,P + E) \overset{\mathtt{!}a}{\longmapsto} P} \tag{45.3b}$$

$$\overline{\$\,(\mathtt{?}a\,.\,P + E) \overset{\mathtt{?}a}{\longmapsto} P} \tag{45.3c}$$

$$\frac{P_1 \parallel P_2 \overset{\alpha}{\mapsto} P_1' \parallel P_2}{P_1 \overset{\alpha}{\mapsto} P_1'} \tag{45.3d}$$

$$\frac{P_1 \overset{\mathtt{!}a}{\longmapsto} P_1' \quad P_2 \overset{\mathtt{?}a}{\longmapsto} P_2'}{P_1 \parallel P_2 \mapsto P_1' \parallel P_2'} \tag{45.3e}$$

Rules (45.3b) and (45.3c) specify that any of the events on which a process is synchronizing may occur. Rule (45.3e) 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:

$$\begin{aligned} V \parallel U &\mapsto \$\,\texttt{!tea}.V + \texttt{?2p}.\$\,\texttt{!cof}.V \parallel \$\,\texttt{!2p}.\$\,\texttt{?cof}.\mathbf{1} \\ &\mapsto \$\,\texttt{!cof}.V \parallel \$\,\texttt{?cof}.\mathbf{1} \\ &\mapsto V \end{aligned}$$

These steps are justified, respectively, by the following pairs of labelled transitions:

$$U \xmapsto{\texttt{!2p}} U' = \$\,\texttt{!2p}.\$\,\texttt{?cof}.\mathbf{1}$$
$$V \xmapsto{\texttt{?2p}} V' = \$\,(\texttt{!tea}.V + \texttt{?2p}.\$\,\texttt{!cof}.V)$$

$$U' \xmapsto{\texttt{!2p}} U'' = \$\,\texttt{?cof}.\mathbf{1}$$
$$V' \xmapsto{\texttt{?2p}} V'' = \$\,\texttt{!cof}.V$$

$$U'' \xmapsto{\texttt{?cof}} \mathbf{1}$$
$$V'' \xmapsto{\texttt{!cof}} V$$

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 nondeterministic choice of events by a process.

45.3 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

$$A = *\,\$\,(?a\,.\,P(\$\,(!a\,.\,\mathbf{1}))),$$

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

$$\$\,(?a\,.\,P(\$\,(!a\,.\,\mathbf{1}))).$$

To start process P we concurrently compose the activator, A, with an initiator process, $\$\,(!a\,.\,\mathbf{1})$. Observe that

$$A \parallel \$\,(!a\,.\,\mathbf{1}) \mapsto A \parallel P(!a\,.\,\mathbf{1}),$$

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:

$$V_1 = *\,\$\,(?v\,.\,V_2) \tag{45.4}$$

$$V_2 = \$\,(?\mathtt{2p}\,.\,\$\,(!\mathtt{tea}\,.\,V_0 + ?\mathtt{2p}\,.\,\$\,(!\mathtt{cof}\,.\,V_0))) \tag{45.5}$$

$$V_0 = \$\,(!v\,.\,\mathbf{1}) \tag{45.6}$$

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 $V_0 \parallel V_1$.

45.4 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:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{45.7a}$$

$$\frac{P \equiv P'}{\nu(a\,.\,P) \equiv \nu(a\,.\,P')} \tag{45.7b}$$

$$\frac{a \# P_2}{\nu(a\,.\,P_1) \parallel P_2 \equiv \nu(a\,.\,P_1 \parallel P_2)} \tag{45.7c}$$

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.

$$\frac{P \overset{\alpha}{\mapsto} P' \quad a \# \alpha}{\nu(a\,.\,P) \overset{\alpha}{\mapsto} \nu(a\,.\,P')} \tag{45.8}$$

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.

45.5 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	::=	$\texttt{rcv}[a](x.P)$	$\texttt{?}a(x).P$
		$\mid$	$\texttt{snd}[a;b](P)$	$\texttt{!}a(b).P$

The event $\texttt{?}a(x).P$ 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	::=	$\texttt{rcv}[a](b)$	$\texttt{?}a(b)$
		$\mid$	$\texttt{snd}[a](b)$	$\texttt{!}a(b)$

The action $\texttt{!}a(b)$ represents a write, or send, of a channel, b, along a channel, a. The action $\texttt{?}a(b)$ 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.

$$\$\left(\texttt{!}a(b).P + E\right) \xmapsto{\texttt{!}a(b)} P \tag{45.9a}$$

$$\$\left(\texttt{?}a(x).P + E\right) \xmapsto{\texttt{?}a(b)} [b/x]P \tag{45.9b}$$

$$\frac{P_1 \xmapsto{\texttt{!}a(b)} P_1' \quad P_2 \xmapsto{\texttt{?}a(b)} P_2'}{P_1 \parallel P_2 \mapsto P_1' \parallel P_2'} \tag{45.9c}$$

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 (45.9b) may be seen as "guessing" that the received data will be b, which is substituted into the resulting process.

45.6 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	$::=$	$\mathtt{rcv}[a](x_1,\ldots,x_k).P$	$?a(x_1,\ldots,x_k).P$
		$\mid$	$\mathtt{snd}[a;b_1,\ldots,b_k](P)$	$!a(b_1,\ldots,b_k).P$
Action	α	$::=$	$\mathtt{rcv}[a](b_1,\ldots,b_k)$	$?a(b_1,\ldots,b_k)$
		$\mid$	$\mathtt{snd}[a](b_1,\ldots,b_k)$	$!a(b_1,\ldots,b_k)$

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.

45.7 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:

$$C(x,g,s) = \$\,(S(g,s) + G(x,g,s)) \tag{45.10}$$
$$S(g,s) = ?s(y)\,.\,C(y,g,s) \tag{45.11}$$
$$G(x,g,s) = !g(x)\,.\,C(x,g,s) \tag{45.12}$$

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.

$$R(c) = *\,\$\,(?c(x,g,s)\,.\,C'(c,x,g,s)) \tag{45.13}$$
$$C'(c,x,g,s) = \$\,(S'(c,g,s) + G'(c,x,g,s)) \tag{45.14}$$
$$S'(c,g,s) = ?s(y)\,.\,\$\,(!c(y,g,s)\,.\,\mathbf{1}) \tag{45.15}$$
$$G'(c,x,g,s) = !g(x)\,.\,\$\,(!c(x,g,s)\,.\,\mathbf{1}) \tag{45.16}$$

The reference cell server repeatedly awaits receipt of a creation message on channel c, 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

$$\nu(c\,.\,R(c) \parallel \nu(g\,.\,\nu(s\,.\,\$\,!c(0,g,s)\,.\,\$\,!s(1)\,.\,\$\,?g(x)\,\ldots.)))$$

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.

45.8 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*, $\texttt{!}a\texttt{(}b\texttt{)}\texttt{.}P$, 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	::=	$\texttt{snd[}a\texttt{](}b\texttt{)}$	$\texttt{!}a\texttt{(}b\texttt{)}$
		|	$\texttt{await(}E\texttt{)}$	$\texttt{\$}\,E$
		|	$\texttt{par(}P_1\texttt{;}P_2\texttt{)}$	$P_1 \parallel P_2$
		|	$\texttt{new(}a\texttt{.}P\texttt{)}$	$\nu\texttt{(}a\texttt{.}P\texttt{)}$
Event	E	::=	$\texttt{null}$	$\mathbf{0}$
		|	$\texttt{rcv[}a\texttt{](}x\texttt{.}P\texttt{)}$	$\texttt{?}a\texttt{(}x\texttt{)}\texttt{.}P$
		|	$\texttt{choice(}E_1\texttt{;}E_2\texttt{)}$	$E_1 + E_2$

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 (45.9a), and adding the following rule for the asynchronous send process:

$$\frac{}{\mathtt{!}a(b) \xmapsto{\mathtt{!}a(b)} \mathbf{1}} \tag{45.17}$$

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 $\mathtt{\$}\,\mathtt{!}a(b)\,.\,\mathbf{1}$, 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

$$\nu(a\,.\,\mathtt{\$}\,((\mathtt{!}a(b)\,.\,P) + E) \parallel \mathtt{\$}\,((\mathtt{?}a(x)\,.\,Q) + F))$$

is represented by the asynchronous process

$$\nu(a\,.\,\nu(a_0\,.\,P' \parallel Q')),$$

where $a_0 \# P$, $a_0 \# Q$, and we define

$$P' = \mathtt{!}a(b, a_0) \parallel \mathtt{\$}\,(\mathtt{?}a_0()\,.\,P + E)$$

and

$$Q' = \mathtt{\$}\,(\mathtt{?}a(x, x_0)\,.\,(\mathtt{!}a_0() \parallel Q) + F).$$

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.

45.9 Definability of Input Choice

It turns out that we may simplify the asynchronous π-calculus even further by eliminating the non-deterministic choice of events by defining it in terms of parallel composition of processes. This means, in fact, that we may do away with the concept of an event entirely, and just have a very simple calculus of processes defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Process	P	::=	`snd[`a`](`b`)`	`!`a`(`b`)`
		|	`rcv[`a`](`x`.`P`)`	`?`a`(`x`).`P
		|	`stop`	**1**
		|	`par(`P_1`;`P_2`)`	$P_1 \parallel P_2$
		|	`new(`a`.`P`)`	$\nu(a.P)$

This reduces the language two three main concepts: channels, communication, and concurrent composition.

The elimination of non-deterministic choice is based on the following intuition. Let P be a process of the form

$$\texttt{\$}\,(\texttt{?}a_1(x_1).P_1 + \ldots + \texttt{?}a_k(x_k).P_k).$$

Interaction with this process by a processing sending a channel, b, along channel a_i involves two separable actions:

1. The transmitted value, b, must be substituted for x_i in P_i to obtain the resulting process, $[b/x_i]P_i$, of the interaction.
2. The other events must be "killed off", since they were not chosen by the interaction.

Ignoring the second action for the time being, the first may be met by simply regarding P as the following parallel composition of processes:

$$\texttt{?}a_1(x_1).P_1 \parallel \ldots \parallel \texttt{?}a_k(x_k).P_k.$$

When concurrently composed with a sending process $\texttt{!}a_i(b)$, this process interactions to yield $[b/x]P_i$, representing the same non-deterministic choice

of interaction. However, the interaction fails to "kill off" the processes that were *not* chosen when the communication along a_i *was* chosen.

To rectify this we modify the encoding of choice to incorporate a protocol for signalling the non-selected processes that they are not eligible to participate in any further communication events. This is achieved by associating a fresh channel with each receive event group of the form illustrated by P above, and arranging that if any of the receiving processes is chosen, then the others become "zombies" that are disabled from further interaction. The process P is represented by the process P' given by the expression

$$\nu(t\,.\,S_t \parallel ?a_1(x_1)\,.\,P_1' \parallel \ldots \parallel ?a_k(x_k)\,.\,P_k'),$$

where P_i' is the process

$$\nu(s, f\,.\,!t(s, f) \parallel ?s()\,.\,(F_t \parallel P_i) \parallel ?f()\,.\,(F_t \parallel !a_i(x_i))).$$

The process S_t signals success when contacted on channel t,

$$S_t = ?t(s, f)\,.\,!s()$$

and the process F_t signals failure when contacted on channel t,

$$F_t = ?t(s, f)\,.\,!f().$$

The process P' allocates a new channel that is shared by all of the processes participating in the encoding of the process P. It then creates $k + 1$ processes, one for each summand, and a "success" process that mediates the protocol. The summands all wait for communication on their respective channels, and the mediating process signals success when contacted. When a concurrently executing process interacts with P' by sending a channel b to P_i' along channel a_i', the protocol is initiated. First, the process P_i' sends a newly allocated success and failure channel to the mediator process, and awaits further communication along these channels. (The new channels serve to identify this particular interaction of P' with its environment.) The mediator signals success, and terminates. The signal activates the receive event along the success channel of P_i', which then activates a new mediator, the "failure" process, to replace the original, "success" process, and also activates P_i since this summand has been chosen for the interaction. All other summands remain active, receiving communications on their respective channels, with the concurrently executing mediator being the "failure" process. Should any of these summands be selected for communication, it is their job as zombies to die off after ensuring that the failing mediator is

reinstated (for the sake of the other zombie processes) and *re-sending* the received message so that it may be propagated to a "living" recipient (that is, one that has not been disabled by a previous interaction with one of its cohort).

