Practical Foundations for Programming Languages

Robert Harper
Carnegie Mellon University

Spring, 2011

[Version 1.16 of 08.27.2011.]

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.

Being in many ways a consolidation of many ideas from the literature on programming languages, I make no attempt to give an exhaustive account of the history or sources of many of the ideas. The notes at the end of each chapter provide some guidance for further reading and background, but are not intended as a complete guide to the literature. For further background please see Girard [33], Martin-Löf [61], Mitchell [70], Pierce [77, 78], and Reynolds [91].

Comments are most welcome, and should be sent to the author at `rwh@cs.cmu.edu`. I am grateful to the following people for their corrections and suggestions: Arbob Ahmad, Andrew Appel, Zena Ariola, Guy E. Blelloch, William Byrd, Luis Caires, Luca Cardelli, Iliano Cervesato, Manuel Chakravarti, Lin Chase, Richard C. Cobbe, Karl Crary, Daniel Dantas, Anupam Datta, Jake Donham, Derek Dreyer, Matthias Felleisen, Dan Friedman, Maia Ginsburg, Kevin Hely, Cao Jing, Gabriele Keller, Danielle Kramer, Akiva Leffert, Ruy Ley-Wild, Dan Licata, Karen Liu, Dave MacQueen, Chris Martens, Greg Morrisett, Tom Murphy, Aleksandar Nanevski, Georg Neis, David Neville, Doug Perkins, Frank Pfenning, Benjamin C. Pierce, Andrew M. Pitts, Gordon D. Plotkin, David Renshaw, John C. Reynolds, Carter T. Schonwald, Dale Schumacher, Dana Scott, Zhong Shao, Robert Simmons, Pawel Sobocinski, Daniel Spoonhower, Paulo Tanimoto, Bernardo Toninho, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, Todd Wilson, Roger Wolff, Luke Zarko, Yu Zhang.

This material is based on work supported by the National Science Foundation under Grant Nos. 0702381 and 0716469. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work was supported in part by the Max Planck Institute for Software Systems in Germany, whose help I gratefully acknowledge.

I thank Espresso a Mano in Pittsburgh, CB2 Cafe in Cambridge, and Thonet Cafe in Saarbruecken for providing a steady supply of coffee and a conducive atmosphere for writing.

Contents

Part I

Judgements and Rules

Chapter 1

Syntactic Objects

Programming languages are languages, a means of expressing computations in a form comprehensible to both people and machines. The syntax of a language specifies the means by which various sorts of phrases (expressions, commands, declarations, and so forth) may be combined to form programs. But what sort of thing are these phrases? What is a program made of?

The informal concept of syntax may be seen to involve several distinct concepts. The *surface*, or *concrete*, syntax is concerned with how phrases are entered and displayed on a computer. The surface syntax is usually thought of as given by strings of characters from some alphabet (say, ASCII or UniCode). The *structural*, or *abstract*, syntax is concerned with the structure of phrases, specifically how they are composed from other phrases. At this level a phrase is a tree, called an *abstract syntax tree*, whose nodes are operators that combine several phrases to form another phrase. The *binding* structure of syntax is concerned with the introduction and use of identifiers: how they are declared, and how declared identifiers are to be used. At this level phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

In this chapter we prepare the ground for all of our later work by defining precisely what are strings, abstract syntax trees, and abstract binding trees. The definitions are a bit technical, but are fundamentally quite simple and intuitive. It is probably best to skim this chapter on first reading, returning to it only as the need arises.

1.1 Strings

An *alphabet* is a (finite or infinite) collection of *characters*. In practice the alphabet is a standardized set such as the UniCode character set. A *string* over an alphabet is either the *null string*, ϵ, consisting of no characters, or the extension of a string by a single character, $c \cdot s$. 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.

It follows from the definition of strings that to show that a property, $\mathcal{P}$, holds of a string s, written $\mathcal{P}(s)$, it suffices to show two things:

1. $\mathcal{P}(\epsilon)$, and
2. if $\mathcal{P}(s)$ and c char, then $\mathcal{P}(c \cdot s)$.

This is called the principle of *string induction*.

The concatenation, $s_1 \hat{} s_2$, of two strings over the same alphabet is defined in the obvious way. Concatentation is also denoted by juxtaposition, and individual characters are often identified with the corresponding unit-length string. This means that *abcd* can be thought of in many ways, for example as the concatenations *ab cd*, *a bcd*, or *abc d*, or even ϵ *abcd* or *abcd* ϵ, as may be convenient in a given situation. It does not matter, however, because string concatenation is associative: $s_1 \hat{} (s_2 \hat{} s_3) = (s_1 \hat{} s_2) \hat{} s_3$. In Chapter 4 we will see that this innocuous-seeming fact is responsible for many of the complications in defining the concrete syntax of a language.

1.2 Abstract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree whose leaves are *variables*, and whose interior nodes are *operators* whose *arguments* are its children. Abstract syntax trees are classified into a variety of *sorts* corresponding to different forms of syntax. A *variable* is an *unknown*, or *indeterminate*, standing for an unspecified, or generic, piece of syntax of a specified sort. Ast's may be combined by an *operator*, which has both a sort and an *arity*, a finite sequence of sorts specifying the number and sorts of its arguments. An operator of sort s and arity $s_1, \ldots, s_n$ combines $n \geq 0$ ast's of sort $s_1, \ldots, s_n$, respectively, into a compound ast of sort s. As a matter of terminology, a *nullary* operator is one that takes no arguments, a *unary* operator takes one, a *binary* operator two, and so forth.

For example, consider a simple language of expressions built from numbers, addition, and multiplication. The abstract syntax of such a language

would consist of a single sort, Exp, and three operators that generate the forms of expression: `num[`n`]` is a nullary operator of sort Exp whenever $n \in \mathbb{N}$; `plus` and `times` are binary operators of sort Exp whose arguments are both of sort Exp. The expression $2 + (3 \times x)$, which involves a variable, x, would be represented by the ast

```
plus(num[2];times(num[3];x))
```

of sort Exp, under the assumption that x is also of this sort.[1]

Let $\mathcal{S}$ be a finite set of sorts. Let $\{ \mathcal{O}_s \}_{s \in \mathcal{S}}$ be a sort-indexed family of *operators*, o, of sort s with arity $\mathsf{ar}(o) = (s_1, \ldots, s_n)$. Let $\{ \mathcal{X}_s \}_{s \in \mathcal{S}}$ be a sort-indexed family of *variables*, x, of each sort s. The family $\mathcal{A}[\mathcal{X}] = \{ \mathcal{A}[\mathcal{X}]_s \}_{s \in \mathcal{S}}$ of ast's of sort s is defined as follows:

1. A variable of sort s is an ast of sort s: if $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. Operators combine ast's: if o is an operator of sort s such that $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

It follows from this definition that the principle of *structural induction* may be used to prove that some property, $\mathcal{P}$, holds of every ast. To show $\mathcal{P}(a)$ holds for every $a \in \mathcal{A}[\mathcal{X}]$, it is enough to show:

1. If $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$.
2. If $o \in \mathcal{O}_s$ and $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, then if $a_1 \in \mathcal{P}_{s_1}$ and $\ldots$ and $a_n \in \mathcal{P}_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{P}_s$.

For example, it is easy to prove by structural induction that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

If $\mathcal{X}$ is a sort-indexed family of variables, we write $\mathcal{X}, x$, where x is a variable of sort s such that $x \notin \mathcal{X}_s$, to stand for the family of sets $\mathcal{Y}$ such that $\mathcal{Y}_s = \mathcal{X}_s \cup \{ x \}$ and $\mathcal{Y}_{s'} = \mathcal{X}_{s'}$ for all $s' \neq s$. The family $\mathcal{X}, x$, where x is a variable of sort s, is said to be the family obtained by *adjoining* the variable x to the family $\mathcal{X}$.

Variables are given meaning by *substitution*. If $a \in \mathcal{A}[\mathcal{X}, x]$ and $b \in \mathcal{A}[\mathcal{X}]$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]$ is defined to be the result of substituting b for every occurrence of x in a. The ast a is called the *target*, and x is called the *subject*, of the substitution. Substitution is defined by the following equations:

[1] In Part II we will discuss in more detail the passage from the informal to the formal representation of syntax.

1. $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(a_1; \ldots; a_n) = o([b/x]a_1; \ldots; [b/x]a_n)$.

For example, we may check that

$$[\texttt{num[2]}/x]\texttt{plus}(x;\texttt{num[3]}) = \texttt{plus}(\texttt{num[2]};\texttt{num[3]}).$$

We may prove by structural induction that substitution on ast's is well-defined.

Theorem 1.1. *If $b \in \mathcal{A}[\mathcal{X}, x]$, then for every $a \in \mathcal{A}[\mathcal{X}]$ there exists a unique $c \in \mathcal{A}[\mathcal{X}]$ such that $[b/x]a = c$*

Proof. By structural induction on a. If $a = x$, then $c = b$ by definition, otherwise if $a = y \neq x$, then $c = y$, also by definition. Otherwise, $a = o(a_1, \ldots, a_n)$, and we have by induction unique $c_1, \ldots, c_n$ such that $[b/x]a_1 = c_1$ and $\ldots$ $[b/x]a_n = c_n$, and so c is $c = o(c_1; \ldots; c_n)$, by definition of substitution. □

In addition to variables we will also have need of a stock of identifiers, called *symbols* or *names* or *parameters*, that are not themselves forms of ast, but which may be used to index a family of operators of a given sort and arity. For example, let Cls be a distinguished sort of classes and let $\mathcal{U}_{\mathsf{Cls}}$ be a set of symbols that are to be thought of as class names. For each name $u \in \mathcal{U}_{\mathsf{Cls}}$, let $\texttt{inst}[u]$ be an operator of sort Exp and arity (Exp). Any such operator may be used to construct an ast, $\texttt{inst}[u](a)$, of sort Exp from an ast, a, of this sort. This expression may be thought of as standing for an instance of the class u with instance data a.[2] The class name is written in square brackets to emphasize that it is merely an *index* for a family of operators, and is not itself to be thought of as a form of ast.

Let $\mathcal{U}$ be a sort-indexed family of symbols, and let $\mathcal{O}$ be a sort-indexed family of operators, any of which may involve parameters from $\mathcal{U}$. The family of sets $\mathcal{A}[\mathcal{U}; \mathcal{X}]$ is defined to be the ast's generated by the operators $\mathcal{O}$, the variables $\mathcal{X}$, and the symbols $\mathcal{U}$. Renaming is extended to symbols in the evident manner, but substitution is defined only for variables in $\mathcal{X}$, and not for symbols. This is as it should be, because symbols are not themselves forms of abstract syntax, and hence it makes no sense to consider replacing a symbol by an ast.

[2] See Chapters 14 and 27 for a full development of this motivating example.

1.3 Abstract Binding Trees

Abstract binding trees, or *abt's*, enrich abstract syntax trees with the means to introduce new variables and symbols, called a *binding*, with a specified range of significance, called its *scope*. The scope of a binding is an abt within which the bound identifier may be used, either as a placeholder (in the case of a variable declaration) or as the index of some operator (in the case of a symbol declaration). Thus the set of active identifiers may be larger within a subtree of an abt than it is within the surrounding tree. Moreover, different subtrees may introduce identifiers with disjoint scopes. The crucial principle is that any use of an identifier should be understood as a reference, or abstract pointer, to its binding. One consequence is that the choice of identifier names is immaterial, so long as one can always associate a unique binding with each use of an identifier.

As a motivating example, consider the expression $\texttt{let}\ x\ \texttt{be}\ a_1\ \texttt{in}\ a_2$, which introduces a variable, x, for use within the expression a_2 to stand for the expression a_1. The variable x is bound by the `let` expression for use within a_2; any use of x within a_1 refers to a different variable that happens to have the same name. For example, in the expression $\texttt{let}\ x\ \texttt{be}\ 7\ \texttt{in}\ x + x$ occurrences of x in the addition refer to the variable introduced by the `let`. On the other hand in the expression $\texttt{let}\ x\ \texttt{be}\ x * x\ \texttt{in}\ x + x$, occurrences of x within the multiplication refer to a different variable than those occurring within the addition. The latter occurrences refer to the binding introduced by the `let`, whereas the former refer to some outer binding not displayed here.

The names of bound variables are immaterial insofar as they determine the same binding. So, for example, the expression $\texttt{let}\ x\ \texttt{be}\ x * x\ \texttt{in}\ x + x$ could just as well have been written $\texttt{let}\ y\ \texttt{be}\ x * x\ \texttt{in}\ y + y$ without changing its meaning. In the former case the variable x is bound within the addition, and in the latter it is the variable y, but the "pointer structure" remains the same. On the other hand the expression $\texttt{let}\ x\ \texttt{be}\ y * y\ \texttt{in}\ x + x$ has a different meaning to these two expressions, because now the variable y within the multiplication refers to a different surrounding variable. Renaming of bound variables is constrained to the extent that it must not alter the reference structure of the expression. For example, the expression $\texttt{let}\ x\ \texttt{be}\ 2\ \texttt{in}\ \texttt{let}\ y\ \texttt{be}\ 3\ \texttt{in}\ x + x$ has a different meaning than the expression $\texttt{let}\ y\ \texttt{be}\ 2\ \texttt{in}\ \texttt{let}\ y\ \texttt{be}\ 3\ \texttt{in}\ y + y$, because the y in the expression `succ(`y`)` in the second case refers to the inner declaration, not the outer one as before.

The concept of an ast may be enriched to account for binding and scope of variable. These enriched ast's are called *abstract binding trees*, or *abt's*

for short. Abt's generalize ast's by allowing an operator to bind any finite number (possibly zero) of variables in each argument position. An argument to an operator is called an *abstractor*, and has the form $x_1, \ldots, x_k\,.\,a$. The sequence of variables $x_1, \ldots, x_k$ are bound within the abt a. (When k is zero, we elide the distinction between $.a$ and a itself.) Written in the form of an abt, the expression $\texttt{let}\ x\ \texttt{be}\ a_1\ \texttt{in}\ a_2$ has the form $\texttt{let}(a_1; x\,.\,a_2)$, which more clearly specifies that the variable x is bound within a_2, and not within a_1. We often write $\vec{x}$ to stand for a finite sequence $x_1, \ldots, x_n$ of distinct variables, and write $\vec{x}\,.\,a$ to mean $x_1, \ldots, x_n\,.\,a$.

To account for binding, the arity of an operator is generalized to consist of a finite sequence of *valences*. The length of the sequence determines the number of arguments, and each valence determines the sort of the argument and the number and sorts of the variables that are bound within it. A valence of the form $(s_1, \ldots, s_k)s$ specifies an argument of sort s that binds k variables of sorts $s_1, \ldots, s_k$ within it. We often write $\vec{s}$ for a finite sequence $s_1, \ldots, s_n$ of sorts, and we say that $\vec{x}$ is of sort $\vec{s}$ to mean that the two sequences have the same length and that each x_i is of sort s_i.

Thus, for example, the arity of the operator `let` is $(\mathsf{Exp}, (\mathsf{Exp})\mathsf{Exp})$, which indicates that it takes two arguments described as follows:

1. The first argument is of sort Exp and binds no variables.
2. The second argument is of sort Exp and binds one variable of sort Exp.

The definition expression $\texttt{let}\ x\ \texttt{be}\ 2 + 2\ \texttt{in}\ x \times x$ is represented by the abstract binding tree

$$\texttt{let(plus(num[2];num[2]);}x\texttt{.times(}x\texttt{;}x\texttt{))}.$$

Let $\mathcal{O}$ be a sort-indexed family of operators, o, with arities, $\mathsf{ar}(o)$. For a given sort-indexed family, $\mathcal{X}$, of variables, the sort-indexed family of abt's, $\mathcal{B}[\mathcal{X}]$, is defined similarly to $\mathcal{A}[\mathcal{X}]$, except that the set of active variables changes for each argument according to which variables are bound within it. A first cut at the definition is as follows:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. If $\mathsf{ar}(o) = ((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if, for each $1 \leq i \leq n$, $\vec{x}_i$ is of sort $\vec{s}_i$ and $a_i \in \mathcal{B}[\mathcal{X}, \vec{x}_i]_{s_i}$, then $o(\vec{x}_1\,.\,a_1\,; \ldots\,; \vec{x}_n\,.\,a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The bound variables are adjoined to the set of active variables within each argument, with the sort of each variable determined by the valence of the operator.

This definition is *almost* correct, but fails to properly account for the behavior of bound variables. An abt of the form $\texttt{let}(a_1; x.\texttt{let}(a_2; x.a_3))$ is ill-formed according to this definition, because the first binding adjoins x to $\mathcal{X}$, which implies that the second cannot also adjoin x to $\mathcal{X}, x$ without causing confusion. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names. This is achieved by altering the second clause of the definition using renaming as follows:[3]

> If $\mathsf{ar}(o) = ((\vec{x}_1)s_1, \ldots, (\vec{x}_n)s_n)$, and if, for each $1 \leq i \leq n$ and for each renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$, where $\vec{x}'_i \notin \mathcal{X}$, we have $\pi_i \cdot a_i \in \mathcal{B}[\mathcal{X}, \vec{x}'_i]$, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The renaming ensures that when we encounter nested binders we avoid collisions. This is called the *freshness condition on binders* since it ensures that all bound variables are "fresh" relative to the surrounding context.

The principle of structural induction extends to abt's, and is called *structural induction modulo renaming*. It states that to show that $\mathcal{P}(a)[\mathcal{X}]$ holds for every $a \in \mathcal{B}[\mathcal{X}]$, it is enough to show the following:

1. if $x \in \mathcal{X}_s$, then $\mathcal{P}[\mathcal{X}]_s(x)$.
2. For every o of sort s and arity $((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if for each $1 \leq i \leq n$, we have $\mathcal{P}[\mathcal{X}, \vec{x}'_i]_{s_i}(\pi_i \cdot a_i)$ for every renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$, then $\mathcal{P}[\mathcal{X}]_s(o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n))$.

The renaming in the second condition ensures that the inductive hypothesis holds for *all* fresh choices of bound variable names, and not just the ones actually given in the abt.

As an example let us define the judgement $x \in a$, where $a \in \mathcal{B}[\mathcal{X}, x]$, to mean that x *occurs free* in a. Informally, this means that x is bound somewhere outside of a, rather than within a itself. If x is bound within a, then those occurrences of x are different from those occurring outside the binding. The following definition ensures that this is the case:

1. $x \in x$.
2. $x \in o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n)$ if there exists $1 \leq i \leq n$ such that for every fresh renaming $\pi : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \pi \cdot a_i$.

[3] The action of a renaming extends to abt's in the obvious way by replacing every occurrence of x by $\pi(x)$, including any occurrences in the variable list of an abstractor as well as within its body.

The first condition states that x is free in x, but not free in y for any variable y other than x. The second condition states that if x is free in some argument, independently of the choice of bound variable names in that argument, then it is free in the overall abt. This implies, in particular, that x is *not* free in $\texttt{let(zero;}x\texttt{.}x\texttt{)}$.

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons), is defined to mean that a and b are identical up to the choice of bound variable names. This relation is defined to be the strongest congruence containing the following two conditions:

1. $x =_\alpha x$.
2. $o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) =_\alpha o(\vec{x}'_1 . a'_1 ; \ldots ; \vec{x}'_n . a'_n)$ if for every $1 \leq i \leq n$, $\pi_i \cdot a_i =_\alpha \pi'_i \cdot a'_i$ for all fresh renamings $\pi_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\pi'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

The idea is that we rename $\vec{x}_i$ and $\vec{x}'_i$ consistently, avoiding confusion, and check that a_i and a'_i are α-equivalent. If $a =_\alpha b$, then a and b are said to be *α-variants* of each other.

Some care is required in the definition of *substitution* of an abt b of sort s for free occurrences of a variable x of sort s in some abt a of some sort, written $[b/x]a$. Substitution is partially defined by the following conditions:

1. $[b/x]x = b$, and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) = o(\vec{x}_1 . a'_1 ; \ldots ; \vec{x}_n . a'_n)$, where, for each $1 \leq i \leq n$, we require that $\vec{x}_i \notin b$, and we set $a'_i = [b/x]a_i$ if $x \notin \vec{x}_i$, and $a'_i = a_i$ otherwise.

If x is bound in some argument to an operator, then substitution does not descend into its scope, for to do so would be to confuse two distinct variables. For this reason we must take care to define a'_i in the second equation according to whether or not $x \in \vec{x}_i$. The requirement that $\vec{x}_i \notin b$ in the second equation is called *capture avoidance*. If some $x_{i,j}$ occurred free in b, then the result of the substitution $[b/x]a_i$ would in general contain $x_{i,j}$ free as well, but then forming $\vec{x}_i . [b/x]a_i$ would *incur capture* by changing the referent of $x_{i,j}$ to be the jth bound variable of the ith argument. In such cases *substitution is undefined* since we cannot replace x by b in a_i without incurring capture.

One way around this is to alter the definition of substitution so that the bound variables in the result are chosen fresh by substitution. By the principle of structural induction we know inductively that, for any renaming

$\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$ with $\vec{x}'_i$ fresh, the substitution $[b/x](\pi_i \cdot a_i)$ is well-defined. Hence we may define

$$[b/x]o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) = o(\vec{x}'_1 . [b/x](\pi_1 \cdot a_1) ; \ldots ; \vec{x}'_n . [b/x](\pi_n \cdot a_n))$$

for some particular choice of fresh bound variable names (any choice will do). There is no longer any need to take care that $x \notin \vec{x}_i$ in each argument, because the freshness condition on binders ensures that this cannot occur, the variable x already being active. Noting that

$$o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) =_\alpha o(\vec{x}'_1 . \pi_1 \cdot a_1 ; \ldots ; \vec{x}'_n . \pi_n \cdot a_n),$$

another way to avoid undefined substitutions is to first choose an α-variant of the target of the substitution whose binders avoid any free variables in the substituting abt, and then perform substitution without fear of incurring capture. In other words substitution is totally defined on α-equivalence classes of abt's.

To avoid all the bureaucracy of binding, we adopt the following *identification convention* throughout this book:

Abstract binding trees are always to be identified up to α-equivalence.

That is, we implicitly work with α-equivalence classes of abt's, rather than abt's themselves. We tacitly assert that all operations and relations on abt's respect α-equivalence, so that they are properly defined on α-equivalence classes of abt's. Whenever we examine an abt, we are choosing a representative of its α-equivalence class, and we have no control over how the bound variable names are chosen. On the other hand experience shows that any operation or property of interest respects α-equivalence, so there is no obstacle to achieving it. Indeed, we might say that a property or operation is legitimate exactly insofar as it respects α-equivalence!

Symbols, as well as variables, may be bound within an argument. Operators indexed by such symbols come into existence within the scope of the symbol, and go out of existence outside of that scope. As an example, consider the family of nullary operators `cls[`a`]` of sort `Exp` indexed by symbols a. (Such an operator might represent a class of data; see Chapter 14 for further development of this idea.) The operator `cls[`a`]` is available only within the scope of the symbol, a, and is otherwise unavailable.

To allow for symbol declaration the valence of an argument is generalized a bit further to specify the number and valences of its bound symbols, as well as of its bound names, $(\vec{r};\vec{s})s$. The sort-indexed family $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is

the set of abt's determined by a fixed set of operators using the symbols, $\mathcal{U}$, and the variables, $\mathcal{X}$. We generally rely on naming conventions to distinguish symbols from variables, reserving u and v for generic symbols, and x and y for generic variables.

1.4 Notes

The concept of abstract syntax has its orgins in the pioneering work of Church, Turing, and Gödel, who first considered the possibility of writing programs that act on representations of programs. Originally programs were represented by natural numbers, using encodings, now called *Gödel-numberings*, based on the prime factorization theorem. Any standard text on mathematical logic, such as Kleene [50], contains a thorough account of such representations. McCarthy's work on Lisp [63, 4] introduced a much more practical and direct representation of syntax as *symbolic expressions*. These ideas were developed further in ML [35], which featured a type system capable of expressing abstract syntax trees. De Bruijn's work on AUTOMATH [74] introduced the idea of using Church's λ notation [21] to account for the binding and scope of variables. These ideas were developed further in the LF language and methodology for formalizing languages and logics [41].

Chapter 2

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. An inductive definition consists of a set of *rules* for deriving *judgements*, or *assertions*, of a variety of forms. Judgements are statements about one or more syntactic objects of a specified sort. The rules specify necessary and sufficient conditions for the validity of a judgement, and hence fully determine its meaning.

2.1 Judgements

We start with the notion of a *judgement*, or *assertion*, about a syntactic object. 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
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 syntactic 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 write a J for the judgement asserting that J 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.

2.2 Inference Rules

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

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

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`:

$$\frac{}{\texttt{zero nat}} \tag{2.2a}$$

$$\frac{a \texttt{ nat}}{\texttt{succ(}a\texttt{) nat}} \tag{2.2b}$$

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

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

$$\frac{}{\texttt{empty tree}} \tag{2.3a}$$

$$\frac{a_1 \texttt{ tree} \quad a_2 \texttt{ tree}}{\texttt{node(}a_1\texttt{;}a_2\texttt{) tree}} \tag{2.3b}$$

These rules specify that a `tree` holds if either a is `empty`, or a is `node(`a_1`;`a_2`)`, where a_1 `tree` and a_2 `tree`. Taking these to be exhaustive, these rules state

that a is a binary tree, which is to say it is either empty, or a node consisting of two children, each of which is also a binary tree.

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

$$\frac{}{\texttt{zero} = \texttt{zero nat}} \tag{2.4a}$$

$$\frac{a = b \text{ nat}}{\texttt{succ}(a) = \texttt{succ}(b) \text{ nat}} \tag{2.4b}$$

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 (2.2b) 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.

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*.

2.3 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 finite 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

$$\frac{\nabla_1 \quad \ldots \quad \nabla_k}{J} \tag{2.5}$$

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`:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}}\,. \tag{2.6}$$

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{empty tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(empty;empty) tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(node(empty;empty);empty) tree}}\,. \tag{2.7}$$

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.

2.4 Rule Induction

Since an inductive definition specifies the *strongest* judgement closed under a collection of rules, we may reason about them by *rule induction*. 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 the property $\mathcal{P}$ holds of the judgement J, we say that $\mathcal{P}$ respects the rule

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

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 called the *inductive hypotheses*, and $\mathcal{P}(J)$ is called the *inductive conclusion* of the inference.

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 defined by a set of rules is both (a) closed under those rules, and (b) sufficient for any other property also closed under those rules. The former 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.

When specialized to Rules (2.2), the principle of rule induction states that to show $\mathcal{P}(a\ \mathsf{nat})$ whenever a nat, it is enough to show:

1. $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$.
2. for every a, if a nat and $\mathcal{P}(a\ \mathsf{nat})$, then ($\mathtt{succ}(a)$ nat and) $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$.

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

Similarly, rule induction for Rules (2.3) states that to show $\mathcal{P}(a\ \mathsf{tree})$ whenever a tree, it is enough to show

1. $\mathcal{P}(\mathtt{empty}\ \mathsf{tree})$.
2. for every a_1 and a_2, if a_1 tree and $\mathcal{P}(a_1\ \mathsf{tree})$, and if a_2 tree and $\mathcal{P}(a_2\ \mathsf{tree})$, then ($\mathtt{node}(a_1;a_2)$ tree and) $\mathcal{P}(\mathtt{node}(a_1;a_2)\ \mathsf{tree})$.

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

We may also show by rule induction 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 2.1. *If* $\mathtt{succ}(a)$ nat, *then* a nat.

Proof. It suffices to show that the property, $\mathcal{P}(a\ \mathsf{nat})$ stating that a nat and that $a = \mathtt{succ}(b)$ implies b nat is closed under Rules (2.2).

Rule (2.2a) Clearly $\mathtt{zero}$ nat, and the second condition holds vacuously, since zero is not of the form $\mathtt{succ}(-)$.

Rule (2.2b) Inductively we know that a nat and that if a is of the form $\mathtt{succ}(b)$, then b nat. We are to show that $\mathtt{succ}(a)$ nat, which is immediate, and that if $\mathtt{succ}(a)$ is of the form $\mathtt{succ}(b)$, then b nat, and we have b nat by the inductive hypothesis.

This completes the proof. □

Using rule induction we may show that equality, as defined by Rules (2.4) is reflexive.

Lemma 2.2. *If* a nat, *then* $a = a$ nat.

Proof. By rule induction on Rules (2.2):

Rule (2.2a) Applying Rule (2.4a) we obtain $\mathtt{zero} = \mathtt{zero}\ \mathsf{nat}$.

Rule (2.2b) Assume that $a = a\ \mathsf{nat}$. It follows that $\mathtt{succ}(a) = \mathtt{succ}(a)\ \mathsf{nat}$ by an application of Rule (2.4b).

□

Similarly, we may show that the successor operation is injective.

Lemma 2.3. *If* $\mathtt{succ}(a_1) = \mathtt{succ}(a_2)\ \mathsf{nat}$, *then* $a_1 = a_2\ \mathsf{nat}$.

Proof. Similar to the proof of Lemma 2.1 on the facing page. □

2.5 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 $a\ \mathsf{list}$ stating that a is a list of natural numbers.

$$\frac{}{\mathtt{nil}\ \mathsf{list}} \tag{2.8a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\mathtt{cons}(a;b)\ \mathsf{list}} \tag{2.8b}$$

The first premise of Rule (2.8b) is an instance of the judgement form $a\ \mathsf{nat}$, which was defined previously, whereas the premise $b\ \mathsf{list}$ 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 requires 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:

$$\frac{}{\texttt{zero even}} \tag{2.9a}$$

$$\frac{a\ \texttt{odd}}{\texttt{succ}(a)\ \texttt{even}} \tag{2.9b}$$

$$\frac{a\ \texttt{even}}{\texttt{succ}(a)\ \texttt{odd}} \tag{2.9c}$$

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

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

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) $\mathcal{P}(a\ \texttt{even})$ iff a nat, and (2) $\mathcal{P}(a\ \texttt{odd})$ iff a nat. The principle of rule induction for Rules (2.9) states that it is sufficient to show the following facts:

1. $\texttt{zero}$ nat, which is derivable by Rule (2.2a).
2. If a nat, then $\texttt{succ}(a)$ nat, which is derivable by Rule (2.2b).
3. If a nat, then $\texttt{succ}(a)$ nat, which is also derivable by Rule (2.2b).

2.6 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:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero};b;b)} \tag{2.10a}$$

$$\frac{\mathsf{sum}(a;b;c)}{\mathsf{sum}(\mathtt{succ}(a);b;\mathtt{succ}(c))} \tag{2.10b}$$

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 2.4. *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 $\mathsf{sum}(a;b;c)$, 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 (2.2). We have two cases to consider:

Rule (2.2a) 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 (2.10a).

Rule (2.2b) 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 (2.10b), 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 (2.10).

Rule (2.10a) 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 2.2 on page 18 we obtain $b = b$ nat.

Rule (2.10b) 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.

□

2.7 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 (2.10). 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, this says that 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 of natural numbers has a partial inverse.

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 (2.10)) using equations:

$$\frac{a\ \mathsf{nat}}{a+\mathtt{zero}=a\ \mathsf{nat}} \tag{2.11a}$$

$$\frac{a+b=c\ \mathsf{nat}}{a+\mathtt{succ}(b)=\mathtt{succ}(c)\ \mathsf{nat}} \tag{2.11b}$$

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 $a + b$ for the unique c such that $a + b = c$ nat.

2.8 Notes

Aczel [2] provides a thorough account of the theory of inductive definitions. The formulation given here is strongly influenced by Martin-Löf's development of the logic of judgements [60, 62].

Chapter 3

Hypothetical and General Judgements

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*. Both enjoy the same *structural properties* expected of entailment, but they differ in that whereas derivability is stable under the addition of new rules, admissibility is, in general, not. A *general judgement* expresses the *universality*, or *genericity*, of a (basic or hypothetical) judgement. There are two forms of general judgement, the *generic* and the *parametric*. The generic judgement expresses generality with respect to all substitution instances for variables in a judgement. The parametric judgement expresses generality with respect to renamings of symbols.

3.1 Hypothetical Judgements

3.1.1 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 basic judgements, to mean that we may derive K from the *expansion* $\mathcal{R}[J_1, \ldots, J_k]$ of the rules $\mathcal{R}$ with the additional axioms

$$\frac{}{J_1} \qquad \cdots \qquad \frac{}{J_k}.$$

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 $\mathcal{R}$. That is, evidence for a hypothetical judgement consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$.

We use capital Greek letters, frequently Γ or Δ, to stand for a finite collection of basic judgements, and write $\mathcal{R}[\Gamma]$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgement in Γ. The judgement $\Gamma \vdash_{\mathcal{R}} K$ means that K is derivable from rules $\mathcal{R}[\Gamma]$, and the judgement $\vdash_{\mathcal{R}} \Gamma$ means that $\vdash_{\mathcal{R}} J$ for each J in Γ. An equivalent way of defining $J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is to say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.1}$$

is *derivable* from $\mathcal{R}$, which means that there is a derivation of J composed of the rules in $\mathcal{R}$ augmented by treating $J_1, \ldots, J_n$ as axioms.

For example, consider the derivability judgement

$$a\ \mathtt{nat} \vdash_{(2.2)} \mathtt{succ(succ}(a)\mathtt{))\ nat} \tag{3.2}$$

relative to Rules (2.2). This judgement is valid for *any* choice of object a, as evidenced by the derivation

$$\dfrac{\dfrac{a\ \mathtt{nat}}{\mathtt{succ}(a)\ \mathtt{nat}}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}} \tag{3.3}$$

which composes Rules (2.2), starting with a nat as an axiom, and ending with succ(succ(a)) nat. Equivalently, the validity of (3.2) may also be expressed by stating that the rule

$$\frac{a\ \mathtt{nat}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}} \tag{3.4}$$

is derivable from Rules (2.2).

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

Theorem 3.1 (Stability)**.** *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \vdash_{\mathcal{R}\cup\mathcal{R}'} J$*.*

Proof. Any derivation of J from $\mathcal{R}[\Gamma]$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}')[\Gamma]$, since any rule in $\mathcal{R}$ is also a rule in $\mathcal{R} \cup \mathcal{R}'$. □

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

Reflexivity Every judgement is a consequence of itself: $\Gamma, J \vdash_{\mathcal{R}} J$. Each hypothesis justifies itself as conclusion.

Weakening If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma, K \vdash_{\mathcal{R}} J$. Entailment is not influenced by unexercised options.

Transitivity If $\Gamma, K \vdash_{\mathcal{R}} J$ and $\Gamma \vdash_{\mathcal{R}} K$, then $\Gamma \vdash_{\mathcal{R}} J$. If we replace an axiom by a derivation of it, the result is a derivation of its consequent without that hypothesis.

Reflexivity follows directly from the meaning of derivability. Weakening follows directly from the definition of derivability. Transitivity is proved by rule induction on the first premise.

3.1.2 Admissibility

Admissibility, written $\Gamma \models_{\mathcal{R}} J$, is a weaker form of hypothetical judgement stating that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. That is, the conclusion J is derivable from rules $\mathcal{R}$ whenever the assumptions Γ are all derivable from rules $\mathcal{R}$. In particular if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the judgement is vacuously true. An equivalent way to define the judgement $J_1, \ldots, J_n \models_{\mathcal{R}} J$ is to state that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.5}$$

is *admissible* relative to the rules in $\mathcal{R}$. This means that given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may construct a derivation of J using the rules in $\mathcal{R}$.

For example, the admissibility judgement

$$\texttt{succ(}a\texttt{) nat} \models_{(2.2)} a\ \texttt{nat} \tag{3.6}$$

is valid, because any derivation of $\texttt{succ(}a\texttt{) nat}$ from Rules (2.2) must contain a sub-derivation of $a\ \texttt{nat}$ from the same rules, which justifies the conclusion. The validity of (3.6) may equivalently be expressed by stating that the rule

$$\frac{\texttt{succ(}a\texttt{) nat}}{a\ \texttt{nat}} \tag{3.7}$$

is admissible for Rules (2.2).

In contrast to derivability the admissibility judgement is *not* stable under extension to the rules. For example, if we enrich Rules (2.2) with the axiom

$$\frac{}{\texttt{succ(junk) nat}} \tag{3.8}$$

(where `junk` is some object for which `junk nat` is not derivable), then the admissibility (3.6) is *invalid*. This is because Rule (3.8) has no premises, and there is no composition of rules deriving `junk nat`. Admissibility is as sensitive to which rules are *absent* from an inductive definition as it is to which rules are *present* in it.

The structural properties of derivability ensure that derivability is stronger than admissibility.

Theorem 3.2. *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \models_{\mathcal{R}} J$*.*

Proof. Repeated application of the transitivity of derivability shows 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

$$\texttt{succ(junk) nat} \vdash_{(2.2)} \texttt{junk nat},$$

yet the admissibility judgement

$$\texttt{succ(junk) nat} \models_{(2.2)} \texttt{junk nat}$$

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. Therefore, the admissibility judgement enjoys the same structural properties as derivability, and hence is a form of hypothetical judgement:

Reflexivity If J is derivable from the original rules, then J is derivable from the original rules: $J \models_{\mathcal{R}} J$.

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 $\Gamma \models_{\mathcal{R}} J$, then $\Gamma, K \models_{\mathcal{R}} J$.

Transitivity If $\Gamma, K \models_{\mathcal{R}} J$ and $\Gamma \models_{\mathcal{R}} K$, then $\Gamma \models_{\mathcal{R}} J$. If the judgements in Γ are derivable, so is K, by assumption, and hence so are the judgements in Γ, K, and hence so is J.

Theorem 3.3. *The admissibility judgement* $\Gamma \models_{\mathcal{R}} J$ *enjoys the structural properties of entailment.*

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. □

If a rule, r, is admissible with respect to a rule set, $\mathcal{R}$, then $\vdash_{\mathcal{R},r} J$ is equivalent to $\vdash_{\mathcal{R}} J$. For if $\vdash_{\mathcal{R}} J$, then obviously $\vdash_{\mathcal{R},r} J$, by simply disregarding r. Conversely, if $\vdash_{\mathcal{R},r} J$, then we may replace any use of r by its expansion in terms of the rules in $\mathcal{R}$. Admissiblity of a rule, r, of the form (3.5) means that any derivations of $J_1,\ldots,J_n$ with respect to rules $\mathcal{R}$ may be transformed into a derivation of J with respect to the same set of rules. It follows by rule induction on $\mathcal{R},r$ that every derivation from the expanded set of rules, $\mathcal{R},r$, may be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to show that $\mathcal{P}(J)$ whenever $\vdash_{\mathcal{R},r} J$, it is sufficient to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$ alone. That is, we need only consider the rules $\mathcal{R}$ in a proof by rule induction to derive $\mathcal{P}(J)$.

3.2 Hypothetical 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 *hypothetical inductive definition* consists of a collection of *hypothetical rules* of the following form:

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\ . \tag{3.9}$$

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 set of global hypotheses for use within that derivation.

In most cases a rule is stated for *all* choices of global context, in which case it is said to be *uniform*. A uniform rule may be given in the *implicit*

form

$$\frac{\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma_n \vdash J_n}{J} \tag{3.10}$$

which stands for the collection of all rules of the form (3.9) in which the global hypotheses have been made explicit.

A hypothetical inductive definition is to be regarded as an ordinary inductive definition of a *formal derivability judgement* $\Gamma \vdash J$ consisting of a finite set of basic judgements, Γ, and a basic judgement, J. A collection of hypothetical rules, $\mathcal{R}$, defines the strongest formal derivability judgement that is *structural* and *closed* under rules $\mathcal{R}$. Structurality means that the formal derivability judgement must be closed under the following rules:

$$\frac{}{\Gamma, J \vdash J} \tag{3.11a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{3.11b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{3.11c}$$

These rules ensure that formal derivability behaves like a hypothetical judgement. By a slight abuse of notation we write $\Gamma \vdash_{\mathcal{R}} J$ to indicate that the $\Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

The principal of *hypothetical rule induction* is just the principal of rule induction applied to the formal hypothetical judgement. So to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under both the rules of $\mathcal{R}$ and under the structural rules. Thus, for each rule of the form (3.10), whether structural or in $\mathcal{R}$, we must show that

$$\text{if } \mathcal{P}(\Gamma\, \Gamma_1 \vdash J_1) \text{ and } \ldots \text{ and } \mathcal{P}(\Gamma\, \Gamma_n \vdash J_n), \text{ then } \mathcal{P}(\Gamma \vdash J).$$

This is just a restatement of the principle of rule induction given in Chapter 2, specialized to the formal derivability judgement $\Gamma \vdash J$.

In practice we usually dispense with the structural rules by the method described in Section 3.1.2 on page 27. By proving that the structural rules are admissible any proof by rule induction may restrict attention to the rules in $\mathcal{R}$ alone. If all of the rules of a hypothetical inductive definition are uniform, the structural rules (3.11b) and (3.11c) are readily seen to be admissible. However, it is typically necessary to include Rule (3.11a) explicitly to ensure reflexivity.

3.3 General Judgements

3.3.1 Generic Derivability

The *generic derivability* judgement $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ states that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the judgement $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$ holds. The renaming ensures that the variables serve only as placeholders; the meaning of the judgement is independent of how the variables are chosen. Evidence for the judgement $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ consists of a *generic derivation*, $\nabla_{\vec{x}}$, such that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the derivation $\nabla_{\vec{x}'}$ is evidence for $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$. The renaming ensures that the meaning of the generic derivability judgement does not depend on the choice of variable names.

For example, the derivation ∇_x given by

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

is evidence for the generic judgement

$$x \mid x\ \mathsf{nat} \vdash^{\mathcal{X}}_{(2.2)} \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}.$$

The generic derivability judgement enjoys the following *structural properties*:

Proliferation If $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$.

Renaming If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [x \leftrightarrow x'] \cdot J$ for any $x' \notin \mathcal{X}, \vec{x}$.

Substitution If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ and $a \in \mathcal{B}[\mathcal{X}, \vec{x}]$, then $\vec{x} \mid [a/x]\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [a/x]J$.

Proliferation is guaranteed by the interpretation of rule schemes as ranging over all expansions of the universe. Renaming is built into the meaning of the generic judgement. Substitution holds as long as the rules themselves are closed under substitution. This need not be the case, but in practice this requirement is usually met.

3.3.2 Parametric Derivability

The *parametric derivability* judgement $\vec{u} \parallel \vec{x} \mid \Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ states that the generic judgement holds *uniformly* for all choices of parameters $\vec{u}$. That is, for all

$\pi : \vec{u} \leftrightarrow \vec{u}'$ such that $\vec{u}' \cap \mathcal{U} = \emptyset$, the generic judgement $\vec{x} \mid \pi \cdot \Gamma \vdash_{\mathcal{R}}^{\mathcal{U},\vec{u}';\mathcal{X}} \pi \cdot J$ is derivable.

The parametric judgement satisfies the following *structural properties*:

Proliferation If $\vec{u} \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$, then $\vec{u}, u \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$.

Renaming If $\vec{u} \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$ and $\pi : \vec{u} \leftrightarrow \vec{u}'$, then $\vec{u}' \parallel \vec{x} \mid \pi \cdot \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} \pi \cdot J$.

Proliferation states that parametric derivability is sensitive only to the presence, but not the absence, of parameters. Renaming states that parametric derivability is independent of the choice of parameters. (There is no analogue of the structural property of substitution for parameters.)

3.4 Generic Inductive Definitions

A *generic inductive definition* admits generic hypothetical judgements in the premises of rules, with the effect of augmenting the variables, as well as the rules, within those premises. A *generic rule* has the form

$$\frac{\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n}{\vec{x} \mid \Gamma \vdash J}\ . \tag{3.12}$$

The variables $\vec{x}$ are the *global variables* of the inference, and, for each $1 \leq i \leq n$, the variables $\vec{x}_i$ are the *local variables* of the ith premise. In most cases a rule is stated for *all* choices of global variables and global hypotheses. Such rules may be given in *implicit form*,

$$\frac{\vec{x}_1 \mid \Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}_n \mid \Gamma_n \vdash J_n}{J}\ . \tag{3.13}$$

A generic inductive definition is just an ordinary inductive definition of a family of *formal generic judgements* of the form $\vec{x} \mid \Gamma \vdash J$. Formal generic judgements are identified up to renaming of variables, so that the latter judgement is treated as identical to the judgement $\vec{x}' \mid \pi \cdot \Gamma \vdash \pi \cdot J$ for any renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$. If $\mathcal{R}$ is a collection of generic rules, we write $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$ to mean that the formal generic judgement $\vec{x} \mid \Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

When specialized to a collection of generic rules, the principle of rule induction states that to show $\mathcal{P}(\vec{x} \mid \Gamma \vdash J)$ whenever $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$. Specifically, for each rule in $\mathcal{R}$ of the form (3.12), we must show that

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

By the identification convention (stated in Chapter 1) the property $\mathcal{P}$ must respect renamings of the variables in a formal generic judgement.

To ensure that the formal generic judgement behaves like a generic judgement, we must always ensure that the following *structural rules* are admissible:

$$\frac{}{\vec{x} \mid \Gamma, J \vdash J} \tag{3.14a}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x} \mid \Gamma, J' \vdash J} \tag{3.14b}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14c}$$

$$\frac{\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash [x \leftrightarrow x'] \cdot J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14d}$$

$$\frac{\vec{x} \mid \Gamma \vdash J \quad \vec{x} \mid \Gamma, J \vdash J'}{\vec{x} \mid \Gamma \vdash J'} \tag{3.14e}$$

$$\frac{\vec{x}, x \mid \Gamma \vdash J \quad a \in \mathcal{B}[\vec{x}]}{\vec{x} \mid [a/x]\Gamma \vdash [a/x]J} \tag{3.14f}$$

The admissibility of Rule (3.14a) is, in practice, ensured by explicitly including it. The admissibility of Rules (3.14b) and (3.14c) is assured if each of the generic rules is uniform, since we may assimilate the additional parameter, x, to the global parameters, and the additional hypothesis, J, to the global hypotheses. The admissibility of Rule (3.14d) is ensured by the identification convention for the formal generic judgement. Rule (3.14f) must be verified explicitly for each inductive definition.

The concept of a generic inductive definition extends to parametric judgements as well. Briefly, rules are defined on formal parametric judgements of the form $\vec{u} \parallel \vec{x} \mid \Gamma \vdash J$, with parameters $\vec{u}$, as well as variables, $\vec{x}$. Such formal judgements are identified up to renaming of both its variables and its parameters to ensure that the meaning is independent of the choice of names. Usually we segregate the hypotheses into two *zones*, written $\vec{u} \parallel \vec{x} \mid \Sigma\ \Gamma \vdash J$, where Σ governs the parameters, $\vec{u}$, and Γ governs the variables, $\vec{x}$. Once separated into zones, it is natural to write this judgement in the form $\vec{x} \mid \Gamma \vdash_{\vec{u} \parallel \Sigma} J$, or even just $\Gamma \vdash_{\Sigma} J$, to reduce notational clutter.

3.5 Notes

The concepts of entailment and generality are fundamental to logic and programming languages. The formulation given here builds on the work of Martin-Löf on the foundations of logic [60, 62] and of Avron on the concept of a consequence relation [9]. Hypothetical and general reasoning are consolidated into a single concept in the AUTOMATH languages [74] and in the LF Logical Framework [41]. These formalisms permit arbitrarily nested combinations of hypothetical and general judgements, whereas the present account restricts attention to general hypothetical judgements on basic judgement forms.

Part II

Levels of Syntax

Chapter 4

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.

4.1 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 lexical structure of a language is usually described using *regular expressions*. For example, the lexical structure of $\mathcal{L}\{\texttt{num str}\}$ may be specified as follows:

Item	itm	::=	kwd | id | num | lit | spl
Keyword	kwd	::=	$\texttt{l}\cdot\texttt{e}\cdot\texttt{t}\cdot\epsilon \mid \texttt{b}\cdot\texttt{e}\cdot\texttt{g}\cdot\epsilon \mid \texttt{i}\cdot\texttt{n}\cdot\epsilon$
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:

$$\frac{s\ \mathsf{str}}{\texttt{ID}[s]\ \mathsf{tok}} \tag{4.1a}$$

$$\frac{n\ \mathsf{nat}}{\texttt{NUM}[n]\ \mathsf{tok}} \tag{4.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{LIT}[s]\ \mathsf{tok}} \tag{4.1c}$$

$$\frac{}{\texttt{LET}\ \mathsf{tok}} \tag{4.1d}$$

$$\frac{}{\texttt{BE}\ \mathsf{tok}} \tag{4.1e}$$

$$\frac{}{\texttt{IN}\ \mathsf{tok}} \tag{4.1f}$$

$$\frac{}{\texttt{ADD}\ \mathsf{tok}} \tag{4.1g}$$

$$\frac{}{\texttt{MUL}\ \mathsf{tok}} \tag{4.1h}$$

$$\frac{}{\texttt{CAT}\ \mathsf{tok}} \tag{4.1i}$$

$$\frac{}{\texttt{LP}\ \mathsf{tok}} \tag{4.1j}$$

$$\frac{}{\texttt{RP}\ \mathsf{tok}} \tag{4.1k}$$

$$\frac{}{\texttt{VB}\ \mathsf{tok}} \tag{4.1l}$$

Rule (4.1a) admits any string as an identifier, even though only certain strings will be treated as identifiers by the lexical analyzer.

Lexical analysis is inductively defined by the following judgement forms:

$s\ \mathsf{charstr} \longleftrightarrow t\ \mathsf{tokstr}$	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

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", s lord states that s is either an alphabetic letter or a digit, and s non-lord states that s does not begin with a letter or digit, and so forth.

$$\frac{}{\epsilon\ \mathsf{charstr} \longleftrightarrow \epsilon\ \mathsf{tokstr}} \tag{4.2a}$$

$$\frac{s = s_1\hat{}s_2\hat{}s_3\ \mathsf{str} \quad s_1\ \mathsf{whs} \quad s_2\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok} \quad s_3\ \mathsf{charstr} \longleftrightarrow ts\ \mathsf{tokstr}}{s\ \mathsf{charstr} \longleftrightarrow t \cdot ts\ \mathsf{tokstr}} \tag{4.2b}$$

$$\frac{s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{4.2c}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{4.2d}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{4.2e}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{4.2f}$$

$$\frac{s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{4.2g}$$

$$\frac{s = \mathtt{l} \cdot \mathtt{e} \cdot \mathtt{t} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \texttt{LET}\ \mathsf{tok}} \tag{4.2h}$$

$$\frac{s = \mathtt{b} \cdot \mathtt{e} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{BE} \ \mathsf{tok}} \tag{4.2i}$$

$$\frac{s = \mathtt{i} \cdot \mathtt{n} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{IN} \ \mathsf{tok}} \tag{4.2j}$$

$$\frac{s = a \cdot s' \ \mathsf{str} \quad a \ \mathsf{ltr} \quad s' \ \mathsf{lds}}{s \ \mathsf{id} \longleftrightarrow \mathtt{ID}[s] \ \mathsf{tok}} \tag{4.2k}$$

$$\frac{s = s_1 \hat{} s_2 \ \mathsf{str} \quad s_1 \ \mathsf{dig} \quad s_2 \ \mathsf{dgs} \quad s \ \mathsf{num} \longleftrightarrow n \ \mathsf{nat}}{s \ \mathsf{num} \longleftrightarrow \mathtt{NUM}[n] \ \mathsf{tok}} \tag{4.2l}$$

$$\frac{s = s_1 \hat{} s_2 \hat{} s_3 \ \mathsf{str} \quad s_1 \ \mathsf{qum} \quad s_2 \ \mathsf{lord} \quad s_3 \ \mathsf{qum}}{s \ \mathsf{lit} \longleftrightarrow \mathtt{LIT}[s_2] \ \mathsf{tok}} \tag{4.2m}$$

$$\frac{s = \mathtt{+} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{ADD} \ \mathsf{tok}} \tag{4.2n}$$

$$\frac{s = \mathtt{*} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{MUL} \ \mathsf{tok}} \tag{4.2o}$$

$$\frac{s = \mathtt{\hat{}} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{CAT} \ \mathsf{tok}} \tag{4.2p}$$

$$\frac{s = \mathtt{(} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{LP} \ \mathsf{tok}} \tag{4.2q}$$

$$\frac{s = \mathtt{)} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{RP} \ \mathsf{tok}} \tag{4.2r}$$

$$\frac{s = \mathtt{|} \cdot \epsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{VB} \ \mathsf{tok}} \tag{4.2s}$$

Rules (4.2) do not specify a *deterministic* algorithm. Rather, Rule (4.2b) applies whenever the input string may be partitioned into three parts, consisting of white space, a lexical item, and the rest of the input. However, the associativity of string concatenation implies that the partitioning is not unique. For example, the string `insert` may be partitioned as `in^sert` or `insert^`ϵ, and hence tokenized as either `IN` followed by `ID[sert]`, or as `ID[insert]` (or, indeed, as two consecutive identifiers in several ways).

One solution to this problem is to impose some extrinsic control criteria on the rules to ensure that they have a unique interpretation. For example, one may insist that Rule (4.2b) apply only when the string s_2 is chosen to be as long as possible so as to ensure that the string `insert` is analyzed as the identifier `ID[insert]`, rather than as two consecutive identifiers, say `ID[ins]` and `ID[ert]`. Moreover, we may impose an ordering on the rules, so that Rule (4.2j) takes priority over Rule (4.2k) to avoid interpreting `in` as an identifier, rather than as a keyword. Another solution is to reformulate the rules so that they are completely deterministic, a technique that will be used in the next section to resolve a similar ambiguity at the level of the concrete syntax.

4.2 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

$$A ::= s_1\, A_1\, s_2\, \ldots\, s_n\, A_n\, s_{n+1} \tag{4.3}$$

we associate an inference rule

$$\frac{s'_1\ A_1 \quad \ldots \quad s'_n\ A_n}{s_1\, s'_1\, s_2\, \ldots\, s_n\, s'_n\, s_{n+1}\ A}\ . \tag{4.4}$$

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:

$$\frac{s'_1\ A_1 \quad \ldots \quad s'_n\ A_n \quad s = s_1 \hat{} s'_1 \hat{} s_2 \hat{} \ldots s_n \hat{} s'_n \hat{} s_{n+1}}{s\ A}\ . \tag{4.5}$$

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 associative, the decomposition is not unique, and so there may be many different ways in which the rule applies.

4.3 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 4.1 on page 37. 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:

$$\frac{s\ \mathsf{num}}{s\ \mathsf{exp}} \tag{4.6a}$$

$$\frac{s\ \mathsf{lit}}{s\ \mathsf{exp}} \tag{4.6b}$$

$$\frac{s\ \mathsf{id}}{s\ \mathsf{exp}} \tag{4.6c}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{ADD}\ s_2\ \mathsf{exp}} \tag{4.6d}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{MUL}\ s_2\ \mathsf{exp}} \tag{4.6e}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{CAT}\ s_2\ \mathsf{exp}} \tag{4.6f}$$

$$\frac{s\ \mathsf{exp}}{\texttt{VB}\ s\ \texttt{VB}\ \mathsf{exp}} \tag{4.6g}$$

$$\frac{s\ \mathsf{exp}}{\texttt{LP}\ s\ \texttt{RP}\ \mathsf{exp}} \tag{4.6h}$$

$$\frac{s_1\ \mathsf{id} \quad s_2\ \mathsf{exp} \quad s_3\ \mathsf{exp}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{exp}} \tag{4.6i}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM}[n]\ \mathsf{num}} \tag{4.6j}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT}[s]\ \mathsf{lit}} \tag{4.6k}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID}[s]\ \mathsf{id}} \tag{4.6l}$$

To emphasize the role of string concatentation, we may rewrite Rule (4.6e), for example, as follows:

$$\frac{s = s_1\, \mathtt{MUL}\, s_2\ \mathsf{str} \quad s_1\ \mathsf{exp} \qquad s_2\ \mathsf{exp}}{s\ \mathsf{exp}}\ . \tag{4.7}$$

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.

4.4 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to avoid 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 (4.6).