45.10 Exercises

Chapter 46

Concurrency

In this chapter we utilize the machinery of process calculus presented in Chapter 45 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.

46.1 Framework

The language $\mathcal{L}\{\texttt{conc}\}$ is an extension of $\mathcal{L}\{\texttt{comm}\}$ (described in Chapter 37) 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	::=	$\texttt{return}(e)$	$\texttt{return}(e)$
		|	$\texttt{letcomp}(e; x.m)$	$\texttt{let}\ \texttt{comp}(x)\ \texttt{be}\ e\ \texttt{in}\ m$
Proc	p	::=	$\texttt{proc}[a](m)$	$\{a : m\}$
		|	$\texttt{par}(p_1; p_2)$	$p_1 \parallel p_2$
		|	$\texttt{new}[\tau](x.p)$	$\nu(x{:}\tau.p)$

The basic form of process is $\texttt{proc}[a](m)$, 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 45, 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 37) 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 $p\ \mathsf{ok}$ by the following rules:

$$\frac{\Sigma; \Gamma \vdash m \sim \tau}{\Sigma, a : \tau\ \texttt{proc}; \Gamma \vdash \{a : m\}\ \mathsf{ok}} \tag{46.1a}$$

$$\frac{\Sigma; \Gamma \vdash p_1\ \mathsf{ok} \quad \Sigma \vdash p_2\ \mathsf{ok}}{\Sigma; \Gamma \vdash p_1 \parallel p_2\ \mathsf{ok}} \tag{46.1b}$$

$$\frac{\Sigma, a : \tau; \Gamma \vdash p\ \mathsf{ok}}{\Sigma; \Gamma \vdash \nu(a{:}\tau\,.\,p)\ \mathsf{ok}} \tag{46.1c}$$

$$\frac{\Sigma; \Gamma \vdash p'\ \mathsf{ok} \quad p \equiv p'}{\Sigma; \Gamma \vdash p\ \mathsf{ok}} \tag{46.1d}$$

Rule (46.1a) specifies that a process of the form $\{a : m\}$ is well-formed if m is a command yielding a value of type τ, where a is a process identifier of type τ. Thus, the type of a process is the type of value that it returns.

Rule (46.1b) states that a parallel composition of processes is well-formed if both processes are well-formed. Rule (46.1c) enriches Σ with a new symbol with a type τ chosen so that p is well-formed under this assumption. Finally, Rule (46.1d) states that typing respects structural congruence. Ordinarily such a rule is left implicit, but we state it explicitly for emphasis.

Each extension of $\mathcal{L}\{\texttt{conc}\}$ considered below may introduce new forms of process governed by new formation and execution rules.

The dynamic semantics of $\mathcal{L}\{\texttt{conc}\}$ is defined by judgements of the form $p \mapsto p'$, where p and p' are processes. 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:

$$\frac{m \mapsto m'}{\texttt{proc[}a\texttt{](}m\texttt{)} \mapsto \texttt{proc[}a\texttt{](}m'\texttt{)}} \tag{46.2a}$$

$$\frac{}{\texttt{proc[}a\texttt{](return(}e\texttt{))} \xmapsto{\texttt{!}a(e)} \texttt{proc[}a\texttt{](return(}e\texttt{))}} \tag{46.2b}$$

$$\frac{p_1 \mapsto p_1'}{\texttt{par(}p_1\texttt{;}p_2\texttt{)} \mapsto p_1' \parallel p_2} \tag{46.2c}$$

$$\frac{p \mapsto p'}{\texttt{new[}\tau\texttt{](}a\texttt{.}p\texttt{)} \mapsto \texttt{new[}\tau\texttt{](}a\texttt{.}p'\texttt{)}} \tag{46.2d}$$

$$\frac{p \equiv q \quad q \mapsto q' \quad q' \equiv p'}{p \mapsto p'} \tag{46.2e}$$

Rule (46.2b) specifies that a process whose execution has completed normally announces this fact to the ambient context by offering the returned value labelled with the process's identifier. This allows for other processes to notice that the process labelled a has terminated, and to recover its returned value.

Each form of computation gives rise to new rules for process execution. These rules generally have the form of transitions of the form

$$\{a : m\} \mapsto \nu(a_1 : \tau_1 \ldots \nu(a_j : \tau_j\,.\,(p_1 \parallel \ldots \parallel p_k))),$$

where $j, k \geq 0$. The transition is often by an action pertinent to the specific form of computation.

46.2 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()`
		$\mid$	`putc(`e`)`	`putc(`e`)`

The static semantics assumes that we have a type `char` of characters:

$$\frac{}{\Sigma\ \Gamma \vdash \texttt{getc()} \sim \texttt{char}} \tag{46.3a}$$

$$\frac{\Sigma\ \Gamma \vdash e : \texttt{char}}{\Sigma\ \Gamma \vdash \texttt{putc(}e\texttt{)} \sim \texttt{char}} \tag{46.3b}$$

Assuming that there are two ports, `in` and `out`, the dynamic semantics of character input/output may be given by the following rules:

$$\frac{}{\{a : \texttt{getc()}\} \xmapsto{\texttt{?in(}c\texttt{)}} \{a : \texttt{return(}c\texttt{)}\}} \tag{46.4a}$$

$$\frac{}{\{a : \texttt{putc(}c\texttt{)}\} \xmapsto{\texttt{!out(}c\texttt{)}} \{a : \texttt{return(}c\texttt{)}\}} \tag{46.4b}$$

(For convenience, we specify that `putc` returns the character that it sent to the output.)

46.3 Mutable Cells

We begin with the representation of mutable storage in $\mathcal{L}\{\texttt{conc}\}$ in which each reference cell is regarded as a concurrent process that enacts a protocol for manipulating its contents. Specifically, the process $\langle l : e \rangle$, where e is a value of some type τ, represents a mutable cell at *location* l with *contents* e of type τ. This process is prepared to send the value e along the channel named l, once again becoming the same process. It is also prepared to receive a value along channel l, which becomes the new contents of the reference cell with location l. Thus we may think of a reference cell as a "server" that emits the current contents of the cell, and that may respond to requests to change its contents.

To model reference cells as processes we extend the grammar of $\mathcal{L}\{\texttt{conc}\}$ to incorporate the machinery developed in Chapter 35 and to introduce a new form of process representing a reference cell:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{ref}(\tau)$	$\tau\,\texttt{ref}$
Expr	e	::=	$\texttt{loc}[l]$	l
Comm	m	::=	$\texttt{ref}(e)$	$\texttt{ref}(e)$
		|	$\texttt{get}(e)$	$\texttt{\^{}}\,e$
		|	$\texttt{set}(e_1;e_2)$	$e_1\,\texttt{<-}\,e_2$
Proc	p	::=	$\texttt{ref}[l](e)$	$\langle l : e\rangle$

The process $\langle l : e\rangle$ represents a mutable cell at location l with contents e, where e is a value.

The static semantics of reference cells is essentially as described in Chapter 37, transposed to the setting of $\mathcal{L}\{\texttt{conc}\}$. The typing rule for references is given as follows:

$$\frac{\Sigma, l : \tau\,\texttt{ref} \vdash e : \tau \quad \Sigma, l : \tau\,\texttt{ref} \vdash e\ \mathsf{val}}{\Sigma, l : \tau\,\texttt{ref} \vdash \langle l : e\rangle\ \mathsf{ok}} \tag{46.5}$$

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.

The dynamic semantics of mutable storage is specified in $\mathcal{L}\{\texttt{conc}\}$ by the following rules:[1]

$$\frac{e\ \mathsf{val}}{\{a : \texttt{ref}(e)\} \mapsto \nu(l : \tau\,\texttt{ref}.\{a : \texttt{return}(l)\} \parallel \langle l : e\rangle)} \tag{46.6a}$$

$$\frac{e\ \mathsf{val}}{\{a : \texttt{\^{}}\,l\} \xmapsto{?l(e)} \{a : \texttt{return}(e)\}} \tag{46.6b}$$

$$\frac{e\ \mathsf{val}}{\{a : l\,\texttt{<-}\,e\} \xmapsto{!l(e)} \{a : \texttt{return}(\langle\rangle)\}} \tag{46.6c}$$

$$\frac{e\ \mathsf{val}}{\langle l : e\rangle \xmapsto{!l(e)} \langle l : e\rangle} \tag{46.6d}$$

$$\frac{e\ \mathsf{val} \quad e'\ \mathsf{val}}{\langle l : e\rangle \xmapsto{?l(e')} \langle l : e'\rangle} \tag{46.6e}$$

[1] For the sake of concision we have omitted the evident rules for evaluation of the constituent expressions of the various forms of command.

Rule (46.6a) gives the semantics of `new`, which allocates a new location, l, which is returned to the calling process, and spawns a new process consisting of a reference cell at location l with contents e. Rule (46.6b) specifies that the execution of the process $\{a : \hat{} l\}$ consists of synchronizing with the reference cell at location l to obtain its contents, continuing with the value so obtained. Rule (46.6c) specifies that execution of $\{a : l \texttt{<-} e\}$ synchronizes with the reference cell at location l to specify its new contents, e'. Rules (46.6d) and (46.6e) specify that a reference cell process, $\langle l : e \rangle$, may interact with other processes via the location l, by either sending the contents, e, of l to a receiver without changing its state, or receiving its new contents, e', from a sender, and changing its contents accordingly.

It is instructive to reconsider the proof of type safety for reference cells given in Chapter 39 from the present point of view. Whereas in Chapter 39 the execution state for a command, m, has the form $m \,@\, \mu$, where μ is a memory mapping locations to values, here the execution state for m is a process that, up to strutural congruence, has the form

$$\nu(l_1 : \tau_1\,\texttt{ref} \ldots \nu(l_k : \tau_k\,\texttt{ref}\,.\,\langle l_1 : e_1\rangle \parallel \langle l_j : e_j\rangle \parallel \{a : m\})). \tag{46.7}$$

The memory has been decomposed into a set of active locations, $l_1, \ldots, l_k$, and a set of concurrent processes $\langle l_1 : e_1\rangle, \ldots, \langle l_j : e_j\rangle$ governing the active locations.

It will turn out to be an invariant of the dynamic semantics that each active location is governed by exactly one process, but the static semantics of processes given by Rules (46.1) are not sufficient to ensure it. (This is as it should be, because the stated property is special to the semantics of reference cells, and not a general property of all possible uses of the process calculus.) The static semantics is sufficient to ensure that if a process of the form (46.7) is well-formed, then for each $1 \leq i \leq j$,

$$l_1 : \tau_1\,\texttt{ref}, \ldots, l_k : \tau_k\,\texttt{ref} \vdash e_i : \tau_i.$$

As discussed in Chapter 35 this condition is necessary for type preservation, because memories may contain cyclic references.

The static semantics of processes is enough to ensure preservation; all that is required is that the contents of each location be type-consistent with its declared type. The static semantics is not, however, sufficient to ensure progress, for we may have fewer reference cell processes than declared locations, and hence the program may "get stuck" referring to the contents of a location, l, for which there is no process of the form $\langle l : e\rangle$ with which to interact. One prove that the following property is an invariant of the

dynamic semantics in the sense that if p satisfies this condition and is well-formed according to Rules (46.1), and $p \mapsto q$, then q also satisfies the same condition:

Lemma 46.1. *If $p \equiv \nu(l:\tau\ \mathtt{ref}.q)$, then $q \equiv \langle l : e\rangle \parallel q'$ for some q' and e.*

For the proof of progress, observe that by inversion of Rules (46.1) and (46.5), if p ok, where

$$p \equiv \nu(l_1:\tau\,\mathtt{ref}\ldots.\nu(l_k:\tau\,\mathtt{ref}.q \parallel \{a : m\})),$$

where l occurs in m, then

$$p \equiv \nu(l:\tau\,\mathtt{ref}.p')$$

for some p'. This, together with Lemma 46.1, ensures that we may make progress in the case that m has the form $\hat{}\,l$ or l `<-` e' for some e'.

46.4 Futures

The semantics of reference cells given in the preceding section makes use of concurrency to model mutable storage. By relaxing the restriction that the content of a cell be a value, we open up further possibilities for exploiting concurrency. In this section we model the concept of a *future*, a memoized, speculatively executed suspension, in the context of the concurrent language, $\mathcal{L}\{\mathtt{conc}\}$.

The syntax of futures is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`fut(`τ`)`	τ `fut`
Expr	e	::=	`loc[`l`]`	l
		|	`pid[`a`]`	a
Comm	m	::=	`fut(`e`)`	`fut(`e`)`
		|	`syn(`e`)`	`syn(`e`)`
Proc	p	::=	`fut[wait][`l`](`a`)`	`[`l `: wait(`a`)]`
		|	`fut[done][`l`](`e`)`	`[`l `: done(`e`)]`

Expressions are enriched to include *locations* of futures, and *process identifiers*, or *pid's*, for synchronization. The command `fut(`e`)` creates a cell whose value is determined by evaluating e concurrently with the caling process. The command `syn(`e`)` synchronizes with the future determined

by e, returning its value once it is available. A future is represented by a process that may be in one of two states, corresponding to whether the computation of its value is pending or finished. A future in the *wait* state has the form $\texttt{fut[wait]}\,[l]\,(a)$, indicating that the value of the future at location l will be determined by the result of executing the process with pid a. A future in the *done* state has the form $\texttt{fut[done]}\,[l]\,(e)$, indicating that the value of the future at location l is e.

The static semantics of futures consists of the evident typing rules for the commands $\texttt{fut}(e)$ and $\texttt{syn}(e)$, together with rules for the new forms of process:

$$\frac{\Sigma\,\Gamma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{fut}(e) \sim \tau\,\texttt{fut}} \tag{46.8a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \tau\,\texttt{fut}}{\Sigma\,\Gamma \vdash \texttt{syn}(e) \sim \tau} \tag{46.8b}$$

$$\frac{\Sigma\,\Gamma \vdash l : \tau\,\texttt{proc}}{\Sigma\,\Gamma \vdash l : \tau\,\texttt{fut}} \tag{46.8c}$$

$$\frac{\Sigma \vdash l : \tau\,\texttt{fut} \quad \Sigma \vdash a : \tau\,\texttt{proc}}{\Sigma \vdash [l : \texttt{wait}(a)]\ \texttt{ok}} \tag{46.8d}$$

$$\frac{\Sigma \vdash l : \tau\,\texttt{fut} \quad \Sigma \vdash e : \tau}{\Sigma \vdash [l : \texttt{done}(e)]\ \texttt{ok}} \tag{46.8e}$$

The dynamic semantics of futures is specified by the following rules:

$$\{a : \texttt{fut}(e)\} \mapsto \nu(l{:}\tau\,\texttt{fut}.\nu(b{:}\tau\,\texttt{proc}.\{a : \texttt{return}(l)\} \parallel [l : \texttt{wait}(b)] \parallel \{b : \texttt{return}(e)\})) \tag{46.9a}$$

$$\{a : \texttt{syn}(l)\} \xmapsto{?_l(e)} \{a : \texttt{return}(e)\} \tag{46.9b}$$

$$[l : \texttt{wait}(a)] \xmapsto{?_a(e)} [l : \texttt{done}(e)] \tag{46.9c}$$

$$[l : \texttt{done}(e)] \xmapsto{!_l(e)} [l : \texttt{done}(e)] \tag{46.9d}$$

Rule (46.9a) specifies that a future is created in the wait state pending termination of the process that evaluates its argument. Rule (46.9b) specifies

that we may only retrieve the value of a future once it has reached the done state. Rules (46.9c) and (46.9d) specify the behavior of futures. A future changes from the wait to the done state when the process that determines its contents has completed execution. Observe that Rule (46.9c) synchronizes with the process labelled b by waiting for that process to announce its termination with its returned value, as described by Rule (46.2b). A future in the done state repeatedly offers its contents to any process that may wish to synchronize with it.

46.5 Fork and Join

The semantics of futures given in Section 46.4 on page 401 may be seen as a combination of the more primitive concepts of *forking* a new process, synchronizing with, or *joining*, another process, creating a *reference cell* to hold the state of the future, and *sum types* to represent the state of the future (either waiting or done). In this section we will focus on the fork and join primitives that underly the semantics of futures.

The syntax of $\mathcal{L}\{\texttt{conc}\}$ is extended with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{proc}(\tau)$	$\tau\,\texttt{proc}$
Expr	e	::=	$\texttt{pid}[a]$	a
Comm	m	::=	$\texttt{fork}(m)$	$\texttt{fork}(m)$
		|	$\texttt{join}(e)$	$\texttt{join}(e)$

The static semantics is given by the following rules:

$$\frac{\Sigma\,\Gamma \vdash m \sim \tau}{\Sigma\,\Gamma \vdash \texttt{fork}(m) \sim \tau\,\texttt{proc}} \tag{46.10a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \tau\,\texttt{proc}}{\Sigma\,\Gamma \vdash \texttt{join}(e) \sim \tau} \tag{46.10b}$$

The dynamic semantics is given by the following rules:

$$\frac{}{\{a : \texttt{fork}(m)\} \mapsto \nu(b.\{a : \texttt{return}(b)\} \parallel \{b : m\})} \tag{46.11a}$$

$$\frac{}{\{a : \texttt{join}(b)\} \xmapsto{?_b(e)} \{a : \texttt{return}(e)\}} \tag{46.11b}$$

Rule (46.11a) creates a new process executing the given command, and returns the pid of the new process to the caling process. Rule (46.11b) synchronizes with the specified process, passing its return value to the caller when it has completed.

46.6 Synchronization

When programming with multiple processes it is necessary to take steps to ensure that they interact in a meaningful manner. For example, if two processes have access to a reference cell representing the current balance in a bank account, it is important to ensure that updates by either process are *atomic* in that they are not compromised by any action of the other process. Suppose that one process is recording accrued interest by increasing the balance by $r\,\%$, and the other is recording a debit of n dollars. Each proceeds by reading the current balance, performing a simple arithmetic computation, and storing the result back to record the result. However, we must ensure that each operation is performed in its entirety without interference from the other in order to preserve the semantics of the transactions. To see what can go wrong, suppose that both processes read the balance, b, then each calculate their own version of the new balance, $b_1 = b + r \times b$ and $b_2 = b - n$, and then both store their results in some order, say b_1 followed by b_2. The resulting balance, b_2, reflects the debit of n dollars, but not the interest accrual! If the stores occur in the opposite order, the new balance reflects the interest accrued, but not the debit. In either case the answer is wrong!

The solution is to ensure that a read-and-update operation is completed in its entirety without affecting or being affected by the actions of any other process. One way to achieve this is to use an *mvar*, which is a reference cell that may, at any time, either hold a value or be empty.[2] Thus an mvar may be in one of two states: *full* or *empty*, according to whether or not it holds a value. A process may *take* the value from a full mvar, thereby rendering it empty, or *put* a value into an empty mvar, thereby rendering it full with that value. No process may take a value from an empty mvar, nor may a process put a value to a full mvar. Any attempt to do so blocks progress until the state of the mvar has been changed by some other process so that it is once again possible to make progress. This simple primitive is sufficient to

[2]The name "mvar" is admittedly cryptic, but is relatively standard. Mvar's are also known as *mailboxes*, since their behavior is similar to that of a postal delivery box.

implement many higher-level constructs such as communication channels, as we shall see shortly.

The syntax of mvar's is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{mvar}(\tau)$	$\tau\,\texttt{mvar}$
Comm	m	::=	$\texttt{mvar}(e)$	$\texttt{mvar}(e)$
		|	$\texttt{take}(e)$	$\texttt{take}(e)$
		|	$\texttt{put}(e_1;e_2)$	$\texttt{put}(e_1;e_2)$
Proc	p	::=	$\texttt{mvar[full]}[l](e)$	$[l : \texttt{full}(e)]$
		|	$\texttt{mvar[empty]}(l)$	$[l : \texttt{empty}]$

The static semantics for commands is analogous to that for reference cells, and is omitted. The rules governing the two new forms of process are as follows:

$$\frac{\Sigma \vdash l : \tau\,\texttt{mvar} \quad \Sigma;\Gamma \vdash e : \tau}{\Sigma;\Gamma \vdash [l : \texttt{full}(e)]\ \mathsf{ok}} \tag{46.12a}$$

$$\frac{\Sigma \vdash l : \tau\,\texttt{mvar}}{\Sigma;\Gamma \vdash [l : \texttt{empty}]\ \mathsf{ok}} \tag{46.12b}$$

The dynamic semantics of mvars is given by the following transition rules:

$$\frac{e\ \mathsf{val}}{\{a : \texttt{mvar}(e)\} \mapsto \nu(l:\tau\,\texttt{mvar}.\{a : \texttt{return}(l)\} \parallel [l : \texttt{full}(e)])} \tag{46.13a}$$

$$\frac{}{\{a : \texttt{take}(l)\} \xrightarrow{?l(e)} \{a : \texttt{return}(e)\}} \tag{46.13b}$$

$$\frac{e\ \mathsf{val}}{\{a : \texttt{put}(l;e)\} \xrightarrow{!l(e)} \{a : \texttt{return}(e)\}} \tag{46.13c}$$

$$\frac{}{[l : \texttt{full}(e)] \xrightarrow{!l(e)} [l : \texttt{empty}]} \tag{46.13d}$$

$$\frac{}{[l : \texttt{empty}] \xrightarrow{?l(e)} [l : \texttt{full}(e)]} \tag{46.13e}$$

Rules (46.13d) and (46.13e) enforce the protocol ensuring that only one process at a time may access the contents of an mvar. If a full mvar synchronizes with a `take` (Rule (46.13b)), then its state changes to empty, precluding further reads of its value. Conversely, if an empty mvar synchronizes

with a `put` (Rule (46.13e)), then its state changes to full with the value specified by the `put`.

Using mvar's it is straightforward to implement communication channels over which processes may send and receive values of some specified type, τ. To be specific, a *channel* is just an mvar containing a queue of messages maintained in the order in which they were received. Sending a message on a channel adds (atomically!) a message to the back of the queue associated with that channel, and receiving a message from a channel removes (again, atomically) a message from the front of the queue. We leave a full development of channels as an instructive exercise for the reader.

46.7 Excercises

Part XVII

Modularity

Chapter 47

Separate Compilation and Linking

Chapter 48

Basic Modules

Chapter 49

Parameterized Modules

Part XVIII

Equivalence

Chapter 50

Equational Reasoning for T

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 $\mathcal{L}\{\texttt{nat} \to\}$ of Chapter 14 in which we can express addition as the function `plus`, the expressions

$$\lambda(x{:}\texttt{nat}.\,\lambda(y{:}\texttt{nat}.\,\texttt{plus}(x)(y)))$$

and

$$\lambda(x{:}\texttt{nat}.\,\lambda(y{:}\texttt{nat}.\,\texttt{plus}(y)(x)))$$

are equal. In other words, the addition function *as programmed in* $\mathcal{L}\{\texttt{nat} \to\}$ 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 sense? It is intuitively obvious that these two expressions are not *definitionally* equivalent, because no opportunity for symbolic execution is afforded by either of them. One may say that these two expressions are definitionally inequivalent because they describe different *algorithms*: one proceeds by recursion on x, the other by recursion on y. On the other hand, the two expressions are interchangeable in any complete computation of a natural number, because the only use we can make of them is to apply them to arguments and compute the result. We say that two functions are *extensionally equivalent* if they give equal results for equal arguments—in particular, they agree on all possible arguments. Since their behavior on arguments is all that matters for calculating observable results, we may expect that extensionally equivalent functions are equal in the sense of being interchangeable in all complete programs. Thinking of the programs in which these functions occur as *observations* of

their behavior, we say that the these functions are *observationally equivalent*. The main result of this chapter is that that observational and extensional equivalence coincide for $\mathcal{L}\{\texttt{nat} \to\}$.

50.1 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 $\mathcal{L}\{\texttt{nat} \to\}$ 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 numeral to which it evaluates.