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

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

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

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

and

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

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 4.3 on the facing page 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

```
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 avoid ambiguity the grammar of $\mathcal{L}\{\texttt{num str}\}$ given in Section 4.3 on page 42 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 subclasses.

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.

4.5 Notes

The literature on parsing is extensive and highly developed. The account given here is meant only as an overview, and as an exercise in inductive definitions. See any textbook on compilers, such as [3], for a thorough discussion of grammars and parsing.

Chapter 5

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. 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. *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. *Formatting* is the inverse process of generating a linear representation of a given piece of abstract syntax.

5.1 Hierarchical and Binding Structure

For the purposes of analysis the most important elements of the syntax of a language are its *hierarchical* and *binding* structure. Ignoring binding and scope, the hierarchical structure of a language may be expressed using abstract syntax trees. Accounting for these requires the additional structure of abstract binding trees. We will define both an ast and an abt representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ in order to compare the two and show how they relate to the concrete syntax described in Chapter 4.

The purely hierarchical abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is generated by

the following operators and their arities:

`num[`n`]`	$()$	(n nat)
`str[`s`]`	$()$	(s str)
`id[`s`]`	$()$	(s str)
`times`	(Exp, Exp)	
`plus`	(Exp, Exp)	
`len`	(Exp)	
`cat`	(Exp, Exp)	
`let[`s`]`	(Exp, Exp)	(s str)

There is one sort, Exp, generated by the above operators. For each n nat there is an operator `num[`n`]` of arity $()$ representing the number n. Similarly, for each s str there is an operator `str[`s`]` of arity $()$, representing a string literal. There are several operators corresponding to functions on numbers and strings.

Most importantly, there are two operators related to identifiers. The first, `id[`s`]`, where s str, represents the identifier with name s thought of as an operator of arity $()$. The second, `let[`s`]`, is a family of operators indexed by s str with two arguments, the binding of the identifier `id[`s`]` and the scope of that binding. These characterizations, however, are *purely informal* in that there is nothing in the "plain" abstract syntax of the language that supports these interpretations. In particular, there is no connection between any occurrences of `id[`s`]` and any occurrence of `let[`s`]` within an expression.

To account for the binding and scope of identifiers requires the greater expressive power of abstract binding trees. An abt representation of $\mathcal{L}\{\texttt{num str}\}$ is defined by the following operators and their arities:

`num[`n`]`	$()$	(n nat)
`str[`s`]`	$()$	(s str)
`times`	(Exp, Exp)	
`plus`	(Exp, Exp)	
`len`	(Exp)	
`cat`	(Exp, Exp)	
`let`	(Exp, (Exp)Exp)	

There is no longer an operator `id[`s`]`; we instead use a variable to refer to a binding site. Correspondingly, the family of operators `let[`s`]` is repalced replaced by a single operator, `let`, of arity (Exp, (Exp)Exp), which binds a variable in its second argument.

To illustrate the relationship between these two representations of the abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, we will first describe the translation from the concrete syntax, given in Chapter 4, to an abstract syntax tree. We will then alter this translation to account for binding and scope, yielding an abstract binding tree.

5.2 Parsing Into Abstract Syntax Trees

We will simultaneously define parsing and formatting as a binary judgement relating the concrete to the abstract syntax. This judgement will have the mode $(\forall, \exists^{\leq 1})$, 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. It will also have the mode $(\exists, \forall)$, which states that each piece of abstract syntax has a (not necessarily unique) representation as a token string in the concrete syntax.

The parsing judgements for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ follow the unambiguous grammar given in Chapter 4:

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

These judgements relate a token string, s, to an expression, e, viewed as an abstract syntax tree.

These judgements are inductively defined simultaneously by the following rules:

$$\frac{n\ \mathsf{nat}}{\texttt{NUM[}n\texttt{]}\ \mathsf{num} \longleftrightarrow \texttt{num[}n\texttt{]}\ \mathsf{expr}} \tag{5.1a}$$

$$\frac{s\ \mathsf{str}}{\texttt{LIT[}s\texttt{]}\ \mathsf{lit} \longleftrightarrow \texttt{str[}s\texttt{]}\ \mathsf{expr}} \tag{5.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{ID[}s\texttt{]}\ \mathsf{id} \longleftrightarrow \texttt{id[}s\texttt{]}\ \mathsf{expr}} \tag{5.1c}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1d}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1e}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1f}$$

$$\frac{s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}}{\mathtt{LP}\ s\ \mathtt{RP}\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1g}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{trm} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1h}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{trm} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\ \mathtt{MUL}\ s_2\ \mathsf{trm} \longleftrightarrow \mathtt{times}(e_1;e_2)\ \mathsf{expr}} \tag{5.1i}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}}{\mathtt{VB}\ s\ \mathtt{VB}\ \mathsf{trm} \longleftrightarrow \mathtt{len}(e)\ \mathsf{expr}} \tag{5.1j}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{exp} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1k}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\ \mathtt{ADD}\ s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus}(e_1;e_2)\ \mathsf{expr}} \tag{5.1l}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\ \mathtt{CAT}\ s_2\ \mathsf{exp} \longleftrightarrow \mathtt{cat}(e_1;e_2)\ \mathsf{expr}} \tag{5.1m}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}} \tag{5.1n}$$

$$\frac{s_1\ \mathsf{id} \longleftrightarrow \mathtt{id}[s]\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr} \quad s_3\ \mathsf{prg} \longleftrightarrow e_3\ \mathsf{expr}}{\mathtt{LET}\ s_1\ \mathtt{BE}\ s_2\ \mathtt{IN}\ s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}[s](e_2;e_3)\ \mathsf{expr}} \tag{5.1o}$$

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 5.1. *If* s prg $\longleftrightarrow$ e expr, *then* s prg *and* e expr, *and similarly for the other parsing judgements.*

Proof. By a straightforward induction on Rules (5.1). □

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

Theorem 5.2. *If* s prg, *then there is a unique* e *such that* s prg $\longleftrightarrow$ e expr, *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 (4.4) as an inductive definition. □

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

Theorem 5.3. *If e expr, then there exists a (not necessarily unique) string s such that s prg and s prg $\longleftrightarrow$ e expr. That is, the parsing judgement has mode $(\exists, \forall)$.*

Proof. By rule induction on Grammar (4.4). □

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

5.3 Parsing Into Abstract Binding Trees

In this section we revise the parser given in Section 5.2 on page 49 to translate from token strings to abstract binding trees to make explicit the binding and scope of identifiers in a program. The revised parsing judgement, s prg $\longleftrightarrow$ e expr, between strings s and abt's e, is defined by a collection of rules similar to those given in Section 5.2 on page 49. These rules take the form of a generic inductive definition (see Chapter 3) in which the premises and conclusions of the rules involve hypothetical judgments of the form

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

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

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

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:

$$\frac{\Gamma \vdash s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr} \qquad \Gamma \vdash s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr} \qquad \Gamma, s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr} \vdash s_3\ \mathsf{prg} \longleftrightarrow e_3\ \mathsf{expr}}{\Gamma \vdash \texttt{LET}\, s_1\, \texttt{BE}\, s_2\, \texttt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \texttt{let}(e_2; x.e_3)\ \mathsf{expr}} \tag{5.2a}$$

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 $\texttt{ID[}s\texttt{]}\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{expr}$ and

`ID[`s`]` id $\longleftrightarrow$ x_2 expr, for the same identifier `ID[`s`]`. By the structural property of exchange, we may choose arbitrarily which to apply to any particular occurrence of `ID[`s`]`, and hence we may parse different occurrences differently.

To rectify this we 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

$$\Gamma \vdash s \text{ prg} \longleftrightarrow e \text{ expr}$$

is reduced to the basic judgement

$$s \text{ prg} \longleftrightarrow e \text{ expr } [S],$$

where S 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:

$$\frac{\begin{array}{ll} s_1 \text{ id} \longleftrightarrow x\ [S] & s_2 \text{ exp} \longleftrightarrow e_2 \text{ expr } [S] \\ S' = S[s_1 \mapsto x] & s_3 \text{ prg} \longleftrightarrow e_3 \text{ expr } [S'] \end{array}}{\texttt{LET}\, s_1\, \texttt{BE}\, s_2\, \texttt{IN}\, s_3 \text{ prg} \longleftrightarrow \texttt{let}(e_2; x.e_3) \text{ expr } [S]} \tag{5.3}$$

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.

$$\frac{S(\texttt{ID[}s\texttt{]}) = x}{\texttt{ID[}s\texttt{]} \text{ id} \longleftrightarrow x\ [S]} \tag{5.4}$$

The premise of this rule states that S maps the identifier `ID[`s`]` 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 `ID[`$s$`]` is an identifier and x is a variable

name. Using this representation it is straightforward to define the following judgement forms:

$$\begin{aligned} S \ \mathsf{symtab} &\qquad \text{well-formed symbol table} \\ S' = S[\texttt{ID[}s\texttt{]} \mapsto x] &\qquad \text{add new association} \\ S(\texttt{ID[}s\texttt{]}) = x &\qquad \text{lookup identifier} \end{aligned}$$

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

5.4 Notes

As noted in Chapter 4, the theory and practice of translation from concrete to abstract syntax is highly developed. Any standard compiler text, such as [3] provides a thorough account, albeit in a different conceptual framework. The formulation given here is based on course materials developed by Frank Pfenning.

Part III

Statics and Dynamics

Chapter 6

Statics

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 *statics* 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" some aspects of 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 statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

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

6.1 Syntax

When defining a language we shall be primarily concerned with its abstract syntax, specified by a collection of operators and their arities. The abstract syntax provides a systematic, unambiguous account of the hierarchical and binding structure of the language, and is therefore to be considered the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through the trouble to set up a fully precise grammar for it.

We will accomplish both of these purposes with a *syntax chart*, whose meaning is best illustrated by example. The following chart summarizes the abstract and concrete syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, which was analyzed in detail in Chapters 4 and 5.

Typ	τ	::=	`num`	`num`	numbers
			`str`	`str`	strings
Exp	e	::=	x	x	variable
			`num[`n`]`	n	numeral
			`str[`s`]`	$"s"$	literal
			`plus(`e_1`;`e_2`)`	$e_1 + e_2$	addition
			`times(`e_1`;`e_2`)`	$e_1 * e_2$	multiplication
			`cat(`e_1`;`e_2`)`	$e_1 \hat{} e_2$	concatenation
			`len(`e`)`	$\lvert e \rvert$	length
			`let(`e_1`;`x`.`e_2`)`	`let` x `be` e_1 `in` e_2	definition

This chart defines two sorts, Typ, ranged over by τ, and Exp, ranged over by e. The chart defines a number of operators and their arities. For example, the operator `let` has arity $(\mathsf{Exp}, (\mathsf{Exp})\mathsf{Exp})$, which specifies that it has two arguments of sort Exp, and binds a variable of sort Exp in the second argument.

6.2 Type System

The role of a type system 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 statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of an inductive definition of generic hypothetical judgements of the form

$$\vec{x} \mid \Gamma \vdash e : \tau,$$

where $\vec{x}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of variables, using the letters x and y for variables that serve as parameters of the typing judgement. We write $x \notin$

$dom(\Gamma)$ to indicate that there is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

The rules defining the statics of $\mathcal{L}\{\texttt{num str}\}$ are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{6.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{6.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{6.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{6.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \tag{6.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \tag{6.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{6.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2} \tag{6.1h}$$

In Rule (6.1h) we tacitly assume that the variable, x, is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the `let` expression.

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

Lemma 6.1 (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 (6.1). □

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 6.2 (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 $\Gamma \vdash e : \tau$. □

In richer languages such inversion principles are more difficult to state and to prove.

6.3 Structural Properties

The statics enjoys the structural properties of the generic hypothetical judgement.

Lemma 6.3 (Weakening). *If* $\Gamma \vdash e' : \tau'$, *then* $\Gamma, x : \tau \vdash e' : \tau'$ *for any* $x \notin dom(\Gamma)$ *and any type* τ.

Proof. By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (6.1h). 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 $z \notin dom(\Gamma)$ and $z \neq x$. 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 (6.1h). □

Lemma 6.4 (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 (6.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where z may be chosen so that $z \neq x$ and $z \notin dom(\Gamma)$. We have by induction and Lemma 6.3 that

1. $\Gamma \vdash [e/x]e_1 : \tau_1$,
2. $\Gamma, z : \tau_1 \vdash [e/x]e_2 : \tau'$.

By the choice of z 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 (6.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 6.3 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 6.4 on the facing 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 6.4 on the preceding 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 6.5 (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 when the variable x occurs more than once in e', because then one copy of e suffices for all occurrences of x in e'.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ given by Rules (6.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introductory* and the *eliminatory*. The introductory forms for a type determine the *values*, or *canonical forms*, of that type. The eliminatory forms determine how to manipulate the values of a type to form a computation of another (possibly the same) type. In $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the introductory forms for the type `num` are the numerals, and those for the type `str` are the literals. The eliminatory forms for the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification will become apparent once we have defined the dynamics of the language in Chapter 7. Then we will see that

[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.

the eliminatory forms are *inverse* to the introductory forms in that they "take apart" what the introductory forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 8.

6.4 Notes

The concept of the static semantics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. The concept of a static semantics in the sense considered here was introduced in the definition of the Standard ML programming language [67], building on much earlier work by Church and others on the typed λ-calculus [11]. The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction [31]. These principles were first explicitly applied to programming languages by Martin-Löf [61, 75].

Chapter 7

Dynamics

The *dynamics* of a language is a description of how programs are to be executed. The most important way to define the dynamics of a language is by the method of *structural dynamics*, which defines a *transition system* that inductively specifies the step-by-step process of executing a program. Another method for presenting dynamics, called *contextual dynamics*, is a variation of structural dynamics in which the transition rules are specified in a slightly different manner. An *equational dynamics* presents the dynamics of a language equationally by a collection of rules for deducing when one program is *definitionally equivalent* to another.

7.1 Transition Systems

A *transition system* is specified by the following four forms of judgment:

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'.

In practice we always arrange things so that no transition is possible from a final state: if s final, then there is no s' state such that $s \mapsto s'$. A state from which no transition is possible is sometimes said to be *stuck*. Whereas all final states are, by convention, stuck, there may be stuck states in a transition system that are not final. 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 *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 there is no s such that $s_n \mapsto s$, 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. The judgement $s \downarrow$ means that there is a complete transition sequence starting from s, which is to say that there exists s' final such that $s \mapsto^* s'$.

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

$$\frac{}{s \mapsto^* s} \tag{7.1a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''} \tag{7.1b}$$

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

When applied to the definition of iterated transition, the principle of rule induction 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 *closed under inverse 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.

$$\frac{}{s \mapsto^0 s} \tag{7.2a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^n s''}{s \mapsto^{n+1} s''} \tag{7.2b}$$

Theorem 7.1. *For all states s and s', $s \mapsto^* s'$ iff $s \mapsto^k s'$ for some $k \geq 0$.*

7.2 Structural Dynamics

A *structural dynamics* for $\mathcal{L}\{\texttt{num str}\}$ is given by a transition system whose states are closed expressions. All states are initial. The final states are the

(closed) values, which represent the completed computations. The judgement e val, which states that e is a value, is inductively defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{7.3a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{7.3b}$$

The transition judgement, $e \mapsto e'$, between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{7.4a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}} \tag{7.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}} \tag{7.4c}$$

$$\frac{s_1\,\hat{}\,s_2 = s\ \mathsf{str}}{\texttt{cat(str[}s_1\texttt{];str[}s_2\texttt{])} \mapsto \texttt{str[}s\texttt{]}} \tag{7.4d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1'\texttt{;}e_2\texttt{)}} \tag{7.4e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1\texttt{;}e_2'\texttt{)}} \tag{7.4f}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{7.4g}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (7.4a), (7.4d), and (7.4g) 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.

Rule (7.4g) 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:

$$\frac{e_1\ \mathsf{val}}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{7.5a}$$

[1] The justification for the terminology "by name" is obscure, but the terminology is firmly established and cannot be changed.

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x.e_2) \mapsto \texttt{let}(e_1';x.e_2)} \tag{7.5b}$$

Rule (7.5b) is an additional search rule specifying that we may evaluate e_1 before e_2. Rule (7.5a) ensures that e_2 is not evaluated until evaluation of e_1 is complete.

A derivation sequence in a structural dynamics 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.

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

Each step in this sequence of transitions is justified by a derivation according to Rules (7.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus}(\texttt{num}[3];\texttt{num}[3]) \mapsto \texttt{num}[6]}\ (7.4a)}{\texttt{plus}(\texttt{plus}(\texttt{num}[3];\texttt{num}[3]);\texttt{num}[4]) \mapsto \texttt{plus}(\texttt{num}[6];\texttt{num}[4])}\ (7.4b)$$

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

The principle of rule induction for the structural dynamics 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 (7.4). For example, we may show by rule induction that structural dynamics of $\mathcal{L}\{\texttt{num str}\}$ is *determinate*, which means that an expression may transition to at most one other expression. The proof a simple lemma relating transition to values:

Lemma 7.2. *For no expression e do we have both e val and $e \mapsto e'$ for some e'.*

Proof. By rule induction on Rules (7.3) and (7.4). □

Lemma 7.3 (Determinacy)**.** *If $e \mapsto e'$ and $e \mapsto e''$, then e' and e'' are α-equivalent.*

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. It is assumed that the primitive operators, such as addition, have a unique value when applied to values. □

Rules (7.4) exemplify the *inversion principle* of language design, which states that the eliminatory forms are *inverse* to the introductory forms of a language. The search rules determine the *principal arguments* of each eliminatory form, and the instruction rules specify how to evaluate an eliminatory form when all of its principal arguments are in introductory form. For example, Rules (7.4) specify that both argument of addition are principal, and specify how to evaluate an addition once its principal arguments are evaluated to numerals. The inversion principle is central to ensuring that a programming language is properly defined, the exact statement of which is given in Chapter 8.

7.3 Contextual Dynamics

A variant of structural dynamics, called *contextual dynamics*, 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.

$$\frac{m + n = p \text{ nat}}{\texttt{plus(num[}m\texttt{];num[}n\texttt{])} \rightsquigarrow \texttt{num[}p\texttt{]}} \tag{7.6a}$$

$$\frac{s \hat{} t = u \text{ str}}{\texttt{cat(str[}s\texttt{];str[}t\texttt{])} \rightsquigarrow \texttt{str[}u\texttt{]}} \tag{7.6b}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \rightsquigarrow [e_1/x]e_2} \tag{7.6c}$$

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.)

$$\frac{}{\circ \text{ ectxt}} \tag{7.7a}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\texttt{plus(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \text{ ectxt}} \tag{7.7b}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\texttt{plus}(e_1;\mathcal{E}_2) \text{ ectxt}} \tag{7.7c}$$

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 dynamics. For example, Rule (7.7c) states that in an expression $\texttt{plus}(e_1;e_2)$, if the first argument, e_1, is a value, then the next instruction step, if any, lies at or within the second 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:

$$\frac{}{e = \circ\{e\}} \tag{7.8a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(\mathcal{E}_1;e_2)\{e\}} \tag{7.8b}$$

$$\frac{e_1 \text{ val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(e_1;\mathcal{E}_2)\{e\}} \tag{7.8c}$$

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 contextual dynamics for $\mathcal{L}\{\texttt{num str}\}$ is defined by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{7.9}$$

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 dynamics 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 dynamics (Rules (7.4)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual dynamics (Rules (7.9)).

Theorem 7.4. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof. From left to right, proceed by rule induction on Rules (7.4). 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 (7.4a), take $\mathcal{E} = \circ$, and

observe that $e \rightsquigarrow e'$. For Rule (7.4b), 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 $\mathcal{E} = \texttt{plus}(\mathcal{E}_1;e_2)$, and observe that $e = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0\}$ and $e' = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0'\}$ with $e_0 \rightsquigarrow e_0'$.

From right to left, observe that if $e \mapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. We prove by induction on Rules (7.8) that $e \mapsto_{\mathsf{s}} e'$. For example, for Rule (7.8a), e_0 is e, e_0' is e', and $e \rightsquigarrow e'$. Hence $e \mapsto_{\mathsf{s}} e'$. For Rule (7.8b), we have that $\mathcal{E} = \texttt{plus}(\mathcal{E}_1;e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathsf{s}} e_1'$. Therefore e is $\texttt{plus}(e_1;e_2)$, e' is $\texttt{plus}(e_1';e_2)$, and therefore by Rule (7.4b), $e \mapsto_{\mathsf{s}} e'$.

□

Since the two transition judgements coincide, contextual dynamics may be seen as an alternative way of presenting a structural dynamics. It has two advantages over structural dynamics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (7.9) in the simpler form

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}\ . \tag{7.10}$$

This formulation is superficially simpler in that it does not make explicit how an expression is to be decomposed into an evaluation context and a reducible expression.

7.4 Equational Dynamics

Another formulation of the dynamics 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 $\mathcal{L}\{\texttt{num str}\}$, 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 expressions in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{7.11a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{7.11b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{7.11c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1;e_2) \equiv \texttt{plus}(e_1';e_2') : \texttt{num}} \tag{7.11d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1;e_2) \equiv \texttt{cat}(e_1';e_2') : \texttt{str}} \tag{7.11e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{let}(e_1;x\,.\,e_2) \equiv \texttt{let}(e_1';x\,.\,e_2') : \tau_2} \tag{7.11f}$$

$$\frac{n_1 + n_2 = n \ \mathsf{nat}}{\Gamma \vdash \texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) \equiv \texttt{num}[n] : \texttt{num}} \tag{7.11g}$$

$$\frac{s_1 \hat{} s_2 = s \ \mathsf{str}}{\Gamma \vdash \texttt{cat}(\texttt{str}[s_1];\texttt{str}[s_2]) \equiv \texttt{str}[s] : \texttt{str}} \tag{7.11h}$$

$$\frac{}{\Gamma \vdash \texttt{let}(e_1;x\,.\,e_2) \equiv [e_1/x]e_2 : \tau} \tag{7.11i}$$

Rules (7.11a) through (7.11c) state that definitional equivalence is an *equivalence relation*. Rules (7.11d) through (7.11f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (7.11g) through (7.11i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. For the sake of concision, Rules (7.11) may be characterized as defining the *strongest congruence* closed under Rules (7.11g), (7.11h), and (7.11i).

Rules (7.11) 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 (7.11). 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 (7.11). A prototypical example is the putative equivalence

$$x : \mathtt{num}, y : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{7.12}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from Rules (7.11). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{7.13}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (7.12), and all of its instances, given by Equation (7.13), may be filled by enriching the notion of equivalence to include a principle 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 dynamics of the expressions involved. (Semantic equivalence is developed rigorously for a related language in Chapter 49.)

Definitional equivalence is sometimes called *symbolic execution*, since it allows any subexpression to be replaced by the result of evaluating it according to the rules of the dynamics of the language.

Theorem 7.5. *$e \equiv e' : \tau$ 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

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

and

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

This is proved by rule induction on Rules (7.11). □

The formulation of definitional equivalence for the by-value dynamics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (7.11i) 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:

$$\frac{e_1 \ \mathsf{val}}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{7.14}$$

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 (7.11f) as follows:

$$\frac{\Xi\,\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Xi, x \ \mathsf{val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{7.15}$$

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

7.5 Notes

The use of transition systems to specify the behavior of programs goes back to the early work of Church and Turing on computability. Turing's approach emphasized the concept of an abstract machine consisting of a finite program together with unbounded memory. Computation proceeds by changing the memory in accordance with the instructions in the program. Much early work on the operational semantics of programming languages, such as Landin's SECD machine [53], emphasized machine models. Church's approach emphasized the language for expressing computations, and defined execution in terms of the programs themselves, rather than in terms of auxiliary concepts such as memories or tapes. Plotkin's elegant formulation of structural operational semantics [82], which we use heavily throughout this book, was inspired by Church's and Landin's ideas [84].

Contextual semantics, which was introduced by Felleisen [29], may be seen as an alternative formulation of structural semantics in which "search rules" are replaced by "context matching". Computation viewed as equational deduction goes back to the early work of Herbrand and Gödel. (In fact, this style of dynamics is sometimes called *Herbrand-Gödel equations*). Church's original formulation of the λ-calculus [21] was based on equational deduction of the kind considered here.

Chapter 8

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 statics and the dynamics. The statics may be seen as predicting that the value of an expression will have a certain form so that the dynamics 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 8.1 (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.

8.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ defined in Chapters 6 and 7 is proved by rule induction on the transition system (rules (7.4)).

Theorem 8.2 (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 (7.4b),

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

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 (7.4g),

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

Assume that $\texttt{let}(e_1;x.e_2) : \tau_2$. By the inversion lemma 6.2 on page 59, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 6.4 on page 60 $[e_1/x]e_2 : \tau_2$, as desired.

It is easy to check that the primitive operations are all type-preserving; for example, if a nat and b nat and $a + b = c$ nat, then c nat.

□

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.

8.2 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 8.3 (Canonical Forms)**.** *If e val and $e : \tau$, then*

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

Proof. By induction on rules (6.1) and (7.3). □

Progress is proved by rule induction on rules (6.1) defining the statics of the language.

Theorem 8.4 (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 (6.1d),

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

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 $\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1'; e_2)$, as required. In the former we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \mapsto e_2'$. In the latter case we have that $\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1; e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 8.3, $e_1 = \texttt{num}[n_1]$ and $e_2 = \texttt{num}[n_2]$, and hence

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

□

Since the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of e, appealing to the inversion theorem at each step to characterize the types of the parts of e. But this approach breaks down when the typing rules are not syntax-directed, that is, when there may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

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

8.3 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. This is because one cannot reliably predict statically whether an expression will turn out to be non-zero when executed (because this is an undecidable property). 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 dynamics 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 dynamics 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:

$$\frac{e_1 \ \mathsf{val}}{\texttt{div}(e_1;\texttt{num[0]}) \ \mathsf{err}} \tag{8.1a}$$

$$\frac{e_1 \ \mathsf{err}}{\mathtt{plus}(e_1;e_2) \ \mathsf{err}} \tag{8.1b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{err}}{\mathtt{plus}(e_1;e_2) \ \mathsf{err}} \tag{8.1c}$$

Rule (8.1a) 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.

Once the error judgement is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{8.2a}$$

$$\frac{}{\mathtt{error} \ \mathsf{err}} \tag{8.2b}$$

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 8.5 (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. □

8.4 Notes

The concept of type safety as it is understood today was first formulated by Milner in his study of the ML type system [65]. This work gave rise to the slogan "well-typed programs cannot go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen observed that one can instead show that ill-defined states cannot arise in a well-typed program [105], giving rise to the slogan "well-typed programs cannot get stuck." However, their formulation relied on a backward analysis showing that a stuck state cannot be well-typed. The formulation given here reformulates this as the progress theorem, which itself relies on the concept of canonical forms of a type introduced by Martin-Löf [75]. The informal concept of type safety is therefore formalized as the conjunction of progress and preservation, which ensure that the state of the dynamics is always well-defined.

Chapter 9

Evaluation Dynamics

In Chapter 7 we defined the evaluation of $\mathcal{L}\{\texttt{num str}\}$ expression using the method of structural dynamics. 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 dynamics* presents the dynamics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Evaluation dynamics suppresses the step-by-step details of determining the value of an expression, and hence does not provide any useful notion of the time complexity of a program. *Cost dynamics* rectifies this by augmenting evaluation dynamics with a *cost measure*. Various cost measures may be assigned to an expression. One example is the number of steps in the structural dynamics required for an expression to reach a value.

9.1 Evaluation Dynamics

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

$$\frac{}{\texttt{num}[n] \Downarrow \texttt{num}[n]} \tag{9.1a}$$

$$\frac{}{\texttt{str}[s] \Downarrow \texttt{str}[s]} \tag{9.1b}$$

$$\frac{e_1 \Downarrow \texttt{num}[n_1] \quad e_2 \Downarrow \texttt{num}[n_2] \quad n_1 + n_2 = n \ \mathsf{nat}}{\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]} \tag{9.1c}$$

$$\frac{e_1 \Downarrow \texttt{str}[s_1] \quad e_2 \Downarrow \texttt{str}[s_2] \quad s_1 \hat{} s_2 = s \ \mathsf{str}}{\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s]} \tag{9.1d}$$

$$\frac{e \Downarrow \mathtt{str}[s] \quad |s| = n \ \mathsf{str}}{\mathtt{len}(e) \Downarrow \mathtt{num}[n]} \tag{9.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow v_2} \tag{9.1f}$$

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 (9.1f) 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 (9.1):

1. Show that $\mathcal{P}(\mathtt{num}[n] \Downarrow \mathtt{num}[n])$.
2. Show that $\mathcal{P}(\mathtt{str}[s] \Downarrow \mathtt{str}[s])$.
3. Show that $\mathcal{P}(\mathtt{plus}(e_1; e_2) \Downarrow \mathtt{num}[n])$, if $\mathcal{P}(e_1 \Downarrow \mathtt{num}[n_1])$, $\mathcal{P}(e_2 \Downarrow \mathtt{num}[n_2])$, and $n_1 + n_2 = n$ nat.
4. Show that $\mathcal{P}(\mathtt{cat}(e_1; e_2) \Downarrow \mathtt{str}[s])$, if $\mathcal{P}(e_1 \Downarrow \mathtt{str}[s_1])$, $\mathcal{P}(e_2 \Downarrow \mathtt{str}[s_2])$, and $s_1 \hat{} s_2 = s$ str.
5. Show that $\mathcal{P}(\mathtt{let}(e_1; x.e_2) \Downarrow v_2)$, 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 9.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on Rules (9.1). All cases except Rule (9.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

9.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$. 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 structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics 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 structural dynamics and the evaluation judgement in the evaluation dynamics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

Theorem 9.2. *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 9.3. *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 7 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 reverse execution.

Lemma 9.4. *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. □

9.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 8.1 on page 75) 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 dynamics. But what if we had instead specified the dynamics 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 dynamics, 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 8.3 on page 78), 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 dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics 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 form of error solely to prove that it cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation dynamics.

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:

$$\frac{}{\texttt{plus(str[}s\texttt{];}e_2\texttt{)}\Uparrow} \tag{9.2a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus(}e_1\texttt{;str[}s\texttt{])}\Uparrow} \tag{9.2b}$$

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

Theorem 9.5. *If $e\Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (9.2). For example, for Rule (9.2a), we observe that $\texttt{str[}s\texttt{]} : \texttt{str}$, and hence $\texttt{plus(str[}s\texttt{];}e_2\texttt{)}$ is ill-typed. □

Corollary 9.6. *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 9.5 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 structural dynamics 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 statics 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 dynamics and the implementation.

9.4 Cost Dynamics

A structural dynamics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation dynamics, 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 dynamics*.

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

$$\frac{}{\mathtt{num}[n] \Downarrow^0 \mathtt{num}[n]} \tag{9.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num}[n_1] \quad e_2 \Downarrow^{k_2} \mathtt{num}[n_2]}{\mathtt{plus}(e_1;e_2) \Downarrow^{k_1+k_2+1} \mathtt{num}[n_1 + n_2]} \tag{9.3b}$$

$$\frac{}{\mathtt{str}[s] \Downarrow^0 \mathtt{str}[s]} \tag{9.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\mathtt{cat}(e_1;e_2) \Downarrow^{k_1+k_2+1} \mathtt{str}[s_1 \hat{} s_2]} \tag{9.3d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{9.3e}$$

Theorem 9.7. *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 dynamics. From right to left proceed by induction on k, with an inner rule induction on the definition of the structural dynamics. □

9.5 Notes

The structural similarity between evaluation dynamics and typing rules was first developed in the definition of Standard ML [67]. Martin-Löf's formulation of type theory as a programming language [75] also stressed evaluation dynamics to define computation. The advantage of evaluation semantics is that it directly defines the relation of interest, that between a program and its outcome. The disadvantage is that it is not as well-suited to metatheory as structural semantics, precisely because it glosses over the fine structure of computation. The concept of a cost dynamics was introduced by Blelloch and Greiner in their study of parallel computation [14, 36]. The sequential cost semantics given here sets the stage for the treatment of parallelism in Chapter 41.

Part IV

Function Types

Chapter 10

Function Definitions and Values

In the language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ we may perform calculations such as the doubling of a given expression, but we cannot express doubling as a concept in itself. To capture the general pattern of doubling, we abstract away from the particular number being doubled using a *variable* to stand for a fixed, but unspecified, number, to express the doubling of an arbitrary number. Any particular instance of doubling may then be obtained by substituting a numeric expression for that variable. In general an expression may involve many distinct variables, necessitating that we specify which of several possible variables is varying in a particular context, giving rise to a *function* of that variable.

In this chapter we will consider two extensions of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with functions. The first, and perhaps most obvious, extension is by adding *function definitions* to the language. A function is defined by binding a name to an abt with a bound variable that serves as the argument of that function. A function is *applied* by substituting a particular expression (of suitable type) for the bound variable, obtaining an expression.

The domain and range of defined functions are limited to the types `nat` and `str`, since these are the only types of expression. Such functions are called *first-order* functions, in contrast to *higher-order functions*, which permit functions as arguments and results of other functions. Since the domain and range of a function are types, this requires that we introduce *function types* whose elements are functions. Consequently, we may form functions of *higher type*, those whose domain and range may themselves be function types.

Historically the introduction of higher-order functions was responsible for a mistake in language design that subsequently was re-characterized as a feature, called *dynamic binding*. Dynamic binding arises from getting the definition of substitution wrong by failing to avoid capture. This makes the names of bound variables important, in violation of the fundamental principle of binding stating that the names of bound variables are unimportant.

10.1 First-Order Functions

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

Exp	e	::=	$\texttt{call}[f](e)$	$f(e)$	call
			$\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e)$	$\texttt{fun}\,f(x_1:\tau_1):\tau_2 = e_2\,\texttt{in}\,e$	definition

The expression $\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e)$ binds the function name f within e to the pattern $x_1.e_2$, 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 $\texttt{call}[f](e)$ instantiates the binding of f with the argument e.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ defines two forms of judgement:

1. Expression typing, $e : \tau$, stating that e has type τ;
2. Function typing, $f(\tau_1) : \tau_2$, stating that f is a function with argument type τ_1 and result type τ_2.

The judgment $f(\tau_1) : \tau_2$ is called the *function header* of f; it specifies the domain type and the range type of a function.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ is defined by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e) : \tau} \tag{10.1a}$$

$$\frac{\Gamma \vdash f(\tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{call}[f](e) : \tau_2} \tag{10.1b}$$

Function substitution, written $[\![x.e/f]\!]e'$, is defined by induction on the structure of e' much like the definition of ordinary substitution. However, a function name, f, is not a form of expression, but rather can only occur in

a call of the form `call[`f`](`e`)`. Function substitution for such expressions is defined by the following rule:

$$\frac{}{[\![x.e/f]\!]\texttt{call[}f\texttt{](}e'\texttt{)} = \texttt{let(}[\![x.e/f]\!]e'\texttt{;}x.e\texttt{)}} \tag{10.2}$$

At call sites to f with argument e', function substitution yields a `let` expression that binds x to the result of expanding any further calls to f within e'.

Lemma 10.1. *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 dynamics of $\mathcal{L}\{\texttt{num str fun}\}$ is defined using function substitution:

$$\frac{}{\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1.e_2\texttt{;}f.e\texttt{)} \mapsto [\![x_1.e_2/f]\!]e} \tag{10.3}$$

Since function substitution replaces all calls to f by appropriate `let` expressions, there is no need to give a rule for function calls.

The safety of $\mathcal{L}\{\texttt{num str fun}\}$ may be obtained as an immediate corollary of the safety theorem for higher-order functions, which we discuss next.

10.2 Higher-Order Functions

The syntactic and semantic similarity between variable definitions and function definitions in $\mathcal{L}\{\texttt{num str fun}\}$ 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 `lam[`τ_1`](`$x.e$`)`. Correspondingly, we must generalize application to have the form `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*, `arr(`τ_1`;`τ_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:

Typ	τ	::=	$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
Exp	e	::=	$\texttt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
			$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$	application

Functions are now "first class" in the sense that a function is an expression of function type.

The statics of $\mathcal{L}\{\texttt{num str} \to\}$ is given by extending Rules (6.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{10.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{10.4b}$$

Lemma 10.2 (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 10.3 (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 dynamics of $\mathcal{L}\{\texttt{num str} \to\}$ extends that of $\mathcal{L}\{\texttt{num str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \texttt{val}} \tag{10.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{10.5b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1} \tag{10.5c}$$

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

Proof. The proof is by induction on rules (10.5), which define the dynamics of the language.

Consider rule (10.5c),

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

Suppose that $\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) : \tau_1$. By Lemma 10.2 on the facing page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 10.3 on the preceding page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 10.5 (Canonical Forms). *If e val and $e : \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam}[\tau_1](x.e_2)$ 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 10.6 (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 (10.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (10.4b). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)$. In the former case, we have by Lemma 10.5 that $e_1 = \mathtt{lam}[\tau_2](x.e)$ for some x and e. But then $\mathtt{ap}(e_1;e_2) \mapsto [e_2/x]e$. □

10.3 Evaluation Dynamics and Definitional Equivalence

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

$$\frac{}{\mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)} \tag{10.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{10.6b}$$

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

Theorem 10.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (10.6). 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 (7.1). 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 (10.5). □

Definitional equivalence for the call-by-name dynamics of $\mathcal{L}\{\mathtt{num}\ \mathtt{str} \to\}$ is defined by a straightforward extension to Rules (7.11).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{10.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2') : \tau} \tag{10.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{10.7c}$$

Definitional equivalence for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (10.7a) to require that the argument be a value. However, to be fully expressive, we must also widen the concept of a value to include all variables that are in scope, so that Rule (10.7a) would apply even when the argument is a variable. The justification for this is that in call-by-value, the parameter of a function stands for the value of its argument, and not for the argument itself. The call-by-value definitional equivalence judgement has the form

$$\Xi\, \Gamma \vdash e_1 \equiv e_2 : \tau,$$

where Ξ is the finite set of hypotheses $x_1\ \mathsf{val}, \ldots, x_k\ \mathsf{val}$ governing the variables in scope at that point. We write $\Xi \vdash e\ \mathsf{val}$ to indicate that e is a value under these hypotheses, so that, for example, $\Xi, x\ \mathsf{val} \vdash x\ \mathsf{val}$.

The rule of definitional equivalence for call-by-value are similar to those for call-by-name, modified to take account of the scopes of value variables. Two illustrative rules are as follows:

$$\frac{\Xi, x\ \mathsf{val}\, \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\, \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{10.8a}$$

$$\frac{\Xi \vdash e_1\ \mathsf{val}}{\Xi\, \Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau}\ . \tag{10.8b}$$

10.4 Dynamic Scope

The dynamics of function application given by Rules (10.5) is defined only for expressions without free variables. When a function is called, the argument is substituted for the function parameter, ensuring that the result remains closed. Moreover, since substitution of closed expressions can never incur capture, the scopes of variables are not disturbed by the dynamics, ensuring that the principles of binding and scope described in Chapter 1 are respected. This treatment of variables is called *static scoping*, or *static binding*, to contrast it with an alternative approach that we now describe.

Another approach, called *dynamic scoping*, or *dynamic binding*, is sometimes advocated as an alternative to static binding. Evaluation is defined for expressions that may contain free variables. Evaluation of a variable is undefined; it is an error to ask for the value of an unbound variable. Function call is defined similarly to dynamic binding, *except* that when a function is called, the argument *replaces* the parameter in the body, possibly *incurring*, rather than avoiding, capture of free variables in the argument. (As we will explain shortly, this behavior is considered to be a feature, not a bug!)

The difference between replacement and substitution may be illustrated by example. Let e be the expression $\lambda\,(x{:}\mathtt{str}.\,y + |x|)$ in which the variable y occurs free, and let e' be the expression $\lambda\,(y{:}\mathtt{str}.\,f(y))$ with free variable f. If we *substitute* e for f in e' we obtain an expression of the form

$$\lambda\,(y'{:}\mathtt{str}.\,\lambda\,(x{:}\mathtt{str}.\,y + |x|)(y')),$$

where the bound variable, y, in e has been renamed to some fresh variable y' so as to avoid capture. If we instead *replace* f by e in e' we obtain

$$\lambda\,(y{:}\mathtt{str}.\,\lambda\,(x{:}\mathtt{str}.\,y + |x|)(y))$$

in which y is no longer free: it has been captured during replacement.

The implications of this seemingly small change to the dynamics of $\mathcal{L}\{\to\}$ are far-reaching. The most obvious implication is that the language is not type safe. In the above example we have that $y : \mathtt{nat} \vdash e : \mathtt{str} \to \mathtt{nat}$, and that $f : \mathtt{str} \to \mathtt{nat} \vdash e' : \mathtt{str} \to \mathtt{nat}$. It follows that $y : \mathtt{nat} \vdash [e/f]e' : \mathtt{str} \to \mathtt{nat}$, but it is easy to see that the result of replacing f by e in e' is ill-typed, regardless of what assumption we make about y. The difficulty, of course, is that the bound occurrence of y in e' has type $\mathtt{str}$, whereas the free occurrence in e must have type $\mathtt{nat}$ in order for e to be well-formed.

One way around this difficulty is to ignore types altogether, and rely on run-time checks to ensure that bad things do not happen, despite the

evident failure of safety. (See Chapter 20 for a full exploration of this approach.) But even if we ignore worries about safety, we are still left with the serious problem that the names of bound variables matter, and cannot be altered without changing the meaning of a program. So, for example, to use expression e', one must bear in mind that the parameter, f, occurs within the scope of a binder for y, a fact that is not revealed by the type of e' (and certainly not if one disregards types entirely!) If we change e' so that it binds a different variable, say z, then we must correspondingly change e to ensure that it refers to z, and not y, in order to preserve the overall behavior of the system of two expressions. This means that e and e' must be developed in tandem, violating a basic principle of modular decomposition. (For more on dynamic scope, please see Chapter 35.)

10.5 Notes

Nearly all programming languages provide some form of function definition mechanism of the kind illustrated here. The main point of the present account is to demonstrate that a more natural, and more powerful, approach is to separate the generic concept of a definition from the specific concept of a function. Function types codify the general notion in a systematic manner that encompasses function definitions as a special case, and moreover, admits passing functions as arguments and returning them as results without special provision. The essential contribution of Church's λ-calculus [21] was to take the notion of function as primary, and indeed to point out that nothing more is needed to obtain a fully expressive programming language. The defining feature of *functional* programming languages is precisely that functions are first-class values that can be handled without special provision or restriction. This, too, is a central feature of *object-oriented* languages: objects consist of methods acting on private data, and are nothing more than functions combined with storage effects.

Chapter 11

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*, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an intrinsic termination proof for each program in the language. Consequently, we may only define *total* functions in the language, those that always return a value for each argument. In essence every program in $\mathcal{L}\{\mathtt{nat} \to\}$ "comes equipped" with a proof of its termination. While this may seem like a shield against infinite loops, it is also a weapon that can be used to show that some programs cannot be written in $\mathcal{L}\{\mathtt{nat} \to\}$. To do so would require a master termination proof for every possible program in the language, something that we shall prove does not exist.

11.1 Statics

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

Typ	τ	::=	$\mathtt{nat}$	$\mathtt{nat}$	naturals
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
Exp	e	::=	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{natrec}(e;e_0;x.y.e_1)$	$\mathtt{natrec}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}$	recursion
			$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.\,e)$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application

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 $\mathtt{natrec}(e;e_0;x.y.e_1)$ 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 $\mathtt{natiter}(e;e_0;y.e_1)$, 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 13) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

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

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}} \tag{11.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{natrec}(e;e_0;x.y.e_1) : \tau} \tag{11.1d}$$

$$\frac{\Gamma, x : \rho \vdash e : \tau}{\Gamma \vdash \mathtt{lam}[\rho](x.e) : \mathtt{arr}(\rho;\tau)} \tag{11.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{11.1f}$$

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

Lemma 11.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

11.2 Dynamics

The dynamics of $\mathcal{L}\{\mathtt{nat} \to\}$ 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 $\mathcal{L}\{\mathtt{nat} \to\}$ are determined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{11.2a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}} \tag{11.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{11.2c}$$

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

$$\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')} \tag{11.3a}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{11.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{11.3c}$$

$$\frac{e \mapsto e'}{\mathtt{natrec}(e;e_0;x.y.e_1) \mapsto \mathtt{natrec}(e';e_0;x.y.e_1)} \tag{11.3d}$$

$$\frac{}{\mathtt{natrec}(\mathtt{z};e_0;x.y.e_1) \mapsto e_0} \tag{11.3e}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{natrec}(\mathtt{s}(e);e_0;x.y.e_1) \mapsto [e,\mathtt{natrec}(e;e_0;x.y.e_1)/x,y]e_1} \tag{11.3f}$$

Rules (11.3e) and (11.3f) specify the behavior of the recursor on `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 11.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

1. *If $\tau = \texttt{nat}$, then $e = \texttt{s(s(}\ldots\texttt{z))}$ 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 11.3 (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'*

11.3 Definability

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ on the natural numbers is *definable* in $\mathcal{L}\{\texttt{nat} \to\}$ iff there exists an expression e_f of type $\texttt{nat} \to \texttt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{n}) \equiv \overline{f(n)} : \texttt{nat}. \tag{11.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is an expression e_f of type $\texttt{nat} \to \texttt{nat}$ such that, when applied to the numeral representing the argument $n \in \mathbb{N}$, is definitionally equivalent to the numeral corresponding to $f(n) \in \mathbb{N}$.

Definitional equivalence for $\mathcal{L}\{\texttt{nat} \to\}$, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{}{\Gamma \vdash \texttt{ap(lam[}\tau\texttt{]}(x.e_2)\texttt{;}e_1\texttt{)} \equiv [e_1/x]e_2 : \tau} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \texttt{natrec(z;}e_0\texttt{;}x.y.e_1\texttt{)} \equiv e_0 : \tau} \tag{11.5b}$$

$$\frac{}{\Gamma \vdash \texttt{natrec(s(}e\texttt{)};}e_0\texttt{;}x.y.e_1\texttt{)} \equiv [e,\texttt{natrec(}e\texttt{;}e_0\texttt{;}x.y.e_1\texttt{)}/x,y]e_1 : \tau} \tag{11.5c}$$

For example, the doubling function, $d(n) = 2 \times n$, is definable in $\mathcal{L}\{\texttt{nat} \to\}$ by the expression $e_d : \texttt{nat} \to \texttt{nat}$ given by

$$\lambda\ (x{:}\texttt{nat}.\,\texttt{natrec}\, x\, \{\texttt{z} \Rightarrow \texttt{z} \mid \texttt{s(}u\texttt{)}\ \texttt{with}\ v \Rightarrow \texttt{s(s(}v\texttt{))}\}).$$

To check that this defines the doubling function, we proceed by induction on $n \in \mathbb{N}$. For the basis, it is easy to check that

$$e_d(\overline{0}) \equiv \overline{0} : \texttt{nat}.$$

For the induction, assume that

$$e_d(\overline{n}) \equiv \overline{d(n)} : \mathtt{nat}.$$

Then calculate using the rules of definitional equivalence:

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

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 that it is not first-order primitive recursive, but is higher-order primitive recursive. The key is to showing that it is definable in $\mathcal{L}\{\mathtt{nat} \to\}$ 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

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

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

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

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

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{11.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{11.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{11.8}$$

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

11.4 Undefinability

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

Theorem 11.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 49.11 on page 501. □

Consequently, values of function type in $\mathcal{L}\{\mathtt{nat} \to\}$ behave like mathematical functions: if $f : \rho \to \tau$ and $e : \rho$, then $f(e)$ evaluates to a value of type τ. Moreover, if $e : \mathtt{nat}$, then there exists a natural number n such that $e \equiv \overline{n} : \mathtt{nat}$.

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 slightly more difficult, the main complication being to ensure

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The undefinability 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.

that α-equivalent expressions are assigned the same Gödel number.) Recall that a general ast, a, has the form $o(a_1, \ldots, a_k)$, where o is an operator of arity k. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let m be the index of o in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of a to be the number

$$2^m\, 3^{n_1}\, 5^{n_2}\, \ldots\, p_k^{n_k},$$

where p_k is the kth prime number (so that $p_0 = 2$, $p_1 = 3$, 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 $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that, for any $e : \mathtt{nat} \to \mathtt{nat}$, $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{nat}$.[2] The determinacy of the dynamics, together with Theorem 11.4 on the facing page, ensure that f_{univ} is a well-defined function. It is called the *universal function* for $\mathcal{L}\{\mathtt{nat} \to\}$ because it specifies the behavior of any expression e of type $\mathtt{nat} \to \mathtt{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the *diagonal function*, $d : \mathbb{N} \to \mathbb{N}$, by the equation $d(m) = f_{univ}(m)(m)$. This function is chosen so that $d(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for this definition will be apparent in a moment.)

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

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

Let e_D be the expression

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

of type $\mathtt{nat} \to \mathtt{nat}$. We then have

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

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

But the termination theorem implies that there exists n such that $e_D(\overline{\ulcorner e_D \urcorner}) \equiv \overline{n}$, and hence we have $\overline{n} \equiv \mathtt{s}(\overline{n})$, which is impossible.

The function f_{univ} is computable (that is, one can write an interpreter for $\mathcal{L}\{\mathtt{nat} \to\}$), but it is not programmable in $\mathcal{L}\{\mathtt{nat} \to\}$ itself. In general a language $\mathcal{L}$ is *universal* if we can write an interpreter for $\mathcal{L}$ in the language $\mathcal{L}$ itself. The foregoing argument shows that $\mathcal{L}\{\mathtt{nat} \to\}$ is *not universal*. Consequently, there are computable numeric functions, such as the diagonal function, that cannot be programmed in $\mathcal{L}\{\mathtt{nat} \to\}$. Consequently, the universal function for $\mathcal{L}\{\mathtt{nat} \to\}$ cannot be programmed in the language. In other words, one cannot write an interpreter for $\mathcal{L}\{\mathtt{nat} \to\}$ in the language itself!

11.5 Notes

$\mathcal{L}\{\mathtt{nat} \to\}$ was introduced by Gödel in his study of proofs of proving the consistency of arithmetic [34]. In this paper Gödel showed how to "compile" proofs in arithmetic into well-typed terms of the language $\mathcal{L}\{\mathtt{nat} \to\}$, and thereby that consistency of arithmetic is equivalent to the termination (more precisely, normalization) of programs in $\mathcal{L}\{\mathtt{nat} \to\}$. This was perhaps the first programming language whose design was directly influenced by consideration of verification (of termination) of its programs.

Chapter 12

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 pay for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like. It is our task as programmers to ensure that the functions defined by recursion are total—all of our loops terminate.

12.1 Statics

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

Typ	τ	$::=$	$\mathtt{nat}$	$\mathtt{nat}$	naturals
			$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
Exp	e	$::=$	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{ifz}(e;e_0;x.e_1)$	$\mathtt{ifz}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$	zero test
			$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{fix}[\tau](x.e)$	$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,e$	recursion

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

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

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{12.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{12.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1) : \tau} \tag{12.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{parr}(\tau_1;\tau_2)} \tag{12.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{12.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau} \tag{12.1g}$$

Rule (12.1g) 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 12.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

12.2 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:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{12.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}} \tag{12.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{12.2c}$$

The bracketed premise on Rule (12.2b) is to be included for the *eager* interpretation of the sucessor operation, and omitted for the *lazy* interpretation. (See Section 12.4 on page 112 for more on this choice, which is further elaborated in Chapter 39).

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

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\} \tag{12.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{12.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{12.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{12.3d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{12.3e}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{12.3f}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e} \tag{12.3g}$$

The bracketed Rule (12.3a) is to be included for an eager interpretation of the successor, and omitted otherwise. Rule (12.3g) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 12.2 (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 (12.3g). Suppose that $\mathtt{fix}[\tau](x.e) : \tau$. By inversion and substitution we have $[\mathtt{fix}[\tau](x.e)/x]e : \tau$, from which the result follows directly by transitivity of the hypothetical judgement. The proof of progress proceeds by induction on the derivation of the typing judgement. For example, for Rule (12.1g) the result follows immediately since we may make progress by unwinding the recursion. □

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

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{z};e_0;x.e_1) \equiv e_0 : \tau} \tag{12.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \equiv [e/x]e_1 : \tau} \tag{12.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}[\tau](x.e) \equiv [\mathtt{fix}[\tau](x.e)/x]e : \tau} \tag{12.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau} \tag{12.4d}$$

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

12.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The drawback is that, in contrast to primitive recursion, the termination of a recursively defined function is not intrinsic to the program itself, but rather must be proved extrinsically by the programmer. The benefit is a much greater freedom in writing programs.

General recursive functions are definable from general recursion and non-recursive functions. Let us write $\mathtt{fun}\,x(y:\tau_1):\tau_2\,\mathtt{is}\,e$ for a recursive function within whose body, $e : \tau_2$, are bound two variables, $y : \tau_1$ standing for the argument and $x : \tau_1 \to \tau_2$ standing for the function itself. The dynamic semantics of this construct is given by the axiom

$$\frac{}{\mathtt{fun}\,x(y:\tau_1):\tau_2\,\mathtt{is}\,e(e_1) \mapsto [\mathtt{fun}\,x(y:\tau_1):\tau_2\,\mathtt{is}\,e, e_1/x, y]e}\ .$$

That is, to apply a recursive function, we substitute the recursive function itself for x and the argument for y in its body.

Recursive functions may be defined in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using a combination of recursion and functions, writing

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

for $\mathtt{fun}\,x(y:\tau_1):\tau_2\,\mathtt{is}\,e$. It is a good exercise to check that the static and dynamic semantics of recursive functions are derivable from this definition.

The primitive recursion construct of $\mathcal{L}\{\mathtt{nat} \to\}$ is defined in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ using recursive functions by taking the expression

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

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 $\phi(m, n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m', n)$ is defined and non-zero, and (2) $\phi(m, n) = 0$. If no such m exists, then $\psi(n)$ is undefined.

Theorem 12.3. *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 set 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 set of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the set 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function, ϕ_{univ}, for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is the partial function on the natural numbers defined by

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

In contrast to $\mathcal{L}\{\mathtt{nat} \rightarrow\}$, the universal function ϕ_{univ} for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is partial (may be undefined for some inputs). It is, in essence, an interpreter that, given the code $\overline{\ulcorner e \urcorner}$ of a closed expression of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, simulates the dynamic semantics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. Since this process may not terminate, the universal function is not defined for all inputs.

By Church's Law the universal function is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. In contrast, we proved in Chapter 11 that the analogous function is *not* definable in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 11.4 on page 102, we may derive the equivalence

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner}))$$

[1] See Chapter 19 for further discussion of Church's Law.

for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. The difference, however, is that this equation is not inconsistent! Rather than being contradictory, it is merely a proof that the expression $e_D(\overline{\ulcorner e_D \urcorner})$ does not terminate when evaluated, for if it did, the result would be a number equal to its own successor, which is impossible.

12.4 Co-Natural Numbers

The dynamics of the successor operation on natural numbers may be taken to be either eager or lazy, according to whether the predecessor of a successor is required to be a value. The eager interpretation represents the standard natural numbers in the sense that if $e : \texttt{nat}$ and e val, then e evaluates to a numeral. The lazy interpretation, however, admits non-standard "natural numbers," such as

$$\omega = \texttt{fix}\, x{:}\texttt{nat}\, \texttt{is}\, \texttt{s}(x).$$

The "number" ω evaluates to $\texttt{s}(\omega)$. This "number" may be thought of as an infinite stack of successors, since whenever we peel off the outermost successor we obtain the same "number" back again. The "number" ω is therefore larger than any other natural number in the sense that one may reach zero by repeatedly taking the predecessor of a natural number, but any number of predecessors on ω leads back to ω itself.

As the scare quotes indicate, it is stretching the terminology to refer to ω as a natural number. Instead one should distinguish a new type, called `conat`, of *lazy natural numbers*, of which ω is an element. The prefix "co-" indicates that the co-natural numbers are "dual" to the natural numbers in the following sense. The natural numbers are inductively defined as the *least* type such that if $e \equiv \texttt{z} : \texttt{nat}$ or $e \equiv \texttt{s}(e') : \texttt{nat}$ for some $e' : \texttt{nat}$, then $e : \texttt{nat}$. Dually, the co-natural numbers may be regarded as the *largest* type such that if $e : \texttt{conat}$, then either $e \equiv \texttt{z} : \texttt{conat}$, or $e \equiv \texttt{s}(e') : \texttt{nat}$ for some $e' : \texttt{conat}$. The difference is that $\omega : \texttt{conat}$, because ω is definitionally equivalent to its own successor, whereas it is not the case that $\omega : \texttt{nat}$, according to these definitions.

The duality between the natural numbers and the co-natural numbers is developed further in Chapter 17, wherein we consider the concepts of inductive and co-inductive types. Eagerness and laziness in general is discussed further in Chapter 39.

12.5 Notes

The language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is inspired by Plotkin's PCF [83]. Plotkin's original motivation was to analyze the relationship between denotational and operational semantics, but many subsequent studies have used PCF as a jumping off point for discussing numerous issues in language design.

Part V

Finite Data Types

Chapter 13

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, for each $i \in I$. 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.

13.1 Nullary and Binary Products

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

Typ	τ	::=	`unit`	`unit`	nullary product
			`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	binary product
Exp	e	::=	`triv`	$\langle\rangle$	null tuple
			`pair`$(e_1;e_2)$	$\langle e_1, e_2\rangle$	ordered pair
			`proj[l]`(e)	$e \cdot \mathtt{l}$	left projection
			`proj[r]`(e)	$e \cdot \mathtt{r}$	right projection

There is no elimination form for the unit type, there being nothing to extract from the null tuple.

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}} \tag{13.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1;e_2) : \mathtt{prod}(\tau_1;\tau_2)} \tag{13.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[l]}(e) : \tau_1} \tag{13.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[r]}(e) : \tau_2} \tag{13.1d}$$

The dynamics of product types is specified by the following rules:

$$\frac{}{\mathtt{triv}\ \mathsf{val}} \tag{13.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{pair}(e_1;e_2)\ \mathsf{val}} \tag{13.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1';e_2)}\right\} \tag{13.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1;e_2')}\right\} \tag{13.2d}$$

$$\frac{e \mapsto e'}{\mathtt{proj[l]}(e) \mapsto \mathtt{proj[l]}(e')} \tag{13.2e}$$

$$\frac{e \mapsto e'}{\mathtt{proj[r]}(e) \mapsto \mathtt{proj[r]}(e')} \tag{13.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{proj[l]}(\mathtt{pair}(e_1;e_2)) \mapsto e_1} \tag{13.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{proj[r]}(\mathtt{pair}(e_1;e_2)) \mapsto e_2} \tag{13.2h}$$

The bracketed rules and premises are to be omitted for a lazy dynamics, and included for an eager dynamics 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 13.1 (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 (13.2). Progress is proved by induction on typing defined by Rules (13.1). □

13.2 Finite Products

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

Typ	τ	::=	$\mathtt{prod[}I\mathtt{]}(i \mapsto \tau_i)$	$\prod_{i\in I}\tau_i$	product
Exp	e	::=	$\mathtt{tuple[}I\mathtt{]}(i \mapsto e_i)$	$\langle e_i\rangle_{i\in I}$	tuple
			$\mathtt{proj[}I\mathtt{][}i\mathtt{]}(e)$	$e \cdot i$	projection

For I a finite index set of size $n \geq 0$, the syntactic form $\mathtt{prod[}I\mathtt{]}(i \mapsto \tau_i)$ specifies an n-argument operator of arity $(0, 0, \ldots, 0)$ whose ith argument is the type τ_i. When it is useful to emphasize the tree structure, such an abt is written in the form $\prod\langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1}\rangle$. Similarly, the syntactic form $\mathtt{tuple[}I\mathtt{]}(i \mapsto e_i)$ specifies an abt constructed from an n-argument operator whose i operand is e_i. This may alternatively be written in the form $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1}\rangle$.

The statics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathtt{tuple[}I\mathtt{]}(i \mapsto e_i) : \mathtt{prod[}I\mathtt{]}(i \mapsto \tau_i)} \tag{13.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{prod[}I\mathtt{]}(i \mapsto e_i) \quad j \in I}{\Gamma \vdash \mathtt{proj[}I\mathtt{][}j\mathtt{]}(e) : \tau_j} \tag{13.3b}$$

In Rule (13.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (13.3a), the index i ranges over the index set I.

The dynamics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}} \tag{13.4a}$$

$$\left\{\frac{e_j \mapsto e_j' \quad (\forall i \neq j)\ e_i' = e_i}{\mathtt{tuple}[I]\,(i \mapsto e_i) \mapsto \mathtt{tuple}[I]\,(i \mapsto e_i')}\right\} \tag{13.4b}$$

$$\frac{e \mapsto e'}{\mathtt{proj}[I]\,[j]\,(e) \mapsto \mathtt{proj}[I]\,[j]\,(e')} \tag{13.4c}$$

$$\frac{\mathtt{tuple}[I]\,(i \mapsto e_i)\ \mathsf{val}}{\mathtt{proj}[I]\,[j]\,(\mathtt{tuple}[I]\,(i \mapsto e_i)) \mapsto e_j} \tag{13.4d}$$

Rule (13.4b) 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 13.2 (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 13.1 on page 118. □

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 $e \cdot \mathtt{l}$ and $e \cdot \mathtt{r}$ are correspondingly defined, respectively, to be $e \cdot 1$ and $e \cdot 2$.

Finite products may also be used to define *labelled tuples*, or *records*, whose components are accessed by symbolic names. If $L = \{l_1, \ldots, l_n\}$ is a finite set of symbols, called *field names*, or *field labels*, then the product type $\prod \langle l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\rangle$ has as values tuples of the form $\langle l_0 : e_0, \ldots, l_{n-1} : e_{n-1}\rangle$ in which $e_i : \tau_i$ for each $0 \leq i < n$. If e is such a tuple, then $e \cdot l$ projects the component of e labeled by $l \in L$.