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 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 $\mathcal{C} = \lambda(x{:}\mathtt{nat}.\,\circ)$, and $e = x + x$, then

$$\mathcal{C}\{e\} = \lambda(x{:}\mathtt{nat}.\,x + x).$$

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 $\mathcal{L}\{\mathtt{nat}\to\}$ is extended to expression contexts by defining the typing judgement

$$\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$$

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 $\mathcal{L}\{\mathtt{nat}\to\}$ (for which see Rules (14.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \tag{50.1a}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})}{\mathtt{s}(\mathcal{C}) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})} \tag{50.1b}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,\mathcal{C}\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{50.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau') \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,e\,\{\mathtt{z} \Rightarrow \mathcal{C}_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{50.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \rhd \tau')}{\mathtt{rec}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow \mathcal{C}_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{50.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \tau_1 \triangleright \tau_2)}{\lambda(x{:}\tau_1.\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1 \to \tau_2)} \tag{50.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{50.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{50.1h}$$

Lemma 50.1. *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 50.2. *If* $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, *and* $\mathcal{C}' : (\Gamma' \triangleright \tau') \rightsquigarrow (\Gamma'' \triangleright \tau'')$, *then* $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma'' \triangleright \tau'')$.

Lemma 50.3. *If* $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ *and* $x \# \Gamma$, *then* $\mathcal{C} : (\Gamma, x : \sigma \triangleright \tau) \rightsquigarrow (\Gamma', x : \sigma \triangleright \tau')$.

Proof. By induction on Rules (50.1). □

A *complete program* is a closed expression of type `nat`.

Definition 50.1. *We say that two complete programs,* e *and* e', *are* Kleene equivalent, *written* $e \simeq e'$, *iff there exists* $n \geq 0$ *such that* $e \mapsto^* \overline{n}$ *and* $e' \mapsto^* \overline{n}$.

Kleene equivalence is evidently reflexive and symmetric; transitivity follows from determinacy of evaluation. Closure under converse evaluation also follows directly from determinacy. It is obviously consistent in that $\overline{0} \not\simeq \overline{1}$.

Definition 50.2. *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* $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$.

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 family of equivalence relations $e_1 \mathrel{\mathcal{E}} e_2 : \tau\ [\Gamma]$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } e \mathrel{\mathcal{E}} e' : \tau\ [\Gamma], \text{ then } \mathcal{C}\{e\} \mathrel{\mathcal{E}} \mathcal{C}\{e'\} : \tau'\ [\Gamma']$$

for every expression context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is *consistent* iff $e \mathrel{\mathcal{E}} e' : \mathtt{nat}\ [\emptyset]$ implies $e \simeq e'$.

Theorem 50.4. *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 $e \mathrel{\mathcal{E}} e' : \tau\ [\Gamma]$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, then by congruence $\mathcal{C}\{e\} \mathrel{\mathcal{E}} \mathcal{C}\{e'\} : \mathtt{nat}\ [\emptyset]$, and hence by consistency $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$. □

A *substitution*, γ, for the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning expressions $e_1 : \tau_1, \ldots, e_n : \tau_n$ to $x_1, \ldots, x_n$, respectively. We write $\hat{\gamma}(e)$ for the substitution $[e_1, \ldots, e_n / x_1, \ldots, x_n]e$.

Lemma 50.5. *If $e \cong e' : \tau\ [\Gamma]$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$. Moreover, if $\gamma \cong \gamma' : \Gamma$, then $\hat{\gamma}(e) \cong \hat{\gamma}'(e) : \tau$ and $\hat{\gamma}(e') \cong \hat{\gamma}'(e') : \tau$.*

Proof. Let $\mathcal{C} : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$ be a program context; we are to show that $\mathcal{C}\{\hat{\gamma}(e)\} \simeq \mathcal{C}\{\hat{\gamma}(e')\}$. Since $\mathcal{C}$ has no free variables, this is equivalent to showing that $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Let $\mathcal{D}$ be the context

$$\lambda(x_1 : \tau_1. \ldots \lambda(x_n : \tau_n. \mathcal{C}\{\circ\}))(e_1) \ldots (e_n),$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $\gamma(x_1) = e_1, \ldots, \gamma(x_n) = e_n$. By Lemma 50.3 on the preceding page we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows directly that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$. Since $e \cong e' : \tau\ [\Gamma]$, we have $\mathcal{D}\{e\} \simeq \mathcal{D}\{e'\}$. But by construction $\mathcal{D}\{e\} \simeq \hat{\gamma}(\mathcal{C}\{e\})$, and $\mathcal{D}\{e'\} \simeq \hat{\gamma}(\mathcal{C}\{e'\})$, so $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Since $\mathcal{C}$ is arbitrary, it follows that $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$.

Defining $\mathcal{D}'$ similarly to $\mathcal{D}$, but based on γ', rather than γ, we may also show that $\mathcal{D}'\{e\} \simeq \mathcal{D}'\{e'\}$, and hence $\hat{\gamma}'(e) \cong \hat{\gamma}'(e') : \tau$. Now if $\gamma \cong \gamma' : \Gamma$, then by congruence we have $\mathcal{D}\{e\} \cong \mathcal{D}'\{e\} : \mathtt{nat}$, and $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\} : \mathtt{nat}$. It follows that $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\} : \mathtt{nat}$, and so, by consistency of observational equivalence, we have $\mathcal{D}\{e'\} \simeq \mathcal{D}'\{e'\}$, which is to say that $\hat{\gamma}(e) \cong \hat{\gamma}'(e') : \tau$. □

Theorem 50.4 licenses the principle of *proof by coinduction*: to show that $e \cong e' : \tau\ [\Gamma]$, it is enough to exhibit a consistent congruence, $\mathcal{E}$, such that $e \mathrel{\mathcal{E}} e' : \tau\ [\Gamma]$. It can be difficult to construct such a relation. In the next section we will provide a general method for doing so that exploits types.

50.2 Extensional 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 extensional equivalence alluded to in the introduction.

Definition 50.3. Extensional equivalence *is a 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:*

$$e \sim e' : \mathtt{nat} \quad \textit{iff} \quad e \simeq e' \mathtt{nat}$$

$$e \sim e' : \tau_1 \to \tau_2 \quad \textit{iff} \quad \textit{if } e_1 \sim e_1' : \tau_1, \textit{ then } e\,(e_1) \sim e'\,(e_1') : \tau_2$$

The definition of extensional equivalence at type `nat` licenses the following principle of *proof by* `nat`*-induction*. To show that $\mathcal{E}\ (e, e')$ whenever $e \sim e' : \mathtt{nat}$, it is enough to show that

1. $\mathcal{E}\ (\overline{0}, \overline{0})$, and
2. if $\mathcal{E}\ (\overline{n}, \overline{n})$, then $\mathcal{E}\ (\overline{n+1}, \overline{n+1})$.

This is, of course, justified by mathematical induction on $n \geq 0$, where $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$ by the definition of Kleene equivalence.

Extensional equivalence is extended to open terms by substitution of related closed terms to obtain related results. If γ and γ' are two substitutions for 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 $\hat{\gamma}(e) \sim \hat{\gamma'}(e') : \tau$ whenever $\gamma \sim \gamma' : \Gamma$.

50.3 Extensional and Observational Equivalence Coincide

In this section we prove the coincidence of observational and extensional equivalence.

Lemma 50.6 (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 from the closure of Kleene equivalence under converse evaluation. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \sim e' : \tau$, and $d \mapsto e$. To show that $d \sim e' : \tau$, we assume $e_1 \sim e_1' : \tau_1$ and show $d(e_1) \sim e'(e_1') : \tau_2$. It follows from the assumption that $e(e_1) \sim e'(e_1') : \tau_2$. Noting that $d(e_1) \mapsto e(e_1)$, the result follows by induction. □

Lemma 50.7 (Consistency)**.** *If $e \sim e' : \mathtt{nat}$, then $e \simeq e'$.*

Proof. By `nat`-induction (without appeal to the inductive hypothesis). If $e \mapsto^* \mathtt{z}$ and $e' \mapsto^* \mathtt{z}$, then $e \simeq e'$; if $e \mapsto^* \mathtt{s}(d)$ and $e' \mapsto^* \mathtt{s}(d')$ then $e \simeq e'$. □

Theorem 50.8 (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 (13.4a), in which $\tau = \tau_1 \rightharpoonup \tau_2$, $e = \lambda(x{:}\tau_1.\, e_2)$ and $e' = \lambda(x{:}\tau_1.\, e_2')$. Since e and e' are values, we are to show that

$$\lambda(x{:}\tau_1.\, \hat{\gamma}(e_2)) \sim \lambda(x{:}\tau_1.\, \widehat{\gamma'}(e_2')) : \tau_1 \rightharpoonup \tau_2.$$

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 (14.1d), for which we are to show that

$$\mathtt{rec}[\tau](\hat{\gamma}(e); \hat{\gamma}(e_0); x.y.\,\hat{\gamma}(e_1)) \sim \mathtt{rec}[\tau](\hat{\gamma'}(e'); \hat{\gamma}(e_0'); x.y.\,\hat{\gamma'}(e_1')) : \tau.$$

By the induction hypothesis applied to the first premise of Rule (14.1d), we have

$$\hat{\gamma}(e) \sim \hat{\gamma'}(e') : \mathtt{nat}.$$

We proceed by nat-induction. It suffices to show that

$$\texttt{rec[}\tau\texttt{]}(\texttt{z};\hat{\gamma}(e_0);x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim \texttt{rec[}\tau\texttt{]}(\texttt{z};\hat{\gamma}'(e_0');x\,.\,y\,.\,\hat{\gamma}'(e_1')) : \tau, \quad (50.2)$$

and that

$$\texttt{rec[}\tau\texttt{]}(\texttt{s}(\overline{n});\hat{\gamma}(e_0);x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim \texttt{rec[}\tau\texttt{]}(\texttt{s}(\overline{n});\hat{\gamma}'(e_0');x\,.\,y\,.\,\hat{\gamma}'(e_1')) : \tau, \quad (50.3)$$

assuming

$$\texttt{rec[}\tau\texttt{]}(\overline{n};\hat{\gamma}(e_0);x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim \texttt{rec[}\tau\texttt{]}(\overline{n};\hat{\gamma}'(e_0');x\,.\,y\,.\,\hat{\gamma}'(e_1')) : \tau. \quad (50.4)$$

To show (50.2), by Lemma 50.6 on the previous page it is enough to show that $\hat{\gamma}(e_0) \sim \hat{\gamma}(e_0') : \tau$. This is assured by the outer inductive hypothesis applied to the second premise of Rule (14.1d).

To show (50.3), define

$$\delta = \gamma[x \mapsto \overline{n}][y \mapsto \texttt{rec[}\tau\texttt{]}(\overline{n};\hat{\gamma}(e_0);x\,.\,y\,.\,\hat{\gamma}(e_1))]$$

and

$$\delta' = \gamma'[x \mapsto \overline{n}][y \mapsto \texttt{rec[}\tau\texttt{]}(\overline{n};\hat{\gamma}(e_0');x\,.\,y\,.\,\hat{\gamma}(e_1'))].$$

By (50.4) we have $\delta \sim \delta' : \Gamma, x : \texttt{nat}, y : \tau$. Consequently, by the outer inductive hypothesis applied to the third premise of Rule (14.1d), and Lemma 50.6 on the preceding page, the required follows. □

Corollary 50.9 (Termination)**.** *If $e : \tau$, then there exists e' val such that $e \mapsto^* e'$.*

Symmetry and transitivity of extensional equivalence are easily established by induction on types; extensional equivalence is therefore an equivalence relation.