13.3 Primitive and Mutual Recursion

In the presence of products we may simplify the primitive recursion construct defined in Chapter 11 so that only the result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\texttt{natiter}\,e\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}$, we may define primitive recursion in the sense of Chapter 11 to be the expression $e' \cdot \texttt{r}$, where e' is the expression

$$\texttt{natiter}\,e\,\{\texttt{z}\Rightarrow\langle \texttt{z}, e_0\rangle \mid \texttt{s}(x)\Rightarrow\langle \texttt{s}(x\cdot\texttt{l}), [x\cdot\texttt{l}, x\cdot\texttt{r}/x_0, x_1]e_1\rangle\}.$$

The idea is to compute inductively both the number, n, and the result of the recursive call on n, from which we can compute both $n+1$ and the result of an additional recursion using e_1. The base case is computed directly as the pair of zero and e_0. It is easy to check that the statics and dynamics of the recursor are preserved by this definition.

We may also use product types to implement *mutual recursion*, which allows several mutually recursive computations to be defined simultaneously. For example, 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 $\texttt{eo}(\bar{0})$ represents E and $\texttt{eo}(\bar{1})$ 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

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

and e_O is the expression

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

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.

13.4 Notes

Product types are the abstract essence of structured data [75]. Structures are tuples whose components are accessible using projections. Most languages have some form of product type, but frequently in a form that is mixed up with representation commitments and restrictions on how they may be used. Rather than introduce *ad hoc* mechanisms for passing multiple arguments or returning multiple results, one may instead systematize the concepts of tupling and pattern matching, and decouple them from function call and return. Similarly, "objects" are just tuples of mutually recursive functions; it seems preferable to break out the building blocks, namely functions, products, and recursion, rather than to amalgamate them into a monolith.

Chapter 14

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.

14.1 Binary and Nullary Sums

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

Typ	τ	::=	$\mathtt{void}$	$\mathtt{void}$	nullary sum
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Exp	e	::=	$\mathtt{abort}[\tau](e)$	$\mathtt{abort}_\tau\, e$	abort
			$\mathtt{in}[\mathtt{l}][\tau](e)$	$\mathtt{l} \cdot e$	left injection
			$\mathtt{in}[\mathtt{r}][\tau](e)$	$\mathtt{r} \cdot e$	right injection
			$\mathtt{case}(e; x_1.e_1; x_2.e_2)$	$\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introductory form. The eliminatory form, $\mathtt{abort}[\tau](e)$, aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary 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.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}\,[\tau]\,(e) : \tau} \tag{14.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[l]}\,[\tau]\,(e) : \tau} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \texttt{in[r]}\,[\tau]\,(e) : \tau} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum}(\tau_1;\tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e; x_1.e_1; x_2.e_2) : \tau} \tag{14.1d}$$

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 dynamics of sums is given by the following rules:

$$\frac{e \mapsto e'}{\texttt{abort}\,[\tau]\,(e) \mapsto \texttt{abort}\,[\tau]\,(e')} \tag{14.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[l]}\,[\tau]\,(e)\ \mathsf{val}} \tag{14.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[r]}\,[\tau]\,(e)\ \mathsf{val}} \tag{14.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[l]}\,[\tau]\,(e) \mapsto \texttt{in[l]}\,[\tau]\,(e')}\right\} \tag{14.2d}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[r]}\,[\tau]\,(e) \mapsto \texttt{in[r]}\,[\tau]\,(e')}\right\} \tag{14.2e}$$

$$\frac{e \mapsto e'}{\texttt{case}(e; x_1.e_1; x_2.e_2) \mapsto \texttt{case}(e'; x_1.e_1; x_2.e_2)} \tag{14.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case}(\texttt{in[l]}\,[\tau]\,(e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1} \tag{14.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case}(\texttt{in[r]}\,[\tau]\,(e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2} \tag{14.2h}$$

The bracketed premises and rules are to be included for an eager dynamics, and excluded for a lazy dynamics.

The coherence of the statics and dynamics is stated and proved as usual.

Theorem 14.1 (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 by induction on Rules (14.2) for preservation, and by induction on Rules (14.1) for progress. □

14.2 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:

Typ	τ	::=	$\mathtt{sum}(\langle i : \tau_i \rangle_{i \in I})$	$\sum_{i \in I} \tau_i$	sum
Exp	e	::=	$\mathtt{in}[\langle i : \tau_i \rangle_{i \in I}]\,[i]\,(e)$	$i \cdot e$	injection
			$\mathtt{case}\,[I]\,(e; \langle i : x_i . e_i \rangle_{i \in I})$	$\mathtt{case}\, e\, \{i \cdot x_i \Rightarrow e_i\}_{i \in I}$	case analysis

The general sum $\sum_{i \in I} \tau_i$ is sometimes written in the form $\sum \langle i : \tau_i \rangle_{i \in I}$. The finite family of types $\langle i : \tau_i \rangle_{i \in I}$ is often abbreviated to $\vec{\tau}$ when the finite index set, I, is clear from context.

The statics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_i \quad i \in I}{\Gamma \vdash \mathtt{in}[\langle i : \tau_i \rangle_{i \in I}]\,[i]\,(e) : \mathtt{sum}(\langle i : \tau_i \rangle_{i \in I})} \tag{14.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}(\langle i : \tau_i \rangle_{i \in I}) \quad (\forall i \in I)\, \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}\,[I]\,(e; \langle i : x_i . e_i \rangle_{i \in I}) : \tau} \tag{14.3b}$$

These rules generalize to the finite case the statics for nullary and binary sums given in Section 14.1 on page 123.

The dynamics of finite sums is defined by the following rules:

$$\frac{\{e \text{ val}\}}{\mathtt{in}[\vec{\tau}]\,[i]\,(e) \text{ val}} \tag{14.4a}$$

$$\left\{ \frac{e \mapsto e'}{\mathtt{in}[\vec{\tau}]\,[i]\,(e) \mapsto \mathtt{in}[\vec{\tau}]\,[i]\,(e')} \right\} \tag{14.4b}$$

$$\frac{e \mapsto e'}{\mathtt{case}\,[I]\,(e; \langle i : x_i . e_i \rangle_{i \in I}) \mapsto \mathtt{case}\,[I]\,(e'; \langle i : x_i . e_i \rangle_{i \in I})} \tag{14.4c}$$

$$\frac{\mathtt{in}[\vec{\tau}]\,[i]\,(e) \text{ val}}{\mathtt{case}\,[I]\,(\mathtt{in}[\vec{\tau}]\,[i]\,(e); \langle i : x_i . e_i \rangle_{i \in I}) \mapsto [e/x_i]e_i} \tag{14.4d}$$

These again generalize the dynamics of binary sums given in Section 14.1 on page 123.

Theorem 14.2 (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 14.1 on page 123. □

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 $I = \{\,\mathtt{l},\mathtt{r}\,\}$ is the two-element index set. The binary sum injections $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$ are defined to be their counterparts, $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$, respectively. Finally, the binary case analysis,

$$\mathtt{case}\, e\, \{\mathtt{l}\cdot x_{\mathtt{l}} \Rightarrow e_{\mathtt{l}} \mid \mathtt{r}\cdot x_{\mathtt{r}} \Rightarrow e_{\mathtt{r}}\},$$

is defined to be the case analysis, $\mathtt{case}\, e\, \{i\cdot x_i \Rightarrow \tau_i\}_{i\in I}$. It is easy to check that the static and dynamics of sums given in Section 14.1 on page 123 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 names for the injections.

14.3 Applications of Sum Types

Sum types have numerous uses, several of which we outline here. More interesting examples arise once we also have recursive types, which are introduced in Part VI.

14.3.1 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!

14.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

Typ	τ	::=	`bool`	`bool`	booleans
Exp	e	::=	`tt`	`tt`	truth
			`ff`	`ff`	falsity
			`if(`e`;`e_1`;`e_2`)`	`if` e `then` e_1 `else` e_2	conditional

The expression `if(`e`;`e_1`;`e_2`)` branches on the value of e : `bool`. We leave a precise formulation of the static and dynamics of this type as an exercise for the reader.

The type `bool` is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{sum(unit;unit)} \tag{14.5a}$$

$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{14.5b}$$

$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{14.5c}$$

$$\texttt{if(}e\texttt{;}e_1\texttt{;}e_2\texttt{)} = \texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \tag{14.5d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. (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 dynamics of the type `bool` is engendered by these definitions.

14.3.3 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.

14.3.4 Options

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

Typ	τ	::=	`opt(`τ`)`	τ `opt`	option
Exp	e	::=	`null`	`null`	nothing
			`just(`e`)`	`just(`e`)`	something
			`ifnull[`τ`](`e`;`e_1`;`x`.`e_2`)`	`check` e `{null` $\Rightarrow e_1$ `| just(`x`)` $\Rightarrow e_2$`}`	null test

The type `opt(`τ`)` represents the type of "optional" values of type τ. The introductory forms are `null`, corresponding to "no value", and `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:

$$\texttt{opt}(\tau) = \texttt{sum}(\texttt{unit}; \tau) \tag{14.6a}$$

$$\texttt{null} = \texttt{in[l][opt}(\tau)\texttt{](triv)} \tag{14.6b}$$

$$\texttt{just}(e) = \texttt{in[r][opt}(\tau)\texttt{]}(e) \tag{14.6c}$$

$$\texttt{ifnull[}\tau\texttt{]}(e; e_1; x_2\texttt{.}e_2) = \texttt{case}(e; \_\texttt{.}e_1; x_2\texttt{.}e_2) \tag{14.6d}$$

We leave it to the reader to examine the statics and dynamics 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 `null` : $\tau \to$ `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

$$\texttt{if null}(e)\ \texttt{then} \ldots \textit{error} \ldots\ \texttt{else} \ldots \textit{proceed} \ldots. \tag{14.7}$$

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,

$$\texttt{ifnull}[\tau](e; e_{error}; x\,.\,e_{ok}) \tag{14.8}$$

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

14.4 Notes

Heterogeneous data structures are ubiquitous. Sums codify heterogeneity, yet few languages properly support them. Much of object-oriented programming is concerned with heterogeneity. Although often confused with types, classes are, as here, tags; dispatch is case analysis, but without the benefit of ensuring that all cases are properly covered. And most such languages succumb to what Tony Hoare calls his "billion dollar mistake," the cursed "null pointer."

Chapter 15

Pattern Matching

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

$$\texttt{let}\ x\ \texttt{be}\ e\ \texttt{in}\ x \cdot \texttt{l} + x \cdot \texttt{r}$$

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

$$\texttt{match}\ e\ \{\langle x_1, x_2\rangle \Rightarrow x_1 + x_2\},$$

using pattern matching to name the components of the pair and refer to them directly. The first argument to the `match` expression is called the *match value* and the second argument consist of a finite sequence of *rules*, separated by vertical bars. In this example there is only one rule, but as we shall see shortly there is, in general, more than one rule in a given `match` expression. Each rule consists of a *pattern*, possibly involving variables, and an *expression* that may involve those variables (as well as any others currently in scope). The value of the `match` is determined by considering each rule in the order given to determine the first rule whose pattern matches the match value. If such a rule is found, the value of the `match` is the value of the expression part of the matching rule, with the variables of the pattern replaced by the corresponding components of the match value.

Pattern matching becomes more interesting, and useful, when combined with sums. The patterns $\texttt{l} \cdot x$ and $\texttt{r} \cdot x$ match the corresponding values of sum type. These may be used in combination with other patterns to express complex decisions about the structure of a value. For example, the following `match` expresses the computation that, when given a pair of type $(\texttt{unit} + \texttt{unit}) \times \texttt{nat}$, either doubles or squares its second component

depending on the form of its first component:

$$\texttt{match}\, e\, \{\langle \texttt{l} \cdot \langle\rangle, x\rangle \Rightarrow x + x \mid \langle \texttt{r} \cdot \langle\rangle, y\rangle \Rightarrow y * y\}. \tag{15.1}$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 13 and 14.

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

15.1 A Pattern Language

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

Exp	e	::=	$\texttt{match}(e; rs)$	$\texttt{match}\, e\, \{rs\}$	case analysis
Rules	rs	::=	$\texttt{rules}[n](r_1; \ldots; r_n)$	$r_1 \mid \ldots \mid r_n$	$(n \geq 0)$
Rule	r	::=	$\texttt{rule}[k](p; x_1, \ldots, x_k.e)$	$p \Rightarrow e$	$(k \geq 0)$
Pat	p	::=	$\texttt{wild}$	$\_$	wild card
			x	x	variable
			$\texttt{triv}$	$\langle\rangle$	unit
			$\texttt{pair}(p_1; p_2)$	$\langle p_1, p_2\rangle$	pair
			$\texttt{in}[\texttt{l}](p)$	$\texttt{l} \cdot p$	left injection
			$\texttt{in}[\texttt{r}](p)$	$\texttt{r} \cdot p$	right injection

The operator $\texttt{match}$ has arity $(0, 0)$, specifying that it takes two operands, the expression to match and a series of rules. A sequence of rules is constructed using the operatator $\texttt{rules}[n]$, which has arity $(0, \ldots, 0)$ specifying that it has $n \geq 0$ operands. Each rule is constructed by the operator $\texttt{rule}[k]$ of arity $(0, k)$ which specifies that it has two operands, binding k variables in the second.

15.2 Statics

The statics of $\mathcal{L}\{pat\}$ makes use of a special form of hypothetical judgement, written

$$x_1 : \tau_1, \ldots, x_k : \tau_k \Vdash p : \tau,$$

with almost the same meaning as

$$x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau,$$

except that each variable is required to be used *at most once* in p. When reading the judgement $\Lambda \Vdash p : \tau$ it is helpful to think of Λ as an *output*,

and p and τ as *inputs*. Given p and τ, the rules determine the hypotheses Λ such that $\Lambda \Vdash p : \tau$.

$$\frac{}{x : \tau \Vdash x : \tau} \tag{15.2a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau} \tag{15.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}} \tag{15.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2} \tag{15.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2} \tag{15.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2} \tag{15.2f}$$

Rule (15.2a) states that a variable is a pattern of type τ. Rule (15.2d) states that a pair pattern consists of two patterns with disjoint variables.

The typing judgments for a rule,

$$p \Rightarrow e : \tau > \tau',$$

and for a sequence of rules,

$$r_1 \mid \ldots \mid r_n : \tau > \tau',$$

specify that rules transform a value of type τ into a value of type τ'. These judgements are inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\,\Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau'} \tag{15.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau > \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'} \tag{15.3b}$$

Using the typing judgements for rules, the typing rule for a `match` expression may be stated quite easily:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'}{\Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'} \tag{15.4}$$

15.3 Dynamics

A *substitution*, θ, is a finite mapping from variables to values. If θ is the substitution $\langle x_1 : e_1 \rangle \otimes \cdots \otimes \langle x_k : e_k \rangle$, we write $\hat{\theta}(e)$ for $[e_1, \ldots, e_k / x_1, \ldots, x_k]e$. The judgement $\theta : \Lambda$ is inductively defined by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{15.5a}$$

$$\frac{\theta : \Lambda \quad \theta(x) = e \quad e : \tau}{\theta : \Lambda, x : \tau} \tag{15.5b}$$

The judgement $\theta \Vdash p \lhd e$ states that the pattern, p, matches the value, e, as witnessed by the substitution, θ, defined on the variables of p. This judgement is inductively defined by the following rules:

$$\frac{}{\langle x : e \rangle \Vdash x \lhd e} \tag{15.6a}$$

$$\frac{}{\emptyset \Vdash \_ \lhd e} \tag{15.6b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle} \tag{15.6c}$$

$$\frac{\theta_1 \Vdash p_1 \lhd e_1 \quad \theta_2 \Vdash p_2 \lhd e_2 \quad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2 \rangle \lhd \langle e_1, e_2 \rangle} \tag{15.6d}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{l} \cdot p \lhd \mathtt{l} \cdot e} \tag{15.6e}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{r} \cdot p \lhd \mathtt{r} \cdot e} \tag{15.6f}$$

These rules simply collect the bindings for the pattern variables required to form a substitution witnessing the success of the matching process.

The judgement $e \perp p$ states that e does not match the pattern p. It is inductively defined by the following rules:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2 \rangle \perp \langle p_1, p_2 \rangle} \tag{15.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2 \rangle \perp \langle p_1, p_2 \rangle} \tag{15.7b}$$

$$\frac{}{\mathtt{l} \cdot e \perp \mathtt{r} \cdot p} \tag{15.7c}$$

$$\frac{e \perp p}{\mathtt{l} \cdot e \perp \mathtt{l} \cdot p} \tag{15.7d}$$

$$\frac{}{\mathtt{r}\cdot e \perp \mathtt{l}\cdot p} \tag{15.7e}$$

$$\frac{e \perp p}{\mathtt{r}\cdot e \perp \mathtt{r}\cdot p} \tag{15.7f}$$

Neither a variable nor a wildcard nor a null-tuple can mismatch any value of appropriate type. A pair can only mismatch a pair pattern due to a mismatch in one of its components. An injection into a sum type can mismatch the opposite injection, or it can mismatch the same injection by having its argument mismatch the argument pattern.

Theorem 15.1. *Suppose that $e : \tau$, e val, and $\Lambda \Vdash p : \tau$. Then either there exists θ such that $\theta : \Lambda$ and $\theta \Vdash p \triangleleft e$, or $e \perp p$.*

Proof. By rule induction on Rules (15.2), making use of the canonical forms lemma to characterize the shape of e based on its type. □

The dynamics of the `match` expression is given in terms of the pattern match and mismatch judgements as follows:

$$\frac{e \mapsto e'}{\mathtt{match}\, e\, \{rs\} \mapsto \mathtt{match}\, e'\, \{rs\}} \tag{15.8a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{match}\, e\, \{\}\ \mathsf{err}} \tag{15.8b}$$

$$\frac{e\ \mathsf{val} \quad \theta \Vdash p_0 \triangleleft e}{\mathtt{match}\, e\, \{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto \hat{\theta}(e_0)} \tag{15.8c}$$

$$\frac{e\ \mathsf{val} \quad e \perp p_0 \quad \mathtt{match}\, e\, \{rs\} \mapsto e'}{\mathtt{match}\, e\, \{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto e'} \tag{15.8d}$$

Rule (15.8b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (15.8c) specifies that the rules are to be considered in order. If the match value, e, matches the pattern, p_0, of the initial rule in the sequence, then the result is the corresponding instance of e_0; otherwise, matching continues by considering the remaining rules.

Theorem 15.2 (Preservation)**.** *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof. By a straightforward induction on the derivation of $e \mapsto e'$. □

15.4 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $\mathcal{L}\{pat\}$ as defined in Section 15.1 on page 132, it would not have much force, because the statics 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 arises when a rule can only match values that are also matched by a 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.

15.4.1 Match Constraints

To express exhaustiveness and irredundancy, we introduce a language of *match constraints* that identify a subset of the closed values of a type. With each rule we associate a constraint 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 constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match constraint also satisfies the match constraint of some preceding rule.

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

Constr	ξ	::=	`all[`τ`]`	$\top$	truth
			`and(`ξ_1`;`ξ_2`)`	$\xi_1 \wedge \xi_2$	conjunction
			`nothing[`τ`]`	$\bot$	falsity
			`or(`ξ_1`;`ξ_2`)`	$\xi_1 \vee \xi_2$	disjunction
			`in[l](`ξ_1`)`	$\mathtt{l} \cdot \xi_1$	left injection
			`in[r](`ξ_2`)`	$\mathtt{r} \cdot \xi_2$	right injection
			`triv`	$\langle\rangle$	unit
			`pair(`ξ_1`;`ξ_2`)`	$\langle \xi_1, \xi_2 \rangle$	pair

It is easy to define the judgement $\xi : \tau$ specifying that the constraint ξ constrains values of type τ.

The *De Morgan Dual*, $\overline{\xi}$, of a match constraint, ξ, is defined by the fol-

lowing rules:

$$\begin{aligned}
\overline{\top} &= \bot \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\bot} &= \top \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\mathtt{l} \cdot \xi_1} &= \mathtt{l} \cdot \overline{\xi_1} \vee \mathtt{r} \cdot \top \\
\overline{\mathtt{r} \cdot \xi_1} &= \mathtt{r} \cdot \overline{\xi_1} \vee \mathtt{l} \cdot \top \\
\overline{\langle\rangle} &= \bot \\
\overline{\langle \xi_1, \xi_2 \rangle} &= \langle \overline{\xi_1}, \xi_2 \rangle \vee \langle \xi_1, \overline{\xi_2} \rangle \vee \langle \overline{\xi_1}, \overline{\xi_2} \rangle
\end{aligned}$$

Intuitively, the dual of a match constraint expresses the negation of that constraint. In the case of the last four rules it is important to keep in mind that these constraints apply only to specific types.

The *satisfaction* judgement, $e \models \xi$, is defined for values e and constraints ξ of the same type by the following rules:

$$\frac{}{e \models \top} \tag{15.9a}$$

$$\frac{e \models \xi_1 \quad e \models \xi_2}{e \models \xi_1 \wedge \xi_2} \tag{15.9b}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{15.9c}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{15.9d}$$

$$\frac{e_1 \models \xi_1}{\mathtt{l} \cdot e_1 \models \mathtt{l} \cdot \xi_1} \tag{15.9e}$$

$$\frac{e_2 \models \xi_2}{\mathtt{r} \cdot e_2 \models \mathtt{r} \cdot \xi_2} \tag{15.9f}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{15.9g}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2 \rangle \models \langle \xi_1, \xi_2 \rangle} \tag{15.9h}$$

The De Morgan dual construction negates a constraint.

Lemma 15.3. *If ξ is a constraint on values of type τ, then $e \models \overline{\xi}$ if, and only if, $e \not\models \xi$.*

We define the *entailment* of two constraints, $\xi_1 \models \xi_2$ to mean that $e \models \xi_2$ whenever $e \models \xi_1$. By Lemma 15.3 we have that $\xi_1 \models \xi_2$ iff $\models \overline{\xi_1} \vee \xi_2$. We often write $\xi_1, \ldots, \xi_n \models \xi$ for $\xi_1 \wedge \ldots \wedge \xi_n \models \xi$ so that in particular $\models \xi$ means $e \models \xi$ for every value $e : \tau$.

15.4.2 Enforcing Exhaustiveness and Redundancy

To enforce exhaustiveness and irredundancy the statics of pattern matching is augmented with constraints that express the set of values matched by a given set of rules. A sequence of rules is *exhaustive* if every value of suitable type satisfies the associated constraint. A rule is *redundant* relative to the preceding rules if every value satisfying its constraint satisfies one of the preceding constraints. A sequence of rules is *irredundant* iff no rule is redundant relative to the rules that precede it in the sequence.

The judgement $\Lambda \Vdash p : \tau\ [\xi]$ augments the judgement $\Lambda \Vdash p : \tau$ 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:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top]} \tag{15.10a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau\ [\top]} \tag{15.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}\ [\langle\rangle]} \tag{15.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2\ [\mathtt{l} \cdot \xi_1]} \tag{15.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2\ [\mathtt{r} \cdot \xi_2]} \tag{15.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad \Lambda_1 \# \Lambda_2}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]} \tag{15.10f}$$

Lemma 15.4. *Suppose that $\Lambda \Vdash p : \tau\ [\xi]$. For every $e : \tau$ such that e val, $e \models \xi$ iff $\theta \Vdash p \triangleleft e$ for some θ, and $e \not\models \xi$ iff $e \perp p$.*

The judgement $\Gamma \vdash r : \tau > \tau'\ [\xi]$ augments the formation judgement for a rule with a match constraint characterizing the pattern component of the rule. The judgement $\Gamma \vdash rs : \tau > \tau'\ [\xi]$ 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.

$$\frac{\Lambda \Vdash p : \tau\ [\xi] \quad \Gamma\,\Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau'\ [\xi]} \tag{15.11a}$$

$$\frac{\begin{array}{c}(\forall 1 \leq i \leq n)\ \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1} \\ \Gamma \vdash r_1 : \tau > \tau'\ [\xi_1] \qquad \ldots \qquad \Gamma \vdash r_n : \tau > \tau'\ [\xi_n]\end{array}}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'\ [\xi_1 \vee \ldots \vee \xi_n]} \tag{15.11b}$$

Rule (15.11b) requires that each successive rule not be redundant relative to the preceding rules. The overall constraint associated to the rule sequence specifies that every value of type τ satisfy the constraint associated with some rule.

The typing rule for `match` expressions demands that the rules that comprise it be exhaustive:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'\ [\xi] \quad \models \xi}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{15.12}$$

Rule (15.11b) ensures that ξ is a disjunction of the match constraints associated to the constituent rules of the match expression. The requirement that ξ be valid amounts to requiring that every value of type τ satisfies the constraint of at least one rule of the match.

Theorem 15.5. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. The exhaustiveness check in Rule (15.12) ensures that if e val and $e : \tau$, then $e \models \xi$. The form of ξ given by Rule (15.11b) ensures that $e \models \xi_i$ for some constraint ξ_i corresponding to the ith rule. By Lemma 15.4 on the facing page the value e must match the ith rule, which is enough to ensure progress. □

15.4.3 Checking Exhaustiveness and Redundancy

Checking exhaustiveness and redundancy reduces to showing that the constraint validity judgement $\models \xi$ is decidable. We will prove this by defining a judgement Ξ incon, where Ξ is a finite set of constraints of the same type, with the meaning that no value of this type satisfies all of the constraints in Ξ. We will then show that either Ξ incon or not.

The rules defining inconsistency of a finite set, Ξ, of constraints of the same type are as follows:

$$\frac{\Xi\ \mathsf{incon}}{\Xi, \top\ \mathsf{incon}} \tag{15.13a}$$

$$\frac{\Xi, \xi_1, \xi_2 \text{ incon}}{\Xi, \xi_1 \wedge \xi_2 \text{ incon}} \tag{15.13b}$$

$$\frac{}{\Xi, \bot \text{ incon}} \tag{15.13c}$$

$$\frac{\Xi, \xi_1 \text{ incon} \quad \Xi, \xi_2 \text{ incon}}{\Xi, \xi_1 \vee \xi_2 \text{ incon}} \tag{15.13d}$$

$$\frac{}{\Xi, \mathtt{l} \cdot \xi_1, \mathtt{r} \cdot \xi_2 \text{ incon}} \tag{15.13e}$$

$$\frac{\Xi \text{ incon}}{\mathtt{l} \cdot \Xi \text{ incon}} \tag{15.13f}$$

$$\frac{\Xi \text{ incon}}{\mathtt{r} \cdot \Xi \text{ incon}} \tag{15.13g}$$

$$\frac{\Xi_1 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \tag{15.13h}$$

$$\frac{\Xi_2 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \tag{15.13i}$$

In Rule (15.13f) we write $\mathtt{l} \cdot \Xi$ for the finite set of constraints $\mathtt{l} \cdot \xi_1, \ldots, \mathtt{l} \cdot \xi_n$, where $\Xi = \xi_1, \ldots, \xi_n$, and similarly in Rules (15.13g), (15.13h), and (15.13i).

Lemma 15.6. *It is decidable whether or not Ξ incon.*

Proof. The premises of each rule involves only constraints that are proper components of the constraints in the conclusion. Consequently, we can simplify Ξ by inverting each of the applicable rules until no rule applies, then determine whether or not the resulting set, Ξ', is contradictory in the sense that it contains $\bot$ or both $\mathtt{l} \cdot \xi$ and $\mathtt{r} \cdot \xi'$ for some ξ and ξ'. □

Lemma 15.7. *Ξ incon iff $\Xi \models \bot$.*

Proof. From left to right we proceed by induction on Rules (15.13). From right to left we may show that if Ξ incon is not derivable, then there exists a value e such that $e \models \Xi$, and hence $\Xi \not\models \bot$. □

15.5 Notes

Pattern-matching against heterogeneous structured data was first explored in the context of logic programming languages, such as Prolog [52, 22], but with an execution model based on proof search. Pattern matching in the form described here is present in the functional languages Miranda [101], Hope [17], Haskell [48], Standard ML [67], and Caml [25].

Chapter 16

Generic Programming

16.1 Introduction

Many programs can be seen as instances of a general pattern applied to a particular situation. Very often the pattern is determined by the types of the data involved. For example, in Chapter 11 the pattern of computing by recursion over a natural number is isolated as the defining characteristic of the type of natural numbers. This concept will itself emerge as an instance of the concept of *type-generic*, or just *generic*, programming.

Suppose that we have a function, f, of type $\rho \to \rho'$ that transforms values of type ρ into values of type ρ'. For example, f might be the doubling function on natural numbers. We wish to extend f to a transformation from type $[\rho/t]\tau$ to type $[\rho'/t]\tau$ by applying f to various spots in the input where a value of type ρ occurs to obtain a value of type ρ', leaving the rest of the data structure alone. For example, τ might be $\mathtt{bool} \times \rho$, in which case f could be extended to a function of type $\mathtt{bool} \times \rho \to \mathtt{bool} \times \rho'$ that sends the pairs $\langle a, b\rangle$ to the pair $\langle a, f(b)\rangle$.

This example glosses over a significant problem of ambiguity of the extension. Given a function f of type $\rho \to \rho'$, it is not obvious in general how to extend it to a function mapping $[\rho/t]\tau$ to $[\rho'/t]\tau$. The problem is that it is not clear which of many occurrences of ρ in $[\rho/t]\tau$ are to be transformed by f, even if there is only one occurrence of ρ. To avoid ambiguity we need a way to mark which occurrences of ρ in $[\rho/t]\tau$ are to be transformed, and which are to be left fixed. This can be achieved by isolating the *type operator*, $t.\tau$, which is a type expression in which a designated variable, t, marks the spots at which we wish the transformation to occur. Given $t.\tau$ and $f : \rho \to \rho'$, we can extend f unambiguously to a function of type

$[\rho/t]\tau \to [\rho'/t]\tau$.

The technique of using a type operator to determine the behavior of a piece of code is called *generic programming*. The power of generic programming depends on which forms of type operator are considered. The simplest case is that of a *polynomial* type operator, one constructed from sum and product of types, including their nullary forms. These may be extended to *positive* type operators, which also permit restricted forms of function types.

16.2 Type Operators

A *type operator* is a type equipped with a designated variable whose occurrences mark the positions in the type where a transformation is to be applied. A type operator is represented by an abstractor $t\,.\,\tau$ such that t type $\vdash \tau$ type. An example of a type operator is the abstractor

$$t\,.\,\texttt{unit} + (\texttt{bool} \times t)$$

in which occurrences of t mark the spots in which a transformation is to be applied. An *instance* of the type operator $t\,.\,\tau$ is obtained by substituting a type, ρ, for the variable, t, within the type τ. We sometimes write $\texttt{Map}[t\,.\,\tau](\rho)$ for the substitution instance $[\rho/t]\tau$.

The *polynomial* type operators are those constructed from the type variable, t, the types `void` and `unit`, and the product and sum type constructors, $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. It is a straightforward exercise to give inductive definitions of the judgement $t\,.\,\tau$ poly stating that the operator $t\,.\,\tau$ is a polynomial type operator.

16.3 Generic Extension

The *generic extension* primitive has the form

$$\texttt{map}[t\,.\,\tau](x\,.\,e';e)$$

with statics given by the following rule:

$$\frac{t \text{ type} \vdash \tau \text{ type} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \texttt{map}[t\,.\,\tau](x\,.\,e';e) : [\rho'/t]\tau} \tag{16.1}$$

The abstractor $x\,.\,e'$ specifies a transformation from type ρ, the type of x, to type ρ', the type of e'. The expression e of type $[\rho/t]\tau$ determines the value

to be transformed to obtain a value of type $[\rho'/t]\tau$. The occurrences of t in τ determine the spots at which the transformation given by $x.e$ is to be performed.

The dynamics of generic extension is specified by the following rules. We consider here only polynomial type operators, leaving the extension to positive type operators to be considered later.

$$\frac{}{\mathtt{map}[t.t]\,(x.e';e) \mapsto [e/x]e'} \tag{16.2a}$$

$$\frac{}{\mathtt{map}[t.\mathtt{unit}]\,(x.e';e) \mapsto \langle\rangle} \tag{16.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 \times \tau_2]\,(x.e';e)\\ \mapsto\\ \langle \mathtt{map}[t.\tau_1]\,(x.e';e\cdot \mathtt{l}), \mathtt{map}[t.\tau_2]\,(x.e';e\cdot \mathtt{r})\rangle\end{array}} \tag{16.2c}$$

$$\frac{}{\mathtt{map}[t.\mathtt{void}]\,(x.e';e) \mapsto \mathtt{abort}(e)} \tag{16.2d}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 + \tau_2]\,(x.e';e)\\ \mapsto\\ \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \Rightarrow \mathtt{l}\cdot \mathtt{map}[t.\tau_1]\,(x.e';x_1) \mid \mathtt{r}\cdot x_2 \Rightarrow \mathtt{r}\cdot \mathtt{map}[t.\tau_2]\,(x.e';x_2)\}\end{array}} \tag{16.2e}$$

Rule (16.2a) applies the transformation $x.e'$ to e itself, since the operator $t.t$ specifies that the transformation is to be perfomed directly. Rule (16.2b) states that the empty tuple is transformed to itself. Rule (16.2c) states that to transform e according to the operator $t.\tau_1 \times \tau_2$, the first component of e is transformed according to $t.\tau_1$ and the second component of e is transformed according to $t.\tau_2$. Rule (16.2d) states that the transformation of a value of type `void` aborts, since there can be no such values. Rule (16.2e) states that to transform e according to $t.\tau_1 + \tau_2$, case analyze e and reconstruct it after transforming the injected value according to $t.\tau_1$ or $t.\tau_2$.

Consider the type operator $t.\tau$ given by $t.\mathtt{unit} + (\mathtt{bool} \times t)$. Let $x.e$ be the abstractor $x.\mathtt{s}(x)$, which increments a natural number. Using Rules (16.2) we may derive that

$$\mathtt{map}[t.\tau]\,(x.e;\mathtt{r}\cdot\langle \mathtt{tt}, n\rangle) \mapsto^* \mathtt{r}\cdot\langle \mathtt{tt}, n+1\rangle.$$

The natural number in the second component of the pair is incremented, since the type variable, t, occurs in that position in the type operator $t.\tau$.

Theorem 16.1 (Preservation). *If* $\mathtt{map}\,[t.\tau]\,(x.e';e) : \tau'$ *and* $\mathtt{map}\,[t.\tau]\,(x.e';e) \mapsto e''$, *then* $e'' : \tau'$.

Proof. By inversion of Rule (16.1) we have

1. t type $\vdash \tau$ type;
2. $x : \rho \vdash e' : \rho'$ for some ρ and ρ';
3. $e : [\rho/t]\tau$;
4. τ' is $[\rho'/t]\tau$.

We proceed by cases on Rules (16.2). For example, consider Rule (16.2c). It follows from inversion that $\mathtt{map}\,[t.\tau_1]\,(x.e';e\cdot\mathtt{l}) : [\rho'/t]\tau_1$, and similarly that $\mathtt{map}\,[t.\tau_2]\,(x.e';e\cdot\mathtt{r}) : [\rho'/t]\tau_2$. It is easy to check that

$$\langle \mathtt{map}\,[t.\tau_1]\,(x.e';e\cdot\mathtt{l}), \mathtt{map}\,[t.\tau_2]\,(x.e';e\cdot\mathtt{r})\rangle$$

has type $[\rho'/t]\tau_1 \times \tau_2$, as required. □

The *positive* type operators extend the polynomial type operators to admit restricted forms of function type. Specifically, $t.\tau_1 \to \tau_2$ is a positive type operator, provided that (1) t *does not occur* in τ_1, and (2) $t.\tau_2$ is a positive type operator. In general, any occurrences of a type variable t in the domain a function type are said to be *negative occurrences*, whereas any occurrences of t within the range of a function type, or within a product or sum type, are said to be *positive occurrences*.[1] A positive type operator is one for which only positive occurrences of the parameter, t, are permitted.

The generic extension according to a positive type operator is defined similarly to the case of a polynomial type operator, with the following additional rule:

$$\frac{}{\mathtt{map}\,[t.\tau_1 \to \tau_2]\,(x.e';e) \mapsto \lambda\,(x_1:\tau_1.\mathtt{map}\,[t.\tau_2]\,(x.e';e(x_1)))} \tag{16.3}$$

[1] The origin of this terminology seems to be that a function type $\tau_1 \to \tau_2$ is analogous to the implication $\phi_1 \supset \phi_2$, which is classically equivalent to $\neg\phi_1 \vee \phi_2$, so that occurrences in the domain are under the negation.

Since t is not permitted to occur within the domain type, the type of the result is $\tau_1 \to [\rho'/t]\tau_2$, assuming that e is of type $\tau_1 \to [\rho/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

It is interesting to consider what goes wrong if we relax the restriction on positive type operators to admit negative, as well as positive, occurrences of the parameter of a type operator. Consider the type operator $t.\tau_1 \to \tau_2$, without restriction on t, and suppose that $x : \rho \vdash e' : \rho'$. The generic extension $\mathtt{map}[t.\tau_1 \to \tau_2]\,(x.e'; e)$ should have type $[\rho'/t]\tau_1 \to [\rho'/t]\tau_2$, given that e has type $[\rho/t]\tau_1 \to [\rho/t]\tau_2$. The extension should yield a function of the form

$$\lambda\,(x_1 : [\rho'/t]\tau_1 . \ldots (e(\ldots(x_1))))$$

in which we apply e to a transformation of x_1 and then transform the result. The trouble is that we are given, inductively, that $\mathtt{map}[t.\tau_1]\,(x.e'; -)$ transforms values of type $[\rho/t]\tau_1$ into values of type $[\rho'/t]\tau_1$, but *we need to go the other way around* in order to make x_1 suitable as an argument for e. But there is no obvious way to obtain the required transformation.

One solution to this is to assume that the fundamental transformation $x.e'$ is *invertible* so that we may apply the inverse transformation on x_1 to get an argument of type suitable for e, then apply the forward transformation on the result, just as in the positive case. Since we cannot invert an arbitrary transformation, we must instead pass both the transformation and its inverse to the generic extension operation so that it can "go backwards" as necessary to cover negative occurrences of the type parameter. So in the general case the generic extension applies only when we are given a *type isomorphism* (a pair of mutually inverse mappings between two types), and then results in another isomorphism pair. We leave the formulation of this as an exercise for the reader.

16.4 Notes

The concept of the functorial action of a type constructor has its roots in category theory [58]. Generic programming is essentially the application of this idea to computation [46].

Part VI

Infinite Data Types

Chapter 17

Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important forms of recursive type. 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 a *recursor*, 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*.

17.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 11. The type `nat` is defined to be the *least* type containing `z` and closed under `s(`$-$`)`. The minimality condition is witnessed by the existence of the recursor, `natiter` e {`z`$\Rightarrow e_0$ | `s(`x`)`$\Rightarrow e_1$}, which transforms a natural number into a value of type τ, given its value for zero, and a transformation from its value on a number to its value on the successor of that number. This operation is well-defined precisely because there are no other natural numbers. Put the other way around, the existence

of this operation expresses the inductive nature of the type $\mathtt{nat}$.

With a view towards deriving the type $\mathtt{nat}$ as a special case of an inductive type, it is useful to consolidate zero and successor into a single introductory form, and to correspondingly consolidate the basis and inductive step of the recursor. This following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \mathtt{unit} + \mathtt{nat}}{\Gamma \vdash \mathtt{fold}_{\mathtt{nat}}(e) : \mathtt{nat}} \tag{17.1a}$$

$$\frac{\Gamma, x : \mathtt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2) : \tau} \tag{17.1b}$$

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introductory form of the type $\mathtt{nat}$. Using this, the expression $\mathtt{z}$ is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{s}(e)$ is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor, $\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2)$, takes as argument the abstractor $x.e_1$ that consolidates the basis and inductive step into a single computation that is given a value of type $\mathtt{unit} + \tau$ yields a value of type τ. Intuitively, if x is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\mathtt{r} \cdot e$, then e_1 computes the inductive step as a function of the result, e, of the recursive call.

The dynamics of the consolidated representation of natural numbers is given by the following rules:

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathsf{val}} \tag{17.2a}$$

$$\frac{e_2 \mapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2) \mapsto \mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2')} \tag{17.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}[x.e_1](\mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \mapsto \\ [\mathtt{map}[t.\mathtt{unit} + t](y.\mathtt{rec}_{\mathtt{nat}}[x.e_1](y); e_2)/x]e_1\end{array}} \tag{17.2c}$$

Rule (17.2c) makes use of generic extension (see Chapter 7) to apply the recursor to the predecessor, if any, of a natural number. The idea is that the result of extending the recursor from the type $\mathtt{unit} + \mathtt{nat}$ to the type $\mathtt{unit} + \tau$ is substituted into the inductive step, given by the expression e_1. If we expand the definition of the generic extension in place, we obtain the

following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}\,[x.e_1]\,(\mathtt{fold}_{\mathtt{nat}}\,(e_2)) \\ \mapsto \\ [\mathtt{case}\,e_2\,\{\mathtt{l}\cdot\_ \Rightarrow \mathtt{l}\cdot\langle\rangle \mid \mathtt{r}\cdot y \Rightarrow \mathtt{r}\cdot\mathtt{rec}_{\mathtt{nat}}\,[x.e_1]\,(y)\}/x]e_1\end{array}}$$

An illustrative example of a coinductive type is the type of *streams* of natural numbers. A stream is an infinite sequence of natural numbers such that an element of the stream can be computed only after computing all preceding elements in that stream. That is, the computations of successive elements of the stream are sequentially dependent in that the computation of one element influences the computation of the next. This characteristic of the introductory form for streams is *dual* to the analogous property of the eliminatory form for natural numbers whereby the result for a number is determined by its result for all preceding numbers.

A stream is characterized by its behavior under the elimination forms for the stream type: $\mathtt{hd}(e)$ returns the next, or head, element of the stream, and $\mathtt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of a recursor, that determines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{nat}} \tag{17.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}(e) : \mathtt{stream}} \tag{17.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\,e\,\mathtt{<hd}(x) \Rightarrow e_1\,\mathtt{\&}\,\mathtt{tl}(x) \Rightarrow e_2\mathtt{>} : \mathtt{stream}} \tag{17.3c}$$

In Rule (17.3c) the current state of the stream is given by the expression e of some type τ, and the head and tail of the stream are determined by the expressions e_1 and e_2, respectively, as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\,e\,\mathtt{<hd}(x) \Rightarrow e_1\,\mathtt{\&}\,\mathtt{tl}(x) \Rightarrow e_2\mathtt{>}\ \mathsf{val}} \tag{17.4a}$$

$$\frac{e \mapsto e'}{\mathtt{hd}(e) \mapsto \mathtt{hd}(e')} \tag{17.4b}$$

$$\frac{}{\texttt{hd(strgen}\,e\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>)} \mapsto [e/x]e_1} \tag{17.4c}$$

$$\frac{e \mapsto e'}{\texttt{tl(}e\texttt{)} \mapsto \texttt{tl(}e'\texttt{)}} \tag{17.4d}$$

$$\begin{array}{c}\texttt{tl(strgen}\,e\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>)} \\ \mapsto \\ \texttt{strgen}\,[e/x]e_2\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>}\end{array} \tag{17.4e}$$

Rules (17.4c) and (17.4e) express the dependency of the head and tail of the stream on its current state. Observe that the tail is obtained by applying the generator to the new state determined by e_2 as a function of the current state.

To derive streams as a special case of a coinductive type, we consolidate the head and the tail into a single eliminatory form, and reorganize the generator correspondingly. This leads to the following statics:

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} : \texttt{nat} \times \texttt{stream}} \tag{17.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \texttt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\,\texttt{(}e_2\texttt{)} : \texttt{stream}} \tag{17.5b}$$

Rule (17.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head, $\texttt{hd(}e\texttt{)}$, and tail, $\texttt{tl(}e\texttt{)}$, of a stream, e, are defined to be the projections $\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \cdot \texttt{l}$ and $\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \cdot \texttt{r}$, respectively. Rule (17.5b) states that a stream may be generated from the state element, e_2, by an expression e_1 that yields the head element and the next state as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\texttt{gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\,\texttt{(}e_2\texttt{)}\ \mathsf{val}} \tag{17.6a}$$

$$\frac{e \mapsto e'}{\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \mapsto \texttt{unfold}_{\texttt{stream}}\texttt{(}e'\texttt{)}} \tag{17.6b}$$

$$\begin{array}{c}\texttt{unfold}_{\texttt{stream}}\texttt{(gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\,\texttt{(}e_2\texttt{))} \\ \mapsto \\ \texttt{map[}t\texttt{.nat} \times t\texttt{]}\,\texttt{(}y\texttt{.gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\,\texttt{(}y\texttt{);}\,[e_2/x]e_1\texttt{)}\end{array} \tag{17.6c}$$

Rule (17.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\begin{array}{c}\texttt{unfold}_{\texttt{stream}}(\texttt{gen}_{\texttt{stream}}[x.e_1](e_2)) \\ \mapsto \\ \langle([e_2/x]e_1)\cdot\texttt{l}, \texttt{gen}_{\texttt{stream}}[x.e_1](([e_2/x]e_1)\cdot\texttt{r})\rangle\end{array}$$

17.2 Statics

We may now give a fully general account of inductive and coinductive types, which are defined in terms of positive type operators. We will consider the language $\mathcal{L}\{\mu_i\mu_f\}$, which extends $\mathcal{L}\{\rightarrow\times+\}$ with inductive and co-inductive types.

17.2.1 Types

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over types. The abstract syntax of inductive and coinductive types is given by the following grammar:

Typ	τ	$::=$	t	t	self-reference
			$\texttt{ind}(t.\tau)$	$\mu_i(t.\tau)$	inductive
			$\texttt{coi}(t.\tau)$	$\mu_f(t.\tau)$	coinductive

Type formation judgements have the form

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

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 $t\ \mathsf{name}$ is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{17.7a}$$

$$\frac{}{\Delta \vdash \texttt{unit}\ \mathsf{type}} \tag{17.7b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{prod}(\tau_1;\tau_2)\ \mathsf{type}} \tag{17.7c}$$

$$\frac{}{\Delta \vdash \texttt{void}\ \mathsf{type}} \tag{17.7d}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \texttt{sum}(\tau_1;\tau_2) \text{ type}} \tag{17.7e}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2) \text{ type}} \tag{17.7f}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \texttt{ind}(t.\tau) \text{ type}} \tag{17.7g}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \texttt{coi}(t.\tau) \text{ type}} \tag{17.8}$$

17.2.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Exp	e	::=	$\texttt{fold}[t.\tau](e)$	$\texttt{fold}(e)$ constructor
			$\texttt{rec}[t.\tau][x.e_1](e_2)$	$\texttt{rec}[x.e_1](e_2)$ recursor
			$\texttt{unfold}[t.\tau](e)$	$\texttt{unfold}(e)$ destructor
			$\texttt{gen}[t.\tau][x.e_1](e_2)$	$\texttt{gen}[x.e_1](e_2)$ generator

The statics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\texttt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{fold}[t.\tau](e) : \texttt{ind}(t.\tau)} \tag{17.9a}$$

$$\frac{\Gamma, x : [\rho/t]\tau \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \texttt{ind}(t.\tau)}{\Gamma \vdash \texttt{rec}[t.\tau][x.e_1](e_2) : \rho} \tag{17.9b}$$

$$\frac{\Gamma \vdash e : \texttt{coi}(t.\tau)}{\Gamma \vdash \texttt{unfold}[t.\tau](e) : [\texttt{coi}(t.\tau)/t]\tau} \tag{17.9c}$$

$$\frac{\Gamma \vdash e_2 : \rho \quad \Gamma, x : \rho \vdash e_1 : [\rho/t]\tau}{\Gamma \vdash \texttt{gen}[t.\tau][x.e_1](e_2) : \texttt{coi}(t.\tau)} \tag{17.9d}$$

17.3 Dynamics

The dynamics of these constructs is given in terms of the generic extension operation described in Chapter 16. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\}$:

$$\frac{}{\texttt{fold}(e) \text{ val}} \tag{17.10a}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{rec}\,[x.e_1]\,(e_2) \mapsto \texttt{rec}\,[x.e_1]\,(e_2')} \tag{17.10b}$$

$$\frac{}{\begin{array}{c}\texttt{rec}\,[x.e_1]\,(\texttt{fold}(e_2)) \\ \mapsto \\ [\texttt{map}\,[t.\tau]\,(y.\texttt{rec}\,[x.e_1]\,(y);e_2)\,/\,x]e_1\end{array}} \tag{17.10c}$$

$$\frac{}{\texttt{gen}\,[x.e_1]\,(e_2)\ \mathsf{val}} \tag{17.10d}$$

$$\frac{e \mapsto e'}{\texttt{unfold}(e) \mapsto \texttt{unfold}(e')} \tag{17.10e}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}(\texttt{gen}\,[x.e_1]\,(e_2)) \\ \mapsto \\ \texttt{map}\,[t.\tau]\,(y.\texttt{gen}\,[x.e_1]\,(y);[e_2/x]e_1)\end{array}} \tag{17.10f}$$

Rule (17.10c) states that to evaluate the recursor on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value, and then perform the inductive step on the result. Rule (17.10f) is simply the dual of this rule for coinductive types.

Lemma 17.1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on Rules (17.10). □

Lemma 17.2. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on Rules (17.9). □

17.4 Notes

The general formulation of inductive and coinductive types for programming was introduced by Mendler [64], making use of the functorial action of a type constructor described in Chapter 16. Mendler's account is based on the interpretation of inductive types as initial algebras, and coinductive types as final co-algebras, for a functor [58, 100].

Chapter 18

Recursive Types

Inductive and coinductive types, such as natural numbers and streams, may be seen as examples of *fixed points* of type operators *up to isomorphism*. An isomorphism between two types, τ_1 and τ_2, is given by two expressions

1. $x_1 : \tau_1 \vdash e_2 : \tau_2$, and
2. $x_2 : \tau_2 \vdash e_1 : \tau_1$

that are mutually inverse to each other.[1] For example, the types $\mathtt{nat}$ and $\mathtt{unit} + \mathtt{nat}$ are isomorphic, as witnessed by the following two expressions:

1. $x : \mathtt{unit} + \mathtt{nat} \vdash \mathtt{case}\, x\, \{\mathtt{l} \cdot \_ \Rightarrow \mathtt{z} \mid \mathtt{r} \cdot x_2 \Rightarrow \mathtt{s}(x_2)\} : \mathtt{nat}$, and
2. $x : \mathtt{nat} \vdash \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{s}(x_2) \Rightarrow \mathtt{r} \cdot x_2\} : \mathtt{unit} + \mathtt{nat}$.

These are called, respectively, the $\mathtt{fold}$ and $\mathtt{unfold}$ operations of the isomorphism $\mathtt{nat} \cong \mathtt{unit} + \mathtt{nat}$. Thinking of $\mathtt{unit} + \mathtt{nat}$ as $[\mathtt{nat}/t](\mathtt{unit} + t)$, this means that $\mathtt{nat}$ is a *fixed point* of the type operator $t\,.\,\mathtt{unit} + t$.

In this chapter we study the language $\mathcal{L}\{+ \times \rightharpoonup \mu\}$, which provides solutions to all type isomorphism equations. The *recursive type* $\mu t\,.\,\tau$ is defined to be a solution to the type isomorphism

$$\mu t\,.\,\tau \cong [\mu t\,.\,\tau/t]\tau.$$

This is witnessed by the operations

$$x : \mu t\,.\,\tau \vdash \mathtt{unfold}(x) : [\mu t\,.\,\tau/t]\tau$$

[1] To make this precise requires a discussion of equivalence of expressions to be taken up in Chapter 49. For now we will rely on an intuitive understanding of when two expressions are equivalent.

and

$$x : [\mu t\,.\,\tau/t]\tau \vdash \texttt{fold}(x) : \mu t\,.\,\tau,$$

which are mutually inverse to each other.

Requiring solutions to all type equations may seem suspicious, since we know by Cantor's Theorem that an isomorphisms such as $X \cong (X \to \mathbf{2})$ is impossible. This negative result tells us not that our requirement is untenable, but rather that *types are not sets*. To permit solution of arbitrary type equations, we must take into account that types describe computations, some of which may not even terminate. Consequently, the function space does not coincide with the set-theoretic function space, but rather is analogous to it (in a precise sense that we shall not go into here).

18.1 Solving Type Isomorphisms

The *recursive type* $\mu t\,.\,\tau$, where $t\,.\,\tau$ is a type operator, represents a solution for t to the isomorphism $t \cong \tau$. The solution is witnessed by two operations, $\texttt{fold}(e)$ and $\texttt{unfold}(e)$, that relate the recursive type $\mu t\,.\,\tau$ to its unfolding, $[\mu t\,.\,\tau/t]\tau$, and serve, respectively, as its introduction and elimination forms.

The language $\mathcal{L}\{+\times\rightharpoonup\mu\}$ extends $\mathcal{L}\{\rightharpoonup\}$ with recursive types and their associated operations.

Typ	τ	::=	t	t	self-reference
			$\texttt{rec}(t\,.\,\tau)$	$\mu t\,.\,\tau$	recursive
Exp	e	::=	$\texttt{fold}[t\,.\,\tau](e)$	$\texttt{fold}(e)$	constructor
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$	destructor

The statics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ consists of two forms of judgement. The first, called *type formation*, is a general hypothetical judgement of the form

$$\Delta \vdash \tau\ \mathsf{type},$$

where Δ has the form $t_1\ \mathsf{type}, \ldots, t_k\ \mathsf{type}$. Type formation is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{18.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{18.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{18.1c}$$

The second form of judgement comprising the statics is the *typing judgement*, which is a hypothetical judgement of the form

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

where we assume that τ type. Typing for $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is inductively defined by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \tag{18.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{18.2b}$$

The dynamics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is specified by one axiom stating that the elimination form is inverse to the introduction form.

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{fold}[t.\tau](e)\ \mathsf{val}} \tag{18.3a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{fold}[t.\tau](e) \mapsto \mathtt{fold}[t.\tau](e')}\right\} \tag{18.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{18.3c}$$

$$\frac{\mathtt{fold}[t.\tau](e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \mapsto e} \tag{18.3d}$$

The bracketed premise and rule are to be included for an *eager* interpretation of the introduction form, and omitted for a *lazy* interpretation.

It is a straightforward exercise to prove type safety for $\mathcal{L}\{+\times\rightharpoonup\mu\}$.

Theorem 18.1 (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 $e \mapsto e'$.

18.2 Recursive Data Structures

One important application of recursive types is to the representation of inductive data types such as the type of natural numbers. We may think of the type `nat` as a solution (up to isomorphism) of the type equation

$$\mathtt{nat} \cong [\mathtt{z} : \mathtt{unit}, \mathtt{s} : \mathtt{nat}]$$

According to this isomorphism every natural number is either zero or the successor of another natural number. A solution is given by the recursive type

$$\mu t.\,[\mathtt{z}:\mathtt{unit},\mathtt{s}:t]. \tag{18.4}$$

The introductory forms for the type `nat` are defined by the following equations:

$$\begin{aligned}\mathtt{z} &= \mathtt{fold}(\mathtt{z}\cdot\langle\rangle)\\ \mathtt{s}(e) &= \mathtt{fold}(\mathtt{s}\cdot e).\end{aligned}$$

The conditional branch may then be defined as follows:

$$\mathtt{ifz}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \;=\; \mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{z}\cdot\_\Rightarrow e_0 \mid \mathtt{s}\cdot x\Rightarrow e_1\},$$

where the "underscore" indicates a variable that does not occur free in e_0. It is easy to check that these definitions exhibit the expected behavior.

As another example, the type `list` of lists of natural numbers may be represented by the recursive type

$$\mu t.\,[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times t]$$

so that we have the isomorphism

$$\mathtt{list}\cong[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times\mathtt{list}].$$

The list formation operations are represented by the following equations:

$$\begin{aligned}\mathtt{nil} &= \mathtt{fold}(\mathtt{n}\cdot\langle\rangle)\\ \mathtt{cons}(e_1;e_2) &= \mathtt{fold}(\mathtt{c}\cdot\langle e_1,e_2\rangle).\end{aligned}$$

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned}&\mathtt{listcase}\,e\,\{\mathtt{nil}\Rightarrow e_0 \mid \mathtt{cons}(x;y)\Rightarrow e_1\} =\\ &\qquad\qquad \mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{n}\cdot\_\Rightarrow e_0 \mid \mathtt{c}\cdot\langle x,y\rangle\Rightarrow e_1\},\end{aligned}$$

where we have used an underscore for a "don't care" variable, and used pattern-matching syntax to bind the components of a pair.

As long as sums and products are evaluated eagerly, 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. If sums or products are evaluated lazily, then the blackboard notation breaks down because it is unable to depict the suspended computations that are present in the data structure. In general there is no substitute for the type itself. Drawings can be helpful, but the type determines the semantics.

We may also represent coinductive types, such as the type of streams of natural numbers, using recursive types. The representation is particularly natural in the case that `fold(`$-$`)` is evaluated lazily, for then we may define the type `stream` to be the recursive type

$$\mu t.\texttt{nat} \times t.$$

This states that every stream may be thought of as a computation of a pair consisting of a number and another stream. If `fold(`$-$`)` is evaluated eagerly, then we may instead consider the recursive type

$$\mu t.\texttt{unit} \to (\texttt{nat} \times t),$$

which expresses the same representation of streams. In either case streams cannot be easily depicted in blackboard notation, not so much because they are infinite, but because there is no accurate way to depict the delayed computation other than by an expression in the programming language. Here again we see that pictures can be helpful, but are not adequate for accurately defining a data structure.

18.3 Self-Reference

In the general recursive expression, `fix[`τ`](`$x.e$`)`, the variable, x, stands for the expression itself. This is ensured by the unrolling transition

$$\texttt{fix[}\tau\texttt{]}(x.e) \mapsto [\texttt{fix[}\tau\texttt{]}(x.e)/x]e,$$

which substitutes the expression itself for x in its body during execution. It is useful to think of x as an *implicit argument* to e, which is to be thought of as a function of x that it implicitly implied to the recursive expression itself whenever it is used. In many well-known languages this implicit argument has a special name, such as `this` or `self`, that emphasizes its self-referential interpretation.

Using this intuition as a guide, we may derive general recursion from recursive types. This derivation shows that general recursion may, like other language features, be seen as a manifestation of type structure, rather than an *ad hoc* language feature. The derivation is based on isolating a type of self-referential expressions of type τ, written $\texttt{self}(\tau)$. The introduction form of this type is (a variant of) general recursion, written $\texttt{self}[\tau](x.e)$, and the elimination form is an operation to unroll the recursion by one step, written $\texttt{unroll}(e)$. The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \texttt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{self}[\tau](x.e) : \texttt{self}(\tau)} \tag{18.5a}$$

$$\frac{\Gamma \vdash e : \texttt{self}(\tau)}{\Gamma \vdash \texttt{unroll}(e) : \tau} \tag{18.5b}$$

The dynamics is given by the following rule for unrolling the self-reference:

$$\frac{}{\texttt{self}[\tau](x.e)\ \mathsf{val}} \tag{18.6a}$$

$$\frac{e \mapsto e'}{\texttt{unroll}(e) \mapsto \texttt{unroll}(e')} \tag{18.6b}$$

$$\frac{}{\texttt{unroll}(\texttt{self}[\tau](x.e)) \mapsto [\texttt{self}[\tau](x.e)/x]e} \tag{18.6c}$$

The main difference, compared to general recursion, is that we distinguish a type of self-referential expressions, rather than impose self-reference at every type. However, as we shall see shortly, the self-referential type is sufficient to implement general recursion, so the difference is largely one of technique.

The type $\texttt{self}(\tau)$ is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type τ to be a function of the expression itself. That is, we seek to define the type $\texttt{self}(\tau)$ so that it satisfies the isomorphism

$$\texttt{self}(\tau) \cong \texttt{self}(\tau) \to \tau.$$

This means that we seek a fixed point of the type operator $t.t \to \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\texttt{rec}(t.t \to \tau),$$

which we take as the definition of $\texttt{self}(\tau)$.

The self-referential expression $\texttt{self}[\tau](x.e)$ is then defined to be the expression

$$\texttt{fold}(\lambda\,(x{:}\texttt{self}(\tau).e)).$$

We may easily check that Rule (18.5a) is derivable according to this definition. The expression $\texttt{unroll}(e)$ is correspondingly defined to be the expression

$$\texttt{unfold}(e)(e).$$

It is easy to check that Rule (18.5b) is derivable from this definition. Moreover, we may check that

$$\texttt{unroll}(\texttt{self}[\tau](y.e)) \mapsto^* [\texttt{self}[\tau](y.e)/y]e.$$

This completes the derivation of the type $\texttt{self}(\tau)$ of self-referential expressions of type τ.

One consequence of admitting the self-referential type $\texttt{self}(\tau)$ is that we may use it to define general recursion at *any* type. To be precise, we may define $\texttt{fix}[\tau](x.e)$ to stand for the expression

$$\texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))$$

in which we have unrolled the recursion at each occurrence of x within e. It is easy to check that this verifies the statics of general recursion given in Chapter 12. Moreover, it also validates the dynamics, as evidenced by the following derivation:

$$\begin{aligned}\texttt{fix}[\tau](x.e) &= \texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))\\ &\mapsto^* [\texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))/x]e\\ &= [\texttt{fix}[\tau](x.e)/x]e.\end{aligned}$$

It follows that recursive types may be used to define a non-terminating expression of every type, namely $\texttt{fix}[\tau](x.x)$. Unlike many other type constructs we have considered, recursive types change the meaning of *every* type, not just those that involve recursion. Recursive types are therefore said to be a *non-conservative extension* of languages such as $\mathcal{L}\{\texttt{nat}\to\}$, which otherwise admits no non-terminating computations.

18.4 Notes

The systematic study of recursive types in programming was initiated by Scott [94, 95] to provide a mathematical model of the untyped λ-calculus.

The derivation of recursion from recursive types is essentially an application of Scott's theory to find the interpretation of a fixed point combinator in a model of the λ-calculus given by a recursive type. The general theory of recursive types was studied by Smyth and Plotkin [96] from a category-theoretic perspective.

Part VII

Dynamic Types

Chapter 19

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!

19.1 The λ-Calculus

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

Exp	u	::=	x	x	variable
			$\lambda(x.u)$	$\lambda\, x.u$	λ-abstraction
			$\mathtt{ap}(u_1;u_2)$	$u_1(u_2)$	application

The statics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical judgements of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression

[1] An apt description of Dana Scott's.

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:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{19.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(u_1; u_2)\ \mathsf{ok}} \tag{19.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda(x.u)\ \mathsf{ok}} \tag{19.1c}$$

The dynamics is given by the following rules:

$$\frac{}{\lambda(x.u)\ \mathsf{val}} \tag{19.2a}$$

$$\frac{}{\mathtt{ap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{19.2b}$$

$$\frac{u_1 \mapsto u_1'}{\mathtt{ap}(u_1; u_2) \mapsto \mathtt{ap}(u_1'; u_2)} \tag{19.2c}$$

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.

Despite the apparent lack of types, $\mathcal{L}\{\lambda\}$ is nevertheless type safe!

Theorem 19.1. *If u ok, then either u val, or there exists u' such that $u \mapsto u'$ and u' ok.*

Proof. Exactly as in preceding chapters. We may show by induction on transition that well-formation is preserved by the dynamics. Since every closed value of $\mathcal{L}\{\lambda\}$ is a λ-abstraction, every closed expression is either a value or can make progress. □

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 u and u' are terms having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u\ \mathsf{ok} \vdash u \equiv u} \tag{19.3a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{19.3b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{19.3c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2')} \tag{19.3d}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \tag{19.3e}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\lambda(x.e_2); e_1) \equiv [e_1/x]e_2} \tag{19.3f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

19.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressiveness. 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 in a gravitational field.[2]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 12. 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.

[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 proper 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.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} = \lambda\, b.\, \lambda\, s.\, b \tag{19.4a}$$

$$\overline{n+1} = \lambda\, b.\, \lambda\, s.\, s(\overline{n}(b)(s)) \tag{19.4b}$$

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:

$$\mathtt{succ} = \lambda\, x.\, \lambda\, b.\, \lambda\, s.\, s(x(b)(s)) \tag{19.5}$$

$$\mathtt{plus} = \lambda\, x.\, \lambda\, y.\, y(x)(\mathtt{succ}) \tag{19.6}$$

$$\mathtt{times} = \lambda\, x.\, \lambda\, y.\, y(\overline{0})(\mathtt{plus}(x)) \tag{19.7}$$

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.

To define $\mathtt{ifz}(u; u_0; x\,.\,u_1)$ requires a bit of ingenuity. We wish to find a term $\mathtt{pred}$ such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{19.8}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{19.9}$$

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.

$$\langle u_1, u_2 \rangle = \lambda f.\, f(u_1)(u_2) \tag{19.10}$$
$$u \cdot \mathtt{l} = u(\lambda x.\, \lambda y.\, x) \tag{19.11}$$
$$u \cdot \mathtt{r} = u(\lambda x.\, \lambda y.\, y) \tag{19.12}$$

It is easy to check that under this encoding $\langle u_1, u_2 \rangle \cdot \mathtt{l} \equiv u_1$, and that a similar equivalence holds for the second projection. We may now define the required representation, u_p, of the predecessor function:

$$u'_p = \lambda x.\, x(\langle \overline{0}, \overline{0} \rangle)(\lambda y.\, \langle y \cdot \mathtt{r}, \mathtt{s}(y \cdot \mathtt{r}) \rangle) \tag{19.13}$$
$$u_p = \lambda x.\, u(x) \cdot \mathtt{l} \tag{19.14}$$

It is easy to check that this gives us the required behavior. Finally, we may define $\mathtt{ifz}(u; u_0; x\,.\,u_1)$ to be the untyped term

$$u(u_0)(\lambda \_.\, [u_p(u)/x]u_1).$$

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*:

$$\mathbf{Y} = \lambda F.\, (\lambda f.\, F(f(f)))(\lambda f.\, F(f(f))). \tag{19.15}$$

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.

19.3 Scott's Theorem

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 11 for a discussion of Gödel-numbering.)

2. Any two non-trivial[3] properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms that respect definitional equivalence are *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, if $\mathcal{A}_0$ and $\mathcal{A}_1$ are inseparable, then neither is 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 19.2. *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 19.3. *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] A property of untyped terms is said to be *trivial* if it either holds for all untyped terms or never holds for any untyped term.

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 19.2 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 19.4. *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. □

19.4 Untyped Means Uni-Typed

The untyped λ-calculus may be faithfully embedded in a typed language with recursive types. This means that every untyped λ-term has a representation as a typed expression in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the λ-calculus in $\mathcal{L}\{+\times\rightharpoonup\mu\}$ (which we could surely do), but rather a direct representation of untyped λ-terms as typed expressions in a language with recursive types.

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 translation, of $\mathcal{L}\{\lambda\}$ into $\mathcal{L}\{+\times\rightharpoonup\mu\}$:

$$x^\dagger = x \tag{19.16a}$$
$$\lambda\, x.\, u^\dagger = \mathtt{fold}(\lambda\ (x\mathtt{:}D.\, u^\dagger)) \tag{19.16b}$$
$$u_1\,(u_2)^\dagger = \mathtt{unfold}(u_1^\dagger)\,(u_2^\dagger) \tag{19.16c}$$

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 20 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.

19.5 Notes

The untyped λ-calculus was introduced by Church [21] in the 1930's as a codification of the informal concept of a computable function. Unlike

the well-known machine models, such as the Turing machine or the random access machine, the λ-calculus directly codifies mathematical and programming practice. The definitive reference for all aspects of the untyped λ-calculus is Barendregt's text [10]. In particular, the proof of Scott's theorem given here is adapted from Barendregt's account. The reduction of untyped to typed λ-calculus via the concept of a recursive type was achieved by Scott in his pioneering work on the semantics of the λ-calculus [93].

Chapter 20

Dynamic Typing

We saw in Chapter 19 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 purported opposition to *statically typed* languages. But the supposed opposition is illusory. Just as the untyped λ-calculus is really unityped, so dynamic languages are special cases of static languages.

20.1 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:

Exp	d	::=	x	variable
			$\texttt{num}(\overline{n})$	numeral
			$\texttt{zero}$	zero
			$\texttt{succ}(d)$	successor
			$\texttt{ifz}(d; d_0; x.d_1)$	$\texttt{ifz}\, d\, \{\texttt{zero} \Rightarrow d_0 \mid \texttt{succ}(x) \Rightarrow d_1\}$
				zero test
			$\texttt{fun}(\lambda\, x.d)$	abstraction
			$\texttt{dap}(d_1; d_2)$	$d_1(d_2)$ application
			$\texttt{fix}(x.d)$	$\texttt{fix}\, x\, \texttt{is}\, d$ recursion

There are two classes of values in $\mathcal{L}\{dyn\}$, the *numbers*, which have the form $\overline{n}$,[1] and the *functions*, which have the form $\lambda(x.d)$. The expressions $\texttt{zero}$ and $\texttt{succ}(d)$ are not in themselves values, but rather are operations that evaluate to classified values.

The concrete syntax of $\mathcal{L}\{dyn\}$ is somewhat deceptive, in keeping with common practice in dynamic languages. For example, the concrete syntax for a number is a bare numeral, $\overline{n}$, but in fact it is just a convenient notation for the classified value, $\texttt{num}(\overline{n})$, of class $\texttt{num}$. Similarly, the concrete syntax for a function is a λ-abstraction, $\lambda(x.d)$, which must be regarded as standing for the classified value $\texttt{fun}(\lambda\, x.d)$ of class $\texttt{fun}$.

The statics of $\mathcal{L}\{dyn\}$ is essentially the same as that of $\mathcal{L}\{\lambda\}$ given in Chapter 19; 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 dynamics of $\mathcal{L}\{dyn\}$ 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.

The value judgement, $d\ \mathsf{val}$, states that d is a fully evaluated (closed) expression:

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{val}} \tag{20.1a}$$

$$\frac{}{\texttt{fun}(\lambda\, x.d)\ \mathsf{val}} \tag{20.1b}$$

[1] The numerals, $\overline{n}$, are n-fold compositions of the form $\texttt{s}(\texttt{s}(\ldots\texttt{s}(\texttt{z})\ldots))$.

The dynamics makes use of judgements that check the class of a value, and recover the underlying λ-abstraction in the case of a function.

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{20.2a}$$

$$\frac{}{\texttt{fun}(\lambda\, x.\, d)\ \mathsf{is\_fun}\ x\,.\,d} \tag{20.2b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\texttt{num}(\_)\ \mathsf{isnt\_fun}} \tag{20.3a}$$

$$\frac{}{\texttt{fun}(\_)\ \mathsf{isnt\_num}} \tag{20.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, $d\ \mathsf{err}$, are defined simultaneously by the following rules:[2]

$$\frac{}{\texttt{zero} \mapsto \texttt{num(z)}} \tag{20.4a}$$

$$\frac{d \mapsto d'}{\texttt{succ}(d) \mapsto \texttt{succ}(d')} \tag{20.4b}$$

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\texttt{succ}(d) \mapsto \texttt{num(s}(\overline{n})\texttt{)}} \tag{20.4c}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{succ}(d)\ \mathsf{err}} \tag{20.4d}$$

$$\frac{d \mapsto d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto \texttt{ifz}(d'; d_0; x\,.\,d_1)} \tag{20.4e}$$

$$\frac{d\ \mathsf{is\_num}\ \texttt{z}}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto d_0} \tag{20.4f}$$

$$\frac{d\ \mathsf{is\_num}\ \texttt{s}(\overline{n})}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto [\texttt{num}(\overline{n})\,/\,x]d_1} \tag{20.4g}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{ifz}(d; d_0; x\,.\,d_1)\ \mathsf{err}} \tag{20.4h}$$

$$\frac{d_1 \mapsto d_1'}{\texttt{dap}(d_1; d_2) \mapsto \texttt{dap}(d_1'; d_2)} \tag{20.4i}$$

[2]The obvious error propagation rules discussed in Chapter 8 are omitted here for the sake of concision.

$$\frac{d_1 \text{ is\_fun } x.d}{\texttt{dap}(d_1;d_2) \mapsto [d_2/x]d} \tag{20.4j}$$

$$\frac{d_1 \text{ isnt\_fun}}{\texttt{dap}(d_1;d_2) \text{ err}} \tag{20.4k}$$

$$\frac{}{\texttt{fix}(x.d) \mapsto [\texttt{fix}(x.d)/x]d} \tag{20.4l}$$

Rule (20.4g) labels the predecessor with the class `num` to maintain the invariant that variables are bound to expressions of $\mathcal{L}\{dyn\}$.

The language $\mathcal{L}\{dyn\}$ enjoys essentially the same safety properties as $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, except that there are more opportunities for errors to arise at run-time.

Theorem 20.1 (Safety)**.** *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 (20.4). 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. □

20.2 Variations and Extensions

The dynamic language $\mathcal{L}\{dyn\}$ defined in Section 20.1 on page 177 closely parallels the static language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 12. One discrepancy, however, is in the treatment of natural numbers. Whereas in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ the zero and successor operations are introductory forms for the type `nat`, in $\mathcal{L}\{dyn\}$ they are elimination forms that act on separately-defined numerals. The point of this representation is to ensure that there is a well-defined class of *numbers* in the language.

It is worthwhile to explore an alternative representation that, superficially, is even closer to $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. Suppose that we eliminate the expression $\texttt{num}(\overline{n})$ from the language, but retain `zero` and $\texttt{succ}(d)$, with the idea that these are to be thought of as introductory forms for numbers in the language. We are faced with the problem that such an expression is well-formed for *any* well-formed d. So, in particular, the expression $\texttt{succ}(\lambda(x.d))$ is a value, as is $\texttt{succ}(\texttt{zero})$. There is no longer a well-defined class of *numbers*, but rather two separate classes of values, zero and successor, with no assurance that the successor is of a number.

The dynamics of the conditional branch changes only slightly, as described by the following rules:

$$\frac{d \mapsto d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto \texttt{ifz}(d'; d_0; x\,.\,d_1)} \tag{20.5a}$$

$$\frac{d\ \textsf{is\_zero}}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto d_0} \tag{20.5b}$$

$$\frac{d\ \textsf{is\_succ}\ d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto [d'/x]d_1} \tag{20.5c}$$

$$\frac{d\ \textsf{isnt\_zero} \quad d\ \textsf{isnt\_succ}}{\texttt{ifz}(d; d_0; x\,.\,d_1)\ \textsf{err}} \tag{20.5d}$$

The foregoing rules are to be augmented by the following rules that check whether a value is of class zero or successor:

$$\frac{}{\texttt{zero}\ \textsf{is\_zero}} \tag{20.6a}$$

$$\frac{}{\texttt{succ}(d)\ \textsf{isnt\_zero}} \tag{20.6b}$$

$$\frac{}{\texttt{succ}(d)\ \textsf{is\_succ}\ d} \tag{20.6c}$$

$$\frac{}{\texttt{zero}\ \textsf{isnt\_succ}} \tag{20.6d}$$

A peculiarity of this formulation of the conditional is that it can only be understood as distinguishing `zero` from `succ(_)`, rather than as distinguishing `zero` from non-`zero`. The reason is that if d is not `zero`, it might be either a successor or a function, and hence its "predecessor" is not well-defined.

Similar considerations arise when enriching $\mathcal{L}\{dyn\}$ with structured data. The classic example is to enrich the language as follows:

Exp	d	$::=$	$\texttt{nil}$	$\texttt{nil}$	null
			$\texttt{cons}(d_1; d_2)$	$\texttt{cons}(d_1; d_2)$	pair
			$\texttt{ifnil}(d; d_0; x, y\,.\,d_1)$	$\texttt{ifnil}\,d\,\{\texttt{nil} \Rightarrow d_0 \mid \texttt{cons}(x; y) \Rightarrow d_1\}$	conditional

The expression $\texttt{ifnil}(d; d_0; x, y\,.\,d_1)$ distinguishes the null structure from the pair of two structures. We leave to the reader the exercise of formulating the dynamics of this extension.

An advantage of dynamic typing is that the constructors `nil` and `cons(`d_1`;`d_2`)` are sufficient to build unbounded, as well as bounded, data structures such as lists or trees. For example, the list consisting of three zero's may be represented by the value