Lemma 50.10 (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 $\mathcal{C}_0 = \lambda(x\!:\!\tau_1.\,\mathcal{C}_2)$ 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 $\gamma_0 \sim \gamma_0' : \Gamma_0$, 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 50.3 on page 422 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 50.11. *If $e \sim e' : \tau\ [\Gamma]$, then $e \cong e' : \tau\ [\Gamma]$.*

Proof. By Lemmas 50.7 on page 423 and 50.10 on the preceding page, and Theorem 50.4 on page 421. □

Corollary 50.12. *If $e : \mathtt{nat}$, then $e \cong \overline{n} : \mathtt{nat}$, for some $n \geq 0$.*

Proof. By Theorem 50.8 on page 423 we have $e \sim e : \tau$. Hence for some $n \geq 0$, we have $e \sim \overline{n} : \mathtt{nat}$, and so by Theorem 50.11, $e \cong \overline{n} : \mathtt{nat}$. □

Lemma 50.13. *For closed expressions $e : \tau$ and $e' : \tau$, if $e \cong e' : \tau$, then $e \sim e' : \tau$.*

Proof. We proceed by induction on the structure of τ. If $\tau = \mathtt{nat}$, consider the empty context to obtain $e \simeq e'$, and hence $e \sim e' : \mathtt{nat}$. If $\tau = \tau_1 \to \tau_2$, 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 50.11 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 50.14. *If $e \cong e' : \tau\ [\Gamma]$, then $e \sim e' : \tau\ [\Gamma]$.*

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$, and that $\gamma \sim \gamma' : \Gamma$. By Theorem 50.11 we have $\gamma \cong \gamma' : \Gamma$, so by Lemma 50.5 on page 421 $\hat{\gamma}(e) \cong \hat{\gamma'}(e') : \tau$. Therefore, by Lemma 50.13, $\hat{\gamma}(e) \sim \hat{\gamma}(e') : \tau$. □

Corollary 50.15. *$e \cong e' : \tau\ [\Gamma]$ iff $e \sim e' : \tau\ [\Gamma]$.*

Theorem 50.16. *If $\Gamma \vdash e \equiv e' : \tau$, then $e \sim e' : \tau\ [\Gamma]$, and hence $e \cong e' : \tau\ [\Gamma]$.*

Proof. By an argument similar to that used in the proof of Theorem 50.8 on page 423 and Lemma 50.10 on page 424, then appealing to Theorem 50.11 on the previous page. □

Corollary 50.17. *If $e \equiv e' : \mathtt{nat}$, then there exists $n \geq 0$ such that $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$.*

Proof. By Theorem 50.16 on the preceding page we have $e \sim e' : \mathtt{nat}$ and hence $e \simeq e'$. □

50.4 Some Laws of Equivalence

In this section we summarize some useful principles of observational equivalence for $\mathcal{L}\{\mathtt{nat} \to\}$. For the most part these may be proved as laws of extensional equivalence, and then transferred to observational equivalence by appeal to Corollary 50.15 on the previous page.

50.4.1 General Laws

Extensional equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{e \cong e : \tau\ [\Gamma]} \tag{50.5a}$$

$$\frac{e' \cong e : \tau\ [\Gamma]}{e \cong e' : \tau\ [\Gamma]} \tag{50.5b}$$

$$\frac{e \cong e' : \tau\ [\Gamma] \quad e' \cong e'' : \tau\ [\Gamma]}{e \cong e'' : \tau\ [\Gamma]} \tag{50.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences.

$$\frac{\Gamma \vdash e \equiv e' : \tau}{e \cong e' : \tau\ [\Gamma]} \tag{50.6a}$$

This is called the *principle of symbolic evaluation*.

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{e \cong e' : \tau\ [\Gamma] \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'\ [\Gamma']} \tag{50.7a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad e_2 \cong e_2' : \tau' \ [\Gamma, x : \tau]}{[e/x]e_2 \cong [e/x]e_2' : \tau' \ [\Gamma]} \tag{50.8a}$$

$$\frac{e_1 \cong e_1' : \tau \ [\Gamma] \quad e_2 \cong e_2' : \tau' \ [\Gamma, x : \tau]}{[e_1/x]e_2 \cong [e_1'/x]e_2' : \tau' \ [\Gamma]} \tag{50.8b}$$

50.4.2 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{e(x) \cong e'(x) : \tau_2 \ [\Gamma, x : \tau_1]}{e \cong e' : \tau_1 \to \tau_2 \ [\Gamma]} \tag{50.9}$$

Consequently, every expression of function type is equivalent to a λ-abstraction:

$$\frac{}{e \cong \lambda(x{:}\tau_1.\, e(x)) : \tau_1 \to \tau_2 \ [\Gamma]} \tag{50.10}$$

50.4.3 Induction Law

An equation involving a free variable, x, of type `nat` can be proved by induction on x.

$$\frac{[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \ [\Gamma] \ (\text{for every } n \in \mathbb{N})}{e \cong e' : \tau \ [\Gamma, x : \mathtt{nat}]} \tag{50.11a}$$

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 $[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \ [\Gamma]$.

50.5 Exercises

Chapter 51

Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The development proceeds long lines similar to those in Chapter 50, but is complicated by the presence of general recursion. The proof depends on the concept of an *admissible relation*, one that admits the principle of *proof by fixed point induction*.

51.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 50, but modified to account for the possibility of non-termination.

The collection of well-formed $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ contexts is inductively defined in a manner directly analogous to that in Chapter 50. Specifically, we define the judgement $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ by rules similar to Rules (50.1), modified for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. (We leave the precise definition as an exercise for the reader.) When Γ and Γ' are empty, we write just $\mathcal{C} : \tau \rightsquigarrow \tau'$.

A *complete program* is a closed expression of type `nat`.

Definition 51.1. *We say that two complete programs, e and e', are* Kleene equivalent, *written $e \simeq e'$, iff for every $n \geq 0$, $e \mapsto^* \overline{n}$ iff $e' \mapsto^* \overline{n}$.*

Kleene equivalence is easily seen to be an equivalence relation and to be closed under converse evaluation. Moreover, $\overline{0} \not\simeq \overline{1}$, and, if e and e' are both divergent, then $e \simeq e'$.

Observational equivalence is defined as in Chapter 50.

Definition 51.2. *We say that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau$ *are* observationally, *or* contextually, equivalent *iff for every program context* $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.

Theorem 51.1. *Observational equivalence is the coarsest consistent congruence.*

Proof. See the proof of Theorem 50.4 on page 421. □

Lemma 51.2 (Substitution and Functionality)**.** *If* $e \cong e' : \tau\ [\Gamma]$ *and* $\gamma : \Gamma$, *then* $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$. *Moreover, if* $\gamma \cong \gamma' : \Gamma$, *then* $\hat{\gamma}(e) \cong \hat{\gamma}'(e) : \tau$ *and* $\hat{\gamma}(e') \cong \hat{\gamma}'(e') : \tau$.

Proof. See Lemma 50.5 on page 421. □

51.2 Extensional Equivalence

Definition 51.3. Extensional equivalence, $e \sim e' : \tau$, *between closed expressions of type* τ *is defined by induction on* τ *as follows:*

$$e \sim e' : \mathtt{nat} \quad \textit{iff} \quad e \simeq e'$$

$$e \sim e' : \tau_1 \to \tau_2 \quad \textit{iff} \quad e_1 \sim e_1' : \tau_1 \textit{ implies } e\,(e_1) \sim e'\,(e_1') : \tau_2$$

Formally, extensional equivalence is defined as in Chapter 50, except that the definition of Kleene equivalence is altered to account for non-termination. Extensional equivalence is extended to open terms by substitution. Specifically, we define $e \sim e' : \tau\ [\Gamma]$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$ whenever $\gamma \sim \gamma' : \Gamma$.

Lemma 51.3 (Strictness)**.** *If* $e : \tau$ *and* $e' : \tau$ *are both divergent, then* $e \sim e' : \tau$.

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows immediately from the definition of Kleene equivalence. If $\tau = \tau_1 \to \tau_2$, then $e(e_1)$ and $e'(e_1')$ diverge, so by induction $e(e_1) \sim e'(e_1') : \tau_2$, as required. □

Lemma 51.4 (Converse Evaluation)**.** *Suppose that* $e \sim e' : \tau$. *If* $d \mapsto e$, *then* $d \sim e' : \tau$, *and if* $d' \mapsto e'$, *then* $e \sim d' : \tau$.

51.3 Extensional and Observational Equivalence Coincide

Theorem 51.5 (Fixed Point Induction)**.** *Suppose that* $x : \tau \vdash e : \tau$*. If*

$$(\forall m \geq 0)\ \mathit{fix}^m\, x{:}\tau\ \mathit{is}\, e \sim \mathit{fix}^m\, x{:}\tau\ \mathit{is}\, e' : \tau,$$

then $\mathit{fix} x{:}\tau\ \mathit{is}\, e \sim \mathit{fix} x{:}\tau\ \mathit{is}\, e' : \tau$*.*

Proof. Define an *applicative context*, $\mathcal{A}$, to be either a hole, $\circ$, or an application of the form $\mathcal{A}(e)$, where $\mathcal{A}$ is an applicative context. (The typing judgement $\mathcal{A} : \rho \rightsquigarrow \tau$ is a special case of the general typing judgment for contexts.) Define extensional equivalence of applicative contexts, written $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau$, by induction on the structure of $\mathcal{A}$ as follows:

1. $\circ \approx \circ : \rho \rightsquigarrow \rho$;
2. if $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau_2 \to \tau$ and $e_2 \sim e_2' : \tau_2$, then $\mathcal{A}(e_2) \approx \mathcal{A}'(e_2') : \rho \rightsquigarrow \tau$.

We prove by induction on the structure of τ, if $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau$ and

$$\text{for every } m \geq 0,\ \mathcal{A}\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e\} \sim \mathcal{A}'\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e'\} : \tau, \tag{51.1}$$

then

$$\mathcal{A}\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e\} \sim \mathcal{A}'\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e'\} : \tau. \tag{51.2}$$

Choosing $\mathcal{A} = \mathcal{A}' = \circ$ (so that $\rho = \tau$) completes the proof.

If $\tau = \mathtt{nat}$, then assume that $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \mathtt{nat}$ and (51.1). By Definition 51.3 on the facing page, we are to show

$$\mathcal{A}\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e\} \simeq \mathcal{A}'\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e'\}.$$

By Corollary 51.14 on page 437 there exists $m \geq 0$ such that

$$\mathcal{A}\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e\} \simeq \mathcal{A}\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e\}.$$

By (51.1) we have

$$\mathcal{A}\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e\} \simeq \mathcal{A}'\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e'\}.$$

By Corollary 51.14 on page 437

$$\mathcal{A}'\{\mathtt{fix}^m\, x{:}\rho\ \mathtt{is}\, e'\} \simeq \mathcal{A}'\{\mathtt{fix}\, x{:}\rho\ \mathtt{is}\, e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 51.3 on page 430, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\, x\mathtt{:}\rho\, \mathtt{is}\, e\}(e_1) \sim \mathcal{A}'\{\mathtt{fix}\, x\mathtt{:}\rho\, \mathtt{is}\, e'\}(e_1') : \tau_2$$

whenever $e_1 \sim e_1' : \tau_1$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (51.1) that for every $m \geq 0$

$$\mathcal{A}_2\{\mathtt{fix}^m\, x\mathtt{:}\rho\, \mathtt{is}\, e\} \sim \mathcal{A}_2'\{\mathtt{fix}^m\, x\mathtt{:}\rho\, \mathtt{is}\, e'\} : \tau_2.$$

Noting that $\mathcal{A}_2 \approx \mathcal{A}_2' : \rho \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\, x\mathtt{:}\rho\, \mathtt{is}\, e\} \sim \mathcal{A}_2'\{\mathtt{fix}\, x\mathtt{:}\rho\, \mathtt{is}\, e'\} : \tau_2,$$

as required. □

Lemma 51.6 (Reflexivity)**.** *If* $\Gamma \vdash e : \tau$, *then* $e \sim e : \tau\ [\Gamma]$.

Proof. The proof proceeds along the same lines as the proof of Theorem 50.8 on page 423. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider Rule (15.1g). Assuming $\gamma \sim \gamma' : \Gamma$, we are to show that

$$\mathtt{fix}\, x\mathtt{:}\tau\, \mathtt{is}\, \hat{\gamma}(e) \sim \mathtt{fix}\, x\mathtt{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e') : \tau.$$

By Theorem 51.5 on the previous page it is enough to show that, for every $m \geq 0$,

$$\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \hat{\gamma}(e) \sim \mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e') : \tau.$$

We proceed by an inner induction on m. When $m = 0$ the result is immediate, since both sides of the desired equivalence diverge. Assuming the result for m, and applying Lemma 51.4 on page 430, it is enough to show that $\hat{\gamma}(e_1) \sim \widehat{\gamma'}(e_1') : \tau$, where

$$e_1 = [\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{51.3}$$

$$e_1' = [\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e')/x]\widehat{\gamma'}(e'). \tag{51.4}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e) \sim \tau : , [\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \hat{\gamma}(e)]$$

then

$$[\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e) \sim \tau : . [[\mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e)]$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows. □

Symmetry and transitivity of eager extensional equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager extensional equivalence is therefore an equivalence relation.

Lemma 51.7 (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 $\mathcal{C}_0$, following along similar lines to the proof of Lemma 51.6 on the preceding page. □

Theorem 51.8. *If* $e \sim e' : \tau\ [\Gamma]$*, then* $e \cong e' : \tau\ [\Gamma]$.

Proof. By consistency and congruence of extensional equivalence. □

Lemma 51.9. *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 empty expression context is a program context. If $\tau = \tau_1 \to \tau_2$, then suppose that $e_1 \sim e_1' : \tau_1$. We are to show that $e(e_1) \sim e'(e_1') : \tau_2$. By Theorem 51.8 $e_1 \cong e_1' : \tau_1$, and hence by Lemma 51.2 on page 430 $e(e_1) \cong e'(e_1') : \tau_2$, from which the result follows by induction. □

Theorem 51.10. *If* $e \cong e' : \tau\ [\Gamma]$*, then* $e \sim e' : \tau\ [\Gamma]$.

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$. Suppose that $\gamma \sim \gamma' : \Gamma$. By Theorem 51.8 we have $\gamma \cong \gamma' : \Gamma$, and so by Lemma 51.2 on page 430 we have

$$\hat{\gamma}(e) \cong \hat{\gamma}'(e') : \tau.$$

Therefore by Lemma 51.9 we have

$$\hat{\gamma}(e) \sim \hat{\gamma}'(e') : \tau.$$

□

Corollary 51.11. $e \cong e' : \tau\ [\Gamma]$ *iff* $e \sim e' : \tau\ [\Gamma]$.

51.4 Compactness

The principle of fixed point induction is derived from a critical property of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, called *compactness*. This property states that only finitely many unwindings of a fixed point expression are needed in a complete 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.

As a technical convenience, we enrich $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with *bounded recursion*, with abstract syntax $\texttt{fix}^m\,[\tau]\,(x.e)$ and concrete syntax $\texttt{fix}^m\,x\texttt{:}\tau\,\texttt{is}\,e$, where $m \geq 0$. The static semantics of bounded recursion is the same as for general recursion:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}^m\,[\tau]\,(x.e) : \tau}\ . \tag{51.5a}$$

The dynamic semantics of bounded recursion is defined as follows:

$$\overline{\texttt{fix}^0\,[\tau]\,(x.e) \mapsto \texttt{fix}^0\,[\tau]\,(x.e)} \tag{51.6a}$$

$$\overline{\texttt{fix}^{m+1}\,[\tau]\,(x.e) \mapsto [\texttt{fix}^m\,[\tau]\,(x.e)\,/\,x]e} \tag{51.6b}$$

If m is positive, the recursive bound is decremented so that subsequent uses of it will be limited to one fewer unrolling. If m reaches zero, the expression steps to itself so that the computation diverges with no result.

The proof of compactness (Theorem 51.13 on the next page) makes use of the stack machine for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 27, augmented with the following transitions for bounded recursive expressions:

$$\overline{k \triangleright \texttt{fix}^0\,x\texttt{:}\tau\,\texttt{is}\,e \mapsto k \triangleright \texttt{fix}^0\,x\texttt{:}\tau\,\texttt{is}\,e} \tag{51.7a}$$

$$\overline{k \triangleright \texttt{fix}^{m+1}\,x\texttt{:}\tau\,\texttt{is}\,e \mapsto k \triangleright \texttt{fix}^m\,x\texttt{:}\tau\,\texttt{is}\,e} \tag{51.7b}$$

It is straightforward to extend the proof of correctness of the stack machine (Corollary 27.4 on page 230) to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function, f, in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$:

$$\texttt{fix}\,f\texttt{:nat} \rightharpoonup \texttt{nat}\,\texttt{is}\,\lambda(x\texttt{:nat}.\,\texttt{ifz}\,x\,\{\texttt{z} \Rightarrow \texttt{s(z)} \mid \texttt{s}(x') \Rightarrow x * f(x')\}).$$

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*, m, on the

recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the m-bounded form of general recursion,

$$\mathtt{fix}^m\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\,\mathtt{is}\,\lambda(x\mathtt{:nat}.\,\mathtt{ifz}\,x\,\{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}).$$

Call this expression $f^{(m)}$. It follows from the definition of f that if $f(\overline{n}) \mapsto^* \overline{p}$, then $f^{(m)}(\overline{n}) \mapsto^* \overline{p}$ for some $m \geq 0$ (in fact, $m = n$ suffices).

When considering expressions of higher type, we cannot expect to get the *same* result from the bounded recursion as from the unbounded. For example, consider the addition function, a, of type $\tau = \mathtt{nat} \rightharpoonup (\mathtt{nat} \rightharpoonup \mathtt{nat})$, given by the expression

$$\mathtt{fix}\,p\mathtt{:}\tau\,\mathtt{is}\,\lambda(x\mathtt{:nat}.\,\mathtt{ifz}\,x\,\{\mathtt{z} \Rightarrow \mathit{id} \mid \mathtt{s}(x') \Rightarrow s \circ (p(x'))\}),$$

where $\mathit{id} = \lambda(y\mathtt{:nat}.\,y)$ is the identity, $e' \circ e = \lambda(x\mathtt{:}\tau.\,e'(e(x)))$ is composition, and $s = \lambda(x\mathtt{:nat}.\,\mathtt{s}(x))$ is the successor function. The application $a(\overline{n})$ terminates after three transitions, regardless of the value of n, resulting in a λ-abstraction. When n is positive, the result contains a *residual* copy of a itself, which is applied to $n-1$ as a recursive call. The m-bounded version of a, written $a^{(m)}$, is also such that $a^{(m)}()$ terminates in three steps, provided that $m > 0$. But the result is not the same, because the residuals of a appear as $a^{(m-1)}$, rather than as a itself.

Turning now to the proof, it is helpful to introduce some notation. Suppose that $x : \tau \vdash e_x : \tau$ for some arbitrary abstractor $x.e_x$. Define $f^{(\omega)} = \mathtt{fix}\,x\mathtt{:}\tau\,\mathtt{is}\,e_x$, and $f^{(m)} = \mathtt{fix}^m\,x\mathtt{:}\tau\,\mathtt{is}\,e_x$, and observe that $f^{(\omega)} : \tau$ and $f^{(m)} : \tau$ for any $m \geq 0$.

The following technical lemma governing the stack machine permits the bound on "passive" occurrences of a recursive expression to be raised without affecting the outcome of evaluation.

Lemma 51.12. *If* $[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$, *where* $e \neq y$, *then* $[f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$.

Proof. By induction on the definition of the transition judgement for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$. □

Theorem 51.13 (Compactness)**.** *Suppose that* $y : \tau \vdash e : \mathtt{nat}$ *where* $y \# f^{(\omega)}$. *If* $[f^{(\omega)}/y]e \mapsto^* \overline{n}$, *then there exists* $m \geq 0$ *such that* $[f^{(m)}/y]e \mapsto^* \overline{n}$.

Proof. We prove simultaneously the stronger statements that if

$$[f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \rhd [f^{(m)}/y]e \mapsto^* \epsilon \lhd \overline{n},$$

and

$$[f^{(\omega)}/y]k \lhd [f^{(\omega)}/y]e \mapsto^* \epsilon \lhd \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \lhd [f^{(m)}/y]e \mapsto^* \epsilon \lhd \overline{n}.$$

(Note that if $[f^{(\omega)}/y]e$ val, then $[f^{(m)}/y]e$ val for all $m \geq 0$.) The result then follows by the correctness of the stack machine (Corollary 27.4 on page 230).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \rhd f^{(\omega)},$$

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \rhd f^{(\omega)} \mapsto [f^{(\omega)}/y]k \rhd [f^{(\omega)}/x]e_x \mapsto^* \epsilon \lhd \overline{n}.$$

Noting that $[f^{(\omega)}/x]e_x = [f^{(\omega)}/y][y/x]e_x$, we have by induction that there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \rhd [f^{(m)}/x]e_x \mapsto^* \epsilon \lhd \overline{n}.$$

By Lemma 51.12 on the preceding page

$$[f^{(m+1)}/y]k \rhd [f^{(m)}/x]e_x \mapsto^* \epsilon \lhd \overline{n}$$

and we need only observe that

$$[f^{(m+1)}/y]k \rhd f^{(m+1)} \mapsto [f^{(m+1)}/y]k \rhd [f^{(m)}/x]e_x$$

to complete the proof. If, on the other hand, the initial step is an unrolling, but $e \neq y$, then we have for some $z \# f^{(\omega)}$ and $z \neq y$

$$[f^{(\omega)}/y]k \rhd \mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_\omega \mapsto [f^{(\omega)}/y]k \rhd [\mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_\omega/z]d_\omega \mapsto^* \epsilon \lhd \overline{n}.$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \rhd [\mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_m/z]d_m \mapsto^* \epsilon \lhd \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 51.12 on the previous page we have

$$[f^{(m+1)}/y]k \rhd [\mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_{m+1}/z]d_{m+1} \mapsto^* \epsilon \lhd \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

Corollary 51.14. *There exists $m \geq 0$ such that $[f^{(\omega)}/y]e \simeq [f^{(m)}/y]e$.*

Proof. If $[f^{(\omega)}/y]e$ diverges, then taking m to be zero suffices. Otherwise, apply Theorem 51.13 on page 435 to obtain m, and note that the required Kleene equivalence follows. □

51.5 Lazy Natural Numbers

In Chapter 15 we considered a variation of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the lazy natural numbers, `lnat`, as base type. This is achieved by specifying that $\texttt{s}(e)$ `val` regardless of the form of e, so that the successor does not evaluate its argument. Using general recursion we may define the infinite number, ω, by $\texttt{fix}\, x\texttt{:lnat is s}(x)$, which consists of an infinite stack of successors. Since the successor is intepreted lazily, ω evaluates to a value, namely $\texttt{s}(\omega)$, its own successor. It follows that the principle of mathematical induction is not valid for the lazy natural numbers. For example, the property of being equivalent to a finite numeral is satisfied by zero and is closed under successor, but fails for ω.

In this section we sketch the modifications to the preceding development for the lazy natural numbers. The main difference is that the definition of extensional equivalence at type `lnat` must be formulated to account for laziness. Rather than being defined *inductively* as the strongest relation closed under specified conditions, we define it *coinductively* as the weakest relation consistent two analogous conditions. We may then show that two expressions are related using the principle of *proof by coinduction*.

If `lnat` is to continue to serve as the observable outcome of a computation, then we must alter the meaning of Kleene equivalence to account for laziness. We adopt the principle that we may observe of a computation only its outermost form: it is either zero or the successor of some other computation. More precisely, we define $e \simeq e'$ iff (a) if $e \mapsto^* \texttt{z}$, then $e' \mapsto^* \texttt{z}$, and *vice versa*; and (b) if $e \mapsto^* \texttt{s}(e_1)$, then $e' \mapsto^* \texttt{s}(e_1')$, and *vice versa*. Note well that we do not require anything of e_1 and e_1' in the second clause. This means that $\overline{1} \simeq \overline{2}$, yet we retain consistency in that $\overline{0} \not\simeq \overline{1}$.

Corollary 51.14 can be proved for the lazy natural numbers by essentially the same argument.

The definition of extensional equivalence at type `lnat` is defined to be the *weakest* equivalence relation, $\mathcal{E}$, between closed terms of type `lnat` satisfying the following *`lnat`-consistency conditions*: if $e\ \mathcal{E}\ e' : \texttt{lnat}$, then

1. If $e \mapsto^* \texttt{z}$, then $e' \mapsto^* \texttt{z}$, and *vice versa*.

2. If $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$ with $e_1 \mathrel{\mathcal{E}} e_1' : \mathtt{lnat}$, and *vice versa*.

It is immediate that if $e \sim e' : \mathtt{lnat}$, then $e \simeq e'$, and so extensional equivalence is consistent. It is also strict in that if e and e' are both divergent expressions of type `lnat`, then $e \sim e' : \mathtt{lnat}$—simply because the preceding two conditions are vacuously true in this case.

This is an example of the more general principle of *proof by* `lnat`*-coinduction*. To show that $e \sim e' : \mathtt{lnat}$, it suffices to exhibit a relation, $\mathcal{E}$, such that

1. $e \mathrel{\mathcal{E}} e' : \mathtt{lnat}$, and

2. $\mathcal{E}$ satisfies the `lnat`-consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in extensional equivalence at type `lnat`, and hence $e \sim e' : \mathtt{lnat}$, as required.