```
cons(zero;cons(zero;cons(zero;nil))).
```

But what to make of this beast?

```
cons(zero;cons(zero;cons(zero;λ(x)x))).
```

It is a perfectly valid expression, but does not correspond to any natural data structure.

The disadvantage of this representation becomes apparent as soon as one wishes to define operations on lists, such as the append function:

$$\texttt{fix}\,a\,\texttt{is}\,\lambda(x.\lambda(y.\texttt{ifnil}(x;y;x_1,x_2.\texttt{cons}(x_1;a(x_2)(y)))))$$

What if x is the second list-like value given above? As it stands, the append function will signal an error upon reaching the function at the end of the list. If, however, y is this value, no error is signalled. This asymmetry may seem innocuous, but it is only one simple manifestation of a pervasive problem with dynamic languages: it is impossible to state within the language even the most rudimentary assumptions about the inputs, such as the assumption that both arguments to the append function ought to be genuine lists.

The conditional expression `ifnil(`d`;`d_0`;`x`,`y`.`d_1`)` is rather *ad hoc* in that it makes a distinction between `nil` and all other values. Why not distinguish successors from non-successors, or functions from non-functions? A more systematic approach is to enrich the language with *predicates* and *destructors*. Predicates determine whether a value is of a specified class, and destructors recover the value labelled with a given class.

Exp	d ::=	`cond(`d`;`d_0`;`d_1`)`	`cond(`d`;`d_0`;`d_1`)`	conditional
		`nil?(`d`)`	`nil?(`d`)`	nil test
		`cons?(`d`)`	`cons?(`d`)`	pair test
		`car(`d`)`	`car(`d`)`	first projection
		`cdr(`d`)`	`cdr(`d`)`	second projection

The conditional `cond(`d`;`d_0`;`d_1`)` distinguishes d between `nil` and *all other values*. If d is not `nil`, the conditional evaluates to d_0, and otherwise evaluates to d_1. In other words the value `nil` represents boolean falsehood,

and all other values represent boolean truth. The predicates `nil?(`d`)` and `cons?(`d`)` test the class of their argument, yielding `nil` if the argument is not of the specified class, and yielding some non-`nil` if so. The destructors `car(`d`)` and `cdr(`d`)`[3] decompose `cons(`d_1`;`d_2`)` into d_1 and d_2, respectively. As an example, the append function may be defined using predicates as follows:

$$\texttt{fix}\, a\, \texttt{is}\, \lambda(x.\lambda(y.\texttt{cond}(x;\texttt{cons}(\texttt{car}(x);a(\texttt{cdr}(x))(y));y))).$$

20.3 Critique of Dynamic Typing

The safety theorem for $\mathcal{L}\{dyn\}$ is often promoted as an advantage of dynamic over static typing. Unlike static languages, which rule out some candidate programs as ill-typed, essentially every piece of abstract syntax in $\mathcal{L}\{dyn\}$ is well-formed, and hence, by Theorem 20.1 on page 180, has a well-defined dynamics. But this can also be seen as a disadvantage, since errors that could be ruled out at compile time by type checking are not signalled until run time in $\mathcal{L}\{dyn\}$. To make this possible, the dynamics of $\mathcal{L}\{dyn\}$ must enforce conditions that need not be checked in a statically typed language.

Consider, for example, the addition function in $\mathcal{L}\{dyn\}$, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[4]

$$\texttt{fun}(\lambda\, x.\texttt{fix}(p.\texttt{fun}(\lambda\, y.\texttt{ifz}(y;x;y'.\texttt{succ}(p(y')))))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda(x.\texttt{fix}\, p\, \texttt{is}\, \lambda(y.\texttt{ifz}\, y\, \{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{succ}(p(y'))\})).$$

It is deceptive, because the concrete syntax obscures the class tags on values, and obscures the use of primitives that check those tags. Let us now examine the costs of these operations in a bit more detail.

First, observe that the body of the fixed point expression is labelled with class `fun`. The dynamics of the fixed point construct binds p to this function. This means that the dynamic class check incurred by the application of p in

[3] This terminology for the projections is archaic, but firmly established in the literature.

[4] This specification imposes no restrictions on the behavior of addition on arguments that are not classified as numbers, but one could make the further demand that the function abort when applied to arguments that are not classified by `num`.

the recursive call is guaranteed to succeed. But $\mathcal{L}\{dyn\}$ offers no means of suppressing this redundant check, because it cannot express the invariant that p is always bound to a value of class `fun`.

Second, observe that the result of applying the inner λ-abstraction is either x, the argument of the outer λ-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class `num`, even though this is guaranteed for all but the base case, which returns the given x, which can be of any class at all. In principle we can check that x is of class `num` once, and observe that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, $\mathcal{L}\{dyn\}$ gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

Third, the argument, y, to the inner function is either the original argument to the addition function, or is the predecessor of some earlier recursive call. But as long as the original call is to a value of class `num`, then the dynamics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in $\mathcal{L}\{dyn\}$, and hence there is no way to avoid the class check imposed by the conditional branch.

Classification is not free—storage is required for the class label, and it takes time to detach the class from a value each time it is used and to attach a class to a value whenever it is created. Although the overhead of classification 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 this is impossible within $\mathcal{L}\{dyn\}$, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

20.4 Notes

The earliest dynamically typed language is Lisp [63], which continues to influence language design a half decade after its invention. Dynamic PCF is essentially the core of Lisp, but with a proper treatment of variable binding, correcting what McCarthy himself as described as an error in the original. Informal discussions of dynamic languages are often confused by the ellision of the dynamic checks that are made explicit here. While the surface syntax of dynamic PCF is essentially the same as that for PCF, minus the type annotations, the underlying dynamics is fundamentally different. It

is for this reason that static PCF cannot be properly seen as a restriction of dynamic PCF by the imposition of a type system, contrary to what seems to be a widely held belief. It is simply not accurate to state that a static type sytem is a *post hoc* restriction imposed on a dynamically typed language.

Chapter 21

Hybrid Typing

A *hybrid* language is one that combines static and dynamic typing by enriching a statically typed language with a distinguished type, `dyn`, of dynamic values. The dynamically typed language considered in Chapter 20 may be embedded into the hybrid language by regarding a dynamically typed program as a statically typed program of type `dyn`. This shows that static and dynamic types are not opposed to one another, but may coexist harmoniously.

The notion of a hybrid language, however, is itself illusory, because the type `dyn` is really a particular recursive type. This shows that there is no need for any special mechanisms to support dynamic typing. Rather, they may be derived from the more general concept of a recursive type. Moreover, this shows that *dynamic typing is but a mode of use of static typing*! The supposed opposition between dynamic and static typing is, therefore, a fallacy: dynamic typing can hardly be opposed to that of which it is but a special case!

21.1 A Hybrid Language

Consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$, which extends $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ (defined in Chapter 12) with the following additional constructs:

Typ	τ	::=	`dyn`	`dyn`	dynamic
Exp	e	::=	`new[`l`](`e`)`	$l \cdot e$	construct
			`cast[`l`](`e`)`	$e \cdot l$	destruct
Cls	l	::=	`num`	`num`	number
			`fun`	`fun`	function

The type dyn is the type of dynamically classified values. The new operation attaches a classifier to a value, and the cast operation checks the classifier and returns the associated value.

The statics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn}\ \rightharpoonup\}$ extends that of $\mathcal{L}\{\mathtt{nat}\ \rightharpoonup\}$ with the following additional rules:

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{new}[\mathtt{num}]\,(e) : \mathtt{dyn}} \tag{21.1a}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{fun}]\,(e) : \mathtt{dyn}} \tag{21.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}]\,(e) : \mathtt{nat}} \tag{21.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}]\,(e) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}} \tag{21.1d}$$

The statics 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 dynamics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn}\ \rightharpoonup\}$ extends that of $\mathcal{L}\{\mathtt{nat}\ \rightharpoonup\}$ with the following rules:

$$\frac{e\ \mathsf{val}}{\mathtt{new}[l]\,(e)\ \mathsf{val}} \tag{21.2a}$$

$$\frac{e \mapsto e'}{\mathtt{new}[l]\,(e) \mapsto \mathtt{new}[l]\,(e')} \tag{21.2b}$$

$$\frac{e \mapsto e'}{\mathtt{cast}[l]\,(e) \mapsto \mathtt{cast}[l]\,(e')} \tag{21.2c}$$

$$\frac{\mathtt{new}[l]\,(e)\ \mathsf{val}}{\mathtt{cast}[l]\,(\mathtt{new}[l]\,(e)) \mapsto e} \tag{21.2d}$$

$$\frac{\mathtt{new}[l']\,(e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l]\,(\mathtt{new}[l']\,(e))\ \mathsf{err}} \tag{21.2e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 21.1 (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 $e' : \mathtt{dyn} \rightharpoonup \mathtt{dyn}$.*

Proof. By a straightforward rule induction on the statics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn}\ \rightharpoonup\}$. □

Theorem 21.2 (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 dynamics, and progress is proved by rule induction on the statics, 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 cast does not match the class of the value. □

21.2 Optimization of Dynamic Typing

The language $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ combines static and dynamic typing by enriching $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the type, dyn, of classified values. It is, for this reason, called a *hybrid* language. Unlike a purely dynamic type system, a hybrid type system can express invariants that are crucial to the optimization of programs in $\mathcal{L}\{dyn\}$.

Let us examine this in the case of the addition function, which may be defined in $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ as follows:

$$\texttt{fun} \cdot \lambda\,(x\texttt{:dyn}.\,\texttt{fix}\,p\texttt{:dyn is fun} \cdot \lambda\,(y\texttt{:dyn}.\,e_{x,p,y})),$$

where

$$x : \texttt{dyn}, p : \texttt{dyn}, y : \texttt{dyn} \vdash e_{x,p,y} : \texttt{dyn}$$

is defined to be the expression

$$\texttt{ifz}\,(y \cdot \texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num} \cdot (\texttt{s}((p \cdot \texttt{fun})(\texttt{num} \cdot y') \cdot \texttt{num}))\}.$$

This is a reformulation of the dynamic addition function given in Section 20.3 on page 183 in which we have made explicit the checking and imposition of classes on values. We will exploit the static type system of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ to optimize this dynamically typed implementation of addition in accordance with the specification given in Section 20.3 on page 183.

First, note 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 rewriting the function as follows:

$$\texttt{fun} \cdot \lambda\,(x\texttt{:dyn}.\,\texttt{fun} \cdot \texttt{fix}\,p\texttt{:dyn} \rightharpoonup \texttt{dyn is}\,\lambda\,(y\texttt{:dyn}.\,e'_{x,p,y})),$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y \cdot \texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num} \cdot (\texttt{s}(p(\texttt{num} \cdot y') \cdot \texttt{num}))\}.$$

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 because it is provided by the caller of the function. However, we may reasonably insist that it is an error to call addition with a non-numeric argument. This canbe enforced by replacing x in the zero branch of the conditional by $x \cdot \texttt{num}$.

Combining these optimizations we obtain the inner loop e''_x defined as follows:

$$\texttt{fix}\,p{:}\texttt{nat} \rightharpoonup \texttt{nat}\,\texttt{is}\,\lambda\,(y{:}\texttt{nat}.\,\texttt{ifz}\,y\,\{\texttt{zero} \Rightarrow x \cdot \texttt{num} \mid \texttt{succ}(y') \Rightarrow \texttt{s}(p(y'))\}).$$

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:

$$\texttt{fun} \cdot \lambda\,(x\texttt{:dyn}.\,\texttt{fun} \cdot \lambda\,(y\texttt{:dyn}.\,\texttt{num} \cdot (e_x''(y \cdot \texttt{num})))).$$

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.

21.3 Static "Versus" Dynamic Typing

There are many attempts to distinguish dynamic from static typing, all 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.

Another way to differentiate dynamic from static languages is to say that whereas static languages check types at compile time, dynamic languages check types at run time. But to say that static languages check types statically is to state a tautology, and to say that dynamic languages check types at run-time is to utter 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 dynamics 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 collections, whereas static languages admit only homogeneous collections. For example, in a dynamic language the elements of a list may be of disparate *classes*, as illustrated by the expression

$$\texttt{cons(s(z);cons}(\lambda(\lambda(x.x))\texttt{;nil)).}$$

But they are nevertheless all of the same *type*! Put the other way around, a static language with a dynamic type is just as capable of representing a heterogeneous collection as is a dynamic language with only one type.

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 21.2 on page 189.

21.4 Reduction to 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:

$$\texttt{dyn} = \mu t\,.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t] \tag{21.3}$$

$$\texttt{new}[\texttt{num}]\,(e) = \texttt{fold}(\texttt{num}\cdot e) \tag{21.4}$$

$$\texttt{new}[\texttt{fun}]\,(e) = \texttt{fold}(\texttt{fun}\cdot e) \tag{21.5}$$

$$\texttt{cast}[\texttt{num}]\,(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow x \mid \texttt{fun}\cdot x \Rightarrow \texttt{error}\} \tag{21.6}$$

$$\texttt{cast}[\texttt{fun}]\,(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow \texttt{error} \mid \texttt{fun}\cdot x \Rightarrow x\} \tag{21.7}$$

One may readily check that the static and dynamics for the type `dyn` are derivable according to these definitions.

This encoding readily generalizes to any number of classes of values: we need only consider additional summands corresponding to each class. For example, to account for the constructors `nil` and $\texttt{cons}(d_1;d_2)$ considered in Chapter 20, the definition of `dyn` is expanded to the recursive type

$$\mu t\,.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t,\texttt{nil}:\texttt{unit},\texttt{cons}:t \times t],$$

with corresponding definitions for the `new` and `cast` operations. This exemplifies the general case: dynamic typing is a mode of use of static types in which classes of values are simply names of summands in a recursive type of dynamic values.

21.5 Notes

The concept of a "hybrid" type system is wholly artificial. It is introduced here as an explanatory bridge between dynamic and static languages. The reality is that dynamic languages are statically typed. The only point of discussing hybrid typing is to show that one can expose more or less of the underlying static type system in a so-called dynamic language. In the general case dynamic typing is but a particular mode of use of static typing, rather than being, as commonly thought, opposed to it. This point of view is essentially due to Scott [93], who also proposed replacing the word "untyped" by "unityped" in programming languages.

Part VIII

Variable Types

Chapter 22

Girard's System F

The languages we have considered so far are all *monomorphic* in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in $\mathcal{L}\{\mathtt{nat}\to\}$ there is a *distinct* identity function for each type τ, namely $\lambda\,(x\!:\!\tau.\,x)$, even though the behavior is the same for each choice of τ. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f\!:\!\tau_2\to\tau_3.\,\lambda\,(g\!:\!\tau_1\to\tau_2.\,\lambda\,(x\!:\!\tau_1.\,f(g(x))))).$$

Each choice of the three types requires a *different* program, even though they all exhibit the same behavior when executed.

Obviously it would be useful to capture the general pattern once and for all, and to instantiate this pattern each time we need it. The expression patterns codify generic (type-independent) behaviors that are shared by all instances of the pattern. Such generic expressions are said to be *polymorphic*. 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*. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 23), and the definability of product, sum, inductive, and coinductive types considered in the preceding chapters. (Only general recursive types extend the expressive power of the language.)

22.1 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 grammar:

Typ	τ	$::=$	t	t	variable
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Exp	e	$::=$	x	x	
			$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{Lam}(t.e)$	$\Lambda(t.e)$	type abstraction
			$\mathtt{App}[\tau](e)$	$e[\tau]$	type application

A *type abstraction*, $\mathtt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* t standing for an unspecified type within e. A *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 statics of $\mathcal{L}\{\to\forall\}$ consists of two judgement forms, the *type formation* judgement,

$$\vec{t} \mid \Delta \vdash \tau\ \mathsf{type},$$

and the *typing judgement*,

$$\vec{t}\,\vec{x} \mid \Delta\ \Gamma \vdash e : \tau.$$

These are generic judgements over *type variables* $\vec{t}$ and *expression variables* $\vec{x}$. They are also hypothetical in a set Δ of *type assumptions* of the form $t\ \mathsf{type}$, where $t \in \mathcal{T}$, and *typing assumptions* of the form $x : \tau$, where $x \in \mathcal{T}$ and $\Delta \vdash \tau\ \mathsf{type}$. As usual we drop explicit mention of the parameter sets, relying on typographical conventions to determine them.

The rules defining the type formation judgement are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{22.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{22.1b}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \mathtt{all}(t.\tau) \text{ type}} \tag{22.1c}$$

The rules defining the typing judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{22.2a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1; \tau_2)} \tag{22.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{22.2c}$$

$$\frac{\Delta, t \text{ type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{22.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \text{ type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{22.2e}$$

Lemma 22.1 (Regularity)**.** *If $\Delta\ \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i$ type for each assumption $x_i : \tau_i$ in Γ, then $\Delta \vdash \tau$ type.*

Proof. By induction on Rules (22.2). □

The statics admits the structural rules for a general hypothetical judgement. In particular, we have the following critical substitution property for type formation and expression typing.

Lemma 22.2 (Substitution)**.**

1. *If Δ, t type $\vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau'$ type.*
2. *If Δ, t type $\Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau$ type, then $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.*
3. *If $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta\ \Gamma \vdash e : \tau$, then $\Delta\ \Gamma \vdash [e/x]e' : \tau'$.*

The second part of the lemma requires substitution into the context, Γ, as well as into the term and its type, because the type variable t may occur freely in any of these positions.

Returning to the motivating examples from the introduction, the polymorphic identity function, I, is written

$$\Lambda(t.\lambda\ (x{:}t.\,x));$$

it has the polymorphic type

$$\forall(t.t \to t).$$

Instances of the polymorphic identity are written $I[\tau]$, where τ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, C, is written

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda\,(f:t_2 \to t_3.\lambda\,(g:t_1 \to t_2.\lambda\,(x:t_1.f(g(x))))))))).$$

The function C has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Instances of C are obtained by applying it to a triple of types, writing $C[\tau_1][\tau_2][\tau_3]$. Each such instance has the type

$$(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3).$$

Dynamics

The dynamics of $\mathcal{L}\{\to\forall\}$ is given as follows:

$$\frac{}{\mathtt{lam}[\tau]\,(x.e)\ \mathsf{val}} \tag{22.3a}$$

$$\frac{}{\mathtt{Lam}(t.e)\ \mathsf{val}} \tag{22.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau_1]\,(x.e);e_2) \mapsto [e_2/x]e} \tag{22.3c}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{22.3d}$$

$$\frac{}{\mathtt{App}[\tau]\,(\mathtt{Lam}(t.e)) \mapsto [\tau/t]e} \tag{22.3e}$$

$$\frac{e \mapsto e'}{\mathtt{App}[\tau]\,(e) \mapsto \mathtt{App}[\tau]\,(e')} \tag{22.3f}$$

Rule (22.3d) endows $\mathcal{L}\{\to\forall\}$ with a call-by-name interpretation of application. One could easily define a call-by-value variant as well.

It is a simple matter to prove safety for $\mathcal{L}\{\to\forall\}$, using familiar methods.

Lemma 22.3 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam}[\tau_1]\,(x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \mathtt{all}(t.\tau')$, then $e = \mathtt{Lam}(t.e')$ with t type $\vdash e' : \tau'$.*

Proof. By rule induction on the statics. □

Theorem 22.4 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on the dynamics. □

Theorem 22.5 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on the statics. □

22.2 Polymorphic Definability

The language $\mathcal{L}\{\to\forall\}$ is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types! This is most naturally expressed using definitional equivalence, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash \lambda\,(x{:}\tau.\,e_2)\,(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{22.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.\,e)\,[\rho] \equiv [\rho/t]e : [\rho/t]\tau} \tag{22.4b}$$

In addition there are rules omitted here specifying that definitional equivalence is reflexive, symmetric, and transitive, and that it is compatible with both forms of application and abstraction.

22.2.1 Products and Sums

The nullary product, or unit, type is definable in $\mathcal{L}\{\to\forall\}$ as follows:

$$\begin{aligned}\mathtt{unit} &= \forall(r.\,r \to r)\\ \langle\rangle &= \Lambda(r.\,\lambda\,(x{:}r.\,x))\end{aligned}$$

The identity function plays the role of the null tuple, since it is the only closed value of this type.

Binary products are definable in $\mathcal{L}\{\to\forall\}$ by using encoding tricks similar to those described in Chapter 19 for the untyped λ-calculus:

$$\begin{aligned}\tau_1 \times \tau_2 &= \forall(r.\,(\tau_1 \to \tau_2 \to r) \to r)\\ \langle e_1, e_2\rangle &= \Lambda(r.\,\lambda\,(x{:}\tau_1 \to \tau_2 \to r.\,x(e_1)(e_2)))\\ e \cdot \mathtt{l} &= e[\tau_1]\,(\lambda\,(x{:}\tau_1.\,\lambda\,(y{:}\tau_2.\,x)))\\ e \cdot \mathtt{r} &= e[\tau_2]\,(\lambda\,(x{:}\tau_1.\,\lambda\,(y{:}\tau_2.\,y)))\end{aligned}$$

The statics given in Chapter 13 is derivable according to these definitions. Moreover, the following definitional equivalences are derivable in $\mathcal{L}\{\rightarrow\forall\}$ from these definitions:

$$\langle e_1, e_2\rangle \cdot \mathtt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle \cdot \mathtt{r} \equiv e_2 : \tau_2.$$

The nullary sum, or void, type is definable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned} \mathtt{void} &= \forall(r.r) \\ \mathtt{abort}[\rho](e) &= e[\rho] \end{aligned}$$

There is no definitional equivalence to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned} \tau_1 + \tau_2 &= \forall(r.(\tau_1 \rightarrow r) \rightarrow (\tau_2 \rightarrow r) \rightarrow r) \\ \mathtt{l} \cdot e &= \Lambda(r.\lambda\,(x{:}\tau_1 \rightarrow r.\lambda\,(y{:}\tau_2 \rightarrow r.x(e)))) \\ \mathtt{r} \cdot e &= \Lambda(r.\lambda\,(x{:}\tau_1 \rightarrow r.\lambda\,(y{:}\tau_2 \rightarrow r.y(e)))) \\ \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} &= \\ e[\rho](\lambda\,(x_1{:}\tau_1.e_1))(\lambda\,(x_2{:}\tau_2.e_2)) & \end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\mathtt{case}\,\mathtt{l}\cdot d_1\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\mathtt{case}\,\mathtt{r}\cdot d_2\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 14 is correctly implemented by these definitions.

22.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\rightarrow\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{natiter}(e_0; e_1; x.e_2) : \tau}.$$

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 we may define the type of natural numbers in $\mathcal{L}\{\to\forall\}$ by the following equations:

$$\begin{aligned}
\mathtt{nat} &= \forall(t.t \to (t \to t) \to t)\\
\mathtt{z} &= \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \to t.z)))\\
\mathtt{s}(e) &= \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \to t.s(e[t](z)(s)))))\\
\mathtt{natiter}(e_0; e_1; x.e_2) &= e_0[\tau](e_1)(\lambda(x{:}\tau.e_2))
\end{aligned}$$

It is a straightforward exercise to check that the static and dynamics given in Chapter 11 is derivable in $\mathcal{L}\{\to\forall\}$ under these definitions.

This shows that $\mathcal{L}\{\to\forall\}$ is *at least as expressive* as $\mathcal{L}\{\mathtt{nat}\to\}$. But is it *more* expressive? Yes! It is possible to show that the evaluation function for $\mathcal{L}\{\mathtt{nat}\to\}$ is definable in $\mathcal{L}\{\to\forall\}$, even though it is not definable in $\mathcal{L}\{\mathtt{nat}\to\}$ itself. However, the same diagonal argument given in Chapter 11 applies here, showing that the evaluation function for $\mathcal{L}\{\to\forall\}$ is not definable in $\mathcal{L}\{\to\forall\}$. We may enrich $\mathcal{L}\{\to\forall\}$ a bit more to define the evaluator for $\mathcal{L}\{\to\forall\}$, but as long as all programs in the enriched language terminate, we will once again have an undefinable function, the evaluation function for that extension.

22.3 Parametricity Overview

A remarkable property of $\mathcal{L}\{\to\forall\}$ is that polymorphic types severely constrain the behavior of their elements. One may prove useful theorems about an expression knowing *only* its type—that is, without ever looking at the code! For example, if i is *any* expression of type $\forall(t.t \to t)$, then it must be the identity function. Informally, when i is applied to a type, τ, and

an argument of type τ, it must return a value of type τ. But since τ is not specified until i is called, the function has no choice but to return its argument, which is to say that it is essentially the identity function. Similarly, if b is *any* expression of type $\forall(t.t \to t \to t)$, then b must be either $\Lambda(t.\lambda\,(x{:}t.\lambda\,(y{:}t.x)))$ or $\Lambda(t.\lambda\,(x{:}t.\lambda\,(y{:}t.y)))$. For when b is applied to two arguments of some type, its only choice to return a value of that type is to return one of the two.

What is remarkable is that these properties of i and b have been derived *without knowing anything about the expressions themselves*, but only their types! The theory of parametricity implies that we are able to derive theorems about the behavior of a program knowing only its type. Such theorems are sometimes called *free theorems* because they come "for free" as a consequence of typing, and require no program analysis or verification to derive (beyond the once-and-for-all proof of Theorem 51.8 on page 523). Free theorems such as those illustrated above underly the experience that in a polymorphic language, well-typed programs tend to behave as expected no further debugging or analysis required. Parametricity so constrains the behavior of a program that it is relatively easy to ensure that the code works just by checking its type. Free theorems also underly the principle of representation independence for abstract types, which is discussed further in Chapter 23.

22.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of $\mathcal{L}\{\to\forall\}$. These are obtained in one of two ways:

1. Restricting type quantification to unquantified types.
2. Restricting the occurrence of quantifiers within types.

22.4.1 Predicative Fragment

The remarkable expressive power of the language $\mathcal{L}\{\to\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let τ be the type $\forall(t.t \to t)$, and, assuming that $e : \tau$, we may apply e to its own type, obtaining the expression $e[\tau]$ of type $\tau \to \tau$. Written out in full, this is the type

$$\forall(t.t \to t) \to \forall(t.t \to t),$$

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of e itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to e again, obtaining the expression $e[\tau](e)$, which is again of type τ — the very type of e!

This property of $\mathcal{L}\{\rightarrow\forall\}$ is called *impredicativity*[1]; the language $\mathcal{L}\{\rightarrow\forall\}$ is said to permit *impredicative (type) quantification*. The distinguishing characteristic of impredicative polymorphism is that it involves a kind of circularity in that the meaning of a quantified type is given in terms of its instances, including the quantified type itself. This quasi-circularity is responsible for the surprising expressive power of $\mathcal{L}\{\rightarrow\forall\}$, and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of $\mathcal{L}\{\rightarrow\forall\}$ terminate).

Contrast this with $\mathcal{L}\{\rightarrow\}$, in which the type of an application of a function is evidently smaller than the type of the function itself. For if $e : \tau_1 \rightarrow \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of e. This situation extends to polymorphism, provided that we impose the restriction that a quantified type can only be instantiated by an un-quantified type. For in that case passage from $\forall(t.\tau)$ to $[\rho/t]\tau$ decreases the number of quantifiers (even if the size of the type expression viewed as a tree grows). For example, the type $\forall(t.t \rightarrow t)$ may be instantiated with the type $u \rightarrow u$ to obtain the type $(u \rightarrow u) \rightarrow (u \rightarrow u)$. This type has more symbols in it than τ, but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called *predicative*[2] *polymorphism*. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the natural numbers are no longer definable in it.

22.4.2 Prenex Fragment

A rather more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is often imposed for the sake of type inference, which permits type annotations to be omitted entirely in the knowledge that they can be recovered from the way the expression is used. We will not discuss type inference here, but we will give a formulation of the prenex fragment

[1] pronounced *im-PRED-ic-a-tiv-it-y*
[2] pronounced *PRED-i-ca-tive*

of $\mathcal{L}\{\to\forall\}$, because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of $\mathcal{L}\{\to\forall\}$ is designated $\mathcal{L}^1\{\to\forall\}$, for reasons that will become clear in the next subsection. It is defined by *stratifying* types into two sorts, the *monotypes* (or *rank*-0 types) and the *polytypes* (or *rank*-1 types). The monotypes are those that do not involve any quantification, and may be used to instantiate the polymorphic quantifier. The polytypes include the monotypes, but also permit quantification over monotypes. These classifications are expressed by the judgements $\Delta \vdash \tau \ \mathsf{mono}$ and $\Delta \vdash \tau \ \mathsf{poly}$, where Δ is a finite set of hypotheses of the form $t \ \mathsf{mono}$, where t is a type variable not otherwise declared in Δ. The rules for deriving these judgements are as follows:

$$\frac{}{\Delta, t \ \mathsf{mono} \vdash t \ \mathsf{mono}} \tag{22.5a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{mono} \quad \Delta \vdash \tau_2 \ \mathsf{mono}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{mono}} \tag{22.5b}$$

$$\frac{\Delta \vdash \tau \ \mathsf{mono}}{\Delta \vdash \tau \ \mathsf{poly}} \tag{22.5c}$$

$$\frac{\Delta, t \ \mathsf{mono} \vdash \tau \ \mathsf{poly}}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{poly}} \tag{22.5d}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The statics of $\mathcal{L}^1\{\to\forall\}$ is given by rules for deriving hypothetical judgements of the form $\Delta\ \Gamma \vdash e : \rho$, where Δ consists of hypotheses of the form $t \ \mathsf{mono}$, and Γ consists of hypotheses of the form $x : \rho$, where $\Delta \vdash \rho \ \mathsf{poly}$. The rules defining this judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{22.6a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)} \tag{22.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{22.6c}$$

$$\frac{\Delta, t \ \mathsf{mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{22.6d}$$

$$\frac{\Delta \vdash \tau \ \mathsf{mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{22.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgements have the form $e : \rho$ for some expression e and polytype ρ.

The restriction on the domain of a λ-abstraction to be a monotype means that a fully general `let` construct is no longer definable—there is no means of binding an expression of polymorphic type to a variable. For this reason it is usual to augment $\mathcal{L}\{\rightarrow\forall_{\mathsf{p}}\}$ with a primitive `let` construct whose statics is as follows:

$$\frac{\Delta \vdash \tau_1 \ \mathsf{poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}\,[\tau_1]\,(e_1; x.e_2) : \tau_2}\ . \tag{22.7}$$

For example, the expression

$$\mathtt{let}\ I : \forall(t.t \rightarrow t)\ \mathtt{be}\ \Lambda(t.\lambda\ (x:t.x))\ \mathtt{in}\ I\,[\tau \rightarrow \tau]\,(I\,[\tau])$$

has type $\tau \rightarrow \tau$ for any polytype τ.

22.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathcal{L}^1\{\rightarrow\forall\}$ may be generalized to form a hierarchy of languages in which the occurrences of polymorphic types are restricted in relation to function types. The key feature of the prenex fragment is that quantified types are not permitted to occur in the domain of a function type. The prenex fragment also prohibits polymorphic types from the range of a function type, but it would be harmless to admit it, there being no significant difference between the type $\rho \rightarrow \forall(t.\tau)$ and the type $\forall(t.\rho \rightarrow \tau)$ (where $t \notin \rho$). This motivates the definition of a hierarchy of fragments of $\mathcal{L}\{\rightarrow\forall\}$ that subsumes the prenex fragment as a special case.

We will define a judgement of the form $\tau\ \mathsf{type}\,[k]$, where $k \geq 0$, to mean that τ is a type of *rank* k. Informally, types of rank 0 have no quantification, and types of rank $k+1$ may involve quantification, but the domains of function types are restricted to be of rank k. Thus, in the terminology of Section 22.4.2 on page 205, a monotype is a type of rank 0 and a polytype is a type of rank 1.

The definition of the types of rank k is defined simultaneously for all k by the following rules. These rules involve hypothetical judgements of the form $\Delta \vdash \tau\ \mathsf{type}\,[k]$, where Δ is a finite set of hypotheses of the form $t_i\ \mathsf{type}\,[k_i]$ for some pairwise distinct set of type variables t_i. The rules defining these judgements are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\,[k] \vdash t\ \mathsf{type}\,[k]} \tag{22.8a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type } [0] \quad \Delta \vdash \tau_2 \text{ type } [0]}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2) \text{ type } [0]} \tag{22.8b}$$

$$\frac{\Delta \vdash \tau_1 \text{ type } [k] \quad \Delta \vdash \tau_2 \text{ type } [k+1]}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2) \text{ type } [k+1]} \tag{22.8c}$$

$$\frac{\Delta \vdash \tau \text{ type } [k]}{\Delta \vdash \tau \text{ type } [k+1]} \tag{22.8d}$$

$$\frac{\Delta, t \text{ type } [k] \vdash \tau \text{ type } [k+1]}{\Delta \vdash \texttt{all}(t.\tau) \text{ type } [k+1]} \tag{22.8e}$$

With these restrictions in mind, it is a good exercise to define the statics of $\mathcal{L}^k\{\to\forall\}$, the restriction of $\mathcal{L}\{\to\forall\}$ to types of rank k (or less). It is most convenient to consider judgements of the form $e : \tau\,[k]$ specifying simultaneously that $e : \tau$ and τ type $[k]$. For example, the rank-limited rules for λ-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1 \text{ type } [0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \texttt{lam[}\tau_1\texttt{]}(x.e_2) : \texttt{arr}(\tau_1;\tau_2)\ [0]} \tag{22.9a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type } [k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \texttt{lam[}\tau_1\texttt{]}(x.e_2) : \texttt{arr}(\tau_1;\tau_2)\ [k+1]} \tag{22.9b}$$

The remaining rules follow a similar pattern.

The rank-limited languages $\mathcal{L}^k\{\to\forall\}$ clarifies the requirement for a primitive `let` construct in $\mathcal{L}^1\{\to\forall\}$. The prenex fragment of $\mathcal{L}\{\to\forall\}$ corresponds to the rank-one fragment $\mathcal{L}^1\{\to\forall\}$. The `let` construct for rank-one types is definable in $\mathcal{L}^2\{\to\forall\}$ from λ-abstraction and application. This definition only makes sense at rank two, since it abstracts over a rank-one polymorphic type.

22.5 Notes

System F was introduced by Girard [32] in the context of proof theory and Reynolds [87] in the context of programming languages. The concept of parametric polymorphism was originally isolated by Strachey, but was not fully developed until the work of Reynolds [86]. One may see the original ML type system [65] as the restriction of System F to rank 1. Extensions to higher ranks give greater expressive power, but at the expense of more difficult type checking and inference problems.

Chapter 23

Abstract Types

Data abstraction is perhaps the most important technique for structuring programs. The main idea is to introduce an *interface* that serves as a contract between the *client* and the *implementor* of an abstract type. The interface specifies what the client may rely on for its own work, and, simultaneously, what the implementor must provide to satisfy the contract. The interface serves to isolate the client from the implementor so that each may be developed in isolation from the other. In particular one implementation may be replaced by another without affecting the behavior of the client, provided that the two implementations meet the same interface and are, in a sense to be made precise below, suitably related to one another. (Roughly, each simulates the other with respect to the operations in the interface.) This property is called *representation independence* for an abstract type.

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! One consequence of this observation is that representation independence is just a use of the parametricity properties of polymorphic

functions discussed in Chapter 22.

23.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

Typ	τ	$::=$	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$	interface
Exp	e	$::=$	$\texttt{pack}[t.\tau][\rho](e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$	implementation
			$\texttt{open}[t.\tau][\rho](e_1;t,x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x:\tau\,\texttt{in}\,e_2$	client

The introductory form for the existential type $\exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$, where ρ is a type and e is an expression of type $[\rho/t]\tau$. The type ρ is called the *representation type* of the package, and the expression e is called the *implementation* of the package. The eliminatory form for existentials is the expression $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x:\tau\,\texttt{in}\,e_2$, which *opens* the package e_1 for use within the *client* e_2 by binding its representation type to t and its implementation to x for use within e_2. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the `open` construct specifies that the type variable, t, and the expression variable, x, are bound within the client. They may be renamed at will by α-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, t, may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to α-equivalence, and is not particularly tied to data abstraction.

23.1.1 Statics

The statics 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.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{23.1a}$$

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \mathtt{pack}[t.\tau]\,[\rho]\,(e) : \mathtt{some}(t.\tau)} \tag{23.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{some}(t.\tau) \quad \Delta, t \text{ type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta\ \Gamma \vdash \mathtt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) : \tau_2} \tag{23.1c}$$

Rule (23.1c) 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.

Lemma 23.1 (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$*. If* $\Delta \vdash \tau_i$ *type for each* $x_i : \tau_i$ *in* Γ*, then* $\Delta \vdash \tau$ *type.*

Proof. By induction on Rules (23.1). □

23.1.2 Dynamics

The (eager or lazy) dynamics of existential types is specified as follows:

$$\frac{\{e \text{ val}\}}{\mathtt{pack}[t.\tau]\,[\rho]\,(e) \text{ val}} \tag{23.2a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{pack}[t.\tau]\,[\rho]\,(e) \mapsto \mathtt{pack}[t.\tau]\,[\rho]\,(e')}\right\} \tag{23.2b}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) \mapsto \mathtt{open}[t.\tau]\,[\tau_2]\,(e_1'; t, x.e_2)} \tag{23.2c}$$

$$\frac{\{e \text{ val}\}}{\mathtt{open}[t.\tau]\,[\tau_2]\,(\mathtt{pack}[t.\tau]\,[\rho]\,(e); t, x.e_2) \mapsto [\rho, e/t, x]e_2} \tag{23.2d}$$

It is important to observe that, according to these rules, *there are no abstract types at run time*! The representation type is propagated to the client by substitution when the package is opened, thereby eliminating the abstraction boundary between the client and the implementor. Thus, data abstraction is a *compile-time discipline* that leaves no traces of its presence at execution time.

23.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\to\forall\}$ to the new constructs.

Theorem 23.2 (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 23.3 (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' such that $e' : [\rho/t]\tau$.*

Proof. By rule induction on the statics, making use of the definition of closed values. □

Theorem 23.4 (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. □

23.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of queues of natural numbers 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, which is assumed to be non-empty.

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, $\exists(t.\tau)$, which serves as the interface of the queue abstraction:

$$\exists(t.\langle \texttt{emp} : t, \texttt{ins} : \texttt{nat} \times t \to t, \texttt{rem} : t \to \texttt{nat} \times t\rangle).$$

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. Here is a very simple implementation, e_l, in which queues are represented as lists:

$$\texttt{pack}\ \texttt{list}\ \texttt{with}\ \langle \texttt{emp} = \texttt{nil}, \texttt{ins} = e_i, \texttt{rem} = e_r \rangle\ \texttt{as}\ \exists(t.\tau),$$

where

$$e_i : \texttt{nat} \times \texttt{list} \to \texttt{list} = \lambda\ (x{:}\texttt{nat} \times \texttt{list}.\, e_i'),$$

and

$$e_r : \texttt{list} \to \texttt{nat} \times \texttt{list} = \lambda\ (x{:}\texttt{list}.\, e_r').$$

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 `list`, and are programmed accordingly.

It is also possible to give another implementation, e_p, of the same interface, $\exists(t.\tau)$, 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:

$$\texttt{pack}\ \texttt{list} \times \texttt{list}\ \texttt{with}\ \langle \texttt{emp} = \langle \texttt{nil}, \texttt{nil} \rangle, \texttt{ins} = e_i, \texttt{rem} = e_r \rangle\ \texttt{as}\ \exists(t.\tau).$$

In this case e_i has type

$$\texttt{nat} \times (\texttt{list} \times \texttt{list}) \to (\texttt{list} \times \texttt{list}),$$

and e_r has type

$$(\texttt{list} \times \texttt{list}) \to \texttt{nat} \times (\texttt{list} \times \texttt{list}).$$

These operations "know" that queues are represented as values of type $\texttt{list} \times \texttt{list}$, and are implemented accordingly.

The important point is that the *same* client type checks regardless of which implementation of queues we choose. This is because the representation type is hidden, or *held abstract*, from the client during type checking.

Consequently, it cannot rely on whether it is `list` or `list` $\times$ `list` or some other type. That is, the client is *independent* of the representation of the abstract type.

23.3 Definability of Existentials

It turns out that it is not necessary to extend $\mathcal{L}\{\rightarrow\forall\}$ 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

$$\texttt{open}\, e_1\, \texttt{as}\, t\, \texttt{with}\, x{:}\tau\, \texttt{in}\, e_2 : \tau_2,$$

where $e_1 : \exists(t.\tau)$, specifies that $e_2 : \tau_2$ under the assumptions t type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau \rightarrow \tau_2),$$

where t may occur in τ (the type of the operations), but not in τ_2 (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned}
\exists(t.\tau) &= \forall(u.\forall(t.\tau \rightarrow u) \rightarrow u)\\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau) &= \Lambda(u.\lambda\,(x{:}\forall(t.\tau \rightarrow u).\,x[\rho](e)))\\
\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2 &= e_1[\tau_2](\Lambda(t.\lambda\,(x{:}\tau.e_2)))
\end{aligned}$$

An existential is encoded as a polymorphic function taking the overall result type, u, as argument, followed by a polymorphic function representing the client with result type u, and yielding a value of type u 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 ρ and an implementation e is a polymorphic function that, when given the result type, t, and the client, x, instantiates x with ρ and passes to it the implementation e.

It is then a straightforward exercise to show that this translation correctly reflects the statics and dynamics of existential types.

23.4 Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types. More precisely, there is a criterion, called *bisimilarity*, for relating two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation by another that is bisimilar to it. This leads to a simple methodology for proving the correctness of *candidate* implementation of an abstract type, which is to show that it is bisimilar to an obviously correct *reference* implementation of it. Since the candidate and the reference implementations are bisimilar, no client may distinguish them from one another, and hence if the client behaves properly with the reference implementation, then it must also behave properly with the candidate.

To derive the definition of bisimilarity of implementations, it is helpful to examine the definition of existentials in terms of universals given in Section 23.3 on the preceding page. It is an immediate consequence of the definition that the client of an abstract type is polymorphic in the representation of the abstract type. A client, c, of an abstract type $\exists(t.\tau)$ has type $\forall(t.\tau \to \tau_2)$, where t does not occur free in τ_2 (but may, of course, occur in τ). Applying the parametricity property described informally in Chapter 22 (and developed rigorously in Chapter 51), this says that if R is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on both of them. The fact that t does not occur in the result type ensures that the behavior of the client is independent of the choice of relation between the implementations, provided that this relation is preserved by the operation that implement it.

To see what this means requires that we specify what is meant by a bisimulation. This is best done by example. Consider the existential type $\exists(t.\tau)$, where τ is the labelled tuple type

$$\langle \mathtt{emp} : t, \mathtt{ins} : \tau \times t \to t, \mathtt{rem} : t \to \tau \times t \rangle.$$

Theorem 51.8 on page 523 ensures that if ρ and ρ' are any two closed types, R is a relation between expressions of these two types, then if any the implementations $e : [\rho/x]\tau$ and $e' : [\rho'/x]\tau$ respect R, then $c\,[\rho]\,e$ behaves the same as $c\,[\rho']\,e'$. It remains to define when two implementations respect the relation R. Let

$$e = \langle \mathtt{emp} = e_{\mathsf{m}}, \mathtt{ins} = e_{\mathsf{i}}, \mathtt{rem} = e_{\mathsf{r}} \rangle$$

and

$$e' = \langle \mathtt{emp} = e'_{\mathsf{m}}, \mathtt{ins} = e'_{\mathsf{i}}, \mathtt{rem} = e'_{\mathsf{r}} \rangle.$$

For these implementations to respect R means that the following three conditions hold:

1. The empty queues are related: $R(e_{\mathsf{m}}, e'_{\mathsf{m}})$.

2. Inserting the same element on each of two related queues yields related queues: if $d : \tau$ and $R(q, q')$, then $R(e_{\mathsf{i}}(d)(q), e'_{\mathsf{i}}(d)(q'))$.

3. If two queues are related, their front elements are the same and their back elements are related: if $R(q, q')$, $e_{\mathsf{r}}(q) \cong \langle d, r\rangle$, $e'_{\mathsf{r}}(q') \cong \langle d', r'\rangle$, then d is d' and $R(r, r')$.

If such a relation R exists, then the implementations e and e' are said to be *bisimilar*. The terminology stems from the requirement that the operations of the abstract type preserve the relation: if it holds before an operation is performed, then it must also hold afterwards, and the relation must hold for the initial state of the queue. Thus each implementation *simulates* the other up to the relationship specified by R.

To see how this works in practice, let us consider informally two implementations of the abstract type of queues specified above. For the reference implementation we choose ρ to be the type `list`, and define the empty queue to be the empty list, insert to add the specified element to the front of the list, and remove to remove the last element of the list. (A remove therefore takes time linear in the length of the list.) For the candidate implementation we choose ρ' to be the type `list` $\times$ `list` consisting of two lists, $\langle b, f\rangle$, where b represents the "back" of the queue, and f represents the "front" of the queue represented in reverse order of insertion. The empty queue consists of two empty lists. To insert d onto $\langle b, f\rangle$, we simply return $\langle \texttt{cons}(d; b), f\rangle$, placing it on the "back" of the queue as expected. To remove an element from $\langle b, f\rangle$ breaks into two cases. If the front, f, of the queue is non-empty, say $\texttt{cons}(d; f')$, then return $\langle d, \langle b, f'\rangle\rangle$ consisting of the front element and the queue with that element removed. If, on the other hand, f is empty, then we must move elements from the "back" to the "front" by reversing b and re-performing the remove operation on $\langle \texttt{nil}, \texttt{rev}(b)\rangle$, where `rev` is the obvious list reversal function.

To show that the candidate implementation is correct, we show that it is bisimilar to the reference implementation. This reduces to specifying a relation, R, between the types `list` and `list` $\times$ `list` such that the three simulation conditions given above are satisfied by the two implementations just described. The relation in question states that $R(l, \langle b, f\rangle)$ iff the list l is the list $\texttt{app}(b)(\texttt{rev}(f))$, where `app` is the evident append function

on lists. That is, thinking of l as the reference representation of the queue, the candidate must maintain that the elements of b followed by the elements of f in reverse order form precisely the list l. It is easy to check that the implementations just described preserve this relation. Having done so, we are assured that the client, c, behaves the same regardless of whether we use the reference or the candidate. Since the reference implementation is obviously correct (albeit inefficient), the candidate must also be correct in that the behavior of any client is unaffected by using it instead of the reference.

23.5 Notes

The connection between abstract types in programming languages and existential types in logic was made by Mitchell and Plotkin [71], although some of the ideas were already present in Reynolds work [87]. The account of representation independence given here is derived from Mitchell [69].

Chapter 24

Constructors and Kinds

The types $\texttt{nat} \to \texttt{nat}$ and $\texttt{nat}\,\texttt{list}$ may be thought of as being built from other types by the application of a *type constructor*, or *type operator*. These two examples differ from each other in that the function space type constructor takes two arguments, whereas the list type constructor takes only one. We may, for the sake of uniformity, think of types such as $\texttt{nat}$ as being built by a type constructor of *no* arguments. More subtly, we may even think of the types $\forall(t.\tau)$ and $\exists(t.\tau)$ as being built up in the same way by regarding the quantifiers as *higher-order* type operators.

These seemingly disparate cases may be treated uniformly by enriching the syntactic structure of a language with a new layer of *constructors*. To ensure that constructors are used properly (for example, that the list constructor is given only one argument, and that the function constructor is given two), we classify constructors by *kinds*. Constructors of a distinguished kind, $\texttt{T}$, are types, which may be used to classify expressions. To allow for multi-argument and higher-order constructors, we will also consider finite product and function kinds. (Later we shall consider even richer kinds.)

The distinction between constructors and kinds on one hand and types and expressions on the other reflects a fundamental separation between the static and dynamic *phase* of processing of a programming language, called the *phase distinction*. The static phase implements the statics and the dynamic phase implements the dynamics. Constructors may be seen as a form of *static data* that is manipulated during the static phase of processing. Expressions are a form of *dynamic data* that is manipulated at run-time. Since the dynamic phase follows the static phase (we only execute well-typed programs), we may also manipulate constructors at run-time.

Adding constructors and kinds to a language introduces more technical complications than might at first be apparent. The main difficulty is that as soon as we enrich the kind structure beyond the distinguished kind of types, it becomes essential to simplify constructors to determine whether they are equivalent. For example, if we admit product kinds, then a pair of constructors is a constructor of product kind, and projections from a constructor of product kind are also constructors. But what if we form the first projection from the pair consisiting of the constructors `nat` and `str`? This should be equivalent to `nat`, since the elimination form if post-inverse to the introduction form. Consequently, any expression (say, a variable) of the one type should also be an expression of the other. That is, typing should respect definitional equivalence of constructors.

There are two main ways to deal with this. One is to introduce a concept of definitional equivalence for constructors, and to demand that the typing judgement for expressions respect definitional equivalence of constructors of kind `T`. This means, however, that we must show that definitional equivalence is decidable if we are to build a complete implementation of the language. The other is to prohibit formation of awkward constructors such as the projection from a pair so that there is never any issue of when two constructors are equivalent (only when they are identical). But this complicates the definition of substitution, since a projection from a constructor variable is well-formed, until you substitute a pair for the variable. Both approaches have their benefits, but the second is simplest, and is adopted here.

24.1 Statics

The syntax of kinds is given by the following grammar:

Kind	κ	::=	`Type`	`T`	types
			`Unit`	1	nullary product
			`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$	binary product
			`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \to \kappa_2$	function

The kinds consist of the kind of types, `T`, the unit kind, `Unit`, and are closed under formation of product and function kinds.

The syntax of constructors is divided into two syntactic sorts, the *neutral*

and the *canonical*, according to the following grammar:

NCon	a	$::=$	u	u	variable
			`proj[l](`a`)`	$a \cdot \mathtt{l}$	first projection
			`proj[r](`a`)`	$a \cdot \mathtt{r}$	second projection
			`app(`a_1`;`c_2`)`	$a_1\,[c_2]$	application
CCon	c	$::=$	`atom(`a`)`	$\widehat{a}$	atomic
			`unit`	$\langle\rangle$	null tuple
			`pair(`c_1`;`c_2`)`	$\langle c_1, c_2\rangle$	pair
			`lam(`u`.`c`)`	$\lambda\, u\,.\, c$	abstraction

The reason to distinguish neutral from canonical constructors is to ensure that it is impossible to apply an elimination form to an introduction form, which demands an equation to capture the inversion principle. For example, the putative constructor $\langle c_1, c_2\rangle \cdot \mathtt{l}$, which would be definitionally equivalent to c_1, is ill-formed according to Grammar (24.1). This is because the argument to a projection must be neutral, but a pair is only canonical, not neutral.

The canonical constructor $\widehat{a}$ is the inclusion of neutral constructors into canonical constructors. However, the grammar does not capture a crucial property of the statics that ensures that only neutral constructors of kind `T` may be treated as canonical. This requirement is imposed to limit the forms of canonical contructors of the other kinds. In particular, variables of function, product, or unit kind will turn out not to be canonical, but only neutral.

The statics of constructors and kinds is specified by the judgements

$\Delta \vdash a \Uparrow \kappa$	neutral constructor formation
$\Delta \vdash c \Downarrow \kappa$	canonical constructor formation

In each of these judgements Δ is a finite set of hypotheses of the form

$$u_1 \Uparrow \kappa_1, \ldots, u_n \Uparrow \kappa_n$$

for some $n \geq 0$. The form of the hypotheses expresses the principle that variables are neutral constructors. The formation judgements are to be understood as generic hypothetical judgements with parameters $u_1, \ldots, u_n$ that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow \kappa} \tag{24.1a}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{l} \Uparrow \kappa_1} \tag{24.1b}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{r} \Uparrow \kappa_2} \tag{24.1c}$$

$$\frac{\Delta \vdash a_1 \Uparrow \kappa_2 \to \kappa \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash a_1\,[c_2] \Uparrow \kappa} \tag{24.1d}$$

$$\frac{\Delta \vdash a \Uparrow \mathtt{T}}{\Delta \vdash \widehat{a} \Downarrow \mathtt{T}} \tag{24.1e}$$

$$\frac{}{\Delta \vdash \langle\rangle \Downarrow 1} \tag{24.1f}$$

$$\frac{\Delta \vdash c_1 \Downarrow \kappa_1 \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \langle c_1, c_2\rangle \Downarrow \kappa_1 \times \kappa_2} \tag{24.1g}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \lambda\, u\,.\, c_2 \Downarrow \kappa_1 \to \kappa_2} \tag{24.1h}$$

Rule (24.1e) specifies that the only neutral constructors that are canonical are those with kind $\mathtt{T}$. This ensures that the language enjoys the following canonical forms property, which is easily proved by inspection of Rules (24.1).

Lemma 24.1. *Suppose that* $\Delta \vdash c \Downarrow \kappa$.

1. *If* $\kappa = 1$, *then* $c = \langle\rangle$.
2. *If* $\kappa = \kappa_1 \times \kappa_2$, *then* $c = \langle c_1, c_2\rangle$ *for some* c_1 *and* c_2 *such that* $\Delta \vdash c_i \Downarrow \kappa_i$ *for* $i = 1, 2$.
3. *If* $\kappa = \kappa_1 \to \kappa_2$, *then* $c = \lambda\, u\,.\, c_2$ *with* $\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2$.

24.2 Higher Kinds

To equip a language, $\mathcal{L}$, with constructors and kinds requires that we augment its statics with hypotheses governing constructor variables, and that we relate constructors of kind $\mathtt{T}$ (types as static data) to the classifiers of dynamic expressions (types as classifiers). To achieve this the statics of $\mathcal{L}$ must be defined to have judgements of the following two forms:

$\Delta \vdash \tau$ type	type formation
$\Delta\ \Gamma \vdash e : \tau$	expression formation

where, as before, Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k$$

for some $k \geq 0$ such that $\Delta \vdash \tau_i$ type for each $1 \leq i \leq k$.

As a general principle, every constructor of kind $\mathtt{T}$ is a classifier:

$$\frac{\Delta \vdash \tau \Uparrow \mathtt{T}}{\Delta \vdash \tau \text{ type}} \,. \tag{24.2}$$

In many cases this is the sole rule of type formation, so that every classifier is a constructor of kind $\mathtt{T}$. However, this need not be the case. In some situations we may wish to have strictly more classifiers than constructors of the distinguished kind.

To see how this might arise, let us consider two extensions of $\mathcal{L}\{\to\forall\}$ from Chapter 22. In both cases we extend the universal quantifier $\forall(t.\tau)$ to admit quantification over an arbitrary kind, written $\forall\, u :: \kappa\,.\,\tau$, but the two languages differ in what constitutes a constructor of kind $\mathtt{T}$. In one case, the *impredicative*, we admit quantified types as constructors, and in the other, the *predicative*, we exclude quantified types from the domain of quantification.

The impredicative fragment includes the following two constructor constants:

$$\frac{}{\Delta \vdash \to \Uparrow \mathtt{T} \to \mathtt{T} \to \mathtt{T}} \tag{24.3a}$$

$$\frac{}{\Delta \vdash \forall_\kappa \Uparrow (\kappa \to \mathtt{T}) \to \mathtt{T}} \tag{24.3b}$$

We regard the classifier $\tau_1 \to \tau_2$ to be the application $\to[\tau_1]\,[\tau_2]$. Similarly, we regard the classifier $\forall\, u :: \kappa\,.\,\tau$ to be the application $\forall_\kappa[\lambda\, u\,.\,\tau]$.

The predicative fragment excludes the constant specified by Rule (24.3b) in favor of a separate rule for the formation of universally quantified types:

$$\frac{\Delta, u \Uparrow \kappa \vdash \tau \text{ type}}{\Delta \vdash \forall\, u :: \kappa\,.\,\tau \text{ type}} \,. \tag{24.4}$$

The point is that $\forall\, u :: \kappa\,.\,\tau$ is a type (as classifier), but is *not* a constructor of kind type.

The significance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

$$\frac{\Delta, u \Uparrow \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(u::\kappa\,.\,e) : \forall\, u :: \kappa\,.\,\tau} \tag{24.5a}$$

$$\frac{\Delta\ \Gamma \vdash e : \forall\, u :: \kappa\,.\,\tau \quad \Delta \vdash c \Downarrow \kappa}{\Delta\ \Gamma \vdash e\,[c] : [c/u]\tau} \tag{24.5b}$$

(Rule (24.5b) makes use of substitution, whose definition requires some care. We will return to this point in Section 24.3.)

Rule (24.5b) makes clear that a polymorphic abstraction quantifies over the constructors of kind κ. When κ is $\mathtt{T}$ this kind may or may not include all of the classifiers of the language, according to whether we are working with the impredicative formulation of quantification (in which the quantifiers are distinguished constants for building constructors of kind $\mathtt{T}$) or the predicative formulation (in which quantifiers arise only as classifiers and not as constructors).

The main idea is that *constructors are static data*, so that a constructor abstraction $\Lambda(u::\kappa\,.\,e)$ of type $\forall\, u :: \kappa\,.\,\tau$ is a mapping from static data c of kind κ to dynamic data $[c/u]e$ of type $[c/u]\tau$. Rule (24.1e) tells us that every constructor of kind $\mathtt{T}$ determines a classifier, but it may or may not be the case that every classifier arises in this manner.

24.3 Hereditary Substitution

Rule (24.5b) involves substitution of a canonical constructor, c, of kind κ into a family of types $u \Uparrow \kappa \vdash \tau$ type. This operation is is written $[c/u]\tau$, as usual. Although the intended meaning is clear, it is in fact impossible to interpret $[c/u]\tau$ as the standard concept of substitution defined in Chapter 1. The reason is that to do so would risk violating the distinction between neutral and canonical constructors. Consider, for example, the case of the family of types

$$u \Uparrow \mathtt{T} \to \mathtt{T} \vdash u\,[d] \Uparrow \mathtt{T},$$

where $d \Uparrow \mathtt{T}$. (It is not important what we choose for d, so we leave it abstract.) Now if $c \Downarrow \mathtt{T} \to \mathtt{T}$, then by Lemma 24.1 on page 222 we have that c is $\lambda\, u'\,.\,c'$. Thus, if interpreted conventionally, substitution of c for u in the given family yields the "constructor" $(\lambda\, u'\,.\,c')\,[d]$, which is not well-formed.

The solution is to define a form of *canonizing substitution* that simplifies such "illegal" combinations as it performs the replacement of a variable by a constructor of the same kind. In the case just sketched this means that we must ensure that

$$[\lambda\, u'\,.\,c'/u]u\,[d] = [d/u']c'.$$

If viewed as a definition this equation is problematic because it switches from substituting for u in the constructor $u\,[d]$ to substituting for u' in the

unrelated constructor c'. Why should such a process terminate? The answer lies in the observation that the kind of u' is definitely smaller than the kind of u, since the former's kind is the domain kind of the latter's function kind. In all other cases of substitution (as we shall see shortly) the size of the target of the substitution becomes smaller; in the case just cited the size may increase, but the type of the target variable decreases. Therefore by a lexicographic induction on the type of the target variable and the structure of the target constructor, we may prove that canonizing substitution is well-defined.

We now turn to the task of making this precise. We will define simultaneously two principal forms of substitution, one of which divides into two cases:

$[c/u:\kappa]a = a'$	canonical into neutral yielding neutral
$[c/u:\kappa]a = c' \Downarrow \kappa'$	canonical into neutral yielding canonical and kind
$[c/u:\kappa]c' = c''$	canonical into canonical yielding canonical

Substitution into a neutral constructor divides into two cases according to whether the substituted variable u occurs in *critical position* in a sense to be made precise below.

These forms of substitution are simultaneously inductively defined by the following rules, which are broken into groups for clarity.

The first set of rules defines substitution of a canonical constructor into a canonical constructor; the result is always canonical.

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\widehat{a'} = \widehat{a''}} \tag{24.6a}$$

$$\frac{[c/u:\kappa]a' = c'' \Downarrow \kappa''}{[c/u:\kappa]\widehat{a'} = c''} \tag{24.6b}$$

$$\frac{}{[u/\langle\rangle:\kappa]=\langle\rangle} \tag{24.6c}$$

$$\frac{[c/u:\kappa]c_1' = c_1'' \quad [c/u:\kappa]c_2' = c_2''}{[c/u:\kappa]\langle c_1', c_2'\rangle = \langle c_1'', c_2''\rangle} \tag{24.6d}$$

$$\frac{[c/u:\kappa]c' = c'' \quad (u \neq u') \quad (u' \notin c)}{[c/u:\kappa]\lambda\, u'.\, c' = \lambda\, u'.\, c''} \tag{24.6e}$$

The conditions on variables in Rule (24.6e) may always be met by renaming the bound variable, u', of the abstraction.

The second set of rules defines substitution of a canonical constructor into a neutral constructor, yielding another neutral constructor.

$$\frac{(u \neq u')}{[c/u:\kappa]u' = u'} \tag{24.7a}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]a' \cdot \mathtt{l} = a'' \cdot \mathtt{l}} \tag{24.7b}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]a' \cdot \mathtt{r} = a'' \cdot \mathtt{r}} \tag{24.7c}$$

$$\frac{[c/u:\kappa]a_1 = a_1' \quad [c/u:\kappa]c_2 = c_2'}{[c/u:\kappa]a_1\,[c_2] = a_1'\,(c_2')} \tag{24.7d}$$

Rule (24.7a) pertains to a *non-critical* variable, which is not the target of substitution. The remaining rules pertain to situations in which the recursive call on a neutral constructor yields a neutral constructor.

The third set of rules defines substitution of a canonical constructor into a neutral constructor, yielding a canonical constructor and its kind.

$$\frac{}{[c/u:\kappa]u = c \Downarrow \kappa} \tag{24.8a}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]a' \cdot \mathtt{l} = c_1' \Downarrow \kappa_1'} \tag{24.8b}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]a' \cdot \mathtt{r} = c_2' \Downarrow \kappa_2'} \tag{24.8c}$$

$$\frac{[c/u:\kappa]a_1' = \lambda\, u'.\, c' \Downarrow \kappa_2' \to \kappa' \quad [c/u:\kappa]c_2' = c_2'' \quad [c_2''/u':\kappa_2']c' = c''}{[c/u:\kappa]a_1'\,[c_2'] = c'' \Downarrow \kappa'} \tag{24.8d}$$

Rule (24.8a) governs a *critical* variable, which is the target of substitution. The substitution transforms it from a neutral constructor to a canonical constructor. This has a knock-on effect in the remaining rules of the group, which analyze the canonical form of the result of the recursive call to determine how to proceed. Rule (24.8d) is the most interesting rule. In the third premise, all three arguments to substitution change as we substitute the (substituted) argument of the application for the parameter of the (substituted) function into the body of that function. Here we require the type of the function in order to determine the type of its parameter.

Theorem 24.2. *Suppose that* $\Delta \vdash c \Downarrow \kappa$, *and* $\Delta, u \Uparrow \kappa \vdash c' \Downarrow \kappa'$, *and* $\Delta, u \Uparrow \kappa \vdash a' \Uparrow \kappa'$. *There exists a unique* $\Delta \vdash c'' \Downarrow \kappa'$ *such that* $[c/u:\kappa]c' = c''$. *Either there exists a unique* $\Delta \vdash a'' \Uparrow \kappa'$ *such that* $[c/u:\kappa]a' = a''$, *or there exists a unique* $\Delta \vdash c'' \Downarrow \kappa'$ *such that* $[c/u:\kappa]a' = c''$, *but not both.*

Proof. Simultaneously by a lexicographic induction with major component the structure of the kind κ, and with minor component determined by Rules (24.1) governing the formation of c' and a'. For all rules except Rule (24.8d) the inductive hypothesis applies to the premise(s) of the relevant formation rules. For Rule (24.8d) we appeal to the major inductive hypothesis applied to κ_2', which is a component of the kind $\kappa_2' \to \kappa'$. □

24.4 Canonization

With hereditary substitution in hand, it is perfectly possible to confine our attention to constructors in canonical form. However, for some purposes it can be useful to admit a more relaxed syntax in which it is possible to form non-canonical constructors that can nevertheless be transformed into canonical form. The prototypical example is the constructor $(\lambda\, u\,.\, c_2)\,[c_1]$, which is malformed according to Rules (24.1), because the first argument of an application is required to be in atomic form, whereas the λ-abstraction is in canonical form. However, if c_1 and c_2 are already canonical, then the malformed application may be transformed into the well-formed canonical form $[v_1/u]c_2$, where substitution is as defined in Section 24.3 on page 224. If c_1 or c_2 are not already canonical we may, inductively, put them into canonical form before performing the substitution, resulting in the same canonical form.

A constructor in *general form* is one that is well-formed with respect to Rules (24.1), but disregarding the distinction between atomic and canonical forms. We write $\Delta \vdash c :: \kappa$ to mean that c is a well-formed constructor of kind κ in general form. The difficulty with admitting general form constructors is that they introduce non-trivial equivalences between constructors. For example, one must ensure that $\langle \mathtt{int}, \mathtt{bool} \rangle \cdot \mathtt{l}$ is equivalent to $\mathtt{int}$ wherever the fomer may occur. With this in mind we will introduce a *canonization* procedure that allows us to define equivalence of general form constructors, written $\Delta \vdash c_1 \equiv c_2 :: \kappa$, to mean that c_1 and c_2 have identical canonical forms (up to α-equivalence).

Canonization of general-form constructors is defined by these two judgements:

1. Canonization: $\Delta \vdash c :: \kappa \Downarrow \overline{c}$: transform general-form constructor c of kind κ to canonical form $\overline{c}$.
2. Atomization: $\Delta \vdash c \Uparrow \underline{c} :: \kappa$: transform general-form constructor c to obtain atomic form $\underline{c}$ of kind κ.

These two judgements are defined simultaneously by the following rules. The canonization judgement is used to determine the canonical form of a general-form constructor; the atomization judgement is an auxiliary to the first that transforms constructors into atomic form. The canonization judgement is to be thought of as having mode $(\forall, \forall, \exists)$, whereas the atomization judgement is to be thought of as having mode $(\forall, \exists, \exists)$.

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \mathtt{T}}{\Delta \vdash c :: \mathtt{T} \Downarrow \widehat{\underline{c}}} \tag{24.9a}$$

$$\frac{}{\Delta \vdash c :: 1 \Downarrow \langle\rangle} \tag{24.9b}$$

$$\frac{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1 \Downarrow \overline{c_1} \quad \Delta \vdash c \cdot \mathtt{r} :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \times \kappa_2 \Downarrow \langle \overline{c_1}, \overline{c_2} \rangle} \tag{24.9c}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c\,[u] :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \to \kappa_2 \Downarrow \lambda\, u\,.\, \overline{c_2}} \tag{24.9d}$$

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow u :: \kappa} \tag{24.9e}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{l} \Uparrow \underline{c} \cdot \mathtt{l} :: \kappa_1} \tag{24.9f}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{r} \Uparrow \underline{c} \cdot \mathtt{r} :: \kappa_2} \tag{24.9g}$$

$$\frac{\Delta \vdash c_1 \Uparrow \underline{c_1} :: \kappa_1 \to \kappa_2 \quad \Delta \vdash c_2 :: \kappa_1 \Downarrow \overline{c_2}}{\Delta \vdash c_1\,[c_2] \Uparrow \underline{c_1}\,[\overline{c_2}] :: \kappa_2} \tag{24.9h}$$

The canonization judgement produces canonical forms, and the atomization judgement produces atomic forms.

Lemma 24.3. *1. If $\Delta \vdash c :: \kappa \Downarrow \overline{c}$, then $\Delta \vdash \overline{c} \Downarrow \kappa$.*

2. If $\Delta \vdash c \Uparrow \underline{c} :: \kappa$, then $\Delta \vdash \underline{c} \Uparrow \kappa$.

Proof. By induction on Rules (24.9). □

Theorem 24.4. *If $\Gamma \vdash c :: \kappa$, then there exists $\overline{c}$ such that $\Delta \vdash c :: \kappa \Downarrow \overline{c}$.*

Proof. By induction on the formation rules for general-form constructors, making use of an analysis of the general-form constructors of kind T. □

24.5 Notes

The classical approach is to consider general-form constructors at the outset, for which substitution is readily defined, and then to test equivalence of general-form constructors by reduction to a common irreducible form. Two main lemmas are required for this approach. First, every constructor must reduce in a finite number of steps to an irreducible form; this is called *normalization*. Second, the relation "has a common irreducible form" must be shown to be transitive; this is called *confluence*. Here we have turned the development on its head by considering only canonical constructors in the first place, then defining substitution using Watkins's method [104]. Having defined substitution it is then straightforward to decide equivalence of general-form constructors by canonization of both sides of a candidate equation.

Part IX

Subtyping

Chapter 25

Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> if τ' is a subtype of τ, then a value of type τ' 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*, rather than *containment*.

25.1 Subsumption

A *subtyping judgement* has the form $\tau' <: \tau$, and states that τ' is a subtype of τ. At a minimum we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{25.1a}$$

$$\frac{\tau'' <: \tau' \quad \tau' <: \tau}{\tau'' <: \tau} \tag{25.1b}$$

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*:

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash e : \tau} \tag{25.2}$$

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 $e : \tau'$ and $\tau' <: \tau$.

25.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 25.4 on page 242 we will examine some of these in more detail from the point of view of type safety.

25.2.1 Numeric Types

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, the set of rational numbers may be considered to consist of ordered pairs (m, n), with $n \neq 0$ and $\gcd(m, n) = 1$, representing the ratio m/n. The set $\mathbb{Z}$ of integers may be isomorphically embedded within $\mathbb{Q}$ by identifying $n \in \mathbb{Z}$ with the ratio $n/1$. 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 as floating point numbers.

25.2.2 Product Types

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 dynamics 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.

Consider a context in which a value of type $\tau = \prod_{j \in J} \tau_j$ is required. The statics of finite products (Rules (13.3)) ensures that the only operation we may perform on a value of type τ, other than to bind it to a variable, is to take the jth projection from it for some $j \in J$ to obtain a value of type τ_j.

Now suppose that e is of type τ'. If the projection $e \cdot j$ is to be well-formed, then τ' must be a finite product type $\prod_{i \in I} \tau'_i$ such that $j \in I$. Moreover, for this to be of type τ_j, it is enough to require that $\tau'_j = \tau_j$. Since $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \cdot \tag{25.3}$$

This rule is sufficient for the required subtyping, but not necessary; we will consider a more liberal form of this rule in Section 25.3 on the next page.