As an application of `lnat`-coinduction, let us consider the proof of Theorem 51.5 on page 431. The overall argument remains as before, but the proof for the type `lnat` must be altered as follows. Suppose that $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \mathtt{lnat}$, and let $a = \mathcal{A}\{\mathtt{fix}\, x\!:\!\rho\, \mathtt{is}\, e\}$ and $a' = \mathcal{A}'\{\mathtt{fix}\, x\!:\!\rho\, \mathtt{is}\, e'\}$. Writing $a^{(m)} = \mathcal{A}\{\mathtt{fix}^m\, x\!:\!\rho\, \mathtt{is}\, e\}$ and $a'^{(m)} = \mathcal{A}'\{\mathtt{fix}^m\, x\!:\!\rho\, \mathtt{is}\, e'\}$, assume that

$$\text{for every } m \geq 0,\ a^{(m)} \sim a'^{(m)} : \mathtt{lnat}.$$

We are to show that

$$a \sim a' : \mathtt{lnat}.$$

Define the functions p_n for $n \geq 0$ on closed terms of type `lnat` by the following equations:

$$p_0(d) = d$$
$$p_{(n+1)}(d) = \begin{cases} d' & \text{if } p_n(d) \mapsto^* \mathtt{s}(d') \\ \textit{undefined} & \text{otherwise} \end{cases}$$

For $n \geq 0$, let $a_n = p_n(a)$ and $a_n' = p_n(a')$. Correspondingly, let $a_n^{(m)} = p_n(a^{(m)})$ and $a_n'^{(m)} = p_n(a_n^{(m)})$. Define $\mathcal{E}$ to be the strongest relation such that $a_n \mathrel{\mathcal{E}} a_n' : \mathtt{lnat}$ for all $n \geq 0$. We will show that the relation $\mathcal{E}$ satisfies the `lnat`-consistency conditions, and so it is contained in extensional equivalence. Since $a \mathrel{\mathcal{E}} a' : \mathtt{lnat}$ (by construction), the result follows immediately.

To show that $\mathcal{E}$ is `lnat`-consistent, suppose that $a_n \mathrel{\mathcal{E}} a_n' : \mathtt{lnat}$ for some $n \geq 0$. We have by Corollary 51.14 on the previous page $a_n \simeq a_n^{(m)}$, for some

$m \geq 0$, and hence, by the assumption, $a_n \simeq {a'_n}^{(m)}$, and so by Corollary 51.14 on page 437 again, ${a'_n}^{(m)} \simeq a'_n$. Now if $a_n \mapsto^* \mathtt{s}(b_n)$, then $a_n^{(m)} \mapsto^* \mathtt{s}(b_n^{(m)})$ for some $b_n^{(m)}$, and hence there exists ${b'_n}^{(m)}$ such that ${a'_n}^{(m)} \mapsto^* {b'_n}^{(m)}$, and so there exists b'_n such that $a'_n \mapsto^* \mathtt{s}(b'_n)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n \mathrel{\mathcal{E}} b'_n : \mathtt{lnat}$ by construction, as required.

51.6 Exercises

1. Call-by-value variant, with recursive functions.

Chapter 52

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 23. 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 $\mathcal{L}\{\to\forall\}$ under a call-by-name interpretation.

52.1 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 $\forall(t.t \to t)$. What function could it be? When instantiated at a type τ it should evaluate to a function g of type $\tau \to \tau$ 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 23, 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.

52.2 Observational Equivalence

The definition of observational equivalence given in Chapters 50 and 51 is based on identifying a type of *answers* that are observable outcomes of complete programs. Values of function type are not regarded as answers, but are treated as "black boxes" with no internal structure, only input-output behavior. In $\mathcal{L}\{\to\forall\}$, however, there are no (closed) base types! Every type is either a function type or a polymorphic type, and hence no types suitable to serve as observable answers.

One way to manage this difficulty is to augment $\mathcal{L}\{\to\forall\}$ with a base type of answers to serve as the observable outcomes of a computation. The only requirement is that this type have two elements that can be immediately distinguished from each other by evaluation. We may achieve this by enriching $\mathcal{L}\{\to\forall\}$ with a base type, **2**, containing two constants, **tt** and **ff**, that serve as possible answers for a complete computation. A complete program is a closed expression of type **2**.

Kleene equivalence is defined for complete programs by requiring that $e \simeq e'$ iff either (a) $e \mapsto^* \mathbf{tt}$ and $e' \mapsto^* \mathbf{tt}$; or (b) $e \mapsto^* \mathbf{ff}$ and $e' \mapsto^* \mathbf{ff}$.

This is obviously an equivalence relation, and it is immediate that $\mathbf{tt} \not\cong \mathbf{ff}$, since these are two distinct constants. As before, we say that a type-indexed family of equivalence relations between closed expressions of the same type is *consistent* if it implies Kleene equivalence at the answer type, $\mathbf{2}$.

To define observational equivalence, we must first define the concept of an expression context for $\mathcal{L}\{\to\forall\}$ as an expression with a "hole" in it. More precisely, we may give an inductive definition of the judgement

$$\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau'),$$

which states that $\mathcal{C}$ is an expression context that, when filled with an expression $\Delta; \Gamma \vdash e : \tau$ yields an expression $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} : \tau$. (We leave the precise definition of this judgement, and the verification of its properties, as an exercise for the reader.)

Definition 52.1. *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* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$.

Lemma 52.1. *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 52.2.

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 $\mathcal{C} : (\Delta; [\rho/t]\Gamma \triangleright [\rho/t]\tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\rho/t]e\} \simeq \mathcal{C}\{[\rho/t]e'\}.$$

Since $\mathcal{C}$ is 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

$$\mathcal{C}' : (\Delta, t; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2}).$$

Therefore, from the assumption,

$$\mathcal{C}'\{e\} \simeq \mathcal{C}'\{e'\}.$$

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 50.5 on page 421. □

52.3 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 50 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 52.2 (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 substitution* that assigns a closed type to each type variable $t \in \Delta$. A type substitution, 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 $x \in dom(\Gamma)$.

Let δ and δ' be two type substitutions of closed types to the type variables in Δ. A *relation assignment*, η, between δ and δ' is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ to 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 $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$, defined as follows.

Definition 52.3 (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:*

$$
\begin{array}{lll}
e \sim e' : t\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e, e') \\
e \sim e' : \mathbf{2}\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e \simeq e' \\
e \sim e' : \tau_1 \to \tau_2\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim e_1' : \tau_1\ [\eta : \delta \leftrightarrow \delta'] \textit{ implies} \\
 & & e\,(e_1) \sim e'\,(e_1') : \tau_2\ [\eta : \delta \leftrightarrow \delta'] \\
e \sim e' : \forall\,(t\,.\,\tau)\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every } \rho,\ \rho',\ \textit{and every } R : \rho \leftrightarrow \rho', \\
 & & e\,[\rho] \sim e'\,[\rho']\ : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']]
\end{array}
$$

Logical equivalence is defined in terms of parameterized logical equivalence by considering all possible interpretations of its free type- and expression variables. An *expression substitution*, γ, for a context Γ, written $\gamma : \Gamma$, is an substitution of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in \mathit{dom}(\Gamma)$. An expression substitution, $\gamma : \Gamma$, 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 52.4 (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 52.3 (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 by the definition of admissibility. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications. □

Lemma 52.4 (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 52.5. *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 52.3 on the facing page and 52.4 on the preceding page. □

Logical Equivalence respects observational equivalence.

Corollary 52.6. *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 52.2 on page 443 and Corollary 52.5. □

Lemma 52.7 (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 $\tau = \tau_1 \to \tau_2$ 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 overly restrictive—every expression satisfies its constraints. This result is sometimes called the *parametricity theorem*.

Theorem 52.8 (Parametricity)**.** *If $\Delta;\Gamma \vdash e : \tau$, then $e \sim e : \tau\ [\Delta;\Gamma]$.*

Proof. By rule induction on the static semantics of $\mathcal{L}\{\to\forall\}$ given by Rules (23.2). We consider two representative cases here.

Rule (23.2d) Suppose $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$. By induction we have that for 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 52.3 on page 446.

Rule (23.2e) Suppose $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$. By induction we have

$$\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e)) : \forall(t.\tau)\ [\eta : \delta \leftrightarrow \delta']$$

Let $\hat{\rho} = \hat{\delta}(\rho)$ and $\hat{\rho'} = \widehat{\delta'}(\rho)$. Define the relation $R : \hat{\rho} \leftrightarrow \hat{\rho'}$ by $R(d, d')$ iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. By Corollary 52.5 on the previous page, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \hat{\rho'}]].$$

By Lemma 52.7 on the preceding page

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))\,[\hat{\delta}(\rho)] && (52.1)\\ &= \hat{\gamma}(\hat{\delta}(e\,[\rho])), && (52.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] &= \widehat{\gamma'}(\widehat{\delta'}(e))\,[\widehat{\delta'}(\rho)] && (52.3)\\ &= \widehat{\gamma'}(\widehat{\delta'}(e\,[\rho])), && (52.4) \end{aligned}$$

from which the result follows.

□

Corollary 52.9. *If $e \cong e' : \tau\ [\Delta;\Gamma]$, then $e \sim e' : \tau\ [\Delta;\Gamma]$.*

Proof. By Theorem 52.8 on page 447 $e \sim e : \tau\ [\Delta;\Gamma]$, and hence by Corollary 52.6 on page 447, $e \sim e' : \tau\ [\Delta;\Gamma]$. □

Lemma 52.10 (Congruence)**.** *If $e \sim e' : \tau\ [\Delta;\Gamma]$ 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 $\mathcal{C}$, following along very similar lines to the proof of Theorem 52.8 on page 447. □

Lemma 52.11 (Consistency)**.** *Logical equivalence is consistent.*

Proof. Follows immediately from the definition of logical equivalence. □

Corollary 52.12. *If $e \sim e' : \tau\ [\Delta;\Gamma]$, then $e \cong e' : \tau\ [\Delta;\Gamma]$.*

Proof. By Lemma 52.11 Logical equivalence is consistent, and by Lemma 52.10, it is a congruence, and hence is contained in observational equivalence. □

Corollary 52.13. *Logical and observational equivalence coincide.*

Proof. By Corollaries 52.9 and 52.12. □

If $d : \tau$ and $d \mapsto e$, then $d \sim e : \tau$, and hence by Corollary 52.12, $d \cong e : \tau$. 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.

52.4 Parametricity Properties

The parametricity theorem enables us to deduce properties of expressions of $\mathcal{L}\{\rightarrow\forall\}$ that hold solely because of their type. The stringencies of parametricity ensure that a polymorphic type has very few inhabitants. For example, we may prove that *every* expression of type $\forall(t.t \rightarrow t)$ behaves like the identity function.

Theorem 52.14. *Let $e : \forall(t.t \rightarrow t)$ be arbitrary, and let id be $\Lambda(t.\lambda(x:t.x))$. Then $e \cong id : \forall(t.t \rightarrow t)$.*

Proof. By Corollary 52.13 on the previous page it is sufficient to show that $e \sim id : \forall(t.t \to t)$. Let ρ and ρ' be arbitrary closed types, let $R : \rho \leftrightarrow \rho'$ be an admissible relation, and suppose that $e_0\ R\ e_0'$. We are to show

$$e[\rho](e_0)\ R\ id[\rho](e_0'),$$

which, given the definition of *id*, is to say

$$e[\rho](e_0)\ R\ e_0'.$$

It suffices to show that $e[\rho](e_0) \cong e_0 : \rho$, for then the result follows by the admissibility of R and the assumption $e_0\ R\ e_0'$.

By Theorem 52.8 on page 447 we have $e \sim e : \forall(t.t \to t)$. Let the relation $S : \rho \leftrightarrow \rho$ be defined by $d\ S\ d'$ iff $d \cong e_0 : \rho$ and $d' \cong e_0 : \rho$. This is clearly admissible, and we have $e_0\ S\ e_0$. It follows that

$$e[\rho](e_0)\ S\ e[\rho](e_0),$$

and so, by the definition of the relation S, $e[\rho](e_0) \cong e_0 : \rho$.

□

In Chapter 23 we showed that product, sum, and natural numbers types are all definable in $\mathcal{L}\{\to\forall\}$. The proof of definability in each case consisted of showing that the type and its associated introduction and elimination forms are encodable in $\mathcal{L}\{\to\forall\}$. The encodings are correct in the (weak) sense that the dynamic semantics of these constructs as given in the earlier chapters is derivable from the dynamic semantics of $\mathcal{L}\{\to\forall\}$ via these definitions. By taking advantage of parametricity we may extend these results to obtain a strong correspondence between these types and their encodings.

As a first example, let us consider the representation of the unit type, `unit`, in $\mathcal{L}\{\to\forall\}$, as defined in Chapter 23 by the following equations:

$$\begin{aligned}\texttt{unit} &= \forall(r.r \to r)\\ \langle\rangle &= \Lambda(r.\lambda(x{:}r.x))\end{aligned}$$

It is easy to see that $\langle\rangle : \texttt{unit}$ according to these definitions. But this merely says that the type `unit` is inhabited (has an element). What we would like to know is that, up to observational equivalence, the expression $\langle\rangle$ is the *only* element of that type. But this is precisely the content of Theorem 52.14 on the previous page! We say that the type `unit` is *strongly definable* within $\mathcal{L}\{\to\forall\}$.

Continuing in this vein, let us examine the definition of the binary product type in $\mathcal{L}\{\to\forall\}$, also given in Chapter 23:

$$\begin{aligned}
\tau_1 \times \tau_2 &= \forall(r.(\tau_1 \to \tau_2 \to r) \to r)\\
\langle e_1, e_2\rangle &= \Lambda(r.\lambda(x{:}\tau_1 \to \tau_2 \to r.\, x(e_1)(e_2)))\\
\mathtt{fst}(e) &= e[\tau_1](\lambda(x{:}\tau_1.\lambda(y{:}\tau_2.\, x)))\\
\mathtt{snd}(e) &= e[\tau_2](\lambda(x{:}\tau_1.\lambda(y{:}\tau_2.\, y)))
\end{aligned}$$

It is easy to check that $\mathtt{fst}(\langle e_1, e_2\rangle) \cong e_1 : \tau_1$ and $\mathtt{snd}(\langle e_1, e_2\rangle) \cong e_2 : \tau_2$ by a direct calculation.

We wish to show that the ordered pair, as defined above, is the unique such expression, and hence that Cartesian products are strongly definable in $\mathcal{L}\{\to\forall\}$. We will make use of a lemma governing the behavior of the elements of the product type whose proof relies on Theorem 52.8 on page 447.

Lemma 52.15. *If $e : \tau_1 \times \tau_2$, then $e \cong \langle e_1, e_2\rangle : \tau_1 \times \tau_2$ for some $e_1 : \tau_1$ and $e_2 : \tau_2$.*

Proof. Expanding the definitions of pairing and the product type, and applying Corollary 52.13 on page 449, we let ρ and ρ' be arbitrary closed types, and let $R : \rho \leftrightarrow \rho'$ be an admissible relation between them. Suppose further that

$$h \sim h' : \tau_1 \to \tau_2 \to t\ [\eta : \delta \leftrightarrow \delta'],$$

where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$ (and are each undefined on $t' \neq t$). We are to show that for some $e_1 : \tau_1$ and $e_2 : \tau_2$,

$$e[\rho](h) \sim h'(e_1)(e_2) : t\ [\eta : \delta \leftrightarrow \delta'],$$

which is to say

$$e[\rho](h)\ R\ h'(e_1)(e_2).$$

Now by Theorem 52.8 on page 447 we have $e \sim e : \tau_1 \times \tau_2$. Define the relation $S : \rho \leftrightarrow \rho'$ by $d\ S\ d'$ iff the following conditions are satisfied:

1. $d \cong h(d_1)(d_2) : \rho$ for some $d_1 : \tau_1$ and $d_2 : \tau_2$;
2. $d' \cong h'(d'_1)(d'_2) : \rho'$ for some $d'_1 : \tau_1$ and $d'_2 : \tau_2$;
3. $d\ R\ d'$.

This is clearly an admissible relation. Noting that

$$h \sim h' : \tau_1 \to \tau_2 \to t\ [\eta' : \delta \leftrightarrow \delta'],$$

where $\eta'(t) = S$ and is undefined for $t' \neq t$, we conclude that $e\,[\rho]\,(h)\ S\ e\,[\rho']\,(h')$, and hence

$$e\,[\rho]\,(h)\ R\ h'\,(d_1')\,(d_2'),$$

as required. □

Now suppose that $e : \tau_1 \times \tau_2$ is such that $\mathtt{fst}(e) \cong e_1 : \tau_1$ and $\mathtt{snd}(e) \cong e_2 : \tau_2$. We wish to show that $e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$. From Lemma 52.15 on the previous page it is easy to deduce that $e \cong \langle \mathtt{fst}(e), \mathtt{snd}(e) \rangle : \tau_1 \times \tau_2$ by congruence and direct calculation. Hence, by congruence we have $e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$.

By a similar line of reasoning we may show that the Church encoding of the natural numbers given in Chapter 23 strongly defines the natural numbers in that the following properties hold:

1. $\mathtt{iter}\,\mathtt{z}\,\{\mathtt{z}{\Rightarrow}e_0 \mid \mathtt{s}(x){\Rightarrow}e_1\} \cong e_0 : \rho$.
2. $\mathtt{iter}\,\mathtt{s}(e)\,\{\mathtt{z}{\Rightarrow}e_0 \mid \mathtt{s}(x){\Rightarrow}e_1\} \cong [\mathtt{iter}\,e\,\{\mathtt{z}{\Rightarrow}e_0 \mid \mathtt{s}(x){\Rightarrow}e_1\}/x]e_1 : \rho$.
3. Suppose that $x : \mathtt{nat} \vdash r(x) : \rho$. If
   (a) $r(\mathtt{z}) \cong e_0 : \rho$, and
   (b) $r(\mathtt{s}(e)) \cong [r(e)/x]e_1 : \rho$,

   then for every $e : \mathtt{nat}$, $r(e) \cong \mathtt{iter}\,e\,\{\mathtt{z}{\Rightarrow}e_0 \mid \mathtt{s}(x){\Rightarrow}e_1\} : \rho$.

The first two equations, which constitute weak definability, are easily established by calculation, using the definitions given in Chapter 23. The third property, the unicity of the iterator, is proved using parametricity by showing that every closed expression of type nat is observationally equivalent to a numeral $\overline{n}$. We then argue for unicity of the iterator by mathematical induction on $n \geq 0$.

Lemma 52.16. *If $e : \mathtt{nat}$, then either $e \cong \mathtt{z} : \mathtt{nat}$, or there exists $e' : \mathtt{nat}$ such that $e \cong \mathtt{s}(e') : \mathtt{nat}$. Consequently, there exists $n \geq 0$ such that $e \cong \overline{n} : \mathtt{nat}$.*

Proof. By Theorem 52.8 on page 447 we have $e \sim e : \mathtt{nat}$. Define the relation $R : \mathtt{nat} \leftrightarrow \mathtt{nat}$ to be the strongest relation such that $d\ R\ d'$ iff either $d \cong \mathtt{z} : \mathtt{nat}$ and $d' \cong \mathtt{z} : \mathtt{nat}$, or $d \cong \mathtt{s}(d_1) : \mathtt{nat}$ and $d' \cong \mathtt{s}(d_1') : \mathtt{nat}$ and $d_1\ R\ d_1'$. It is easy to see that $\mathtt{z}\ R\ \mathtt{z}$, and if $e\ R\ e'$, then $\mathtt{s}(e)\ R\ \mathtt{s}(e')$. Letting $\mathtt{zero} = \mathtt{z}$ and $\mathtt{succ} = \lambda(x{:}\mathtt{nat}.\,\mathtt{s}(x))$, we have

$$e\,[\mathtt{nat}]\,(\mathtt{zero})\,(\mathtt{succ})\ R\ e\,[\mathtt{nat}]\,(\mathtt{zero})\,(\mathtt{succ}).$$

The result follows by the induction principle arising from the definition of R as the strongest relation satisfying its defining conditions. □

A straightforward extension of this argument shows that, up to observational equivalence, inductive and coinductive types are strongly definable in $\mathcal{L}\{\to\forall\}$.

52.5 Exercises

Chapter 53

Representation Independence

This chapter must be revised to be consistent with Chapter 52.

Parametricity is the essence of representation independence. The typing rules for `open` given in 24.1 on page 206 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 50 and 52, 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 52 for polymorphic languages. This condition is called *bisimilarity*.

53.1 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 24 that the client, e_c, of the abstract type $\exists(t.\tau)$ is such that t type, $x : \tau \vdash e_c : \tau_c$ for some type τ_c such that $t \# \tau_c$. It follows from Theorem 52.8 on page 447 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

$$\texttt{open}\, e_1'\, \texttt{as}\, t\, \texttt{with}\, x\texttt{:}\tau\, \texttt{in}\, e_c \sim \texttt{open}\, e_2'\, \texttt{as}\, t\, \texttt{with}\, x\texttt{:}\tau\, \texttt{in}\, e_c : \tau_c.$$

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 $t \# \tau_c$ 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.

53.2 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
$$\texttt{nil @ rev(nil)} \cong \texttt{nil} : \texttt{nat list},$$
which follows by symbolic execution, using the definitions of the operations involved.

2. For `insert`, we assume that $m \sim n : \mathtt{nat}$ and $l\ R\ \langle b, f\rangle$, and prove that

$$\mathtt{QL.insert}(m, l)\ R\ \mathtt{QFB.insert}(n, \langle b, f\rangle).$$

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

$$m :: l \cong (n :: b)\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}.$$

Calculating, we obtain

$$(n :: b)\ \mathtt{@}\ \mathtt{rev}(f) \cong n :: (b\ \mathtt{@}\ \mathtt{rev}(f)) : \mathtt{nat\ list}$$

and

$$n :: (b\ \mathtt{@}\ \mathtt{rev}(f)) \cong n :: l : \mathtt{nat\ list},$$

since $l \cong b\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}$. Since $m \sim n : \mathtt{nat}$, 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 $l \cong b\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}$. We are to show

$$\mathtt{QL.remove}(l) \sim \mathtt{QFB.remove}(\langle b, f\rangle) : \mathtt{nat} \times t\ [[t \mapsto R] : [t \mapsto \rho_{\mathsf{ls}}] \leftrightarrow [t \mapsto \rho_{\mathsf{fb}}]].$$

Assuming that the queue is non-empty, so that removing an element is well-defined, it can be shown that $l \cong l'\ \mathtt{@}\ [m] : \mathtt{nat\ list}$ 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 $f \cong n :: f' : \mathtt{nat\ list}$ for some n and f'. Then by the definition of `QFB.remove`,

$$\mathtt{QFB.remove}(\langle b, f\rangle) \cong \langle n, \langle b, f'\rangle\rangle : \mathtt{nat} \times t,$$

taking equality at type t to be the relation R. We must show that

$$\langle m, l'\rangle \sim \langle n, \langle b, f'\rangle\rangle : \mathtt{nat} \times t,$$

with t equality being R. This means that we must show that $m = n$ and $l' \cong b\ \mathtt{@}\ \mathtt{rev}(f') : \mathtt{nat\ list}$.

Calculating from our assumptions,

$$\begin{aligned} l &\cong l'\ \mathtt{@}\ [m] \\ &\cong b\ \mathtt{@}\ \mathtt{rev}(f) \\ &\cong b\ \mathtt{@}\ \mathtt{rev}(n :: f') \\ &\cong b\ \mathtt{@}\ (\mathtt{rev}(f')\ \mathtt{@}\ [n]) \\ &\cong (b\ \mathtt{@}\ \mathtt{rev}(f'))\ \mathtt{@}\ [n], \end{aligned}$$

from which the result follows. Finally, if f is empty, then it can be shown that $b \cong b' \mathtt{@} [n] : \mathtt{nat\,list}$ for some b' and n. But then

$$\mathtt{rev}(b) \cong n :: \mathtt{rev}(b') : \mathtt{nat\,list},$$

which reduces to the case for f non-empty.

This completes the proof — by representation independence the reference and candidate implementations are equivalent.

53.3 Exercises

Part XIX

Working Drafts of Chapters