The argument for Rule (25.3) is based on a dynamics in which we may evaluate $e \cdot j$ regardless of the actual form of e, provided only that it has a field indexed by $j \in J$. Is this a reasonable assumption?

One common case is that I and J are initial segments of the natural numbers, say $I = [0..m-1]$ and $J = [0..n-1]$, so that the product types may be thought of as m- and n-tuples, respectively. The containment $I \subseteq J$ amounts to requiring that $m \geq n$, which is to say that a tuple type is regarded as a subtype of all of its prefixes. When specialized to this case, Rule (25.3) may be stated in the form

$$\frac{m \geq n}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \cdot \tag{25.4}$$

One way to justify this rule is to consider elements of the subtype to be consecutive sequences of values of type $\tau_0, \ldots, \tau_{m-1}$ from which we may calculate the jth projection for any $0 \leq j < n \leq m$, regardless of whether or not m is strictly bigger than n.

Another common case is when I and J are finite sets of symbols, so that projections are based on the field name, rather than its position. When specialized to this case, Rule (25.3) takes the following form:

$$\frac{m \geq n}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \cdot \tag{25.5}$$

Here we are taking advantage of the implicit identification of labeled tuple types up to reordering of fields, so that the rule states that any field of the supertype must be present in the subtype with the same type.

25.2.3 Sum Types

By an argument dual to the one given for finite product types we may derive a related subtyping rule for finite sum types. If a value of type $\sum_{j \in J} \tau_j$ is

required, the statics of sums (Rules (14.3)) ensures that the only non-trivial operation that we may perform on that value is a J-indexed case analysis. If we provide a value of type $\sum_{i \in I} \tau_i'$ instead, no difficulty will arise so long as $I \subseteq J$ and each τ_i' is equal to τ_i. If the containment is strict, some cases cannot arise, but this does not disrupt safety. This leads to the following subtyping rule for finite sums:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i \mathrel{<:} \sum_{j \in J} \tau_j}\ . \tag{25.6}$$

Note well the reversal of the containment as compared to Rule (25.3).

When I and J are initial segments of the natural numbers, we obtain the following special case of Rule (25.6):

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] \mathrel{<:} [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{25.7}$$

One may also consider a form of width subtyping for unlabeled 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 to ensure type safety.

25.3 Variance

In addition to basic subtyping principles such as those considered in Section 25.2 on page 234, it is also important to consider the effect of subtyping on 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. It is said to be *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

25.3.1 Product Types

Finite product types are *covariant* in each field. For if e is of type $\prod_{i \in I} \tau_i'$, and the projection $e \cdot j$ is expected to be of type τ_j, then it is sufficient to require that $j \in I$ and $\tau_j' \mathrel{<:} \tau_j$. This is summarized by the following rule:

$$\frac{(\forall i \in I)\ \tau_i' \mathrel{<:} \tau_i}{\prod_{i \in I} \tau_i' \mathrel{<:} \prod_{i \in I} \tau_i} \tag{25.8}$$

It is implicit in this rule that the dynamics of projection must not be sensitive to the precise type of any of the fields of a value of finite product type.

When specialized to n-tuples, Rule (25.8) reads as follows:

$$\frac{\tau_1' <: \tau_1 \quad \ldots \quad \tau_n' <: \tau_n}{\langle \tau'_1, \ldots, \tau'_n \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} . \tag{25.9}$$

When specialized to symbolic labels, the covariance principle for finite products may be re-stated as follows:

$$\frac{\tau_1' <: \tau_1 \quad \ldots \quad \tau_n' <: \tau_n}{\langle l_1 : \tau_1', \ldots, l_n : \tau_n' \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} . \tag{25.10}$$

25.3.2 Sum Types

Finite sum types are also covariant, because each branch of a case analysis on a value of the supertype expects a value of the corresponding summand, for which it is sufficient to provide a value of the corresponding subtype summand:

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\sum_{i \in I} \tau_i' <: \sum_{i \in I} \tau_i} \tag{25.11}$$

When specialized to symbolic labels as index sets, we obtain the following formulation of the covariance principle for sum types:

$$\frac{\tau_1' <: \tau_1 \quad \ldots \quad \tau_n' <: \tau_n}{[l_1 : \tau_1', \ldots, l_n : \tau_n'] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} . \tag{25.12}$$

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 τ_i' instead.

25.3.3 Function Types

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 $e : \tau_1 \to \tau_2'$. This means that if $e_1 : \tau_1$, then $e(e_1) : \tau_2'$. If $\tau_2' <: \tau_2$, then $e(e_1) : \tau_2$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau_2' <: \tau_2}{\tau_1 \to \tau_2' <: \tau_1 \to \tau_2} \tag{25.13}$$

Every function that delivers a value of type τ_2' also delivers a value of type τ_2, provided that $\tau_2' <: \tau_2$. 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 $e : \tau_1 \to \tau_2$. This means that e may be applied to any value of type τ_1 to obtain a value of type τ_2. Hence, by the subsumption principle, it may be applied to any value of a subtype, τ_1', of τ_1, and it will still deliver a value of type τ_2. Consequently, we may just as well think of e as having type $\tau_1' \to \tau_2$.

$$\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2} \tag{25.14}$$

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!

Combining these rules we obtain the following general principle of contra- and co-variance for function types:

$$\frac{\tau_1' <: \tau_1 \quad \tau_2' <: \tau_2}{\tau_1 \to \tau_2' <: \tau_1' \to \tau_2} \tag{25.15}$$

Beware of the reversal of the ordering in the domain!

25.3.4 Quantified Types

The extension of subtyping to quantified types requires a judgement of the form $\Delta \vdash \tau' <: \tau$, where $\Delta \vdash \tau'$ type and $\Delta \vdash \tau$ type. The variance principles for the quantifiers may then be stated so that both are covariant in the quantified type:

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \forall(t.\tau') <: \forall(t.\tau)} \tag{25.16a}$$

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \exists(t.\tau') <: \exists(t.\tau)} \tag{25.16b}$$

The judgement $\Delta \vdash \tau' <: \tau$ states that τ' is a subtype of τ uniformly in the type variables declared in Δ. Consequently, we may derive the principle of substitution:

Lemma 25.1. *If Δ, t type $\vdash \tau' <: \tau$, and $\Delta \vdash \rho$ type, then $\Delta \vdash [\rho/t]\tau' <: [\rho/t]\tau$.*

Proof. By induction on the subtyping derivation. □

It is easy to check that the above variance principles for the quantifiers are consistent with the principle of subsumption. For example, a package of the subtype $\exists(t.\tau')$ consists of a representation type, ρ, and an implementation, e, of type $[\rho/t]\tau'$. But if t type $\vdash \tau' <: \tau$, we have by substitution that $[\rho/t]\tau' <: [\rho/t]\tau$, and hence e is also an implementation of type $[\rho/t]\tau$. This is sufficient to ensure that the package is also of the supertype.

It is natural to extend subtyping to the quantifiers by allowing quantification over all subtypes of a specified type. This is called *bounded quantification*. To express bounded quantification we consider additional hypotheses of the form $t <: \rho$, expressing that t is a variable that may only be instantiated to subtypes of ρ.

$$\frac{}{\Delta, t \text{ type}, t <: \tau \vdash t <: \tau} \tag{25.17a}$$

$$\frac{\Delta \vdash \tau :: \mathtt{T}}{\Delta \vdash \tau <: \tau} \tag{25.17b}$$

$$\frac{\Delta \vdash \tau'' <: \tau' \quad \Delta \vdash \tau' <: \tau}{\Delta \vdash \tau'' <: \tau} \tag{25.17c}$$

$$\frac{\Delta \vdash \rho' <: \rho \quad \Delta, t \text{ type}, t <: \rho' \vdash \tau' <: \tau}{\Delta \vdash \forall\, t <: \rho.\, \tau' <: \forall\, t <: \rho'.\, \tau'} \tag{25.17d}$$

$$\frac{\Delta \vdash \rho' <: \rho \quad \Delta, t \text{ type}, t <: \rho' \vdash \tau' <: \tau}{\Delta \vdash \exists\, t <: \rho'.\, \tau' <: \exists\, t <: \rho.\, \tau} \tag{25.17e}$$

Rule (25.17d) states that the universal quantifier is contravariant in its bound, whereas Rule (25.17e) states that the existential quantifier is covariant in its bound.

25.3.5 Recursive Types

The variance principle for recursive types is rather subtle, and has been the source of errors in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mu t.\, [\mathtt{empty} : \mathtt{unit}, \mathtt{binode} : \langle \mathtt{data} : \mathtt{nat}, \mathtt{lft} : t, \mathtt{rht} : t \rangle],$$

and the type of "bare" binary trees, without labels on the nodes,

$$\mu t.\, [\mathtt{empty} : \mathtt{unit}, \mathtt{binode} : \langle \mathtt{lft} : t, \mathtt{rht} : t \rangle].$$

Is either a subtype of the other? Intuitively, one might expect the type of labeled 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:

$$\mu t.\,[\mathtt{empty}:\mathtt{unit},\mathtt{binode}:\langle \mathtt{lft}:t,\mathtt{rht}:t\rangle,\mathtt{trinode}:\langle \mathtt{lft}:t,\mathtt{mid}:t,\mathtt{rht}:t\rangle].$$

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:

$$\frac{t\ \mathsf{type} \vdash \tau' <: \tau}{\mu t.\,\tau' <: \mu t.\,\tau}\ \textbf{??} \tag{25.18}$$

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 $\tau' <: \tau$.

Rule (25.18) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. Deriving this requires checking that the subtyping relationship

$$\langle \mathtt{data}:\mathtt{nat},\mathtt{lft}:t,\mathtt{rht}:t\rangle <: \langle \mathtt{lft}:t,\mathtt{rht}:t\rangle,$$

holds generically in t, which is evidently the case.

Unfortunately, Rule (25.18) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\tau' = \mu t.\,\langle \mathtt{a}:t\to\mathtt{nat},\mathtt{b}:t\to\mathtt{int}\rangle$$

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 (25.18) we may derive $\tau' <: \tau$, which we will show to be incorrect. Let $e : \tau'$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} = \lambda\,(x:\tau'.4),\mathtt{b} = \lambda\,(x:\tau'.q((\mathtt{unfold}(x)\cdot\mathtt{a})(x)))\rangle),$$

where q : $\texttt{nat} \to \texttt{nat}$ is the discrete square root function. Since $\tau' <: \tau$, it follows that $e : \tau$ as well, and hence

$$\texttt{unfold}(e) : \langle \texttt{a} : \tau \to \texttt{int}, \texttt{b} : \tau \to \texttt{int} \rangle.$$

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 (25.18) 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. To do so we make use of bounded quantification to state the rule of subsumption for recursive types:

$$\frac{\Delta, t\ \mathsf{type}, t'\ \mathsf{type}, t' <: t \vdash \tau' <: \tau \quad \Delta, t'\ \mathsf{type} \vdash \tau'\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mu t'.\tau' <: \mu t.\tau}. \tag{25.19}$$

That is, to check whether $\mu t'.\tau' <: \mu t.\tau$, we assume that $t' <: t$, since t' and t stand for the respective recursive types, and check that $\tau' <: \tau$ under this assumption. It is instructive to check that the unsound subtyping is *not* derivable using this rule: the subtyping assumption is at odds with the contravariance of the function type in its domain.

25.4 Safety

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.

As a representative case we will sketch the proof of safety for a language with subtyping for product types. The subtyping relation is defined by Rules (25.3) and (25.8). We assume that the statics includes subsumption, Rule (25.2).

Lemma 25.2 (Structurality)**.**

1. *The tuple subtyping relation is reflexive and transitive.*
2. *The typing judgement* $\Gamma \vdash e : \tau$ *is closed under weakening and substitution.*

Proof.

1. Reflexivity is proved by induction on the structure of types. Transitivity is proved by induction on the derivations of the judgements $\tau'' <: \tau'$ and $\tau' <: \tau$ to obtain a derivation of $\tau'' <: \tau$.
2. By induction on Rules (13.3), augmented by Rule (25.2).

□

Lemma 25.3 (Inversion)**.**

1. *If* $e \cdot j : \tau$*, then* $e : \prod_{i \in I} \tau_i$*,* $j \in I$*, and* $\tau_j <: \tau$*.*
2. *If* $\langle e_i \rangle_{i \in I} : \tau$*, then* $\prod_{i \in I} \tau'_i <: \tau$ *where* $e_i : \tau'_i$ *for each* $i \in I$*.*
3. *If* $\tau' <: \prod_{j \in J} \tau_j$*, then* $\tau' = \prod_{i \in I} \tau'_i$ *for some* I *and some types* τ'_i *for* $i \in I$*.*
4. *If* $\prod_{i \in I} \tau'_i <: \prod_{j \in J} \tau_j$*, then* $J \subseteq I$ *and* $\tau'_j <: \tau_j$ *for each* $j \in J$*.*

Proof. By induction on the subtyping and typing rules, paying special attention to Rule (25.2). □

Theorem 25.4 (Preservation)**.** *If* $e : \tau$ *and* $e \mapsto e'$*, then* $e' : \tau$*.*

Proof. By induction on Rules (13.4). For example, consider Rule (13.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$, $e' = e_k$. By Lemma 25.3 we have that $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, $k \in J$, and $\tau_k <: \tau$. By another application of Lemma 25.3 for each $i \in I$ there exists τ'_i such that $e_i : \tau'_i$ and $\prod_{i \in I} \tau'_i <: \prod_{j \in J} \tau_j$. By Lemma 25.3 again, we have $J \subseteq I$ and $\tau'_j <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaing cases are similar. □

Lemma 25.5 (Canonical Forms)**.** *If e val and $e : \prod_{j \in J} \tau_j$, then e is of the form $\langle e_i \rangle_{i \in I}$, where $J \subseteq I$, and $e_j : \tau_j$ for each $j \in J$.*

Proof. By induction on Rules (13.3) augmented by Rule (25.2). □

Theorem 25.6 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By induction on Rules (13.3) augmented by Rule (25.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. Rule (13.4d) follows from Lemma 25.5. □

25.5 Notes

Subtyping is perhaps the most widely misunderstood concept in programming languages. Subtyping, a static concept, is often confused with subclassing, a dynamic concept (see Chapters 14 and 27 for more on classes). Subtyping is principally a convenience, akin to type inference, that makes some programs simpler to write. But the subsumption rule cuts both ways. Inasmuch as it allows the implicit passage from τ' to τ whenever τ' is a subtype of τ, it also weakens the meaning of a type assertion $e : \tau$ to mean that e has some type contained in the type τ. This precludes expressing the requirement that e has *exactly* the type τ, or that two expresses jointly have the *same* type. And it is precisely this weakness that creates so many of the difficulties with subtyping.

Much has been written about subtyping. Among the earliest uses of subtyping in a full-scale language is in Standard ML [67] whose module system is based on two concepts of subtyping, called *enrichment* and *realization*. The former corresponds to the notion of product subtyping considered in this chapter, and the latter correponds to the notion of type definitions considered in Chapter 26. The first systematic studies of subtyping include those by Mitchell [68], Reynolds [88], and Cardelli [18]. The meta-theory of bounded quantification and subtyping for recursive types is suprisingly complex. See Pierce's text [77] for a thorough treatment of these topics, and their application to object-oriented programming.

Chapter 26

Singleton 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 $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ binds t to τ within e, so that one may write expressions such as

$$\texttt{def}\ t\ \texttt{is}\ \texttt{nat} \times \texttt{nat}\ \texttt{in}\ \lambda\,(x{:}t.\, \texttt{s}(x \cdot \texttt{l})).$$

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 a type definition abbreviates the polymorphic instantiation $\Lambda(t.e)[\tau]$, which binds t to τ within e. This does, indeed, capture the dynamics of type abbreviation, but it fails to validate the intended statics. The difficulty is that, according to this interpretation of type definitions, 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 $\texttt{s}(x \cdot \texttt{l})$ fails to type check, unless the binding of t were exposed.

The interpetation of type definition 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 $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ would be taken as a primitive

form of expression whose statics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \mathtt{def}\ t\ \mathtt{is}\ \tau\ \mathtt{in}\ e : \tau'} \tag{26.1}$$

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.

The importance of singleton kinds lies in their role in program modules, which are the subject of Chapters 47 and 48.

26.1 Overview

The central organizing principle of type theory is *compositionality*. To ensure that a program may be decomposed into separable parts, we ensure that the composition of a program from constituent parts is mediated by the types of those parts. Put in other terms, the only thing that one portion of a program "knows" about another is its type. For example, the formation rule for addition of natural numbers depends only on the type of its arguments (both must have type `nat`), and not on their specific form or value. But in the case of a type abbreviation of the form $\mathtt{def}\ t\ \mathtt{is}\ \tau\ \mathtt{in}\ e$, the principle of compositionality dictates that the only thing that e "knows" about the type variable t is its kind, namely `T`, and not its binding, namely τ. This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of t to capture its identity. This may be achieved through the notion of a *singleton kind*. Informally, the kind $\mathtt{S}(\tau)$ is the kind of types that are definitionally equivalent to τ. That is, up to definitional equality, this kind has only one inhabitant, namely τ. Consequently, if $u :: \mathtt{S}(\tau)$ is a variable of singleton kind, then within its scope, the variable u is synonymous

with τ. Thus we may represent $\mathtt{def}\ t\ \mathtt{is}\ \tau\ \mathtt{in}\ e$ by $\Lambda(t::\mathtt{S}(\tau).e)[\tau]$, 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 $\mathtt{T}$, and hence is a type. Otherwise, a variable, u, of 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 $\mathtt{S}(\tau) :<: \mathtt{T}$, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind $\mathtt{T}$ within the singleton kind $\mathtt{S}(\tau)$. 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 $\mathtt{S}(\tau)$ represents a *family of kinds* indexed by constructors of kind $\mathtt{T}$. This, in turn, implies that we must generalize the product and function kinds to *dependent products* and *dependent functions*. 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. 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.

Third, it is useful to consider singletons not just of kind $\mathtt{T}$, but also of higher kinds. To support this we introduce *higher singletons*, written $\mathtt{S}(c::\kappa)$, 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.

26.2 Singletons

The syntax of singleton kinds is given by the following grammar:

$$\mathsf{Kind}\quad \kappa \quad ::= \quad \mathtt{S}(c) \quad \mathtt{S}(c) \quad \text{singleton}$$

Informally, the singleton kind, $\mathtt{S}(c)$, classifies constructors that are equivalent (in a sense to be made precise shortly) to c. For the time being we tacitly include the constructors and kinds given in Chapter 24 (but see Section 26.3 on page 249).

The following judgement forms comprise the statics of singletons:

$\Delta \vdash \kappa \ \mathsf{kind}$	kind formation
$\Delta \vdash \kappa_1 \equiv \kappa_2$	kind equivalence
$\Delta \vdash c :: \kappa$	constructor formation
$\Delta \vdash c_1 \equiv c_2 :: \kappa$	constructor equivalence
$\Delta \vdash \kappa_1 :<: \kappa_2$	sub-kinding

These judgements are defined simultaneously by a collection of rules including the following:

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) \ \mathsf{kind}} \tag{26.2a}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash c :: \mathtt{S}(c)} \tag{26.2b}$$

$$\frac{\Delta \vdash c :: \mathtt{S}(d)}{\Delta \vdash c \equiv d :: \mathtt{Type}} \tag{26.2c}$$

$$\frac{\Delta \vdash c :: \kappa_1 \quad \Delta \vdash \kappa_1 :<: \kappa_2}{\Delta \vdash c :: \kappa_2} \tag{26.2d}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) :<: \mathtt{Type}} \tag{26.2e}$$

Omitted for brevity are rules stating that sub-kinding is a reflexive and transitive; that constructor and kind equivalence are reflexive, symmetric, transitive, and preserved by kind and constructor formation.

The significance of these rules is best appreciated by considering the behavior of variables of singleton kind. Suppose that $\Delta \vdash u :: \mathtt{S}(c)$ is such a variable. Then by Rule (26.2c) we may deduce that $\Delta \vdash u \equiv c :: \mathtt{T}$. In essence the declaration of u with a singleton kind serves to define u to be the constructor (of kind $\mathtt{T}$) specified by its kind.

Taking this a step further, the existential type $\exists\, u :: \mathtt{S}(c)\,.\,\tau$ is the type of packages whose representation type is (equivalent to) c—it is an abstract type whose identity is revealed by assigning it a singleton kind. By the general principles of equivalence we have that the type $\exists\, u :: \mathtt{S}(c)\,.\,\tau$ is equivalent to the type $\exists\, \_ :: \mathtt{S}(c)\,.\,[c/u]\tau$, wherein we have propagated the equivalence of u and c into the type τ. On the other hand we may also "forget" the definition of u, since the subtyping

$$\exists\, u :: \mathtt{S}(c)\,.\,\tau <: \exists\, u :: \mathtt{T}\,.\,\tau$$

is derivable using the following variance rule for existentials over a kind:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_2 \quad \Delta, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Delta \vdash \exists\, u :: \kappa_1\,.\,\tau_1 <: \exists\, u :: \kappa_2\,.\,\tau_2} \tag{26.3}$$

Similarly, we may derive the subtyping

$$\forall\, u :: \mathtt{T}\,.\,\tau <: \forall\, u :: \mathtt{S}(c)\,.\,\tau$$

from the following variance rule for universals over a kind:

$$\frac{\Delta \vdash \kappa_2 :<: \kappa_1 \quad \Delta, u :: \kappa_2 \vdash \tau_1 <: \tau_2}{\Delta \vdash \forall\, u :: \kappa_1\,.\,\tau_1 <: \forall\, u :: \kappa_2\,.\,\tau_2} \tag{26.4}$$

Informally, the displayed subtyping states that a polymorphic function that may be applied to *any* type is one that may only be applied to a particular type, c.

These examples show that singleton kinds express the idea of a scoped definition of a type variable in a way that is not tied to an *ad hoc* definition mechanism, but rather arises naturally from general principles of binding and scope. We will see in Chapters 47 and 48 more sophisticated uses of singletons to manage the interaction among program modules.

26.3 Dependent Kinds

While it is perfectly possible to add singleton kinds to the framework of higher kinds introduced in Chapter 24, to do so would be to shortchange the expressiveness of the language. Using higher kinds we can express the kind of constructors that, when applied to a type, yield a specific type, say `int`, as result, namely $\mathtt{T} \to \mathtt{S(int)}$. But we cannot express the kind of constructors that, when applied to a type, yield *that very type* as result, for there is no way for the result kind to refer to the argument of the function. Similarly, using product kinds we can express the kind of pairs whose first component is `int` and whose second component is an arbitrary type, namely $\mathtt{S(int)} \times \mathtt{T}$. But we cannot express the kind of pairs whose second component is equivalent to its first component, for there is no way for the kind of the second component to make reference to the first component itself.

To express such concepts requires a generalization of product and function kinds in which the kind of the second component of a pair may mention the first component of that pair, or the kind of the result of a function may mention the argument to which it is applied. Such kinds are called *dependent kinds* because they involve kinds that mention, or depend upon, constructors (of kind `T`). The syntax of dependent kinds is given by the

following grammar:

Kind	κ	$::=$	$\mathtt{S}(c)$	$\mathtt{S}(c)$	singleton
			$\Sigma(\kappa_1; u.\kappa_2)$	$\Sigma\, u::\kappa_1.\kappa_2$	dependent product
			$\Pi(\kappa_1; u.\kappa_2)$	$\Pi\, u::\kappa_1.\kappa_2$	dependent function
Con	c	$::=$	u	u	variable
			$\mathtt{pair}(c_1; c_2)$	$\langle c_1, c_2\rangle$	pair
			$\mathtt{proj[l]}(c)$	$c \cdot \mathtt{l}$	first projection
			$\mathtt{proj[r]}(c)$	$c \cdot \mathtt{r}$	second projection
			$\mathtt{lam}[\kappa](u.c)$	$\lambda\,(u::\kappa.\, c)$	abstraction
			$\mathtt{app}(c_1; c_2)$	$c_1[c_2]$	application

As a notational convenience when there is no dependency in a kind we write $\kappa_1 \times \kappa_2$ for $\Sigma\,\_::\kappa_1.\kappa_2$, and $\kappa_1 \to \kappa_2$ for $\Pi\,\_::\kappa_1.\kappa_2$, where the "blank" stands for an irrelevant variable.

The syntax of dependent kinds differs from that given in Chapter 24 for higher kinds in that we do not draw a distinction between atomic and canonical constructors, and consider that substitution is defined conventionally, rather than hereditarily. This simplifies the syntax, but at the expense of leaving open the decidability of constructor equivalence. The method of hereditary substitution considered in Chapter 24 may be extended to singleton kinds, but we will not develop this extension here. Instead we will simply assert without proof that equivalence of well-formed constructors is decidable.

The dependent product kind $\Sigma\, u::\kappa_1.\kappa_2$ classifies pairs $\langle c_1, c_2\rangle$ of constructors in which c_1 has kind κ_1 and c_2 has kind $[c_1/u]\kappa_2$. For example, the kind $\Sigma\, u::\mathtt{T}.\mathtt{S}(u)$ classifies pairs $\langle c, c\rangle$, where c is a constructor of kind $\mathtt{T}$. More generally, this kind classifies pairs of the form $\langle c_1, c_2\rangle$ where c_1 and c_2 are equivalent, but not necessarily identical, constructors. The dependent function kind $\Pi\, u::\kappa_1.\kappa_2$ classifies constructors c that, when applied to a constructor c_1 of kind κ_1 yield a constructor of kind $[c_1/u]\kappa_2$. For example, the kind $\Pi\, u::\mathtt{T}.\mathtt{S}(u)$ classifies constructors that, when applied to a constructor, c, yield a constructor equivalent to c; a constructor of this kind is essentially the identity function. We may, of course, combine these to form kinds such as

$$\Pi\, u::\mathtt{T} \times \mathtt{T}.\mathtt{S}(u \cdot \mathtt{r}) \times \mathtt{S}(u \cdot \mathtt{l}),$$

which classifies functions that swaps the components of a pair of types. (Such examples may lead one to surmise that the behavior of any constructor may be pinned down precisely using dependent kinds. We shall see in Section 26.4 on page 253 that this is indeed the case.)

The formation, introduction, and elimination rules for the product kind are as follows:

$$\frac{\Delta \vdash \kappa_1 \text{ kind} \quad \Delta, u :: \kappa_2 \vdash \kappa_2 \text{ kind}}{\Delta \vdash \Sigma\, u :: \kappa_1 \,.\, \kappa_2 \text{ kind}} \tag{26.5a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: [c_1/u]\kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle :: \Sigma\, u :: \kappa_1 \,.\, \kappa_2} \tag{26.5b}$$

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1 \,.\, \kappa_2}{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1} \tag{26.5c}$$

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1 \,.\, \kappa_2}{\Delta \vdash c \cdot \mathtt{r} :: [c_1/u]\kappa_2} \tag{26.5d}$$

In Rule (26.5a), observe that the variable, u, may occur in the kind κ_2 by appearing in a singleton kind. Correspondingly, Rules (26.5b), (26.5c), and (26.5d) substitute a constructor for this variable.

Constructor equivalence is defined to be an equivalence relation that is compatible with all forms of constructors and kinds, so that a constructor may always be replaced by an equivalent constructor and the result will be equivalent. The following equivalence axioms govern the constructors associated with the dependent product kind:

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{l} \equiv c_1 :: \kappa_1} \tag{26.6a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{r} \equiv c_2 :: \kappa_2} \tag{26.6b}$$

The subkinding rule for the dependent product kind specifies that it is covariant in both positions:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 :<: \kappa_2'}{\Delta \vdash \Sigma\, u :: \kappa_1 \,.\, \kappa_2 :<: \Sigma\, u :: \kappa_1' \,.\, \kappa_2'} \tag{26.7}$$

The congruence rule for equivalence of dependent product kinds is formally similar:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \Sigma\, u :: \kappa_1 \,.\, \kappa_2 \equiv \Sigma\, u :: \kappa_1' \,.\, \kappa_2'} \tag{26.8}$$

Notable consequences of these rules include the subkindings

$$\Sigma\, u :: \mathtt{S(int)} \,.\, \mathtt{S}(u) :<: \Sigma\, u :: \mathtt{T} \,.\, \mathtt{S}(u)$$

and

$$\Sigma\, u :: \mathtt{T} \,.\, \mathtt{S}(u) :<: \mathtt{T} \times \mathtt{T},$$

and the equivalence

$$\Sigma\, u{::}\mathtt{S(int)}.\mathtt{S}(u) \equiv \mathtt{S(int)} \times \mathtt{S(int)}.$$

Subkinding is used to "forget" information about the identity of the components of a pair, and equivalence is used to "propagate" such information within a kind.

The formation, introduction, and elimination rules for dependent function kinds are as follows:

$$\frac{\Delta \vdash \kappa_1 \text{ kind} \quad \Delta, u :: \kappa_2 \vdash \kappa_2 \text{ kind}}{\Delta \vdash \Pi\, u{::}\kappa_1 . \kappa_2 \text{ kind}} \tag{26.9a}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2}{\Delta \vdash \lambda\ (u{::}\kappa_1.\, c) :: \Pi\, u{::}\kappa_1 . \kappa_2} \tag{26.9b}$$

$$\frac{\Delta \vdash c :: \Pi\, u{::}\kappa_1 . \kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash c\,[c_1] :: [c_1/u]\kappa_2} \tag{26.9c}$$

Rule (26.9b) specifies that the result kind of a λ-abstraction depends uniformly on the argument, u. Correspondingly, Rule (26.9c) specifies that the kind of an application is obtained by substitution of the argument into the result kind of the function itself.

The following rule of equivalence governs the constructors associated with the dependent product kind:

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash (\lambda\ (u{::}\kappa_1.\, c))\,[c_1] \equiv [c_1/u]c :: \kappa_2} \tag{26.10}$$

The subtyping rule for the dependent function kind specifies that it is contravariant in its domain and covariant in its range:

$$\frac{\Delta \vdash \kappa_1' :\!<\!: \kappa_1 \quad \Delta, u :: \kappa_1' \vdash \kappa_2 :\!<\!: \kappa_2'}{\Delta \vdash \Pi\, u{::}\kappa_1 . \kappa_2 :\!<\!: \Pi\, u{::}\kappa_1' . \kappa_2'} \tag{26.11}$$

The equivalence rule is similar, except that the symmetry of equivalence obviates a choice of variance:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \Pi\, u{::}\kappa_1 . \kappa_2 \equiv \Pi\, u{::}\kappa_1' . \kappa_2'} \tag{26.12}$$

Rule (26.11) gives rise to the subkinding

$$\Pi\, u{::}\mathtt{T.S(int)} :\!<\!: \Pi\, u{::}\mathtt{S(int).T},$$

which illustrates the co- and contra-variance of the dependent function kind. In words this means that a constructor function that takes any type and delivers the type `int` is also a constructor that takes the type `int` and delivers a type. Rule (26.12) gives rise to the equivalence

$$\Pi\, u::\mathtt{S(int)}.\mathtt{S}(u) \equiv \mathtt{S(int)} \to \mathtt{S(int)},$$

which propagates information about the argument into the range kind. Combining these two rules we may derive the subkinding

$$\Pi\, u::\mathtt{T}.\mathtt{S}(u) :<: \mathtt{S(int)} \to \mathtt{S(int)}.$$

Intuitively, a constructor function that yields its argument is, in particular, a constructor function that may only be applied to `int`, and yields `int`. Formally, by contravariance we have the subkinding

$$\Pi\, u::\mathtt{T}.\mathtt{S}(u) :<: \Pi\, u::\mathtt{S(int)}.\mathtt{S}(u),$$

and by sharing propagation we may derive the indicated superkind.

26.4 Higher Singletons

Although singletons are restricted to constructors of kind `T`, we may use dependent product and function kinds to define singletons of every kind. Specifically, we wish to define the kind $\mathtt{S}(c::\kappa)$, where c is of kind κ, that classifies constructors equivalent to c. When $\kappa = \mathtt{T}$ this is, of course, just $\mathtt{S}(c)$; the problem is to define singletons for the higher kinds $\Sigma\, u::\kappa_1.\kappa_2$ and $\Pi\, u::\kappa_1.\kappa_2$.

To see what is involved, suppose that $c :: \kappa_1 \times \kappa_2$. The singleton kind $\mathtt{S}(c::\kappa_1 \times \kappa_2)$ should classify constructors equivalent to c. If we assume, inductively, that singletons have been defined for κ_1 and κ_2, then we need only observe that c is equivalent to $\langle c\cdot\mathtt{l}, c\cdot\mathtt{r}\rangle$. For then the singleton $\mathtt{S}(c::\kappa_1 \times \kappa_2)$ may be defined to be $\mathtt{S}(c\cdot\mathtt{l}::\kappa_1) \times \mathtt{S}(c\cdot\mathtt{r}::\kappa_2)$. Similarly, suppose that $c :: \kappa_1 \to \kappa_2$. Using the equivalence of c and $\lambda\,(u::\kappa_1 \to \kappa_2.\, c[u])$, we may define $\mathtt{S}(c::\kappa_1 \to \kappa_2)$ to be $\Pi\, u::\kappa_1.\mathtt{S}(c[u]::\kappa_2)$.

In general the kind $\mathtt{S}(c::\kappa)$ is defined by induction on the structure of κ by the following kind equivalences:

$$\frac{\Delta \vdash c :: \mathtt{S}(c')}{\Delta \vdash \mathtt{S}(c::\mathtt{S}(c')) \equiv \mathtt{S}(c)} \tag{26.13a}$$

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1.\kappa_2}{\Delta \vdash \mathtt{S}(c::\Sigma\, u::\kappa_1.\kappa_2) \equiv \Sigma\, u::\mathtt{S}(c\cdot\mathtt{l}::\kappa_1).\mathtt{S}(c\cdot\mathtt{r}::\kappa_2)} \tag{26.13b}$$

$$\frac{\Delta \vdash c :: \Pi\, u::\kappa_1\,.\,\kappa_2}{\Delta \vdash \mathtt{S}(c::\Pi\, u::\kappa_1\,.\,\kappa_2) \equiv \Pi\, u::\kappa_1\,.\,\mathtt{S}(c\,[u]::\kappa_2)} \tag{26.13c}$$

The sensibility of these equations relies on Rule (26.2c) together with the following principles of constructor equivalence, called *extensionality principles*:

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1\,.\,\kappa_2}{\Delta \vdash c \equiv \langle c\cdot \mathtt{l}, c\cdot \mathtt{r}\rangle :: \Sigma\, u::\kappa_1\,.\,\kappa_2} \tag{26.14a}$$

$$\frac{\Delta \vdash c :: \Pi\, u::\kappa_1\,.\,\kappa_2}{\Delta \vdash c \equiv \lambda\,(u::\kappa_1.\,c\,[u]) :: \Pi\, u::\kappa_1\,.\,\kappa_2} \tag{26.14b}$$

Rule (26.2c) states that the only constructors of kind $\mathtt{S}(c')$ are those equivalent to c', and Rules (26.14a) and (26.14b) state that the only members of the dependent product and function types are, respectively, pairs and λ-abstractions of the appropriate kinds.

The higher singleton is the most precise kind for any well-formed constructor:

Lemma 26.1. *If* $\Delta \vdash c :: \kappa$, *then* $\Delta \vdash \mathtt{S}\,(c::\kappa) :<: \kappa$ *and* $\Delta \vdash c :: \mathtt{S}\,(c::\kappa)$.

The proof of this lemma is surprisingly intricate; the reader is referred to the references below for details.

26.5 Notes

Singleton kinds were introduced by Stone and Harper [99] to isolate the central concept of type sharing in the ML module system [67, 42, 55]. The cited work by Stone and Harper provides a full development of the metatheory of singleton kinds, including decidability of constructor equivalence. Crary [26] provides another, more elementary, proof of decidability based on an extension of hereditary substitution to singleton kinds.

Part X

Classes and Methods

Chapter 27

Dynamic Dispatch

It frequently arises that the values of a type are partitioned into a variety of *classes*, each classifying data with distinct internal structure. A good example is provided by the type of points in the plane, which may be classified according to whether they are represented in cartesian or polar form. Both are represented by a pair of real numbers, but in the cartesian case these are the x and y coordinates of the point, whereas in the polar case these are its distance, ρ, from the origin and its angle, θ, with the polar axis. A classified value is said to be an *instance* of, or an *object* of its class. The class determines the type of the classified data, which is called the *instance type* of the class. The classified data itself is called the *instance data* of the object.

Functions that act on classified values are called *methods*. The behavior of a method is determined by the class of its argument. The method is said to *dispatch* on the class of the argument. Because it happens at run-time, this is called, rather grandly, *dynamic dispatch*. For example, the distance of a point from the origin is calculated differently according to whether the point is represented in cartesian or polar form. In the former case the required distance is $\sqrt{x^2 + y^2}$, whereas in the latter it is simply ρ itself. Similarly, the quadrant of a cartesian point may be determined by examining the sign of its x and y coordinates, and the quadrant of a polar point may be calculated by taking the integral part of the angle θ divided by $\pi/2$.

27.1 The Dispatch Matrix

Since each method acts by dispatch on the class of its argument, we may envision the entire system of classes and methods as a matrix, e_{dm}, called the *dispatch matrix*, whose rows are classes, whose columns are methods,

and whose (c, d)-entry is the code for method d acting on an argument of class c, expressed as a function of the instance data of the object. Thus, the dispatch matrix has a type of the form

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d),$$

where C is the set of class names, D is the set of method names, τ^c is the instance type associated with class c and ρ_d is the result type of method d. The instance type is the same for all methods acting on a given class, and the result type is the same for all classes acted on by a given method.

As an illustrative example, let us consider the type of points in the plane classified into two classes, `cart` and `pol`, corresponding to the cartesian and polar representations. The instance data for a cartesian point has the type

$$\tau^{\mathtt{cart}} = \langle \mathtt{x} : \mathtt{real}, \mathtt{y} : \mathtt{real} \rangle,$$

and the instance data for a polar point has the type

$$\tau^{\mathtt{pol}} = \langle \mathtt{r} : \mathtt{real}, \mathtt{th} : \mathtt{real} \rangle.$$

Consider two methods acting on points, `dist` and `quad`, which compute, respectively, the squared distance of a point from the origin and the quadrant of a point. The distance method is given by the tuple $e_{\mathtt{dist}} = \langle \mathtt{cart} = e^{\mathtt{cart}}_{\mathtt{dist}}, \mathtt{pol} = e^{\mathtt{pol}}_{\mathtt{dist}} \rangle$ of type

$$\langle \mathtt{cart} : \tau^{\mathtt{cart}} \to \rho_{\mathtt{dist}}, \mathtt{pol} : \tau^{\mathtt{pol}} \to \rho_{\mathtt{dist}} \rangle,$$

where $\rho_{\mathtt{dist}} = \mathtt{real}$ is the result type,

$$e^{\mathtt{cart}}_{\mathtt{dist}} = \lambda\ (u : \tau^{\mathtt{cart}}.\ (u \cdot \mathtt{x})^2 + (u \cdot \mathtt{y})^2)$$

is the distance computation for a cartesian point, and

$$e^{\mathtt{pol}}_{\mathtt{dist}} = \lambda\ (v : \tau^{\mathtt{pol}}.\ (v \cdot \mathtt{r})^2)$$

is the distance computation for a polar point. Similarly, the quadrant method is given by the tuple $e_{\mathtt{quad}} = \langle \mathtt{cart} = e^{\mathtt{cart}}_{\mathtt{quad}}, \mathtt{pol} = e^{\mathtt{pol}}_{\mathtt{quad}} \rangle$ of type

$$\langle \mathtt{cart} : \tau^{\mathtt{cart}} \to \rho_{\mathtt{quad}}, \mathtt{pol} : \tau^{\mathtt{pol}} \to \rho_{\mathtt{quad}} \rangle,$$

where $\rho_{\mathtt{quad}} = \mathtt{[I, II, III, IV]}$ is the type of quadrants, and $e^{\mathtt{cart}}_{\mathtt{quad}}$ and $e^{\mathtt{pol}}_{\mathtt{quad}}$ are expressions that compute the quadrant of a point in rectangular and polar forms, respectively.

Now let $C = \{\,\texttt{cart}, \texttt{pol}\,\}$ and let $D = \{\,\texttt{dist}, \texttt{quad}\,\}$, and define the dispatch matrix, e_{dm}, to be the value of type

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d)$$

such that, for each class c and method d,

$$e_{\mathsf{dm}} \cdot c \cdot d \mapsto^* e_d^c.$$

That is, the entry in the dispatch matrix, e_{dm}, for class c and method d is defined to be the implementation of that method acting on an instance of that class.

There are two main ways to organize a system of classes and methods, according to whether we wish to place emphasis on the classes, thought of as a collection of methods acting on its instances, or on the methods, thought of as a collection of classes on which the methods act. These are, respectively, the *class-based* and the *method-based* organizations. Languages that place special emphasis on classes and methods, called *object-oriented languages*,[1] usually adopt one or the other of these organizations as a central design principle, but there is no fundamental reason why one is preferable to the other. Ideally one should be able to program in either style according to the needs of a particular problem.

Both organizations provide a definition of *object types* in a language with sums and products. The definition consists of these components:

- The type of objects, which are instances of the classes on which the methods act.
- The operation `new[`c`](`e`)` that creates an object of the class c with instance data given by the expression e.
- The operation $e \Leftarrow d$ that invokes method d on the instance given by the expression e.

27.2 Method-Based Organization

The method-based organization starts with the *transpose* of the dispatch matrix, which has the type

$$\prod_{d \in D} \prod_{c \in C} (\tau^c \to \rho_d).$$

[1] The term "object-oriented" itself speaks to the vagueness of the concept. It has little, if any, definite meaning, and is mainly used to express approval.

By observing that each row of the transposed dispatch matrix determines a method, we obtain the *method vector*, e_{mv}, of type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} \left(\sum_{c \in C} \tau^c \right) \to \rho_d.$$

Each entry of the method vector consists of a *dispatcher* that determines the result as a function of the instance data associated with a given object. This organization makes it easy to add new methods for a given collection of classes by simply defining a new function of this type. It makes adding a new class relatively more difficult, however, since doing so requires that each method be updated to account for the new forms of object.

An object is a value of type $\tau = \sum_{c \in C} \tau^c$, the sum over the classes of the instance types. For example, the type of points in the plane is the sum type

$$[\mathtt{cart} : \tau^{\mathtt{cart}}, \mathtt{pol} : \tau^{\mathtt{pol}}].$$

Each point is labelled with its class, specifying its representation as having either cartesian or polar form.

An instance of a class c is just the instance data labelled with its class to form an element of the object type:

$$\mathtt{new}[c](e) \triangleq c \cdot e.$$

For example, a cartesian point with coordinates x_0 and y_0 is given by the expression

$$\mathtt{new}[\mathtt{cart}](\langle \mathtt{x} = x_0, \mathtt{y} = y_0 \rangle) \triangleq \mathtt{cart} \cdot \langle \mathtt{x} = x_0, \mathtt{y} = y_0 \rangle.$$

Similarly, a polar point with distance ρ_0 and angle θ_0 is given by the expression

$$\mathtt{new}[\mathtt{pol}](\langle \mathtt{r} = r_0, \mathtt{th} = \theta_0 \rangle) \triangleq \mathtt{pol} \cdot \langle \mathtt{r} = r_0, \mathtt{th} = \theta_0 \rangle.$$

The method-based organization consolidates the implementation of each method into the *method vector*, e_{mv} of type τ_{mv}, defined by $\langle e_d \rangle_{d \in D}$, where for each $d \in D$ the expression $e_d : \tau \to \rho_d$ is

$$\lambda\,(\mathit{this} : \tau.\, \mathtt{case}\, \mathit{this}\, \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d(u)\}_{c \in C}).$$

Each entry in the method vector may be thought of as a *dispatch function* that determines the action of that method on each class of object.

In the case of points in the plane, the method vector has the product type

$$\langle \mathtt{dist} : \tau \to \rho_{\mathtt{dist}}, \mathtt{quad} : \tau \to \rho_{\mathtt{quad}} \rangle.$$

The dispatch function for the `dist` method has the form

$$\lambda\,(\mathit{this}:\tau.\,\texttt{case}\,\mathit{this}\,\{\texttt{cart}\cdot u \Rightarrow e_{\mathsf{dm}}\cdot\texttt{cart}\cdot\texttt{dist}(u)\mid\texttt{pol}\cdot v\Rightarrow e_{\mathsf{dm}}\cdot\texttt{pol}\cdot\texttt{dist}(v)\}),$$

and the dispatch function for the `quad` method has the similar form

$$\lambda\,(\mathit{this}:\tau.\,\texttt{case}\,\mathit{this}\,\{\texttt{cart}\cdot u \Rightarrow e_{\mathsf{dm}}\cdot\texttt{cart}\cdot\texttt{quad}(u)\mid\texttt{pol}\cdot v\Rightarrow e_{\mathsf{dm}}\cdot\texttt{pol}\cdot\texttt{quad}(v)\}).$$

The *message send* operation, $e \Leftarrow d$, applies the dispatch function for method d to the object e:

$$e \Leftarrow d \triangleq e_{\mathsf{mv}}\cdot d(e).$$

Thus we have, for each class, c, and method, d,

$$\begin{aligned}(\texttt{new}[c]\,(e)) \Leftarrow d &\mapsto^* e_{\mathsf{mv}}\cdot d(c\cdot e)\\ &\mapsto^* e_{\mathsf{dm}}\cdot c\cdot d(e)\end{aligned}$$

That is, the message send invokes the implementation of the method d on the instance data for the given object.

27.3 Class-Based Organization

The class-based organization starts with the observation that the dispatch matrix may be reorganized to "factor out" the instance data for each method acting on that class to obtain the *class vector*, e_{cv}, of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c\in C}(\tau^c \to (\prod_{d\in D}\rho_d)).$$

Each row of the class vector consists of a *constructor* that determines the result of each of the methods when acting on given instance data. This organization makes it easy to add a new class to the program; we need only define the method tuple on the instance data for the new class. It makes adding a new method relatively more difficult, however, because we must extend the interpretation of each class to account for it.

An object has the type $\rho = \prod_{d\in D}\rho_d$ consisting of the product over the methods of the result types of the methods. For example, in the case of points in the plane, the type ρ is the product type

$$\langle\texttt{dist}:\rho_{\texttt{dist}},\texttt{quad}:\rho_{\texttt{quad}}\rangle.$$

Each component specifies the result of each of the methods acting on that object.

The message send operation, $e \Leftarrow d$, is just the projection $e \cdot d$. So, in the case of points in the plane, $e \Leftarrow \texttt{dist}$ is the projection $e \cdot \texttt{dist}$, and similarly $e \Leftarrow \texttt{quad}$ is the projection $e \cdot \texttt{quad}$.

The class-based organization consolidates the implementation of each class into a *class vector*, e_{cv}, a tuple of type τ_{cv} consisting of the *constructor* of type $\tau^c \to \rho$ for each class $c \in C$. The class vector is defined by $e_{\mathsf{cv}} = \langle e^c \rangle_{c \in C}$, where for each $c \in C$ the expression e^c is

$$\lambda\,(u : \tau^c.\, \langle e_{\mathsf{dm}} \cdot c \cdot d(u) \rangle_{d \in D}).$$

For example, the constructor for the class `cart` is the function $e^{\texttt{cart}}$ given by the expression

$$\lambda\,(u : \tau^{\texttt{cart}}.\, \langle \texttt{dist} = e_{\mathsf{dm}} \cdot \texttt{cart} \cdot \texttt{dist}(u), \texttt{quad} = e_{\mathsf{dm}} \cdot \texttt{cart} \cdot \texttt{quad}(u) \rangle).$$

Similarly, the constructor for the class `pol` is the function $e^{\texttt{pol}}$ given by the expression

$$\lambda\,(u : \tau^{\texttt{pol}}.\, \langle \texttt{dist} = e_{\mathsf{dm}} \cdot \texttt{pol} \cdot \texttt{dist}(u), \texttt{quad} = e_{\mathsf{dm}} \cdot \texttt{pol} \cdot \texttt{quad}(u) \rangle).$$

The class vector, e_{cv}, in this case is the tuple $\langle \texttt{cart} = e^{\texttt{cart}}, \texttt{pol} = e^{\texttt{pol}} \rangle$ of type $\langle \texttt{cart} : \tau^{\texttt{cart}} \to \rho, \texttt{pol} : \tau^{\texttt{pol}} \to \rho \rangle$.

An instance of a class is obtained by applying the constructor for that class to the instance data:

$$\texttt{new}[c](e) \triangleq e_{\mathsf{cv}} \cdot c(e).$$

For example, a cartesian point is obtained by writing $\texttt{new}[\texttt{cart}](\langle \texttt{x} = x_0, \texttt{y} = y_0 \rangle)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \texttt{cart}(\langle \texttt{x} = x_0, \texttt{y} = y_0 \rangle).$$

Similarly, a polar point is obtained by writing $\texttt{new}[\texttt{pol}](\texttt{r} = r_0, \texttt{th} = \theta_0)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \texttt{pol}(\langle \texttt{r} = r_0, \texttt{th} = \theta_0 \rangle).$$

It is easy to check for this organization of points that for each class c and method d, we may derive

$$\begin{aligned} (\texttt{new}[c](e)) \Leftarrow d &\mapsto^* (e_{\mathsf{cv}} \cdot c(e)) \cdot d \\ &\mapsto^* e_{\mathsf{dm}} \cdot c \cdot d(e) \end{aligned}$$

The outcome is, of course, the same as for the method-based organization.

27.4 Self-Reference

It is often useful to allow methods to create new objects or to send messages to objects. This is not possible using the dispatch matrix described in Section 27.1 on page 257, for the simple reason that object creation and message send are defined in terms of the dispatch matrix itself, and are sensitive to whether we are using a class- or method-based organization. To allow for this form of self-reference, we will generalize the type of the dispatch matrix so that each method is given access to both the class and method vector representations:

$$\prod_{c \in C} \prod_{d \in D} \forall (t \,.\, \tau_{\mathsf{cv}} \to \tau_{\mathsf{mv}} \to \tau^c \to \rho_d).$$

The type variable, t, is an abstract type representing the object type. The types τ_{cv} and τ_{mv}, are, respectively, the type of the class and method vectors, defined in terms of the abstract type t. They are defined by the equations

$$\tau_{\mathsf{cv}} = \prod_{c \in C} (\tau^c \to t),$$

and

$$\tau_{\mathsf{mv}} = (\prod_{d \in D} t \to \rho_d).$$

The component of the class vector corresponding to a class, c, is a constuctor that builds a value of the abstract object type, t, from the instance data for c. The component of the method vector corresponding to a method, d, is a dispatcher that yields a result of type ρ_d when applied to a value of the abstract object type, t.

In accordance with the revised type of the dispatch matrix the entry corresponding to class c and method d has the form

$$\Lambda (t \,.\, \lambda\ (cv : \tau_{\mathsf{cv}} \,.\, \lambda\ (mv : \tau_{\mathsf{mv}} \,.\, \lambda\ (u : \tau^c \,.\, e)))).$$

The arguments cv and mv are used to create new instances and to send messages to instances. A new object of c' with instance data e' is created by writing $cv \cdot c'(e')$, and a message d' is sent to an object e' by writing $mv \cdot d'(e')$. Either the class c' or the method d' may well be the class c or method d themselves.

For the method-based organization the method vector, e_{mv}, has the self-referential type $[\tau/t]\tau_{\mathsf{mv}}$ `self` in which the object type is specified to be

the sum of the instance types of each of the classes. It is defined by the expression

$$\texttt{self}\ mv\ \texttt{is}\ \langle d = \lambda\ (this : \tau.\ \texttt{case}\ this\ \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d\,[\tau]\,(e'_{\mathsf{cv}})\,(e'_{\mathsf{mv}})\,(u)\}_{c \in C})\rangle_{d \in D}.$$

The class vector argument, e'_{cv}, is the tuple of tagging operations

$$\langle c = \lambda\ (u : \tau^c.\ c \cdot u)\rangle_{c \in C},$$

and the method vector argument, e'_{mv}, is the recursive unrolling of the method vector itself, namely $\texttt{unroll}(mv)$. The message send operation $e \Leftarrow d$ is given by the expression $\texttt{unroll}(e_{\mathsf{mv}}) \cdot d(e)$, whereas objection creation, $\texttt{new}[c](e)$, is defined, as before, to be $c \cdot e$.

For the class-based organization the class vector, e_{cv}, has the type $[\rho/t]\tau_{\mathsf{cv}}$ $\texttt{self}$ in which the object type is the product of the result types of each of the methods. It is defined by the expression

$$\texttt{self}\ cv\ \texttt{is}\ \langle c = \lambda\ (u : \tau^c.\ \langle d = e_{\mathsf{dm}} \cdot c \cdot d\,[\rho]\,(e''_{\mathsf{cv}})\,(e''_{\mathsf{mv}})\,(u)\rangle_{d \in D})\rangle_{c \in C}$$

where the class vector argument, e''_{cv}, is $\texttt{unroll}(cv)$, and the method vector argument, e''_{mv}, is the tuple of projections,

$$\langle d = \lambda\ (this : \rho.\ this \cdot d)\rangle_{d \in D}.$$

Object creation, $\texttt{new}[c](e)$ is defined by the expression $\texttt{unroll}(e_{\mathsf{cv}}) \cdot c(e)$, whereas message send, $d \Leftarrow e$, is defined, as before, by $e \cdot d$.

The symmetries between the two organizations are striking. They are a reflection of the fundamental symmetries between sum and product types.

27.5 Notes

The term "object-oriented" means many things to many people, but certainly dynamic dispatch is one of its central concepts. Both class-based and method-based organizations are popular, and have strong adherents, but, given the inherent symmetries of the situation, it seems impossible to argue that one is superior to the other. The literature on object-oriented programming is extensive. Pierce's text [77] discusses many more aspects of object-oriented programming than are considered here, and provides many good references to the literature.

Chapter 28

Inheritance

Dynamic dispatch is a means of organizing a collection of methods, D, acting on a collection of classes, C. A dispatch matrix assigns to each class, $c \in C$, and each method, $d \in D$, a function of type $\tau^c \to \rho_d$ mapping the instance data for c to the result type for d. For fixed C and D one may define a dispatch matrix by providing such a function for each entry. However, it is sometimes useful to build a new dispatch matrix by extending an old one with new classes or methods, providing definitions for the new classes and methods, and either *inheriting* or *overriding* the entries corresponding to the old classes and methods. Methodologically speaking, the modifications to the old entries are meant to be relatively few, but nothing precludes a wholesale redefinition of every method of a class, or every class of a method.

As a consequence, knowing that one dispatch matrix is derived from another tells us nothing about the behavior of their respective instances.[1] Unfortunately, inheritance is often confused with subtyping. Inheritance is a statement about how a body of code came into being, whereas subtyping is a statement about how a body of code can be expected to behave. Many languages seek to ensure that if one class inherits from another, then the type of objects of the subclass is a subtype of the type of objects of the superclass. As we shall see, this is not always the case.

Some languages permit one dispatch matrix to be determined by inheritance and overriding from several dispatch tables simultaneously, whereas limit themselves to at most one. Languages that impose this restriction are said to admit *single inheritance*, whereas those that do not are said to ad-

[1] In view of this one may doubt the significance of programming methodologies that stress inheritance as a central principle.

mit *multiple inheritance*. The main difficulty with multiple inheritance is that there must be some means of resolving the ambiguity that arises when more than one parent provides a class or method that the child wishes to inherit. With single inheritance no such ambiguity can arise. The main reason to limit attention to the single inheritance case is to avoid problems that arise when inheritance is confused with, or conflated with, subtyping. For then there may be several inequivalent ways to regard one type as a subtype of another, leading to semantic ambiguities. But solely as a mechanism for defining dispatch matrices, there seems to be no fundamental reason to restrict to single inheritance.

In this chapter we will consider the most common form of inheritance, called *subclassing*, which is defined for the class-based organization described in Chapter 27. We are given a class vector, e_{cv}, and we are to define a new class vector, e^*_{cv}, by adding a *subclass*, c^* of the *superclasses* $c_1, \ldots, c_n$, where $n \geq 0$. A dual form of inheritance, for which there is no established terminology, may be defined for the method-based organization given in Chapter 27. We will not develop this idea in detail here, since the development follows along similar lines to subclassing.

28.1 Subclassing

Let $e_{\mathsf{cv}} : (\prod_{c \in C} \tau^c \to \prod_{d \in D} \rho_d)$ be a class vector as described in Chapter 27, and suppose that $c_1, \ldots, c_n \in C$. To define a subclass, $c_* \notin C$, of the superclasses $c_1, \ldots, c_n$, requires the following information:

1. The instance type τ^{c_*} of the new class such that $\tau^{c_*} <: \tau^{c_i}$ for each $1 \leq i \leq n$.

2. The subset $D_{\mathsf{inh}} \subseteq D$ of *inherited* methods. The remainder, $D_{\mathsf{ovr}} = D \setminus D_{\mathsf{inh}}$, is the set of *overridden* methods.

3. For each $d \in D_{\mathsf{ovr}}$, an expression $e_d : \tau^{c_*} \to \rho_d$, defining the action of the overridden method, d, on instances of the class c_*.

The subtyping requirement on the type τ^{c_*} ensures that the inherited methods are applicable to instances of the new class.

The extended class vector, e^*_{cv}, is defined from this data as follows:

1. For each class $c \in C$, the constructor $e^*_{\mathsf{cv}} \cdot c$ is equivalent to the constructor $e_{\mathsf{cv}} \cdot c$. All existing classes are preserved intact.

2. The action of method d on class c_* is determined by whether it is an inherited or an overridden method:

   (a) If $d \in D_{\mathsf{inh}}$ is an inherited method, then, for some $1 \leq i \leq n$ and any instance data e_* of type τ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_*) \cdot d$ is equivalent to the method $e^*_{\mathsf{cv}} \cdot c_i(e_*) \cdot d$. If there is more than one parent class, the choice of i must be made by some convention to avoid ambiguity.

   (b) If $d \in D_{\mathsf{ovr}}$ is an overridden method, then for any instance data e_* of type τ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_*) \cdot d$ is equivalent to $e_d(e_*)$, where e_d is given by the subclass definition.

The resulting class vector, e^*_{cv}, is of type

$$\prod_{c \in C \cup \{ c_* \}} (\tau^c \to \prod_{d \in D} \rho_d).$$

The requirement that $\tau^{c_*} <: \tau^{c_i}$ ensures that the instance data for the subclass may be acted upon by a method of the superclass. The condition on inherited methods ensures that the message $\texttt{new}[c_*](e_*) \Leftarrow d$ is equivalent to the message $\texttt{new}[c_i](e_*) \Leftarrow d$, and the condition on overridden methods ensure that $\texttt{new}[c_*](e_*) \Leftarrow d$ is equivalent to $e_d(e_*)$.

So far the object type $\rho = \prod_{d \in D} \rho_d$ is unaffected in the passage from the super- to the subclass. It is also possible to allow each class to determine the methods supported on it, and to allow the result types of methods to vary with each class. To account for the first possibility, we consider a family of sets $\{ D_c \}_{c \in C}$ in which D_c is the set of methods associated to class c. To account for the second possibility, let ρ^c_d be the result type of method d when acting on an object of class $c \in C$. The class vector then has the type

$$\prod_{c \in C} \tau^c \to (\prod_{d \in D_c} \rho^c_d).$$

The type $\rho^c = \prod_{d \in D_c} \rho^c_d$ is then the type of instances of class c.

Under this generalization we may define a subclas, c_*, of the superclasses by specifying the following information:

1. The instance type, τ^{c_*}, for the new class such that $\tau^{c_*} <: \tau^{c_i}$ for each $1 \leq i \leq n$.

2. The set $D_{c_*} = D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$ of inherited, overridden, and extending methods associated with class c_* such that $D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} \subseteq \bigcup_{1 \leq i \leq n} D_{c_i}$.

3. For each $d \in D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$, an expression $e_d : \tau^{c_*} \to \rho_d^{c_*}$ providing the implementation of method d for the new class c_*.

Since each class determines its own collection of methods, we now require that $D_{c_i} \cap D_{c_j} = \emptyset$ whenever $i \neq j$, so that no method is provided by more than one superclass.

This data determines a new class vector, e^*_{cv}, defined from the old class vector e_{cv}, as follows:

1. For each class $c \in C$, the constructor $e^*_{\mathsf{cv}} \cdot c$ is equivalent to the constructor $e_{\mathsf{cv}} \cdot c$.

2. The action of the method d on an instance of class c_* is defined as follows:

   (a) If $d \in D_{\mathsf{inh}}$ is an inherited method, then, for a unique $1 \leq i \leq n$ and any instance data e_* of type τ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_*) \cdot d$ is equivalent to the method $e^*_{\mathsf{cv}} \cdot c_i(e_*) \cdot d$.

   (b) If $d \in D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$ is an overridden or extending method, then for any instance data e_* of type τ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_*) \cdot d$ is equivalent to $e_d(e_*)$, where e_d is given by the subclass definition.

There is no ambiguity in the choice of inherited methods, since the superclass method suites are assumed to be disjoint.

The type of the new class vector is

$$\prod_{c \in C \uplus \{c_*\}} (\tau^c \to (\prod_{d \in D_c} \rho_d^c)).$$

The object type for instances of a class $c \in C$ is the product type $\rho^c = \prod_{d \in D_c} \rho_d^c$. The object type, ρ^{c_*}, of the new class, c_*, need not be a subtype of any old object type ρ^{c_i}. For example, the subclass result type of an overriding method need not bear any relationship to any superclass result type for that method. Moreover, the subclass need not provide all of the methods provided by the superclasses.

This means that *inheritance does not imply subtyping*. However, if we impose the requirements that (1) if $d \in D_{\mathsf{ovr}}$, then $\rho_d^{c_*} <: \rho_d^{c_i}$ for the unique superclass c_i providing the method d, and (2) $D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} = \bigcup_{1 \leq i \leq n} D_{c_i}$, then the instance type of a subclass will be a subtype of that of the superclass, so that a subclass object may be used wherever a superclass object is required. Many object-oriented languages insist on this condition as a requirement for inheritance. This policy is not, however, always sustainable. Certain forms of inheritance, such as those supporting "self type", are incompatible with this requirement.

28.2 Notes

Adherents of object-oriented languages differ on the importance of inheritance. Most such languages support inheritance, but it is a common practice to avoid it when building systems, precisely because it is anti-modular by its very nature. (To understand a subclass, one must fully understand the code, not the interface, of the superclass.) The emphasis on the class-based organization in many extant object-oriented languages seems misplaced, given the inherent symmetries of the situation. Confusion between subclassing and subtyping is rampant, as is the confusion between types and classes. Fundamentally these concepts are, and should be kept, separate. Pierce's text provides a thorough discussion of the complex interactions between inheritance and subtyping, particularly in the presence of polymorphism [77].

Part XI

Control Effects

Chapter 29

Control Stacks

The technique of structural dynamics 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 dynamics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

29.1 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*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{29.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k;f\ \mathsf{stack}} \tag{29.1b}$$

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{29.2a}$$

$$\frac{}{\mathtt{ifz}(-;e_1;x.e_2)\ \mathsf{frame}} \tag{29.2b}$$

$$\frac{}{\mathtt{ap}(-;e_2)\ \mathsf{frame}} \tag{29.2c}$$

The frames correspond to searchrules in the dynamics 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 $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \rhd \mathtt{z} \mapsto k \lhd \mathtt{z}} \tag{29.3a}$$

$$\frac{}{k \rhd \mathtt{s}(e) \mapsto k;\mathtt{s}(-) \rhd e} \tag{29.3b}$$

$$\frac{}{k;\mathtt{s}(-) \lhd e \mapsto k \lhd \mathtt{s}(e)} \tag{29.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor, and evaluate e; when that returns with e', we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \rhd \mathtt{ifz}(e;e_1;x.e_2) \mapsto k;\mathtt{ifz}(-;e_1;x.e_2) \rhd e} \tag{29.4a}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x.e_2) \lhd \mathtt{z} \mapsto k \rhd e_1} \tag{29.4b}$$

$$\overline{k;\mathtt{ifz}(-;e_1;x.e_2) \triangleleft \mathtt{s}(e) \mapsto k \triangleright [e/x]e_2} \tag{29.4c}$$

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined, we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions and recursion.

$$\overline{k \triangleright \mathtt{lam}[\tau](x.e) \mapsto k \triangleleft \mathtt{lam}[\tau](x.e)} \tag{29.5a}$$

$$\overline{k \triangleright \mathtt{ap}(e_1;e_2) \mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1} \tag{29.5b}$$

$$\overline{k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau](x.e) \mapsto k \triangleright [e_2/x]e} \tag{29.5c}$$

$$\overline{k \triangleright \mathtt{fix}[\tau](x.e) \mapsto k \triangleright [\mathtt{fix}[\tau](x.e)/x]e} \tag{29.5d}$$

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 39 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:

$$\overline{\epsilon \triangleright e \text{ initial}} \tag{29.6a}$$

$$\frac{e \text{ val}}{\epsilon \triangleleft e \text{ final}} \tag{29.6b}$$

29.2 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:

$$\overline{\epsilon : \tau} \tag{29.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k;f : \tau} \tag{29.7b}$$

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\overline{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{29.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-;e_1;x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{29.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-;e_2) : \mathtt{arr}(\tau_2;\tau) \Rightarrow \tau} \tag{29.8c}$$

The two forms of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{29.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{29.9b}$$

We leave the proof of safety of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ as an exercise.

Theorem 29.1 (Safety)**.** *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

29.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ 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

$$\epsilon \triangleright \mathtt{ap}(e_1;e_2) \mapsto \epsilon;\mathtt{ap}(-;e_2) \triangleright e_1,$$

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

$$k \triangleright \mathtt{ap}(e_1'; e_2) \mapsto k; \mathtt{ap}(-; e_2) \triangleright e_1'.$$

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 dynamics described in Chapter 9, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 29.2. *If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 9.2 on page 83 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 29.3. *If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then $e \mapsto^* e'$.*

Corollary 29.4. *$e \mapsto^* \overline{n}$ 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.

29.3.1 Completeness

Proof of Lemma 29.2 on the previous page. The proof is by induction on an evaluation dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{29.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1;e_2) &\mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1 \\ &\mapsto^* k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

29.3.2 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:

$$\frac{k \bowtie e = e'}{k \triangleright e \not\mapsto e'} \tag{29.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \not\mapsto e'} \tag{29.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{29.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k;\mathtt{s}(-) \bowtie e = e'} \tag{29.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1;e_2;x.e_3) = e'}{k;\mathtt{ifz}(-;e_2;x.e_3) \bowtie e_1 = e'} \tag{29.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1;e_2) = e}{k;\mathtt{ap}(-;e_2) \bowtie e_1 = e} \tag{29.12d}$$

These judgements both define total functions.

Lemma 29.5. *The judgement $s \mathrel{\rightarrowtail\!\!\!\!\!\!\circ}\; e$ has mode $(\forall, \exists!)$, and the judgement $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique e' such that $k \bowtie e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

Lemma 29.6. *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

Proof. The proof is by rule induction on the transition $e \mapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, k.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (29.12) that $k; \mathtt{ap}(-; e_2) \bowtie e_1 = d$ and $k; \mathtt{ap}(-; e_2) \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x.e); e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k'; \mathtt{ap}(-; c_2)$. It follows from Rules (29.12) that $k' \bowtie \mathtt{ap}(e; c_2) = d$ and $k' \bowtie \mathtt{ap}(e'; c_2) = d'$. But by the SOS rules $\mathtt{ap}(e; c_2) \mapsto \mathtt{ap}(e'; c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 29.3 on page 277.

Proof of Lemma 29.3 on page 277. 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 $s' = k; \mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $k; \mathtt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k; \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); -) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k; \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); -) \bowtie e_2 = e'$ 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 29.6.

□

29.4 Notes

The abstract machine considered here is typical of a wide class of machines that make control flow explicit in the state. The prototype is Landin's SECD machine [53], which may be seen as a linearization of a structural operational semantics [82]. An advantage of a machine model is that the explicit treatment of control is natural for languages that allow the control state to be explicitly manipulated (see Chapter 31 for a prime example). A disadvantage is that one is required to make explicit the control state of the computation, rather than leave it implicit as in structural operational semantics. Which is better depends wholly on the situation at hand, though historically there has been greater emphasis on abstract machines than on structural semantics.

Chapter 30

Exceptions

Exceptions effect a non-local transfer of control from the point at which the exception is *raised* to an 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.

30.1 Failures

A *failure* is a control mechanism that permits a computation to refuse to return a value to the point of its evaluation. Failure can be detected by *catching* it, diverting evaluation to another expression, called a *handler*. Failure can be turned into success, provided that the handler does not itself fail.

The following grammar defines the syntax of failures:

Exp	e	::=	`fail`	`fail`	failure
			`catch`$(e_1;e_2)$	`catch` e_1 `ow` e_2	handler

The expression `fail` aborts the current evaluation, and the expression `catch`$(e_1;e_2)$ handles any failure in e_1 by evaluating e_2 instead.

The statics of failures is straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{30.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1;e_2) : \tau} \tag{30.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, since either might determine the value of that expression.

The dynamics of failures may be given using *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.

Stack unwinding can be defined directly using structural dynamics, but we prefer to make use of the stack machine defined in Chapter 29. In addition to states of the form $k \triangleright e$, which evaluates the expression e on the stack k, and $k \triangleleft e$, which passes the value e to the stack k, we make use of an additional form of state, $k \blacktriangleleft$, which passes a failure up the stack to the nearest enclosing handler.

The set of frames defined in Chapter 29 is extended with the additonal form $\texttt{catch}(-;e_2)$. The transition rules given in Chapter 29 are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail} \mapsto k \blacktriangleleft} \tag{30.2a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1;e_2) \mapsto k;\texttt{catch}(-;e_2) \triangleright e_1} \tag{30.2b}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \triangleleft v \mapsto k \triangleleft v} \tag{30.2c}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \blacktriangleleft \mapsto k \triangleright e_2} \tag{30.2d}$$

$$\frac{}{k;f \blacktriangleleft \mapsto k \blacktriangleleft} \tag{30.2e}$$

As a notational convenience, we require that Rule (30.2e) apply only if none of the preceding rules apply. Evaluating `fail` propagates a failure up the stack. The act of raising an exception may itself raise an exception. 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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{30.3a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}} \tag{30.3b}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 29 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different: a well-typed program does not get stuck, but it may well result in an uncaught failure!

Theorem 30.1 (Safety)**.** 1. *If s ok and $s \mapsto s'$, then s' ok.*

2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

30.2 Exceptions

Failures are simplistic in that they do not distinguish different causes, and hence do not permit handlers to react differently to different circumstances. An *exception* is a generalization of a failure that associates a value with the failure. This value is passed to the handler, allowing it to discriminate between various forms of failures, and to pass data appropriate to that form of failure. The type of values associated with exceptions is discussed in Section 30.3 on the next page. For now, we simply assume that there is some type, τ_{exn}, of values associated with a failure.

The syntax of exceptions is given by the following grammar:

Exp	e	::=	$\texttt{raise}[\tau](e)$	$\texttt{raise}(e)$	exception
			$\texttt{handle}(e_1; x.e_2)$	$\texttt{handle}\, e_1\, \texttt{ow}\, x \Rightarrow e_2$	handler

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\texttt{handle}(e_1; x.e_2)$ binds a variable, x, in the handler, e_2, to which the associated value of the exception is bound, should an exception be raised during the execution of e_1.

The statics of exceptions generalizes that of failures:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau} \tag{30.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1; x.e_2) : \tau} \tag{30.4b}$$

The dynamics of exceptions is a mild generalization of the dynamics of failures in which we generalize the failure state, $k \blacktriangleleft$, to the exception state, $k \blacktriangleleft e$, which passes a value of type τ_{exn} along with the failure. The syntax of stack frames is extended to include $\texttt{raise}[\tau](-)$ and $\texttt{handle}(-; x.e_2)$. The dynamics of exceptions is specified by the following rules:

$$\overline{k \triangleright \texttt{raise}[\tau](e) \mapsto k;\texttt{raise}[\tau](-) \triangleright e} \tag{30.5a}$$

$$\overline{k;\texttt{raise}[\tau](-) \triangleleft e \mapsto k \blacktriangleleft e} \tag{30.5b}$$

$$\overline{k;\texttt{raise}[\tau](-) \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.5c}$$

$$\overline{k \triangleright \texttt{handle}(e_1; x.e_2) \mapsto k;\texttt{handle}(-; x.e_2) \triangleright e_1} \tag{30.5d}$$

$$\overline{k;\texttt{handle}(-; x.e_2) \triangleleft e \mapsto k \triangleleft e} \tag{30.5e}$$

$$\overline{k;\texttt{handle}(-; x.e_2) \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{30.5f}$$

$$\frac{(f \neq \texttt{handle}(-; x.e_2))}{k; f \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{30.5g}$$

It is a straightforward exercise to extend the safety theorem given in Section 30.1 on page 281 to exceptions.

30.3 Exception Type

The statics of exceptions is parameterized by the type of exception values, τ_{exn}. This type may be chosen arbitrarily, but it must be shared by all exceptions in a program to ensure type safety. For otherwise a handler cannot tell what type of value to expect from an exception, compromising safety.

But how do we choose the type of exceptions? A very naïve choice would be to take τ_{exn} to be the type `str`, so that, for example, one may write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. This is fine as far as it goes, but a handler for such an exception would have to interpret the string if it is to distinguish one exception from another!

Motivated by this, we might choose τ_{exn} to be `nat`, which amounts to saying that exceptional conditions are coded as natural numbers.[1] This does allow the handler to distinguish one source of failure from another, but makes no provision for associating data with the failure. Moreover, it forces the programmer to impose a single, global convention for indexing the causes of failure, compromising modular development and evolution.

The first concern—how to associate data specific to the type of failure—can be addressed by taking τ_{exn} to be a labelled sum type whose classes are the forms of failure, and whose associated types determine the form of the data attached to the exception. For example, the type τ_{exn} might have the form

$$\tau_{exn} = \texttt{[div:unit,fnf:string,...]}.$$

The class `div` might represent an arithmetic fault, with no associated data, and the class `fnf` might represent a "file not found" error, with associated data being the name of the file.

Using a sum type for τ_{exn} makes it easy for the handler to discriminate on the source of the failure, and to recover the associated data without fear of a type safety violation. For example, we might write

```
try e1 ow x ⇒
  match x {
    div ⟨⟩ ⇒ e_div
  | fnf s ⇒ e_fnf }
```

to handle the exceptions specified by the sum type given in the preceding paragraph.

The problem with choosing a sum type for τ_{exn} is that it imposes a *static classification* of the sources of failure in a program. There must be one, globally agreed-upon type that classifies all possible forms of failure, and specifies their associated data. Using sums in this manner impedes modular

[1] In Unix these are called `errno`'s, for *error numbers*.

development and evolution, since all of the modules comprising a system must agree on the one, central type of exception values. A better approach is to use *dynamic classification* for exception values by choosing τ_{exn} to be an *extensible sum*, one to which new classes may be added at execution time. This allows separate program modules to introduce their own failure classification scheme without worrying about interference with one another; the initialization of the module generates new classes at run-time that are guaranteed to be distinct from all other classes previously or subsequently generated. (See Chapter 36 for more on dynamic classification.)

30.4 Encapsulation

It is sometimes useful to distinguish expressions that can fail or raise an exception from those that cannot. An expression is called *fallible*, or *exceptional*, if it can fail or raise an exception during its evaluation, and is *infallible*, or *unexceptional*, otherwise. The concept of fallibility is intentionally permissive in that an infallible expression may be considered to be (vacuously) fallible, whereas infallibility is intended to be strict in that an infallible expression cannot fail. Consequently, if e_1 and e_2 are two infallible expressions both of whose values are required in a computation, we may evaluate them in either order without affecting the outcome. If, on the other hand, one or both are fallible, then the outcome of the computation is sensitive to the evaluation order (whichever fails first determines the overall result).

To formalize this distinction we distinguish two *modes* of expression, the fallible and the infallible, linked by a *modality* classifying the fallible expressions of a type.

Type	τ	::=	`fallible(`τ`)`	τ `fallible`	fallible
Fall	f	::=	`fail`	`fail`	failure
			`succ(`e`)`	`succ` e	success
			`try(`e`;`x`.`f_1`;`f_2`)`	`let fall(`x`) be` e `in` f_1 `ow` f_2	handler
Infall	e	::=	x	x	variable
			`fall(`f`)`	`fall` f	fallible
			`try(`e`;`x`.`e_1`;`e_2`)`	`let fall(`x`) be` e `in` e_1 `ow` e_2	handler

The type τ `fallible` is the type of encapsulated fallible expressions of type τ. Fallible expressions include failures, successes (infallible expressions thought of as vacuously fallible), and handlers that intercept failures,

but which may itself fail. Infallible expressions include variables, encapsulated fallible expressions, and handlers that intercepts failures, always yielding an infallible result.

The statics of encapsulated failures consists of two judgement forms, $\Gamma \vdash e : \tau$ for infallible expressions and $\Gamma \vdash f \sim \tau$ for fallible expressions. These judgements are defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{30.6a}$$

$$\frac{\Gamma \vdash f \sim \tau}{\Gamma \vdash \mathtt{fall}(f) : \mathtt{fallible}(\tau)} \tag{30.6b}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{try}(e; x.e_1; e_2) : \tau'} \tag{30.6c}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} \sim \tau} \tag{30.6d}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{succ}(e) \sim \tau} \tag{30.6e}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash f_1 \sim \tau' \quad \Gamma \vdash f_2 \sim \tau'}{\Gamma \vdash \mathtt{try}(e; x.f_1; f_2) \sim \tau'} \tag{30.6f}$$

Rule (30.6c) specifies that a handler may be used to turn a fallible expression (encapsulated by e) into an infallible computation, provided that the result is infallible regardless of whether the encapsulated expression succeeds or fails.

The dynamics of encapsulated failures is readily derived, though some care must be taken with the elimination form for the modality.

$$\frac{}{\mathtt{fall}(f)\ \mathsf{val}} \tag{30.7a}$$

$$\frac{}{k \triangleright \mathtt{try}(e; x.e_1; e_2) \mapsto k; \mathtt{try}(-; x.e_1; e_2) \triangleright e} \tag{30.7b}$$

$$\frac{}{k; \mathtt{try}(-; x.e_1; e_2) \triangleleft \mathtt{fall}(f) \mapsto k; \mathtt{try}(-; x.e_1; e_2); \mathtt{fall}(-) \triangleright f} \tag{30.7c}$$

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \tag{30.7d}$$

$$\frac{}{k \triangleright \mathtt{succ}(e) \mapsto k;\mathtt{succ}(-) \triangleright e} \tag{30.7e}$$

$$\frac{}{k;\mathtt{succ}(-) \triangleleft e \mapsto k \triangleleft \mathtt{succ}(e)} \tag{30.7f}$$

$$\frac{e \text{ val}}{k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \triangleleft \mathtt{succ}(e) \mapsto k \triangleright [e/x]e_1} \tag{30.7g}$$

$$\frac{}{k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \blacktriangleleft \mapsto k \triangleright e_2} \tag{30.7h}$$

We have omitted the rules for the fallible form of handler; they are similar to Rules (30.7b) to (30.7b) and (30.7g) to (30.7h), albeit with infallible subexpressions e_1 and e_2 replaced by fallible subexpressions f_1 and f_2.

An initial state has the form $k \triangleright e$, where e is an infallible expression, and k is a stack of suitable type. Consequently, a fallible expression, f, can only be evaluated on a stack of the form

$$k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-)$$

in which a handler for any failure that may arise from f is present. Therefore, a final state has the form $\epsilon \triangleleft e$, where e val; no uncaught failure can arise.

30.5 Notes

Various forms of exceptions were explored in the many dialects of Lisp (see, for example, [98]). The original formulation of ML as a metalanguage for mechanized logic [35] made extensive use of exceptions (called "failures") to implement tactics and tacticals. Most modern languages now include an exception mechanism of the kind considered here.

The essential distinction between the exception *mechanism* and exception *values* is often misunderstood. The two have no relationship to one another. Exception values are often dynamically classified (see Chapter 36), but dynamic classification has many more uses than just exception values. Another common misconception is to link exceptions with fluid binding (for which see Chapter 35). As the account given here makes clear, there is absolutely no relationship between exceptions and fluid binding. Exceptions are simply a particular use of the monad associated with option types, as described in Section 30.4 on page 286.

Chapter 31

Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *continuations*; they are values that can be passed and returned at will in a computation. Continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus 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.

Why are continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor.

31.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type `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](e1;e2)`, which restores the value of e_1 to the control stack that is the value of e_2.

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 → 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 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:τ' → τ,k:τ cont):τ' cont = ....
```

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. This is the continuation that, when given a value v', applies f to it, and throws the result to k. We can seize this continuation using `letcc`, writing

```
throw f(letcc x:τ' cont in ...) to k
```

At the point of the ellipsis the variable x is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code:

```
fun compose (f:τ' → τ, k:τ cont):τ' cont =
    letcc ret:τ' cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

The type of `ret` is that of a continuation-expecting continuation!

31.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\to\}$ expressions with these additional forms:

Type	τ	::=	`cont(`τ`)`	τ `cont`	continuation
Expr	e	::=	`letcc[`τ`](`x`.`e`)`	`letcc` x `in` e	mark
			`throw[`τ`](`e_1`;`e_2`)`	`throw` e_1 `to` e_2	goto
			`cont(`k`)`	`cont(`k`)`	continuation

The expression `cont(`k`)` is a reified control stack, which arises during evaluation.

The statics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \mathtt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \mathtt{letcc}[\tau](x.e) : \tau} \tag{31.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \mathtt{cont}(\tau_1)}{\Gamma \vdash \mathtt{throw}[\tau'](e_1; e_2) : \tau'} \tag{31.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The statics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \mathtt{cont}(k) : \mathtt{cont}(\tau)} \tag{31.2}$$

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 dynamics we extend $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \ \mathsf{exp}}{\mathtt{throw}[\tau](-; e_2) \ \mathsf{frame}} \tag{31.3a}$$

$$\frac{e_1 \ \mathsf{val}}{\mathtt{throw}[\tau](e_1; -) \ \mathsf{frame}} \tag{31.3b}$$

Every reified control stack is a value:

$$\frac{k \ \mathsf{stack}}{\mathtt{cont}(k) \ \mathsf{val}} \tag{31.4}$$

The transition rules for the continuation constructs are as follows:

$$\overline{k \triangleright \mathtt{letcc}[\tau](x.e) \mapsto k \triangleright [\mathtt{cont}(k)/x]e} \tag{31.5a}$$

$$\overline{k \triangleright \mathtt{throw}[\tau](e_1; e_2) \mapsto k; \mathtt{throw}[\tau](-; e_2) \triangleright e_1} \tag{31.5b}$$

$$\frac{e_1 \ \mathsf{val}}{k; \mathtt{throw}[\tau](-; e_2) \triangleleft e_1 \mapsto k; \mathtt{throw}[\tau](e_1; -) \triangleright e_2} \tag{31.5c}$$

$$\overline{k; \mathtt{throw}[\tau](v; -) \triangleleft \mathtt{cont}(k') \mapsto k' \triangleleft v} \tag{31.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of $\mathcal{L}\{\rightarrow\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ given in Chapter 29.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau']\,(-; e_2) : \tau \Rightarrow \tau'} \tag{31.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\mathtt{throw}[\tau']\,(e_1; -) : \mathtt{cont}(\tau) \Rightarrow \tau'} \tag{31.6b}$$

The rest of the definitions remain as in Chapter 29.

Lemma 31.1 (Canonical Forms)**.** *If $e : \mathtt{cont}(\tau)$ and $e\ \mathsf{val}$, then $e = \mathtt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 31.2 (Safety)**.** 1. *If $s\ \mathsf{ok}$ and $s \mapsto s'$, then $s'\ \mathsf{ok}$.*

2. *If $s\ \mathsf{ok}$, then either $s\ \mathsf{final}$ or there exists s' such that $s \mapsto s'$.*

31.3 Coroutines

The distinction between a routine and a subroutine is the distinction between a manager and a worker. The routine calls upon the subroutine to accomplish a piece of work, and the subroutine returns to the routine when its work is done. The relationship is asymmetric in that there is a clear distinction between the *caller*, the main routine, and the *callee*, the subroutine. Often it is useful to consider a symmetric situation in which two routines each call the other to help accomplish a task. Such a pair of routines are called *coroutines*; their relationship to one another is symbiotic rather than parasitic.

The key to implementing a subroutine is for the caller to pass to the callee a continuation representing the return point of the subroutine call. When the subroutine is finished, it calls the continuation passed to it by the calling routine. Since the subroutine is finished at that point, there is no need for the callee to pass a continuation back to the caller. The key to implementing coroutines is to have each routine treat the other as a subroutine of itself. In particular, whenever a coroutine cedes control to its caller, it provides a continuation that the caller may use to cede control back to the callee, in the process providing a continuation for itself. (This raises an interesting question of how the whole process gets started. We'll return to this shortly.)

To see how a pair of coroutines is implemented, let us consider the type of each routine in the pair. A routine is a continuation accepting two arguments, a datum to be passed to that routine when it is resumed, and a

continuation to be resumed when the routine is finished its task. The datum represents the state of the computation, and the continuation is a coroutine that accepts arguments of the same form. Thus, the type of a coroutine must satisfy the type isomorphism

$$\tau\,\texttt{coro} \cong (\tau \times \tau\,\texttt{coro})\,\texttt{cont}.$$

So we may take $\tau\,\texttt{coro}$ to be the recursive type

$$\tau\,\texttt{coro} \triangleq \mu t\,.\,(\tau \times t)\,\texttt{cont}.$$

Up to isomorphism, the type $\tau\,\texttt{coro}$ is the type of continuations that accept a value of type τ, representing the state of the coroutine, and the partner coroutine, a value of the same type.

A coroutine, r, passes control to another coroutine, r', by evaluating the expression $\texttt{resume}(\langle s, r'\rangle)$, where s is the current state of the computation. Doing so creates a new coroutine whose entry point is the return point (calling site) of the application of `resume`. Therefore the type of `resume` is

$$\tau \times \tau\,\texttt{coro} \to \tau \times \tau\,\texttt{coro}.$$

The definition of `resume` is as follows:

$$\lambda\,(\langle s, r'\rangle : \tau \times \tau\,\texttt{coro}.\,\texttt{letcc}\,k\,\texttt{in}\,\texttt{throw}\,\langle s, \texttt{fold}(k)\rangle\,\texttt{to}\,\texttt{unfold}(r'))$$

When applied, `resume` seizes the current continuation, and passes the state, s, and the seized continuation (packaged as a coroutine) to the called coroutine.

But how do we create a system of coroutines in the first place? Since the state is explicitly passed from one routine to the other, a coroutine may be defined as a state transformation function that, when activated with the current state, determines the next state of the computation. A system of coroutines is created by establishing a joint exit point to which the result of the system is thrown, and creating a pair of coroutines that iteratively transform the state and pass control to the partner routine. If either routine wishes to terminate the computation, it does so by throwing a result value to their common exit point. Thus, a coroutine may be specified by a function of type

$$(\rho, \tau)\,\texttt{rout} \triangleq \rho\,\texttt{cont} \to \tau \to \tau,$$

where ρ is the result type and τ is the state type of the system of coroutines.

The set up a system of coroutines we define a function `run` that, given two routines, creates a function of type $\tau \to \rho$ that, when applied to the

initial state, computes a result of type ρ. The computation consists of a cooperating pair of routines that share a common exit point. The definition of `run` begins as follows:

$$\lambda\,\langle r_1, r_2\rangle.\,\lambda\,s_0.\,\texttt{letcc}\,x_0\,\texttt{in}\,\texttt{let}\,r_1'\,\texttt{be}\,r_1(x_0)\,\texttt{in}\,\texttt{let}\,r_2'\,\texttt{be}\,r_2(x_0)\,\texttt{in}\ldots$$

Given two routines, `run` establishes their common exit point, and passes this continuation to both routines. By throwing to this continuation either routine may terminate the computation with a result of type ρ. The body of the `run` function continues as follows:

$$\texttt{rep}(r_2')(\texttt{letcc}\,k\,\texttt{in}\,\texttt{rep}(r_1')(\langle s_0, \texttt{fold}(k)\rangle))$$

The auxiliary function `rep` creates an infinite loop that transforms the state and passes control to the other routine:

$$\lambda\,t.\,\texttt{fix}\,l\,\texttt{is}\,\lambda\,\langle s, r\rangle.\,l(\texttt{resume}(\langle t(s), r\rangle)).$$

The system is initialized by starting routine r_1 with the initial state, and arranging that, when it cedes control to its partner, it starts routine r_2 with the resulting state. At that point the system is bootstrapped: each routine will resume the other on each iteration of the loop.

A good example of coroutining arises whenever we wish to interleave input and output in a computation. We may achieve this using a coroutine between a *producer* routine and a *consumer* routine. The producer emits the next element of the input, if any, and passes control to the consumer with that element removed from the input. The consumer processes the next data item, and returns control to the producer, with the result of processing attached to the output. The input and output are modeled as lists of type $\tau_i\,\texttt{list}$ and $\tau_o\,\texttt{list}$, respectively, which are passed back and forth between the routines.[1] The routines exchange messages according to the following protocol. The message $\texttt{OK}(\langle i, o\rangle)$ is sent from the consumer to producer to acknowledge receipt of the previous message, and to pass back the current state of the input and output channels. The message $\texttt{EMIT}(\langle v, \langle i, o\rangle\rangle)$, where v is a value of type $\tau_i\,\texttt{opt}$, is sent from the producer to the consumer to emit the next value (if any) from the input, and to pass the current state of the input and output channels to the consumer.

This leads to the following implementation of the producer/consumer model. The type τ of the state maintained by the routines is the labelled

[1] In practice the input and output state are implicit, but we prefer to make them explicit for the sake of clarity.

sum type

$$[\texttt{OK} : \tau_i\ \texttt{list} \times \tau_o\ \texttt{list}, \texttt{EMIT} : \tau_i\ \texttt{opt} \times (\tau_i\ \texttt{list} \times \tau_o\ \texttt{list})].$$

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer, P, is defined by the expression

$$\lambda\, x_0.\, \lambda\, msg.\, \texttt{case}\ msg\ \{b_1 \mid b_2 \mid b_3\},$$

where the first branch, b_1, is

$$\texttt{OK} \cdot \langle \texttt{nil}, os \rangle \Rightarrow \texttt{EMIT} \cdot \langle \texttt{null}, \langle \texttt{nil}, os \rangle \rangle$$

and the second branch, b_2, is

$$\texttt{OK} \cdot \langle \texttt{cons}(i; is), os \rangle \Rightarrow \texttt{EMIT} \cdot \langle \texttt{just}(i), \langle is, os \rangle \rangle,$$

and the third branch, b_3, is

$$\texttt{EMIT} \cdot \_ \Rightarrow \texttt{error}.$$

In words, if the input is exhausted, the producer emits the value `null`, along with the current channel state. Otherwise, it emits `just(`i`)`, where i is the first remaining input, and removes that element from the passed channel state. The producer cannot see an `EMIT` message, and signals an error if it should occur.

The consumer, C, is defined by the expression

$$\lambda\, x_0.\, \lambda\, msg.\, \texttt{case}\ msg\ \{b'_1 \mid b'_2 \mid b'_3\},$$

where the first branch, b'_1, is

$$\texttt{EMIT} \cdot \langle \texttt{null}, \langle \_, os \rangle \rangle \Rightarrow \texttt{throw}\ os\ \texttt{to}\ x_0,$$

the second branch, b'_2, is

$$\texttt{EMIT} \cdot \langle \texttt{just}(i), \langle is, os \rangle \rangle \Rightarrow \texttt{OK} \cdot \langle is, \texttt{cons}(f(i); os) \rangle,$$

and the third branch, b'_3, is

$$\texttt{OK} \cdot \_ \Rightarrow \texttt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to x_0 as the ultimate value of the computation. If

it is present, the function f (unspecified here) of type $\tau_i \to \tau_o$ is applied to transform the input to the output, and the result is added to the output channel. If the message `OK` is received, the consumer signals an error, as the producer never produces such a message.

The initial state, s_0, has the form $\texttt{OK} \cdot \langle is, os \rangle$, where *is* and *os* are the initial input and output channel state, respectively. The computation is created by the expression

$$\texttt{run}(\langle P, C \rangle)(s_0),$$

which sets up the coroutines as described earlier.

While it is relatively easy to visualize and implement coroutines involving only two partners, it is more complex, and less useful, to consider a similar pattern of control among $n \geq 2$ participants. In such cases it is more common to structure the interaction as a collection of n routines, each of which is a coroutine of a central *scheduler*. When a routine resumes its partner, it passes control to the scheduler, which determines which routine to execute next, again as a coroutine of itself. When structured as coroutines of a scheduler, the individual routines are called *threads*. A thread *yields* control by resuming its partner, the scheduler, which then determines which thread to execute next as a coroutine of itself. This pattern of control is called *cooperative multi-threading*, since it is based on explicit yields, rather than implicit yields imposed by asynchronous events such as timer interrupts.

31.4 Notes

Continuations are a ubiquitous notion in programming languages. Reynolds's survey [90] provides an excellent account of the history and literature on continuations. The account given here draws on the work of Felleisen [29].

Part XII

Types and Propositions

Chapter 32

Constructive Logic

The correspondence between *propositions* and *types*, and the associated correspondence between *proofs* and *programs*, is the central organizing principle of programming languages. A type specifies a behavior, and a program implements it. Similarly, a proposition poses a problem, and a proof solves it. A statics relates a program to the type it implements, and a dynamics relates a program to its simplification by an execution step. Similarly, a formal logical system relates a proof to the proposition it proves, and proof reduction relates equivalent proofs. The structural rule of substitution underlies the decomposition of a program into separate modules. Similarly, the structural rule of transitivity underlies the decomposition of a theorem into lemmas.

These correspondences are neither accidental nor incidental. The *propositions as types principle*, identifies propositions with types and proofs with programs. According to this principle, a proposition *is* the type of its proofs, and a proof *is* a program of that type. Consequently, every theorem has *computational content*, the its proof viewed as a program, and every program has *mathematical content*, the proof that the program represents.

Can every conceivable form of proposition also be construed as a type? Does every type correspond to a proposition? Must every proof have computational content? Is every program a proof of a theorem? To answer these questions would require a book of its own (and still not settle the matter). From a constructive perspective we may say that type theory enriches logic to incorporate not only types of proofs, but also types for the objects of study. In this sense logic is a particular mode of use of type theory. If we think of type theory as a comprehensive view of mathematics, this implies that, contrary to conventional wisdom, logic is based on math-

ematics, rather than mathematics on logic!

In this chapter we introduce the propositions-as-types correspondence for a particularly simple system of logic, called *propositional contructive logic*. In Chapter 33 we will extend the correspondence to *propositional classical logic*. This will give rise to a computational interpretation of classical proofs that makes essential use of continuations.

32.1 Constructive Semantics

Constructive logic is concerned with two judgements, ϕ prop, stating that ϕ expresses a proposition, and ϕ true, stating that ϕ is a true proposition. What distinguishes constructive from non-constructive logic is that a proposition is not conceived of as merely a truth value, but instead as a *problem statement* whose solution, if it has one, is given by a proof. A proposition is said to be *true* exactly when it has a proof, in keeping with ordinary mathematical practice. *There is no other criterion of truth than the existence of a proof.*

This principle has important, possibly surprising, consequences, the most important of which is that we cannot say, in general, that a proposition is either true or false. If for a proposition to be true means to have a proof of it, what does it mean for a proposition to be false? It means that we have a *refutation* of it, showing that it cannot be proved. That is, a proposition is false if we can show that the assumption that it is true (has a proof) contradicts known facts. 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.

In light of this it should be clear that not every proposition is either true or false. For if ϕ expresses an unsolved problem, such as the famous $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem, then we have neither a proof nor a refutation of it (the mere absence of a proof not being a refutation). Such a problem is *undecided*, precisely because it is unsolved. Since there will always be unsolved problems (there being infinitely many propositions, but only finitely many proofs at a given point in the evolution of our knowledge), we cannot say that every proposition is *decidable*, that is, either true or false.

Having said that, some propositions *are* decidable, and hence may be considered to be either true or false. 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. This argument does not

extend to the real numbers. To get an idea of why not, consider the presentation of a real number by its decimal expansion. At any finite time we will have explored only a finite initial segment of the expansion, which is not enough to determine if it is, say, less than 1. For if we have determined the expansion to be $0.99\ldots9$, we cannot decide at any time, short of infinity, whether or not the number is 1. (This argument is not a proof, because one may wonder whether there is some other representation of real numbers that admits such a decision to be made finitely, but it turns out that this is not the case.)

The constructive attitude is simply to accept the situation as inevitable, and make our peace with that. When faced with a problem we have no choice but to roll up our sleeves and try to prove it or refute it. There is no guarantee of success! Life's hard, but we muddle through somehow.

32.2 Constructive Logic

The judgements ϕ prop and ϕ true of constructive logic are rarely of interest by themselves, but rather in the context of a hypothetical judgement 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 judgement ϕ true.

The structural properties of the hypothetical judgement, when specialized to constructive logic, define what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{32.1a}$$

$$\frac{\Gamma \vdash \phi_1 \text{ true} \quad \Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma \vdash \phi_2 \text{ true}} \tag{32.1b}$$

$$\frac{\Gamma \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{32.1c}$$

$$\frac{\Gamma, \phi_1 \text{ true}, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{32.1d}$$

$$\frac{\Gamma_1, \phi_2 \text{ true}, \phi_1 \text{ true}, \Gamma_2 \vdash \phi \text{ true}}{\Gamma_1, \phi_1 \text{ true}, \phi_2 \text{ true}, \Gamma_2 \vdash \phi \text{ true}} \tag{32.1e}$$

The last two rules are implicit in that we regard Γ as a *set* of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

32.2.1 Rules of Provability

The syntax of propositional logic is given by the following grammar:

Prop	ϕ ::=	`true`	$\top$	truth
		`false`	$\bot$	falsity
		`and`$(\phi_1;\phi_2)$	$\phi_1 \wedge \phi_2$	conjunction
		`or`$(\phi_1;\phi_2)$	$\phi_1 \vee \phi_2$	disjunction
		`imp`$(\phi_1;\phi_2)$	$\phi_1 \supset \phi_2$	implication

The connectives of propositional logic are given meaning by rules that determine (a) what constitutes a "direct" proof of a proposition formed from a given connective, and (b) how to exploit the existence of such a proof in an "indirect" proof of another proposition. These are called the *introduction* and *elimination* rules for the connective. The principle of *conservation of proof* states that these rules are inverse to one another — the elimination rule cannot extract more information (in the form of a proof) than was put into it by the introduction rule, and the introduction rules can be used to reconstruct a proof from the information extracted from it by the elimination rules.

Truth Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it.

$$\frac{}{\Gamma \vdash \top\ \mathsf{true}} \tag{32.2a}$$

(no elimination rule)

(32.2b)

Conjunction Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi_1\ \mathsf{true} \quad \Gamma \vdash \phi_2\ \mathsf{true}}{\Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}} \tag{32.3a}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}}{\Gamma \vdash \phi_1\ \mathsf{true}} \tag{32.3b}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}}{\Gamma \vdash \phi_2\ \mathsf{true}} \tag{32.3c}$$

Implication Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma \vdash \phi_1 \supset \phi_2 \text{ true}} \tag{32.4a}$$

$$\frac{\Gamma \vdash \phi_1 \supset \phi_2 \text{ true} \quad \Gamma \vdash \phi_1 \text{ true}}{\Gamma \vdash \phi_2 \text{ true}} \tag{32.4b}$$

Falsehood Falsehood expresses the trivially false (refutable) proposition.

(no introduction rule)

(32.5a)

$$\frac{\Gamma \vdash \bot \text{ true}}{\Gamma \vdash \phi \text{ true}} \tag{32.5b}$$

Disjunction Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi_1 \text{ true}}{\Gamma \vdash \phi_1 \vee \phi_2 \text{ true}} \tag{32.6a}$$

$$\frac{\Gamma \vdash \phi_2 \text{ true}}{\Gamma \vdash \phi_1 \vee \phi_2 \text{ true}} \tag{32.6b}$$

$$\frac{\Gamma \vdash \phi_1 \vee \phi_2 \text{ true} \quad \Gamma, \phi_1 \text{ true} \vdash \phi \text{ true} \quad \Gamma, \phi_2 \text{ true} \vdash \phi \text{ true}}{\Gamma \vdash \phi \text{ true}} \tag{32.6c}$$

Negation The negation, $\neg\phi$, of a proposition, ϕ, may be defined as the implication $\phi \supset \bot$. This means that $\neg\phi$ true if ϕ true $\vdash \bot$ true, which is to say that the truth of ϕ is *refutable* in that we may derive a proof of falsehood from any purported proof of ϕ. Because constructive truth is identified with the existence of a proof, the implied semantics of negation is rather strong. In particular, a problem, ϕ, is *open* exactly when we can neither affirm nor refute it. This is in contrast to the classical conception of truth, which assigns a fixed truth value to each proposition, so that every proposition is either true or false.

32.2.2 Rules of Proof

The key to the propositions-as-types principle is to make explict the forms of proof. The basic 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. (Sometimes

p is called a "proof term", but we will simply call p a "proof.") 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.

$$\frac{}{\Gamma \vdash \texttt{trueI} : \top} \tag{32.7a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \texttt{andI}(p_1; p_2) : \phi_1 \wedge \phi_2} \tag{32.7b}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash \texttt{andE[l]}(p_1) : \phi_1} \tag{32.7c}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash \texttt{andE[r]}(p_1) : \phi_2} \tag{32.7d}$$

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Gamma \vdash \texttt{impI}[\phi_1](x\,.\,p_2) : \phi_1 \supset \phi_2} \tag{32.7e}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2 \quad \Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \texttt{impE}(p; p_1) : \phi_2} \tag{32.7f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \texttt{falseE}[\phi](p) : \phi} \tag{32.7g}$$

$$\frac{\Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \texttt{orI[l]}[\phi_2](p_1) : \phi_1 \vee \phi_2} \tag{32.7h}$$

$$\frac{\Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \texttt{orI[r]}[\phi_1](p_2) : \phi_1 \vee \phi_2} \tag{32.7i}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \phi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \phi}{\Gamma \vdash \texttt{orE}[\phi_1; \phi_2](p; x\,.\,p_1; y\,.\,p_2) : \phi} \tag{32.7j}$$

32.3 Propositions as Types

Reviewing the rules of proof for constructive logic, we observe a striking correspondence between them and the rules for forming expressions of various types. For example, the introduction rule for conjunction specifies that a proof of a conjunction consists of a pair of proofs, one for each conjunct, and the elimination rule inverts this, allowing us to extract a proof of each conjunct from any proof of a conjunction. There is an obvious analogy with the static semantics of product types, whose introductory form is a pair and whose eliminatory forms are projections.

This correspondence extends to other forms of proposition as well, as summarized by the following chart relating a proposition, ϕ, to a type ϕ^*:

Proposition	*Type*
$\top$	`unit`
$\bot$	`void`
$\phi_1 \wedge \phi_2$	$\phi_1^* \times \phi_2^*$
$\phi_1 \supset \phi_2$	$\phi_1^* \to \phi_2^*$
$\phi_1 \vee \phi_2$	$\phi_1^* + \phi_2^*$

It is obvious that this correspondence is invertible, so that we may associate a proposition with each product, sum, or function type.

Importantly, this correspondence extends to the introductory and eliminatory forms of proofs and programs as well:

Proof	*Program*
`trueI`	$\langle\rangle$
`falseE`$[\phi](p)$	`abort`(p^*)
`andI`$(p_1; p_2)$	$\langle p_1^*, p_2^* \rangle$
`andE[l]`(p)	$p^* \cdot \mathtt{l}$
`andE[r]`(p)	$p^* \cdot \mathtt{r}$
`impI`$[\phi_1](x_1 . p_2)$	$\lambda\,(x_1 : \phi_1^* .\, p_2^*)$
`impE`$(p; p_1)$	$p^*(p_1^*)$
`orI[l]`$[\phi_2](p)$	$\mathtt{l} \cdot p^*$
`orI[r]`$[\phi_1](p)$	$\mathtt{r} \cdot p^*$
`orE`$[\phi_1; \phi_2](p; x_1 . p_1; x_2 . p_2)$	$\mathtt{case}\, p^* \,\{\mathtt{l} \cdot x_1 \Rightarrow p_1^* \mid \mathtt{r} \cdot x_2 \Rightarrow p_2^*\}$

Here again the correspondence is easily seen to be invertible, so that we may regard a program of a product, sum, or function type as a proof of the corresponding proposition.

Theorem 32.1.

1. *If ϕ prop, then ϕ^* type.*
2. *If $\Gamma \vdash p : \phi$, then $\Gamma^* \vdash p^* : \phi^*$.*

The foregoing correspondence between the statics of propositions and proofs on one hand, and types and programs on the other extends also to the dynamics, by applying the inversion principle stating that eliminatory forms are post-inverse to introductory forms. The dynamic correspondence may be expressed by the validity of these definitional equivalences under the static correspondences given above:

$$\begin{aligned}
\texttt{andE[l]}(\texttt{andI}(p;q)) &\equiv p \\
\texttt{andE[r]}(\texttt{andI}(p;q)) &\equiv q \\
\texttt{impE}(\texttt{impI}[\phi](x\,.\,p_2);p_1) &\equiv [p_1/x]p_2 \\
\texttt{orE}[\phi_1;\phi_2](\texttt{orI[l]}[\phi_2](p);x_1\,.\,p_2;x_2\,.\,p_2) &\equiv [p/x_1]p_1 \\
\texttt{orE}[\phi_1;\phi_2](\texttt{orI[r]}[\phi_1](p);x_1\,.\,p_1;x_2\,.\,p_2) &\equiv [p/x_2]p_2
\end{aligned}$$

Observe that these equations are all valid under the static correspondence given above. For example, the first of these equations corresponds to the definitional equivalence $\langle e_1, e_2\rangle \cdot \texttt{l} \equiv e_1$, which is valid for the lazy interpretation of ordered pairs.

The significance of the dynamic correspondence is that it assigns *computational content* to proofs: a proof in constructive propositional logic may be read as a program. Put the other way around, it assigns *logical content* to programs: every expression of product, sum, or function type may be read as a proof of a proposition.

32.4 Notes

The propositions-as-types principle is sometimes erroneously called the *Curry-Howard Isomorphism*. This terminology obscures the essential contributions of Arend Heyting [45], Andrei Kolmogorov [51], Nicolaas de-Bruijn [74], and Per Martin-Löf [75] to the development of this deep correspondence between logic and computation.

Chapter 33

Classical Logic

In constructive logic a proposition is true exactly when it has a *proof*, a derivation of it from axioms and assumptions, and is false exactly when it has a *refutation*, a derivation of a contradiction from the assumption that it is true. Constructive logic is a logic of positive evidence. To affirm or deny a proposition requires a proof, either of the proposition itself, or of a contradiction, under the assumption that it has a proof. We are not always in a position to affirm or deny a proposition. An *open problem* is one for which we have neither a proof nor a refutation—so that, constructively speaking, it is neither true nor false!

In contrast classical logic (the one we learned in school) is a logic of perfect information in which every proposition is either true or false. One may say that classical logic corresponds to "god's view" of the world—there are no open problems, rather all propositions are either true or false. Put another way, to assert that every proposition is either true or false is to *weaken* the notion of truth to encompass all that is *not false*, dually to the constructively (and classically) valid interpretation of falsity as all that is *not true*. The symmetry between truth and falsity is appealing, but there is a price to pay for this: the meanings of the logical connectives are weaker in the classical case than in the constructive.

A prime example is provided by the *law of the excluded middle*, the assertion that $\phi \vee \neg\phi$ true is valid for all propositions ϕ. Constructively, this principle is not universally valid, because it would mean that every proposition either has a proof or a refutation, which is manifestly not the case. Classically, however, the law of the excluded middle is valid, because every proposition is either true or false. The discrepancy between the constructive and classical interpretations can be attributed to the different meanings

given to disjunction and negation by the two logics. In particular the classical truth of a disjunction cannot guarantee the constructive truth of one or the other disjunct. Something other than a constructive proof must be admitted as evidence for a disjunction if the law of the excluded middle is to hold true. And it is precisely for this reason that a classical proof expresses less than does a constructive proof of the same proposition.

Despite this weakness, classical logic admits a computational interpretation similar to, but somewhat less expressive than, that of constructive logic. The dynamics of classical proofs is derived from the complementarity of truth and falsity. A computation is initiated by juxtaposing a proof and a refutation—or, in programming terms, an expression and a *continuation*, or *control stack*. Continuations are essential to the meaning of classical proofs. In particular, the proof of the law of the excluded middle will be seen to equivocate between proving and refuting a proposition, using continuations to avoid getting caught in a contradiction.

33.1 Classical Logic

In constructive logic a connective is defined by giving its introduction and elimination rules. In classical logic a connective is defined by giving its truth and falsity conditions. Its truth rules correspond to introduction, and its falsity rules to elimination. The symmetry between truth and falsity is expressed by the principle of indirect proof. To show that ϕ true it is enough to show that ϕ false entails a contradiction, and, conversely, to show that ϕ false it is enough to show that ϕ true leads to a contradiction. While the second of these is constructively valid, the first is fundamentally classical, expressing the principle of indirect proof.

33.1.1 Provability and Refutability

There are three judgement forms in classical logic:

1. ϕ true, stating that the proposition ϕ is provable;
2. ϕ false, stating that the proposition ϕ is refutable;
3. #, stating that a contradiction has been derived.

We will consider hypothetical judgements of 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 basic judgement forms. The hypotheses are divided into two "zones" for convenience. We let Γ stand for a finite set of "true" hypotheses, and Δ stand for a finite set of "false" hypotheses.

The rules of classical logic are organized around the symmetry between truth and falsity, which is mediated by the contradiction judgement.

The hypothetical judgement is reflexive:

$$\frac{}{\Delta, \phi \text{ false } \Gamma \vdash \phi \text{ false}} \tag{33.1a}$$

$$\frac{}{\Delta\ \Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{33.1b}$$

The remaining rules are stated so that the structural properties of weakening, contraction, and transitivity are admissible.

A contradiction arises when a proposition is judged to be both true and false. A proposition is true if its falsity is absurd, and is false if its truth is absurd.

$$\frac{\Delta\ \Gamma \vdash \phi \text{ false} \quad \Delta\ \Gamma \vdash \phi \text{ true}}{\Delta\ \Gamma \vdash \#} \tag{33.1c}$$

$$\frac{\Delta, \phi \text{ false } \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi \text{ true}} \tag{33.1d}$$

$$\frac{\Delta\ \Gamma, \phi \text{ true} \vdash \#}{\Delta\ \Gamma \vdash \phi \text{ false}} \tag{33.1e}$$

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \top \text{ true}} \tag{33.1f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ true} \quad \Delta\ \Gamma \vdash \phi_2 \text{ true}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ true}} \tag{33.1g}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ false}} \tag{33.1h}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ false}} \tag{33.1i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \bot \text{ false}} \tag{33.1j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{true}} \tag{33.1k}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{true}} \tag{33.1l}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{false} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{false}} \tag{33.1m}$$

Negation inverts the sense of each judgement:

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{false}}{\Delta\ \Gamma \vdash \neg\phi\ \mathsf{true}} \tag{33.1n}$$

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \neg\phi\ \mathsf{false}} \tag{33.1o}$$

An implication is true if its conclusion is true whenever the assumption is true, and is false if its conclusion is false yet its assumption is true.

$$\frac{\Delta\ \Gamma, \phi_1\ \mathsf{true} \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2\ \mathsf{true}} \tag{33.1p}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2\ \mathsf{false}} \tag{33.1q}$$

33.1.2 Proofs and Refutations

The dynamics of classical proofs is most easily explained by introducing a notation for the derivations of each of the judgement forms of classical logic:

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

$$\underbrace{u_1 \div \phi_1, \ldots, u_m \div \phi_m}_{\Delta}\ \underbrace{x_1 : \psi_1, \ldots, x_n : \psi_n}_{\Gamma} \vdash J,$$

in which we have labelled the truth and falsity assumptions with variables.

A contradiction arises whenever a proposition is both true and false:

$$\frac{\Delta\ \Gamma \vdash k \div \phi \quad \Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash k \# p} \tag{33.2a}$$

Truth and falsity are defined symmetrically in terms of contradiction:

$$\frac{\Delta, u \div \phi\ \Gamma \vdash k \# p}{\Delta\ \Gamma \vdash \mathtt{ccr}(u \div \phi.k \# p) : \phi} \tag{33.2b}$$

$$\frac{\Delta\ \Gamma, x : \phi \vdash k \# p}{\Delta\ \Gamma \vdash \mathtt{ccp}(x : \phi.k \# p) \div \phi} \tag{33.2c}$$

Reflexivity corresponds to the use of a variable hypothesis:

$$\frac{}{\Delta, u \div \phi\ \Gamma \vdash u \div \phi} \tag{33.2d}$$

$$\frac{}{\Delta\ \Gamma, x : \phi \vdash x : \phi} \tag{33.2e}$$

The other structure properties are admissible.

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \langle\rangle : \top} \tag{33.2f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \langle p_1, p_2\rangle : \phi_1 \wedge \phi_2} \tag{33.2g}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1}{\Delta\ \Gamma \vdash \mathtt{fst};k_1 \div \phi_1 \wedge \phi_2} \tag{33.2h}$$

$$\frac{\Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{snd};k_2 \div \phi_1 \wedge \phi_2} \tag{33.2i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \mathtt{abort} \div \bot} \tag{33.2j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1}{\Delta\ \Gamma \vdash \mathtt{inl}(p_1) : \phi_1 \vee \phi_2} \tag{33.2k}$$

$$\frac{\Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \mathtt{inr}(p_2) : \phi_1 \vee \phi_2} \tag{33.2l}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{case}(k_1; k_2) \div \phi_1 \vee \phi_2} \tag{33.2m}$$

Negation inverts the sense of each judgement:

$$\frac{\Delta\ \Gamma \vdash k \div \phi}{\Delta\ \Gamma \vdash \mathtt{not}(k) : \neg\phi} \tag{33.2n}$$

$$\frac{\Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash \mathtt{not}(p) \div \neg\phi} \tag{33.2o}$$

An implication is true if its conclusion is true whenever the assumption is true, and is false if its conclusion if false yet its assumption is true.

$$\frac{\Delta\ \Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \lambda\,(x{:}\phi_1.\, p_2) : \phi_1 \supset \phi_2} \tag{33.2p}$$

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{app}(p_1); k_2 \div \phi_1 \supset \phi_2} \tag{33.2q}$$

33.2 Deriving Elimination Forms

The price of achieving a symmetry between truth and falsity in classical logic is that we must very often rely on the principle of indirect proof: to show that a proposition is true, we often must derive a contradicton from the assumption of its falsity. For example, a proof of

$$(\phi \wedge (\psi \wedge \theta)) \supset (\theta \wedge \phi)$$

in classical logic has the form

$$\lambda\,(w{:}\phi \wedge (\psi \wedge \theta).\, \mathtt{ccr}(u \div \theta \wedge \phi.\, k \mathbin{\#} w)),$$

where k is the refutation

$$\mathtt{fst}; \mathtt{ccp}(x : \phi.\, \mathtt{snd}; \mathtt{ccp}(y : \psi \wedge \theta.\, \mathtt{snd}; \mathtt{ccp}(z : \theta.\, u \mathbin{\#} \langle z, x\rangle) \mathbin{\#} y) \mathbin{\#} w).$$

And yet in constructive logic this proposition has a direct proof that avoids the circumlocutions of proof by contradiction:

$$\lambda\,(w{:}\phi \wedge (\psi \wedge \theta).\, \mathtt{andI}(\mathtt{andE[r]}(\mathtt{andE[r]}(w)); \mathtt{andE[l]}(w))).$$

But this proof cannot be expressed (as is) in classical logic, because classical logic lacks the elimination forms of constructive logic.

However, we may package the use of indirect proof into a slightly more palatable form by deriving the elimination rules of constructive logic. For example, the rule

$$\frac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}$$

is derivable in classical logic:

$$\frac{\dfrac{\dfrac{}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi\ \mathsf{false}}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{false}} \quad \dfrac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}}{\dfrac{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}}$$

The other elimination forms are derivable in a similar manner, in each case relying on indirect proof to construct a proof of the truth of a proposition from a derivation of a contradiction from the assumption of its falsity.

The derivations of the elimination forms of constructive logic are most easily exhibited using proof and refutation expressions, as follows:

$$\begin{aligned}
\mathtt{falseE}[\phi](p) &= \mathtt{ccr}(u \div \phi.\mathtt{abort} \mathbin{\#} p)\\
\mathtt{andE}[\mathtt{l}](p) &= \mathtt{ccr}(u \div \phi.\mathtt{fst}; u \mathbin{\#} p)\\
\mathtt{andE}[\mathtt{r}](p) &= \mathtt{ccr}(u \div \psi.\mathtt{snd}; u \mathbin{\#} p)\\
\mathtt{impE}(p_1; p_2) &= \mathtt{ccr}(u \div \psi.\mathtt{app}(p_2); u \mathbin{\#} p_1)\\
\mathtt{orE}[\phi;\psi](p_1; x.p_2; y.p) &= \mathtt{ccr}(u \div \gamma.\mathtt{case}(\mathtt{ccp}(x : \phi.u \mathbin{\#} p_2); \mathtt{ccp}(y : \psi.u \mathbin{\#} p)) \mathbin{\#} p_1)
\end{aligned}$$

It is straightforward to check that the expected elimination rules hold. For example, the rule

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi \supset \psi \quad \Delta\ \Gamma \vdash p_2 : \phi}{\Delta\ \Gamma \vdash \mathtt{impE}(p_1; p_2) : \psi} \tag{33.3}$$

is derivable using the definition of $\mathtt{impE}(p_1; p_2)$ given above. By suppressing proof terms, we may derive the corresponding provability rule

$$\frac{\Delta\ \Gamma \vdash \phi \supset \psi\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \psi\ \mathsf{true}}\ . \tag{33.4}$$

33.3 Proof Dynamics

The dynamics of classical logic arises from the simplification of the contradiction between a proof and a refutation of a proposition. To make this explicit we will define a transition system whose states are contradictions $k \# p$ consisting of a proof, p, and a refutation, k, of the same proposition. The steps of the computation consist of simplifications of the contradictory state based on the form of p and k.

The truth and falsity rules for the connectives play off one another in a pleasing manner:

$$\mathtt{fst};k \# \langle p_1, p_2\rangle \mapsto k \# p_1 \tag{33.5a}$$
$$\mathtt{snd};k \# \langle p_1, p_2\rangle \mapsto k \# p_2 \tag{33.5b}$$
$$\mathtt{case}(k_1;k_2) \# \mathtt{inl}(p_1) \mapsto k_1 \# p_1 \tag{33.5c}$$
$$\mathtt{case}(k_1;k_2) \# \mathtt{inr}(p_2) \mapsto k_2 \# p_2 \tag{33.5d}$$
$$\mathtt{not}(p) \# \mathtt{not}(k) \mapsto k \# p \tag{33.5e}$$
$$\mathtt{app}(p_1);k \# \lambda\,(x{:}\phi.\, p_2) \mapsto k \# [p_1/x]p_2 \tag{33.5f}$$

The rules of indirect proof give rise to the following transitions:

$$\mathtt{ccp}(x : \phi.k_1 \# p_1) \# p_2 \mapsto [p_2/x]k_1 \# [p_2/x]p_1 \tag{33.5g}$$
$$k_1 \# \mathtt{ccr}(u \div \phi.k_2 \# p_2) \mapsto [k_1/u]k_2 \# [k_1/u]p_2 \tag{33.5h}$$

The first of these defines the behavior of the refutation of ϕ that proceeds by contradicting the assumption that ϕ is true. This refutation is activated by presenting it with a proof of ϕ, which is then substituted for the assumption in the new state. Thus, "ccp" stands for "call with current proof." The second transition defines the behavior of the proof of ϕ that proceeds by contradicting the assumption that ϕ is false. This proof is activated by presenting it with a refutation of ϕ, which is then substituted for the assumption in the new state. Thus, "ccr" stands for "call with current refutation."

Rules (33.5g) to (33.5h) overlap in that there are two possible transitions for a state of the form

$$\mathtt{ccp}(x : \phi.k_1 \# p_1) \# \mathtt{ccr}(u \div \phi.k_2 \# p_2),$$

one to the state $[p/x]k_1 \# [p/x]p_1$, where p is $\mathtt{ccr}(u \div \phi.k_2 \# p_2)$, and one to the state $[k/u]k_2 \# [k/u]p_2$, where k is $\mathtt{ccp}(x : \phi.k_1 \# p_1)$. The dynamics of classical logic is therefore *non-deterministic*. To avoid this one may impose a priority ordering among the two cases, preferring one transition

over the other when there is a choice. Preferring the first corresponds to a "lazy" dynamics for proofs, because we pass the unevaluated proof, p, to the refutation on the left, which is thereby activated. Preferring the second corresponds to an "eager" dynamics for proofs, in which we pass the unevaluated refutation, k, to the proof, which is thereby activated.

Theorem 33.1 (Preservation)**.** *If $k \div \phi$, $p : \phi$, and $k \# p \mapsto k' \# p'$, then there exists ϕ' such that $k' \div \phi'$ and $p' : \phi'$.*

Proof. By rule induction on the dynamics of classical logic. □

Theorem 33.2 (Progress)**.** *If $k \div \phi$ and $p : \phi$, then either $k \# p$ final or $k \# p \mapsto k' \# p'$.*

Proof. By rule induction on the statics of classical logic. □

To initiate computation we postulate that `halt` is a refutation of any proposition. The initial and final states of computational are defined as follows:

$$\frac{}{\texttt{halt} \mathbin{\#} p \ \mathsf{initial}} \tag{33.6a}$$

$$\frac{p \ \mathsf{canonical}}{\texttt{halt} \mathbin{\#} p \ \mathsf{final}} \tag{33.6b}$$

The judgement p canonical states that p is a canonical proof, which is defined to be any proof other than an indirect proof.

33.4 Law of the Excluded Middle

The law of the excluded middle is derivable in classical logic:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vee \neg\phi \ \mathsf{false}, \phi \ \mathsf{true} \vdash \phi \ \mathsf{true}}}{\phi \vee \neg\phi \ \mathsf{false}, \phi \ \mathsf{true} \vdash \phi \vee \neg\phi \ \mathsf{true}} \quad \dfrac{}{\phi \vee \neg\phi \ \mathsf{false}, \phi \ \mathsf{true} \vdash \phi \vee \neg\phi \ \mathsf{false}}}{\phi \vee \neg\phi \ \mathsf{false}, \phi \ \mathsf{true} \vdash \#}}{\phi \vee \neg\phi \ \mathsf{false} \vdash \phi \ \mathsf{false}}}{\phi \vee \neg\phi \ \mathsf{false} \vdash \neg\phi \ \mathsf{true}}}{\phi \vee \neg\phi \ \mathsf{false} \vdash \phi \vee \neg\phi \ \mathsf{true}} \qquad \dfrac{}{\phi \vee \neg\phi \ \mathsf{false} \vdash \phi \vee \neg\phi \ \mathsf{false}}}{\phi \vee \neg\phi \ \mathsf{false} \vdash \#}}{\phi \vee \neg\phi \ \mathsf{true}}$$

When written out using explicit proofs and refutations, we obtain the proof term $p_0 : \phi \vee \neg\phi$:

$$\mathtt{ccr}(u \div \phi \vee \neg\phi\,.\,u \,\#\, \mathtt{inr}(\mathtt{not}(\mathtt{ccp}(x : \phi\,.\,u \,\#\, \mathtt{inl}(x))))).$$

To understand the computational meaning of this proof, let us juxtapose it with a refutation, $k \div \phi \vee \neg\phi$, and simplify it using the dynamics given in Section 33.3 on page 316. The first step is the transition

$$k \,\#\, \mathtt{ccr}(u \div \phi \vee \neg\phi\,.\,u \,\#\, \mathtt{inr}(\mathtt{not}(\mathtt{ccp}(x : \phi\,.\,u \,\#\, \mathtt{inl}(x)))))$$
$$\mapsto$$
$$k \,\#\, \mathtt{inr}(\mathtt{not}(\mathtt{ccp}(x : \phi\,.\,k \,\#\, \mathtt{inl}(x)))),$$

wherein we have replicated k so that it occurs in two places in the result state. By virtue of its type the refutation k must have the form $\mathtt{case}(k_1; k_2)$, where $k_1 \div \phi$ and $k_2 \div \neg\phi$. Continuing the reduction, we obtain:

$$\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inr}(\mathtt{not}(\mathtt{ccp}(x : \phi\,.\,\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(x))))$$
$$\mapsto$$
$$k_2 \,\#\, \mathtt{not}(\mathtt{ccp}(x : \phi\,.\,\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(x))).$$

By virtue of its type k_2 must have the form $\mathtt{not}(p_2)$, where $p_2 : \phi$, and hence the transition proceeds as follows:

$$\mathtt{not}(p_2) \,\#\, \mathtt{not}(\mathtt{ccp}(x : \phi\,.\,\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(x)))$$
$$\mapsto$$
$$\mathtt{ccp}(x : \phi\,.\,\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(x)) \,\#\, p_2.$$

Observe that p_2 is a valid proof of ϕ! Proceeding, we obtain

$$\mathtt{ccp}(x : \phi\,.\,\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(x)) \,\#\, p_2$$
$$\mapsto$$
$$\mathtt{case}(k_1; k_2) \,\#\, \mathtt{inl}(p_2)$$
$$\mapsto$$
$$k_1 \,\#\, p_2$$

The first of these two steps is the crux of the matter: the refutation, $k = \mathtt{case}(k_1; k_2)$, which was replicated at the outset of the derivation, is re-used, but with a different argument. At the first use, the refutation, k, which is provided by the context of use of the law of the excluded middle, is presented with a proof $\mathtt{inr}(p_1)$ of $\phi \vee \neg\phi$. That is, the proof behaves as though

the right disjunct of the law is true, which is to say that ϕ is false. If the context is such that it inspects this proof, it can only be by providing the proof, p_2, of ϕ that refutes the claim that ϕ is false. Should this occur, the proof of the law of the excluded middle *backtracks* the context, providing instead the proof $\texttt{inl}(p_2)$ to k, which then passes p_2 to k_1 without further incident. The proof of the law of the excluded middle baldly asserts $\neg\phi$ $\texttt{true}$, regardless of the form of ϕ. Then, if caught in its lie by the context providing a proof of ϕ, *changes its mind* and asserts to the *original* context, k, after all! No further reversion is possible, because the context has itself provided a proof, p_2, of ϕ.

The law of the excluded middle illustrates that classical proofs are to be thought of as *interactions* between proofs and refutations, which is to say interactions between a proof and the context in which it is used. In programming terms this corresponds to an abstract machine with an explicit control stack, or continuation, representing the context of evaluation of an expression. That expression may access the context (stack, continuation) to effect backtracking as necessary to maintain the perfect symmetry between truth and falsity. The penalty is that a closed proof of a disjunction no longer need reveal which disjunct it proves, for as we have just seen, it may, on further inspection, change its mind!

33.5 Notes

The computational interpretation of classical logic was first explored by Griffin [38] and Murthy [72]. The account given here was influenced by Wadler's formulation [103] of Gentzen's sequent calculus [31], transposed by Aleks Nanevski to natural deduction using multiple judgement forms.

Part XIII

Symbols

Chapter 34

Symbols

A *symbol* is an atomic datum with no internal structure. Whereas variables are given meaning by substitution, symbols are given meaning by a collection of primitives associated with it. In subsequent chapters we shall consider several interpretations of symbols, giving rise to concepts such as fluid binding, dynamic classification, assignable variables, and communication channels.

A "new" symbol, a, with associated type, ρ, is introduced within a scope, e, by the declaration $\nu\, a\!:\!\rho\, \mathtt{in}\, e$. The meaning of the type ρ varies according to the interpretation of symbols under consideration. It is important to emphasize, however, that a symbol is not a form of expression, and hence is not to be considered an element of its associated type. The expression, e, in the symbol declaration $\nu\, a\!:\!\rho\, \mathtt{in}\, e$, is the *scope* of the symbol. Since bound identifiers may always be renamed within their scope, the declared symbol a may be regarded as "new" in that it is guaranteed to be distinct from all other symbols in scope at the point of the declaration.

The dynamics of symbol declaration defines the *extent*, or range of significance, of the declared symbol. Under a *scoped*, or *stack-like*, dynamics, the extent of a symbol is just its scope. Once the scope of the declaration has been evaluated, the symbol may be deallocated—the statics will ensure that the result cannot depend on that symbol. Under a *scope-free*, or *heap-like*, dynamics the extent of a symbol is unlimited. A symbol may escape the scope of its declaration, which means that it cannot be deallocated once the scope has been evaluated.

34.1 Symbol Declaration

The ability to declare a new symbol is shared by all applications of symbols in subsequent chapters. The syntax for symbol declaration is given by the following grammar:

$$\mathsf{Exp}\quad e \quad ::= \quad \mathtt{new}[\tau](a.e) \quad \nu\, a{:}\tau\, \mathtt{in}\, e \quad \text{generation}$$

The statics of symbol declaration makes use of a *signature*, or *symbol context*, that associates a type to each of a finite set of symbols. We use the letter Σ to range over signatures, which are finite sets of pairs $a : \tau$, where a is a symbol and τ is a type. The typing judgement $\Gamma \vdash_\Sigma e : \tau$ is parameterized by a signature, Σ, associating types to symbols.

The statics of symbol declaration is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma,a:\rho} e : \tau \quad \tau\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{new}[\rho](a.e) : \tau} \tag{34.1}$$

Informally, the requirement ρ mobile ensures that the returned value is meaningful outside of the body of the declaration. If the scope of a is confined to the body, then the type ρ must be restricted so that a value of this type cannot depend on a. If the scope of a is not restricted to the body, then no restrictions on ρ need be imposed: all types are mobile.

34.1.1 Scoped Dynamics

The scoped dynamics of symbol declaration is given by a transition judgement of the form $e \underset{\Sigma}{\mapsto} e'$ indexed by a signature, Σ, specifying the active symbols of the transition. Either e or e' may involve the symbols declared in Σ, but no others.

$$\frac{e \underset{\Sigma,a:\rho}{\longmapsto} e'}{\mathtt{new}[\rho](a.e) \underset{\Sigma}{\mapsto} \mathtt{new}[\rho](a.e')} \tag{34.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{new}[\rho](a.e) \underset{\Sigma}{\mapsto} e} \tag{34.2b}$$

Rule (34.2a) specifies that evaluation takes place within the scope of the declaration of a symbol. Rule (34.2b) specifies that the declared symbol may be deallocated once its scope has been evaluated, provided that the

value of the body is a value relative to the active symbols surrounding the declaration.

The definition of the judgement τ mobile must be chosen to ensure that the following *mobility condition* is satisfied:

If τ *mobile,* $\vdash_{\Sigma,a:\rho} e : \tau$*, and* e $\mathsf{val}_{\Sigma,a:\rho}$*, then* $\vdash_{\Sigma} e : \tau$ *and* e val_{Σ}.

For example, in the presence of symbolic references (see Section 34.2 on the next page below), a function type cannot be deemed mobile, since a function may contain a reference to a local symbol. The type `nat` may only be deemed mobile if the successor is evaluated eagerly, for otherwise a symbolic reference may occur within a value of this type, invalidating the condition.

Theorem 34.1 (Preservation)**.** *If* $\vdash_{\Sigma} e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$*, then* $\vdash_{\Sigma} e' : \tau$.

Proof. By induction on the dynamics of symbol declaration. Rule (34.2a) follows directly by induction, applying Rule (34.1). Rule (34.2b) follows directly from the condition on mobility. □

Theorem 34.2 (Progress)**.** *If* $\vdash_{\Sigma} e : \tau$*, then either* $e \underset{\Sigma}{\mapsto} e'$*, or* e val_{Σ}.

Proof. There is only one rule to consider, Rule (34.1). By induction we have either $e \underset{\Sigma,a:\rho}{\longmapsto} e'$, in which case Rule (34.2a) applies, or e $\mathsf{val}_{\Sigma,a:\rho}$, in which case by the mobility condition we have e val_{Σ}, and hence Rule (34.2b) applies. □

34.1.2 Scope-Free Dynamics

The scope-free dynamics of symbols is defined by a transition system between states of the form $\nu\,\Sigma\,\{\,e\,\}$, where Σ is a signature and e is an expression over this signature. The judgement $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$ states that evaluation of e relative to symbols Σ results in the expression e' in the extension Σ' of Σ.

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{new[}\rho\mathtt{]}\,(a\,.\,e)\,\} \mapsto \nu\,\Sigma, a : \rho\,\{\,e\,\}} \tag{34.3}$$

Rule (34.3) specifies that symbol generation enriches the signature with the newly introduced symbol by extending the signature for all future transitions.

All other rules of the dynamics must be changed accordingly to account for the allocated symbols. For example, the dynamics of function application cannot simply be inherited from Chapter 12, but must be reformulated as follows:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\}\mapsto\nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,e_1(e_2)\,\}\mapsto\nu\,\Sigma'\,\{\,e_1'(e_2)\,\}} \tag{34.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\lambda\,(x:\tau.\,e)\,(e_2)\,\}\mapsto\nu\,\Sigma\,\{\,[e_2/x]e\,\}} \tag{34.4b}$$

These rules shuffle around the signature so as to account for symbol declarations within the constituent expressions of the application. Similar rules must be given for all other constructs of the language.

Theorem 34.3 (Preservation)**.** *If* $\nu\,\Sigma\,\{\,e\,\}\mapsto\nu\,\Sigma'\,\{\,e'\,\}$ *and* $\vdash_\Sigma e:\tau$*, then* $\Sigma'\supseteq\Sigma$ *and* $\vdash_{\Sigma'} e':\tau$*.*

Proof. There is only one rule to consider, Rule (34.3), which is easily handled by inversion of Rule (34.1). □

Theorem 34.4 (Progress)**.** *If* $\vdash_\Sigma e:\tau$*, then either* e val_Σ *or* $\nu\,\Sigma\,\{\,e\,\}\mapsto\nu\,\Sigma'\,\{\,e'\,\}$ *for some* Σ' *and* e'*.*

Proof. Immediate, by Rule (34.3). □

When other types are considered, the proofs of progress and preservation must be re-done to account for the allocated symbols. In particular, a canonical forms lemma for a type must be stated to account for the active symbols. In Chapters 43 and 44 we will introduce methods for dynamicsstreamlining such proofs in the presence of symbols.

34.2 Symbolic References

Symbols are not themselves values, but they may be used to form values. One useful example is provided by the type $\tau\,\mathtt{sym}$ of *symbolic references*. A value of this type has the form $\mathtt{\&}\,a$, where a is a symbol in the signature. To compute with a reference we may branch according to whether it is a reference to a specified symbol or not. The syntax symbolic references is given by the following chart:

Typ	τ	::=	$\mathtt{sym}(\tau)$	$\tau\,\mathtt{sym}$	symbols
Exp	e		$\mathtt{sym}[a]$	$\mathtt{\&}\,a$	reference
			$\mathtt{is}[a]\,[t.\,\tau]\,(e;e_1;e_2)$	$\mathtt{if}\,e\,\mathtt{is}\,a\,\mathtt{then}\,e_1\,\mathtt{ow}\,be_2$	comparison

The expression $\texttt{sym}[a]$ is a reference to the symbol a, a value of type $\texttt{sym}(\tau)$. The expression $\texttt{is}[a][t.\tau](e; e_1; e_2)$ compares the value of e, which must be a reference to some symbol b, with the given symbol, a. If b is a, the expression evaluates to e_1, and otherwise to e_2.

34.2.1 Statics

The typing rules for symbolic references are as follows:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\rho} \texttt{sym}[a] : \texttt{sym}(\rho)} \tag{34.5a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\rho} e : \texttt{sym}(\rho') \quad \Gamma \vdash_{\Sigma,a:\rho} e_1 : [\rho'/t]\tau \quad \Gamma \vdash_{\Sigma,a:\rho} e_2 : [\rho/t]\tau}{\Gamma \vdash_{\Sigma,a:\rho} \texttt{is}[a][t.\tau](e; e_1; e_2) : [\rho'/t]\tau} \tag{34.5b}$$

Rule (34.5a) is the introduction rule for the type $\texttt{sym}(\rho)$. It states that if a is a symbol with associated type ρ, then $\texttt{sym}[a]$ is an expression of type $\texttt{sym}(\rho)$. Rule (34.5b) is the elimination rule for the type $\texttt{sym}(\rho)$. The type associated to the given symbol, a, is not required to be the same as the type of the symbol referred to by the expression e. If e evaluates to a reference to a, then these types will coincide, but if it refers to another symbol, $b \neq a$, then these types may well differ.

With this in mind, let us examine carefully Rule (34.5b). *A priori* there is a discrepancy between the type, ρ, of a and the type, ρ', of the symbol referred to by e. This discrepancy is mediated by the type operator $t.\tau$.[1] Regardless of the outcome of the comparison, the overall type of the expresion is $[\rho'/t]\tau$. To ensure safety, we must ensure that this is a valid type for the result, regardless of whether the comparison succeeds or fails. If e evaluates to the symbol a, then we "learn" that the types ρ' and ρ coincide, since the specified and referenced symbol coincide. This is reflected by the type $[\rho/t]\tau$ for e_1. If e evaluates to some other symbol, $a' \neq a$, then the comparison evaluates to e_2, which is required to have type $[\rho'/t]\tau$; no further information about the type of the symbol is acquired in this branch.

34.2.2 Dynamics

The (scoped) dynamics of symbolic references is given by the following rules:

$$\frac{}{\texttt{sym}[a]\ \mathsf{val}_{\Sigma,a:\rho}} \tag{34.6a}$$

[1] See Chapter 16 for a discussion of type operators.

$$\frac{}{\mathtt{is}[a]\,[t.\tau]\,(\mathtt{sym}[a];e_1;e_2) \underset{\Sigma,a:\rho}{\longmapsto} e_1} \tag{34.6b}$$

$$\frac{(a \neq a')}{\mathtt{is}[a]\,[t.\tau]\,(\mathtt{sym}[a'];e_1;e_2) \underset{\Sigma,a:\rho,a':\rho'}{\longmapsto} e_2} \tag{34.6c}$$

$$\frac{e \underset{\Sigma,a:\rho}{\longmapsto} e'}{\mathtt{is}[a]\,[t.\tau]\,(e;e_1;e_2) \underset{\Sigma,a:\rho}{\longmapsto} \mathtt{is}[a]\,[t.\tau]\,(e';e_1;e_2)} \tag{34.6d}$$

Rules (34.6b) and (34.6c) specify that $\mathtt{is}[a]\,[t.\tau]\,(e;e_1;e_2)$ branches according to whether the value of e is a reference to the symbol, a, or not.

34.2.3 Safety

To ensure that the mobility condition is satisfied, it is important that symbolic reference types *not* be deemed mobile.

Theorem 34.5 (Preservation)**.** *If $\vdash_\Sigma e : \tau$ and $e \underset{\Sigma}{\mapsto} e'$, then $\vdash_\Sigma e' : \tau$.*

Proof. By rule induction on Rules (34.6). The most interesting case is Rule (34.6b). When the comparison is positive, the types ρ and ρ' must be the same, since each symbol has at most one associated type. Therefore, e_1, which has type $[\rho'/t]\tau$, also has type $[\rho/t]\tau$, as required. □

Lemma 34.6 (Canonical Forms)**.** *If $\vdash_\Sigma e : \mathtt{sym}(\rho)$ and e val_Σ, then $e = \mathtt{sym}[a]$ for some a such that $\Sigma = \Sigma', a : \rho$.*

Proof. By rule induction on Rules (34.5), taking account of the definition of values. □

Theorem 34.7 (Progress)**.** *Suppose that $\vdash_\Sigma e : \tau$. Then either e val_Σ, or there exists e' such that $e \underset{\Sigma}{\mapsto} e'$.*

Proof. By rule induction on Rules (34.5). For example, consider Rule (34.5b), in which we have that $\mathtt{is}[a]\,[t.\tau]\,(e;e_1;e_2)$ has some type τ and that $e : \mathtt{sym}(\rho)$ for some ρ. By induction either Rule (34.6d) applies, or else we have that e val_Σ, in which case we are assured by Lemma 34.6 that e is $\mathtt{sym}[a]$ for some symbol b of type ρ declared in Σ. But then progress is assured by Rules (34.6b) and (34.6c), since equality of symbols is decidable (either a is b or it is not). □

34.3 Notes

The concept of a symbol in a programming language was considered by McCarthy in the original formulation of Lisp [63]. Unfortunately, symbols were, and often continue to be, confused with variables, leading to numerous pathologies of language design. While frequently tied to dynamically typed (that is, unityped) languages, there is no impediment to integrating symbols into a statically typed language. The formulation given here was influenced by the work of Pitts and Stark [81] on "names" in statically typed languages.

Chapter 35

Fluid Binding

In this chapter we return to the concept of dynamic scoping of variables that was criticized in Chapter 10. There it was observed that dynamic scoping is problematic for at least two reasons:

- Bound variables may not always be renamed in an expression without changing its meaning.
- Since the scopes of variables are resolved dynamically, type safety is compromised.

These violations of the expected behavior of variables is intolerable, since it is at variance with long-standing mathematical practice and because it compromises the modularity of programs.

It is possible, however, to recover a type-safe analogue of dynamic scoping by divorcing it from the fundamental concept of a *variable*, and instead introducing a new mechanism, called *fluid*, or *dynamic*, *binding* of *symbols*. The main idea is to associate with each symbol a binding to a value of its associated type, and to allow these bindings to be updated within a specified scope. This provides the flexibility of dynamic scoping without interfering with the behavior of variables.

35.1 Statics

The language $\mathcal{L}\{\texttt{fluid}\}$ extends the language $\mathcal{L}\{\texttt{sym}\}$ defined in Chapter 34 with the following additional constructs:

Exp	e	$::=$	$\texttt{put}[a](e_1;e_2)$	$\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2$	binding
			$\texttt{get}[a]$	$\texttt{get}\,a$	retrieval

As in Chapter 34, we use a to stand for some unspecified *symbol*. The expression `get[`a`]` evaluates to the value of the current binding of a, if it has one, and is stuck otherwise. The expression `put[`a`](`e_1`;`e_2`)` 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 symbol a is not bound by the `put` expression, but is instead a parameter of it.

The statics of $\mathcal{L}\{\texttt{fluid}\}$ is defined by judgements of the form

$$\Gamma \vdash_\Sigma e : \tau,$$

where Σ is a finite set $a_1 : \tau_1, \ldots, a_k : \tau_k$ of declarations of the pairwise distinct symbols $a_1, \ldots, a_k$, and Γ is, as usual, a finite set $x_1 : \tau_1, \ldots, x_n : \tau_n$ of declarations of the pairwise distinct variables $x_1, \ldots, x_n$.

The statics of $\mathcal{L}\{\texttt{fluid}\}$ extends that of $\mathcal{L}\{\texttt{sym}\}$ (see Chapter 34) with the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{get[}a\texttt{]} : \tau} \tag{35.1a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau_1} e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma,a:\tau_1} e_2 : \tau_2}{\Gamma \vdash_{\Sigma,a:\tau_1} \texttt{put[}a\texttt{]}(e_1;e_2) : \tau_2} \tag{35.1b}$$

Rule (35.1b) specifies that the symbol a is a parameter of the expression that must be declared in Σ.

35.2 Dynamics

We assume a stack-like dynamics for symbols, as described in Chapter 34. The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ maintains an association of values to symbols that changes in a stack-like manner during execution. We define a family of transition judgements of the form $e \xmapsto[\Sigma]{\mu} e'$, where Σ is as in the statics, and μ is a finite function mapping some subset of the symbols declared in Σ to values of appropriate type. If μ is defined for some symbol a, then it has the form $\mu' \otimes \langle a : e \rangle$ for some μ' and value e. If, on the other hand, μ is undefined for some symbol a, we may regard it as having the form $\mu' \otimes \langle a : \bullet \rangle$. 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 dynamics of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following rules:

$$\frac{e \ \mathsf{val}_{\Sigma,a:\tau}}{\texttt{get[}a\texttt{]} \xmapsto[\Sigma,a:\tau]{\mu \otimes \langle a : e \rangle} e} \tag{35.2a}$$

$$\frac{e_1 \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_1'}{\mathtt{put}\,[a]\,(e_1; e_2) \overset{\mu}{\underset{\Sigma}{\longmapsto}} \mathtt{put}\,[a]\,(e_1'; e_2)} \tag{35.2b}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma,a:\tau} \quad e_2 \overset{\mu\otimes\langle a:e_1\rangle}{\underset{\Sigma,a:\tau}{\longmapsto}} e_2'}{\mathtt{put}\,[a]\,(e_1; e_2) \overset{\mu\otimes\langle a:\_\rangle}{\underset{\Sigma,a:\tau}{\longmapsto}} \mathtt{put}\,[a]\,(e_1; e_2')} \tag{35.2c}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma,a:\tau} \quad e_2\ \mathsf{val}_{\Sigma,a:\tau}}{\mathtt{put}\,[a]\,(e_1; e_2) \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_2} \tag{35.2d}$$

Rule (35.2a) specifies that $\mathtt{get}\,[a]$ evaluates to the current binding of a, if any. Rule (35.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (35.2c) 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 (35.2d) eliminates the fluid binding for a once evaluation of the extent of the binding has completed.

According to the dynamics defined by Rules (35.2), there is no transition of the form $\mathtt{get}\,[a] \overset{\mu}{\underset{\Sigma}{\longmapsto}} e$ if $\mu(a) = \bullet$. The judgement $e\ \mathsf{unbound}_\Sigma$ states that execution of e will lead to such a "stuck" state, and is inductively defined by the following rules:

$$\frac{\mu(a) = \bullet}{\mathtt{get}\,[a]\ \mathsf{unbound}_\mu} \tag{35.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\mu}{\mathtt{put}\,[a]\,(e_1; e_2)\ \mathsf{unbound}_\mu} \tag{35.3b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{unbound}_\mu}{\mathtt{put}\,[a]\,(e_1; e_2)\ \mathsf{unbound}_\mu} \tag{35.3c}$$

In a larger language it would also be necessary to include error propagation rules of the sort discussed in Chapter 8.

35.3 Type Safety

Define the auxiliary judgement $\mu : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{35.4a}$$

$$\frac{\vdash_\Sigma e : \tau \quad \mu : \Sigma}{\mu \otimes \langle a : e \rangle : \Sigma, a : \tau} \tag{35.4b}$$

$$\frac{\mu : \Sigma}{\mu \otimes \langle a : \bullet \rangle : \Sigma, a : \tau} \tag{35.4c}$$

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 (equivalently, bound to a "black hole").

Theorem 35.1 (Preservation). *If $e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'$, where $\mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*

Proof. By rule induction on Rules (35.2). Rule (35.2a) is handled by the definition of $\mu : \Sigma$. Rule (35.2b) follows immediately by induction. Rule (35.2d) is handled by inversion of Rules (35.1). Finally, Rule (35.2c) is handled by inversion of Rules (35.1) and induction. □

Theorem 35.2 (Progress). *If $\vdash_\Sigma e : \tau$ and $\mu : \Sigma$, then either e val_Σ, or e $\mathsf{unbound}_\mu$, or there exists e' such that $e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'$.*

Proof. By induction on Rules (35.1). For Rule (35.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\mu : \Sigma$, we have either $\mu(a) = \bullet$ or $\mu(a) = e$ for some e such that $\vdash_\Sigma e : \tau$. In the former case we have e $\mathsf{unbound}_\mu$, and in the latter we have $\mathtt{get}[a] \overset{\mu}{\underset{\Sigma}{\longmapsto}} e$. For Rule (35.1b), we have by induction that either e_1 val_Σ or e_1 $\mathsf{unbound}_\mu$, or $e_1 \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_1'$. In the latter two cases we may apply Rule (35.2b) or Rule (35.3b), respectively. If e_1 val_Σ, we apply induction to obtain that either e_2 val_Σ, in which case Rule (35.2d) applies; e_2 $\mathsf{unbound}_\mu$, in which case Rule (35.3b) applies; or $e_2 \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_2'$, in which case Rule (35.2c) applies. □

35.4 Some Subtleties

Fluid binding in the context of a first-order language is easy to understand. If the expression $\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2$ 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 $\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2$ 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.

Consider the expression

$$\texttt{put}\,17\,\texttt{for}\,a\,\texttt{in}\,\lambda\,(x\texttt{:nat}.\,x + \texttt{get}\,a), \tag{35.5}$$

which has type $\texttt{nat} \to \texttt{nat}$, 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 17, and returns the function $\lambda\,(x\texttt{:nat}.\,x + \texttt{get}\,a)$, which refers to the symbol a. If this value were returned from the body of the declaration, applying it would result in an unbound symbol error.

Contrast this with the superficially similar expression

$$\texttt{let}\,y\,\texttt{be}\,17\,\texttt{in}\,\lambda\,(x\texttt{:nat}.\,x + y), \tag{35.6}$$

in which we have replaced the fluid-bound symbol, a, by a statically bound variable, y. This expression evaluates to $\lambda\,(x\texttt{:nat}.\,x + 17)$, which adds 17 to its argument when applied. There is never any possibility of an unbound symbol 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.

One way to think about this situation is to consider that fluid-bound symbols serve as an alternative to passing additional arguments to a function to specialize its value whenever it is called. To see this, let e stand for the value of expression (35.5), a λ-abstraction whose body is dependent on the binding of the symbol a. To use this function safely, it is necessary that the programmer provide a binding for a prior to calling it. For example, the expression

$$\texttt{put}\,7\,\texttt{for}\,a\,\texttt{in}\,(e(9))$$

evaluates to 15, and the expression

$$\texttt{put}\,8\,\texttt{for}\,a\,\texttt{in}\,(e(9))$$

evaluates to 17. Writing just $e(9)$, without a surrounding binding for a, results in a run-time error attempting to retrieve the binding of the unbound symbol a.

This behavior may be simulated by adding an additional argument to the function value that will be bound to the current binding of the symbol a at the point where the function is called. Instead of using fluid binding, one would provide an additional argument at each call site, writing

$$e'(7)(9)$$

and

$$e'(8)(9),$$

respectively, were e' is the λ-abstraction

$$\lambda\,(y{:}\mathtt{nat}.\,\lambda\,(x{:}\mathtt{nat}.\,x+y)).$$

Additional arguments can be cumbersome, though, especially when several call sites provide the same binding for a. Using fluid binding we may write

$$\mathtt{put}\,7\,\mathtt{for}\,a\,\mathtt{in}\,\langle e(8), e(9)\rangle,$$

whereas using an additional argument we must write

$$\langle e'(7)(8), e'(7)(9)\rangle.$$

However, such redundancy can be mitigated by simply factoring out the common part, writing

$$\mathtt{let}\,f\,\mathtt{be}\,e'(7)\,\mathtt{in}\,\langle f(8), f(9)\rangle.$$

A significant drawback of using fluid binding is that the requirement to provide a binding for a is not apparent in the type of e, whereas the type of e' reflects the demand for an additional argument. One may argue that the type system *should* record the dependency of a computation on a specified set of fluid-bound symbols. For example, the expression e might be given a type of the form $\mathtt{nat} \to_a \mathtt{nat}$, reflecting the demand that a binding for a be provided at the call site.

35.5 Fluid References

The `get` and `put` operations for fluid binding are indexed by a symbol that must be given as part of the syntax of the operator. Rather than insist that the target symbol be given statically, it is useful to be able to defer until run-time the choice of fluid on which a `get` or `put` acts. This may be achieved

by introducing *references* to fluids, which allow the name of a fluid to be represented as a value. References come equipped with analogues of the `get` and `put` primitives, but for a dynamically determined symbol.

The syntax of references as an extension to $\mathcal{L}\{\texttt{fluid}\}$ is given by the following grammar:

Typ	τ	::=	$\texttt{fluid}(\tau)$	$\tau\,\texttt{fluid}$	fluid
Exp	e	::=	$\texttt{fl}[a]$	$\texttt{fl}[a]$	reference
			$\texttt{getfl}(e)$	$\texttt{getfl}\,e$	retrieval
			$\texttt{putfl}(e;e_1;e_2)$	$\texttt{putfl}\,e\,\texttt{is}\,e_1\,\texttt{in}\,e_2$	binding

The expression $\texttt{fl}[a]$ is the symbol a considered as a value of type $\texttt{fluid}(\tau)$. The expressions $\texttt{getfl}(e)$ and $\texttt{putfl}(e;e_1;e_2)$ are analogues of the `get` and `put` operations for fluid-bound symbols.

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{fl}[a] : \texttt{fluid}(\tau)} \tag{35.7a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{fluid}(\tau)}{\Gamma \vdash_\Sigma \texttt{getfl}(e) : \tau} \tag{35.7b}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{fluid}(\tau) \quad \Gamma \vdash_\Sigma e_1 : \tau \quad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \texttt{putfl}(e;e_1;e_2) : \tau_2} \tag{35.7c}$$

Since we are using a scoped dynamics, references to fluids cannot be deemed mobile.

The dynamics of references consists of resolving the referent and deferring to the underlying primitives acting on symbols.

$$\frac{}{\texttt{fl}[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{35.8a}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\texttt{getfl}(e) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{getfl}(e')} \tag{35.8b}$$

$$\frac{}{\texttt{getfl}(\texttt{fl}[a]) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{get}[a]} \tag{35.8c}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\texttt{putfl}(e;e_1;e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{putfl}(e';e_1;e_2)} \tag{35.8d}$$

$$\frac{}{\texttt{putfl(fl[}a\texttt{];}e_1\texttt{;}e_2\texttt{)} \xmapsto[\Sigma]{\mu} \texttt{put[}a\texttt{](}e_1\texttt{;}e_2\texttt{)}} \tag{35.8e}$$

35.6 Notes

The concept of dynamic binding arose from the confusion of variables and symbols in early dialects of Lisp. When properly separated, variables retain their substitutive meaning, and symbols give rise to a separate concept of fluid binding. Allen's monograph [4] contains a thorough discussion of the implementation of fluid binding. The formulation given here draws on Allen's account and on more recent work by Nanevski [73].

Chapter 36

Dynamic Classification

In Chapters 14 and 27 we investigated the use of sums for the classification of values of disparate type. Every value of a classified type is labelled with a symbol that determines the type of the instance data. A classified value is decomposed by pattern matching against a known class, which reveals the type of the instance data.

Under this representation the possible classes of an object are fully determined *statically* by its type. However, it is sometimes useful to allow the possible classes of data value to be determined *dynamically*. A typical situation of this kind arises when two components of a program wish to "share a secret"—that is, to compute a value that is opaque to intermediaries. This can be accomplished by creating a fresh class that is known only to the two "end points" of the communication who may create instances of this class, and pattern match against it to recover the underlying datum. In this sense dynamic classification may be regarded as a *perfect encryption* mechanism in which the class serves as an absolutely unbreakable encryption key under which data may be protected from intruders. It is absolutely unbreakable because, by α-equivalence, it is impossible to "guess" the name of a bound symbol.[1]

One may wonder why a program would ever need to keep a secret from itself, but there are many useful applications of this. For example, a program may consist of many independent processes communicating over an insecure network. Perfect encryption by dynamic classification supports the creation of *private channels* between processes; see Chapter 44 for fur-

[1] In practice this is implemented using probabilistic techniques to avoid the need for a central arbiter of unicity of symbol names. However, such methods require a source of randomness, which may be seen as just such an arbiter in disguise. There is no free lunch.

ther details. Exceptions are another, less obvious, application of dynamic classification. An exception involves two parties, the raiser and the handler. Raising an exception may be viewed as sending a message to a *specific* handler (rather than to any handler that wishes to intercept it). This may be enforced by classifying the exception value with a dynamically generated class that is recognized by the intended handler, and no other.

36.1 Dynamic Classes

A dynamic class is a symbol that may be generated at run-time. A classified value consists of a symbol of type τ together with a value of that type. To compute with a classified value, it is compared with a known class. If the value is of this class, the underlying instance data is passed to the positive branch, otherwise the negative branch is taken, where it may be matched against other known classes.

36.1.1 Statics

The syntax of the language `clsfd` of dynamic classification is given by the following grammar:

Typ	τ	$::=$	`clsfd`	`clsfd`	classified
Exp	e	$::=$	`in[`a`](`e`)`	$a \cdot e$	instance
			`isin[`a`](`e`;`x`.`e_1`;`e_2`)`	`match` e `as` $a \cdot x \Rightarrow e_1$ `ow` $\Rightarrow e_2$	comparison

The expression `in[`a`](`e`)` is a classified value with class a and underlying value e. The expression `isin[`a`](`e`;`x`.`e_1`;`e_2`)` checks whether the class of the value given by e is a. If so, the classified value is passed to e_1; if not, the expression e_2 is evaluated instead.

The statics of `clsfd` is defined by the following rules:

$$\frac{\Gamma \vdash_{\Sigma,a:\rho} e : \rho}{\Gamma \vdash_{\Sigma,a:\rho} \texttt{in[}a\texttt{](}e\texttt{)} : \texttt{clsfd}} \tag{36.1a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\rho} e : \texttt{clsfd} \quad \Gamma, x : \rho \vdash_{\Sigma,a:\rho} e_1 : \tau \quad \Gamma \vdash_{\Sigma,a:\rho} e_2 : \tau}{\Gamma \vdash_{\Sigma,a:\rho} \texttt{isin[}a\texttt{](}e\texttt{;}x\texttt{.}e_1\texttt{;}e_2\texttt{)} : \tau} \tag{36.1b}$$

The type associated to the symbol in the signature determines the type of the instance data.

36.1.2 Dynamics

To maximize the flexibility in the use of dynamic classification, we will employ an unscoped dynamics for symbol generation. Within this framework the dynamics of classification is given by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\mathtt{in}[a]\,(e)\ \mathsf{val}_{\Sigma,a:\tau}} \tag{36.2a}$$

$$\frac{\nu\,\Sigma\,\{\,e\,\}\mapsto\nu\,\Sigma'\,\{\,e'\,\}}{\nu\,\Sigma\,\{\,\mathtt{in}[a]\,(e)\,\}\mapsto\nu\,\Sigma'\,\{\,\mathtt{in}[a]\,(e')\,\}} \tag{36.2b}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\nu\,\Sigma\,\{\,\mathtt{isin}[a]\,(\mathtt{in}[a]\,(e);x\,.\,e_1;e_2)\,\}\mapsto\nu\,\Sigma\,\{\,[e/x]e_1\,\}} \tag{36.2c}$$

$$\frac{(a\neq a')}{\nu\,\Sigma\,\{\,\mathtt{isin}[a]\,(\mathtt{in}[a']\,(e');x\,.\,e_1;e_2)\,\}\mapsto\nu\,\Sigma\,\{\,e_2\,\}} \tag{36.2d}$$

$$\frac{\nu\,\Sigma\,\{\,e\,\}\mapsto\nu\,\Sigma'\,\{\,e'\,\}}{\nu\,\Sigma\,\{\,\mathtt{isin}[a]\,(e;x\,.\,e_1;e_2)\,\}\mapsto\nu\,\Sigma'\,\{\,\mathtt{isin}[a]\,(e';x\,.\,e_1;e_2)\,\}} \tag{36.2e}$$

The dynamics of the elimination form for the type `clsfd` relies on *disequality* of names (specifically, Rule (36.2d)). Since disequality is not preserved under substitution, it is not sensible to consider any language construct whose dynamics relies on such a substitution. To see what goes wrong, consider the expression

$$\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,a\cdot\_\Rightarrow\mathtt{tt}\,\mathtt{ow}\Rightarrow\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{ff}\,\mathtt{ow}\Rightarrow\mathtt{tt}.$$

This is easily seen to evaluate to `ff`, since the outer conditional is on the class a, which is *a priori* different from b. However, if we substitute b for a in this expression we obtain

$$\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{tt}\,\mathtt{ow}\Rightarrow\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{ff}\,\mathtt{ow}\Rightarrow\mathtt{tt},$$

which evaluate to `tt`, since now the outer conditional governs the evaluation. (The relevance of this example will become apparent in Chapter 52, which discusses a proposed language extension based on such a substitution.)

36.1.3 Safety

Theorem 36.1 (Safety)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$*, then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$*.*
2. *If* $\vdash_\Sigma e : \tau$*, then either* $e\ \mathsf{val}_\Sigma$ *or* $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$ *for some* e' *and* Σ'*.*

Proof. Similar to the safety proofs given in Chapters 14, 15, and 34. □

36.2 Class References

The type $\mathtt{class}(\tau)$ has as values references to classes.

Typ	τ	::=	$\mathtt{class}(\tau)$	$\tau\,\mathtt{class}$	class reference
Exp	e	::=	$\mathtt{cls}[a]$	$\mathtt{\&}\,a$	reference
			$\mathtt{mk}(e_1;e_2)$	$\mathtt{mk}(e_1;e_2)$	instance
			$\mathtt{isofcls}(e_0;e_1;x\,.\,e_2;e_3)$	$\mathtt{isofcls}(e_0;e_1;x\,.\,e_2;e_3)$	dispatch

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \mathtt{cls}[a] : \mathtt{class}(\tau)} \tag{36.3a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{class}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{mk}(e_1;e_2) : \mathtt{clsfd}} \tag{36.3b}$$

$$\frac{\Gamma \vdash_\Sigma e_0 : \mathtt{class}(\rho) \quad \Gamma \vdash_\Sigma e_1 : \mathtt{clsfd} \quad \Gamma, x:\rho \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma e_3 : \tau}{\Gamma \vdash_\Sigma \mathtt{isofcls}(e_0;e_1;x\,.\,e_2;e_3) : \tau} \tag{36.3c}$$

The corresponding dynamics is given by these rules:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\} \mapsto \nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,\mathtt{mk}(e_1;e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{mk}(e_1';e_2)\,\}} \tag{36.4a}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad \nu\,\Sigma\,\{\,e_2\,\} \mapsto \nu\,\Sigma'\,\{\,e_2'\,\}}{\nu\,\Sigma\,\{\,\mathtt{mk}(e_1;e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{mk}(e_1;e_2')\,\}} \tag{36.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{mk}(\mathtt{cls}[a];e)\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{in}[a]\,(e)\,\}} \tag{36.4c}$$

$$\frac{\nu\,\Sigma\,\{\,e_0\,\} \mapsto \nu\,\Sigma'\,\{\,e_0'\,\}}{\nu\,\Sigma\,\{\,\mathtt{isofcls}(e_0;e_1;x\,.\,e_2;e_3)\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{isofcls}(e_0';e_1;x\,.\,e_2;e_3)\,\}} \tag{36.4d}$$

$$\nu\,\Sigma\,\{\,\texttt{isofcls(cls[}a\texttt{];}e_1\texttt{;}x\,\texttt{.}\,e_2\texttt{;}e_3\texttt{)}\,\}\mapsto\nu\,\Sigma\,\{\,\texttt{isin[}a\texttt{](}e_1\texttt{;}x\,\texttt{.}\,e_2\texttt{;}e_3\texttt{)}\,\} \tag{36.4e}$$

Rules (36.4d) and (36.4e) specify that the first argument is evaluated to determine the target class, which is then used to check whether the second argument, a classified data value, if of the target class. This may be seen as a two-stage pattern matching process in which evaluation of e_0 determines the pattern against which to match the classified value of e_1.

36.3 Definability of Dynamic Classes

The type `clsfd` may be defined in terms of symbolic references, product types, and existential types by the type expression

$$\texttt{clsfd}\triangleq\exists\texttt{(}t\,\texttt{.}\,t\,\texttt{sym}\times t\texttt{)}.$$

The introductory form, `in[`a`](`e`)`, where a is a symbol whose associated type is ρ and e is an expression of type ρ, is defined to be the package

$$\texttt{pack}\,\rho\,\texttt{with}\,\langle\texttt{\&}\,a,e\rangle\,\texttt{as}\,\exists\texttt{(}t\,\texttt{.}\,t\,\texttt{sym}\times t\texttt{)}.$$

The eliminatory form, `isin[`a`](`e`;`$x\,\texttt{.}\,e_1$`;`e_2`)`, is defined in terms of symbol comparison as defined in Chapter 34. Suppose that the overall type of the conditional is τ and that the type associated to the symbol a is ρ. The type of e must be `clsfd`, defined as above, and the type of the branches e_1 and e_2 must be τ, with x assumed to be of type ρ in e_1. The conditional is defined to be the expression

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle x,y\rangle\texttt{:}\,t\,\texttt{sym}\times t\,\texttt{in}\,\texttt{(}e_{body}\texttt{(}y\texttt{))},$$

where e_{body} is an expression to be defined shortly. The comparison opens the package, e, representing the classified value, and decomposes it into a type, t, a symbol, x, of type t `sym`, and an underlying value, y, of type t. The expression e_{body}, which is to be defined shortly, will have the type $t\to\tau$, so that the application to y is type correct.

The expression e_{body} compares the symbolic reference, x, to the symbol, a, of type ρ, and yields a value of type $t\to\tau$ regardless of the outcome. It is therefore defined to be the expression

$$\texttt{is[}a\texttt{][}u\,\texttt{.}\,u\to\tau\texttt{](}x\texttt{;}e_1'\texttt{;}e_2'\texttt{)}$$

where, in accordance with Rule (34.5b), e_1' has type $[\rho/u](u \to \tau) = \rho \to \tau$, and e_2' has type $[t/u](u \to \tau) = t \to \tau$. The expression e_1' "knows" that the abstract type, t, is ρ, the type associated to the symbol a, because the comparison has come out positively. On the other hand, e_2' does not "learn" anything about the identity of t.

It remains to choose the expressions e_1' and e_2'. In the case of a positive comparison, we wish to pass the classified value to the expression e_1 by substitution for the variable x. This is accomplished by defining e_1' to be the expression

$$\lambda\,(x{:}\rho.\,e_1) : \rho \to \tau.$$

In the case of a negative comparison no value is to be propagated to e_2. We therefore define e_2' to be the expression

$$\lambda\,(\_{:}t.\,e_2) : t \to \tau.$$

We may then check that the statics and dynamics given in Section 36.1 on page 340 are derivable, given the definitions of the type of classified values and its introductory and eliminator forms.

36.4 Classifying Secrets

Dynamic classification may be used to enforce *confidentiality* and *integrity* of data values in a program. A value of type `clsfd` may only be constructed by *sealing* it with some class, a, and may only be deconstructed by a case analysis that includes a branch for a. By controlling which parties in a multi-party interaction have access to the classifier, a, we may control how classified values are created (ensuring their *integrity*) and how they are inspected (ensuring their *confidentiality*). Any party that lacks access to a cannot decipher a value classified by a, nor may it create a classified value with this class. Because classes are dynamically generated symbols, they provide an absolute confidentiality guarantee among parties in a computation.[2]

Consider the following simple protocol for controlling the integrity and confidentiality of data in a program. A fresh symbol, a, is introduced, and we return a pair of functions of type

$$(\tau \to \texttt{clsfd}) \times (\texttt{clsfd} \to \tau\,\texttt{opt}),$$

[2]Of course, this guarantee is for programs written in conformance with the statics given here. If the abstraction imposed by the type system is violated, no guarantees of confidentiality can be made.

called the *constructor* and *destructor* functions for that class.

```
newsym a:τ in
  ⟨ λ(x:τ.a · x),
    λ(x:clsfd.match x as a · y ⇒ null ow ⇒ just(y)) ⟩.
```

The first function creates a value classified by a, and the second function recovers the instance data of a value classified by a. Outside of the scope of the declaration the symbol a is an absolutely unguessable secret.

To enforce the *integrity* of a value of type τ, it is sufficient to ensure that only trusted parties have access to the constructor. To enforce the *confidentiality* of a value of type τ, it is sufficient to ensure that only trusted parties have access to the destructor. Ensuring the integrity of a value amounts to associating an invariant to it that is maintained by the trusted parties that may create an instance of that class. Ensuring the confidentiality of a value amounts to propagating the invariant to parties that may decipher it.

36.5 Notes

Dynamic classification appears in Standard ML as the type, `exn`, of values associated with exceptions [67]. Unfortunately the utility of this type is obscured by a too-close association with its original application: a new class is allocated by an `exception` declaration! Yet, as we have seen here, dynamic classification is useful for much more than just exceptions. It can be used to model perfect encryption and to control the flow of information among the parties in a computation. This application was popularized in a different disguise in the π-calculus [66], under the name "channel passing," once again obscuring the underlying mechanism by associating it with a particular application. Here we have instead presented dynamic classification as a concept in its own right, with several different applications.

Part XIV

Storage Effects

Chapter 37

Modernized Algol

Modernized Algol, or $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$, is an imperative, block-structured programming language based on the classic language Algol. $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ may be seen as an extension to $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with a new syntactic sort of *commands* that act on *assignables* by retrieving and altering their contents. Assignables are introduced by *declaring* them for use within a specified scope; this is the essence of block structure. Commands may be combined by sequencing, and may be iterated using recursion.

$\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ maintains a careful separation between *pure* expressions, whose meaning does not depend on any assignables, and *impure* commands, whose meaning is given in terms of assignables. This ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, and allows for expressions to be manipulated much as in **PCF**. Commands, on the other hand, have a tightly constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is that it adheres to the *stack discipline*, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline. This avoids the need for more complex forms of storage management, at the expense of reducing the expressiveness of the language. (Relaxing this restriction is the subject of Chapter 38.)

37.1 Basic Commands

The syntax of the language $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ of modernized Algol distinguishes pure *expressions* from impure *commands*. The expressions include those of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ (as described in Chapter 12), augmented with one additional

construct, and the commands are those of a simple imperative programming language based on assignment. The language maintains a sharp distinction between *mathematical variables*, or just *variables*, and *assignable variables*, or just *assignables*. Variables are introduced by λ-abstraction, and are given meaning by substitution. Assignables are introduced by a declaration, and are given meaning by assignment and retrieval of their *contents*, which is, for the time being, restricted to natural numbers. Expressions evaluate to values, and have no effect on assignables. Commands are executed for their effect on assignables, and also return a value. Composition of commands not only sequences their execution order, but also passes the value returned by the first to the second before it is executed. The returned value of a command is, for the time being, restricted to the natural numbers. (But see Section 37.3 on page 357 for the general case.)

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is given by the following grammar, from which we have omitted repetition of the expression syntax of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ for the sake of brevity.

Typ	τ	$::=$	$\texttt{cmd}$	$\texttt{cmd}$	command
Exp	e	$::=$	$\texttt{do}(m)$	$\texttt{do}\,m$	encapsulation
Cmd	m	$::=$	$\texttt{ret}(e)$	$\texttt{ret}\,e$	return
			$\texttt{bnd}(e; x.m)$	$\texttt{bnd}\,x \leftarrow e\,;m$	sequence
			$\texttt{dcl}(e; a.m)$	$\texttt{dcl}\,a := e\,\texttt{in}\,m$	new assignable
			$\texttt{get}[a]$	a	fetch
			$\texttt{set}[a](e)$	$a := e$	assign

The expression $\texttt{do}(m)$ consists of the unevaluated command, m, thought of as a value of type $\texttt{cmd}$. The command, $\texttt{ret}(e)$, returns the value of the expression e without having any effect on the assignables. The command $\texttt{bnd}(e; x.m)$ evaluates e to an encapulated command, which is then executed and its returned value is substituted for x prior to executing m. The command $\texttt{dcl}(e; a.m)$ introduces a new assignable, a, for use within the command, m, whose initial contents is given by the expression, e. The command $\texttt{get}[a]$ returns the current contents of the assignable, a, and the command $\texttt{set}[a](e)$ changes the contents of the assignable a to the value of e, and returns that value.

37.1.1 Statics

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ consists of two forms of judgement:

1. Expression typing: $\Gamma \vdash_\Sigma e : \tau$.

2. Command formation: $\Gamma \vdash_\Sigma m \ \mathsf{ok}$.

The context, Γ, specifies the types of variables, as usual, and the signature, Σ, consists of a finite set of assignables. These judgements are inductively defined by the following rules:

$$\frac{\Gamma \vdash_\Sigma m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{do}(m) : \mathtt{cmd}} \tag{37.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \ \mathsf{ok}} \tag{37.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd} \quad \Gamma, x : \mathtt{nat} \vdash_\Sigma m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \ \mathsf{ok}} \tag{37.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat} \quad \Gamma \vdash_{\Sigma,a} m \ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \ \mathsf{ok}} \tag{37.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \mathtt{get}[a] \ \mathsf{ok}} \tag{37.1e}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \mathtt{nat}}{\Gamma \vdash_{\Sigma,a} \mathtt{set}[a](e) \ \mathsf{ok}} \tag{37.1f}$$

Rule (37.1a) is the introductory rule for the type `cmd`, and Rule (37.1c) is the corresponding eliminatory form. Rule (37.1d) introduces a new assignable for use within a specified command. The name, a, of the assignable is bound by the declaration, and hence may be renamed to satisfy the implicit constraint that it not already be present in Σ. Rule (37.1e) states that the command to retrieve the contents of an assignable, a, returns a natural number. Rule (37.1f) states that we may assign a natural number to an assignable.

37.1.2 Dynamics

The dynamics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup\}$ is defined in terms of a *memory*, μ, a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgement forms:

1. $e \ \mathsf{val}_\Sigma$, stating that e is a value relative to Σ.

2. $e \underset{\Sigma}{\mapsto} e'$, stating that the expression e steps to the expression e'.

These judgements are inductively defined by the following rules, together with the rules defining the dynamics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ (see Chapter 12). It is important, however, that the successor operation be given an *eager*, rather than *lazy*, dynamics so that a closed value of type `nat` is a numeral (for reasons that will be explained in Section 37.3 on page 357).

$$\frac{}{\mathtt{do}(m)\ \mathsf{val}_\Sigma} \tag{37.2a}$$

Rule (37.2a) states that an encapsulated command is a value.

The dynamics of commands is defined in terms of states $m \parallel \mu$, where μ is a memory mapping assignables to values, and m is a command. There are two judgements governing such states:

1. $m \parallel \mu\ \mathsf{final}_\Sigma$. The state $m \parallel \mu$ is fully executed.
2. $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$. The state $m \parallel \mu$ steps to the state $m' \parallel \mu'$; the set of active assignables is given by the signature Σ.

These judgements are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{ret}(e) \parallel \mu\ \mathsf{final}_\Sigma} \tag{37.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{ret}(e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{37.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{bnd}(e; x.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(e'; x.m) \parallel \mu} \tag{37.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd}(\mathtt{do}(\mathtt{ret}(e)); x.m) \parallel \mu \underset{\Sigma}{\mapsto} [e/x]m \parallel \mu} \tag{37.3d}$$

$$\frac{m_1 \parallel \mu \underset{\Sigma}{\mapsto} m_1' \parallel \mu'}{\mathtt{bnd}(\mathtt{do}(m_1); x.m_2) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(\mathtt{do}(m_1'); x.m_2) \parallel \mu'} \tag{37.3e}$$

$$\frac{}{\mathtt{get}[a] \parallel \mu \otimes \langle a : e\rangle \underset{\Sigma,a}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes \langle a : e\rangle} \tag{37.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{set}[a](e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{set}[a](e') \parallel \mu} \tag{37.3g}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{set[}a\mathtt{]}(e) \parallel \mu \otimes \langle a : \_ \rangle \underset{\Sigma}{\mapsto} \mathtt{ret}(e) \parallel \mu \otimes \langle a : e \rangle} \tag{37.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{dcl}(e; a\,.\,m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a\,.\,m) \parallel \mu} \tag{37.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad m \parallel \mu \otimes \langle a : e \rangle \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes \langle a : e' \rangle}{\mathtt{dcl}(e; a\,.\,m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a\,.\,m') \parallel \mu'} \tag{37.3j}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad e'\ \mathsf{val}_{\Sigma,a}}{\mathtt{dcl}(e; a\,.\,\mathtt{ret}(e')) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{37.3k}$$

Rule (37.3a) specifies that a `ret` command is final if its argument is a value. Rules (37.3c) to (37.3e) specify the dynamics of sequential composition. The expression, e, must, by virtue of the type system, evaluate to an encapsulated command, which is to be executed to determine its return value, which is then substituted into m before executing it.

Rules (37.3i) to (37.3k) define the concept of *block structure* in a programming language. Declarations adhere to the *stack discipline* in that an assignable is allocated for the duration of evaluation of the body of the declaration, and deallocated after evaluation of the body is complete. Therefore the lifetime of an assignable can be identified with its scope, and hence we may visualize the dynamic lifetimes of assignables as being nested inside one another, in the same manner as their static scopes are nested inside one another. This stack-like behavior of assignables is a characteristic feature of what are known as *Algol-like languages*.

37.1.3 Safety

The judgement $m \parallel \mu\ \mathsf{ok}_\Sigma$ is defined by the rule

$$\frac{\vdash_\Sigma m\ \mathsf{ok} \quad \mu : \Sigma}{m \parallel \mu\ \mathsf{ok}_\Sigma} \tag{37.4}$$

where the auxiliary judgement $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a : \rho \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e\ \mathsf{val}_\varnothing \text{ and } \vdash_\varnothing e : \mathtt{nat}}{\mu : \Sigma} \tag{37.5}$$

That is, the memory must bind a number to each location in Σ.

Theorem 37.1 (Preservation)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$*.*
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$*, with* $\vdash_\Sigma m$ *ok and* $\mu : \Sigma$*, then* $\vdash_\Sigma m'$ *ok and* $\mu' : \Sigma$*.*

Proof. Simultaneously, by induction on Rules (37.2) and (37.3).

Consider Rule (37.3j). Assume that $\vdash_\Sigma \texttt{dcl}(e; a.m)\ \texttt{ok}$ and $\mu : \Sigma$. By inversion of typing we have $\vdash_\Sigma e : \texttt{nat}$ and $\vdash_{\Sigma,a} m\ \texttt{ok}$. Since $e\ \mathsf{val}_\Sigma$ and $\mu : \Sigma$, we have $\mu \otimes \langle a : e \rangle : \Sigma, a$. By induction we have $\vdash_{\Sigma,a} m'\ \texttt{ok}$ and $\mu' \otimes \langle a : e \rangle : \Sigma, a$, from which the result follows immediately.

Consider Rule (37.3k). Assume that $\vdash_\Sigma \texttt{dcl}(e; a.\texttt{ret}(e'))\ \texttt{ok}$ and $\mu : \Sigma$. By inversion we have $\vdash_\Sigma e : \texttt{nat}$, $\vdash_{\Sigma,a} \texttt{ret}(e')\ \texttt{ok}$, and hence that $\vdash_{\Sigma,a} e' : \texttt{nat}$. But since $e'\ \mathsf{val}_{\Sigma,a}$, we also have $\vdash_\Sigma e' : \texttt{nat}$, as required. □

Theorem 37.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* $e\ \mathsf{val}_\Sigma$*, or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$*.*
2. *If* $\vdash_\Sigma m$ *ok and* $\mu : \Sigma$*, then either* $m \parallel \mu\ \mathsf{final}_\Sigma$ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'*.*

Proof. Simultaneously, by induction on Rules (37.1). Consider Rule (37.1d). By the first inductive hypothesis we have either $e \underset{\Sigma}{\mapsto} e'$ or $e\ \mathsf{val}_\Sigma$. In the former case Rule (37.3i) applies. In the latter, we have by the second inductive hypothesis either $m \parallel \mu \otimes \langle a : e \rangle\ \mathsf{final}_{\Sigma,a}$ or $m \parallel \mu \otimes \langle a : e \rangle \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes \langle a : e' \rangle$. In the former case we apply Rule (37.3k), and in the latter, Rule (37.3j). □

A variant of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ treats the operation $\texttt{get}[a]$ as a form of *expression*, rather than as a form of *command*. This allows us to write expressions such as $a + b$ for the sum of the contents of assignables a and b, rather than have to write a command that explicitly fetches the contents of a and b, returning their sum.

To allow for this we must enrich the dynamics of expressions to allow access to the bindings of the active assignables, writing $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$ to state that one step of evaluation of the expression e relative to Σ and μ results in the expression e'. The definition of this judgement includes the rule

$$\frac{}{\texttt{get}[a] \overset{\mu \otimes \langle a : e \rangle}{\underset{\Sigma,a}{\longmapsto}} e}, \tag{37.6}$$

which allows an expression to depend on the contents of an assignable.

37.2 Some Programming Idioms

The language $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is designed to expose the elegant interplay between the execution of an expression for its value and the execution of a command for its effect on assignables. In this section we show how to derive several standard idioms of imperative programming in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$.

We define the *sequential composition* of commands, written $\{x \leftarrow m_1 \,;\, m_2\}$, to stand for the command $\texttt{bnd}\, x \leftarrow \texttt{do}\, (m_1) \,;\, m_2$. This generalizes to an n-ary form by defining

$$\{x_1 \leftarrow m_1 \,;\, \ldots\, x_{n-1} \leftarrow m_{n-1} \,;\, m_n\},$$

to stand for the iterated composition

$$\{x_1 \leftarrow m_1 \,;\, \ldots \{x_{n-1} \leftarrow m_{n-1} \,;\, m_n\}\}.$$

We sometimes write just $\{m_1 \,;\, m_2\}$ for the composition $\{\_ \leftarrow m_1 \,;\, m_2\}$ in which the returned value from m_1 is ignored; this generalizes in the obvious way to an n-ary form.

A related idiom, the command $\texttt{run}\, e$, executes an encapsulated command and returns its result; it for $\texttt{bnd}\, x \leftarrow e \,;\, \texttt{ret}\, x$.

The *conditional* command, $\texttt{if}\, (m)\, m_1\, \texttt{else}\, m_2$, executes either m_1 or m_2 according to whether the result of executing m is zero or not:

$$\{x \leftarrow m \,;\, \texttt{run}\, (\texttt{ifz}\, x\, \{\texttt{z} \Rightarrow \texttt{do}\, m_1 \mid \texttt{s}(\_) \Rightarrow \texttt{do}\, m_2\})\}.$$

The returned value of the conditional is the value returned by the selected command.

The *while loop* command, $\texttt{while}\, (m_1)\, m_2$, repeatedly executes the command m_2 while the command m_1 yields a non-zero number. It is defined as follows:

$$\texttt{run}\, (\texttt{fix}\, \mathit{loop} : \texttt{cmd}\, \texttt{is}\, \texttt{do}(\texttt{if}\, (m_1)\, \{\texttt{ret}\, \texttt{z}\}\, \texttt{else}\, \{m_2 \,;\, \texttt{run}\, \mathit{loop}\})).$$

This commands runs the self-referential encapsulated command that, when executed, first executes m_1, branching on the result. If the result is zero, the loop returns zero (arbitrarily). If the result is non-zero, the command m_2 is executed and the loop is repeated.

A *procedure* is a function of type $\tau \rightharpoonup$ `cmd` that takes an argument of some type, τ, and yields an unexecuted command as result. A *procedure call* is the composition of a function application with the activation of the resulting command. If e_1 is a procedure and e_2 is its argument, then the procedure call `call` e_1`(`e_2`)` is defined to be the command `run (`e_1`(`e_2`))`, which immediately runs the result of applying e_1 to e_2.

As an example, here is a procedure of type `nat` $\rightharpoonup$ `cmd` that returns the factorial of its argument:

```
λx:nat. do {
  dcl r := 1 in
  dcl a := x in
  { while ( a ) {
      y ← r
    ; z ← a
    ; r := (x-z+1)× y
    ; a := z-1
    }
    ; r
  }
}
```

The loop maintains that invariant that the contents of `r` is the factorial of x minus the contents of `a`. Initialization makes this invariant true, and it is preserved by each iteration of the loop, so that upon completion of the loop the assignable `a` contains 0 and `r` contains the factorial of x, as required.

If, as described in Section 37.2 on the preceding page, we admit assignables as forms of expression, this example may be written as follows:

```
λx:nat. do {
  dcl r := 1 in
  dcl a := x in
  { while ( ret (a) ) {
      r := (x-a+1)× r
    ; a := a-1
    }
    ; ret ( r )
  }
}
```

The test governing the `while` loop is the command that returns the contents of the assignable a. However, if assignables are forms of expression,

it makes sense to change the syntax of the `while` command so that the test condition is an expression, rather than a command. In this case the expression would simply be `a`, the expression that returns the contents of the assignable a, rather than the more awkward command that returns its contents.

37.3 Typed Commands and Typed Assignables

So far we have restricted the type of the returned value of a command, and the contents of an assignable, to be `nat`. Can this restriction be relaxed, while adhering to the stack discipline?

The key to admitting other types of returned value and assignable variables is to consider the proof of Theorem 37.1 on page 354. There we relied on the fact that a value of type `nat` is a composition of successors, starting from zero, to ensure that the value is well-typed even in the absence of the locally declared assignable, a. The proof breaks down, and indeed the preservation theorem is false, when the return type of a command or the contents type of an assignable is unrestricted.

For example, if we may return values of procedure type, then we may violate safety as follows:

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \texttt{ret}\, (\lambda\, (x\texttt{:nat.do}\, \{a := x\})).$$

This command, when executed, allocates a new assignable, a, and returns a procedure that, when called, assigns its argument to a. But this makes no sense, because the assignable, a, is deallocated when the body of the declaration returns, but the returned value still refers to it! If the returned procedure is called, execution will get stuck in the attempt to assign to a.

A similar example shows that admitting assignables of arbitrary type is also unsound:

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \{b := \lambda\, (x\texttt{:nat.do}\, \{a := x\})\, \texttt{;ret}\, \texttt{z}\}.$$

We assign to it a procedure that uses a locally declared assignable, a, and then leaves the scope of the declaration. If we then call the procedure stored in b, execution will get stuck attempting to assign to the non-existent assignable, a!

To admit declarations to return values and to admit assignables of types other than `nat`, we must rework the statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ to record the returned type of a command and to record the type of the contents of each

assignable. First, we generalize the finite set, Σ, of active assignables to assign a type to each active assignable so that Σ has the form of a finite set of assumptions of the form $a : \tau$, where a is an assignable. Second, we replace the judgement $\Gamma \vdash_\Sigma m$ ok by the more general form $\Gamma \vdash_\Sigma m \sim \tau$, stating that m is a well-formed command returning a value of type τ. Third, the type cmd must be generalized to $\texttt{cmd}(\tau)$, which is written in examples as τ cmd, to specify the return type of the encapsulated command.

The statics given in Section 37.1.1 on page 350 may be generalized to admit typed commands and typed assignables, as follows:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \texttt{do}(m) : \texttt{cmd}(\tau)} \tag{37.7a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{ret}(e) \sim \tau} \tag{37.7b}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\tau) \quad \Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \texttt{bnd}(e; x.m) \sim \tau'} \tag{37.7c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \text{ mobile} \quad \Gamma \vdash_{\Sigma, a:\tau} m \sim \tau' \quad \tau' \text{ mobile}}{\Gamma \vdash_\Sigma \texttt{dcl}(e; a.m) \sim \tau'} \tag{37.7d}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} \texttt{get}[a] \sim \tau} \tag{37.7e}$$

$$\frac{\Gamma \vdash_{\Sigma, a:\tau} e : \tau}{\Gamma \vdash_{\Sigma, a:\tau} \texttt{set}[a](e) \sim \tau} \tag{37.7f}$$

Apart from the generalization to track returned types and content types, the most important change is to require that the types τ and τ' be a mobile type in Rule (37.7d).

As in Chapter 34, these rules make use of the judgement τ mobile, which states that the type τ is *mobile*. The definition of this judgement is guided by the following *mobility condition*:

$$\textit{if } \tau \text{ mobile}, \vdash_{\Sigma, a:\rho} e : \tau \textit{ and } e \text{ val}_{\Sigma, a:\rho}, \textit{ then } \vdash_\Sigma e : \tau \textit{ and } e \text{ val}_\Sigma. \tag{37.8}$$

Since the successor is evaluated eagerly, the type nat may be deemed mobile:

$$\frac{}{\texttt{nat mobile}} \tag{37.9}$$

Since the body of a procedure may involve an assignable, no procedure type may be considered mobile, nor may the type of commands returning a given type, for similar reasons. On the other hand, a product of mobile

types may safely be deemed mobile, provided that pairing is evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 \times \tau_2 \text{ mobile}} \tag{37.10}$$

Similarly, sums may be deemed mobile so long as the injections are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 + \tau_2 \text{ mobile}} \tag{37.11}$$

Laziness defeats mobility, because values may contain suspended computations that depend on an assignable. For example, if the successor operation for the natural numbers were evaluated lazily, then $\mathtt{s}(e)$ would be a value for any expression, e, including one that refers to an assignable, a. Similarly, if pairing were lazy, then products may not be deemed mobile, and if injections were evaluated lazily, then sums may not either.

What about function types other than procedure types? One may think they are mobile, because a pure expression cannot depend on an assignable. While this is indeed the case, the mobility condition need not hold. For example, consider the following value of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$:

$$\lambda\,(x{:}\mathtt{nat}.\,(\lambda\,(\_{:}\mathtt{cmd}\,\mathtt{z}))(\mathtt{do}\,\{a\})).$$

Although the assignable a is not actually needed to compute the result, it nevertheless occurs in the value, in violation of the safety condition.

Theorem 37.3 (Preservation for Typed Commands)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$, *then* $\vdash_\Sigma e' : \tau$.
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$, *with* $\vdash_\Sigma m \sim \tau$ *and* $\mu : \Sigma$, *then* $\vdash_\Sigma m' \sim \tau$ *and* $\mu' : \Sigma$.

Theorem 37.4 (Progress for Typed Commands)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* e val_Σ, *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.
2. *If* $\vdash_\Sigma m \sim \tau$ *and* $\mu : \Sigma$, *then either* $m \parallel \mu$ final_Σ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'.

The proofs of Theorems 37.3 and 37.4 follows very closely the proof of Theorems 37.1 on page 354 and 37.2 on page 354. The main difference is that we appeal to the mobility condition in the case of a declaration.

37.4 Capabilities

The commands a and $a := e$ operate on a statically specified target assignable, a. That is, a must be in scope at the point where the command occurs. Since a is a static parameter of these commands, and not an argument determined at run-time, it would appear, at first glance, that there is no way to operate on an assignable that is not determined until run-time. For example, how can we write a procedure that, for a dynamically specified assignable, adds two to the contents of that assignable?

One way is to use a *capability* to operate on that assignment. A capability is an encapsulated command that operates on an assignable when it is activated. The *get capability*, or *getter*, for an assignable a of type τ is the command $\texttt{do}\,\{a\}$ of type $\tau\,\texttt{cmd}$ that, when executed, returns the contents of a. The *set capability*, or *setter*, for a is the procedure $\lambda\,(x\!:\!\tau.\,\texttt{do}\,\{a := x\})$ that, when applied, assigns its argument to a. Since capabilities are pairs of procedures, they are not mobile.

A general double-increment procedure that operates on any assignable, regardless of whether it is in scope, may be programmed as follows:

$$\lambda\,(\mathit{get}\!:\!\texttt{nat cmd}.\,\lambda\,(\mathit{set}\!:\!\texttt{nat} \rightharpoonup \texttt{nat cmd}.\,\texttt{do}\,\{x \leftarrow \mathit{get}\,;\mathit{set}(\texttt{s(s(}x\texttt{))})\}).$$

The procedure is to be called with a getter and a setter for the same assignable. When executed, it invokes the getter to obtain the contents of that assignable, and then invokes the setter to assign its contents to be two more than the value it contained.

Although it is natural to consider the get and set capabilities for an assignable as a pair, it can be useful to separate them to provide limited access to an assignable in a particular context. If only the get capability is passed to a procedure, then its result may depend on the contents of the underlying assignable, but may not alter it. Similarly, if only the set capability is passed, then it may alter the contents of the underlying assignable, but cannot access its current contents. It is also useful to consider other forms of capability than simple getters and setters. For example, one could define an increment and a decrement capability for an assignable, and pass one or both to a procedure to limit how it may influence the value of that assignable. The possibilities are endless.

37.5 References

Returning to the double-increment example, the type does not constrain the caller to provide get and set capabilities that act on the same assignable. One way to ensure this is to introduce a *name*, or *reference*, to an assignable as a form of value. A reference may be thought of as a token that provides access to the get and set capabilities of an assignable. Moreover, two references may be tested for equality, so that one may determine at run-time whether they refer to the same underlying assignable.[1]

A *reference* is a value of type $\texttt{ref}(\tau)$, where τ is the type of the contents of the assignable to which it refers. A (closed) value of reference type is the name of an assignable thought of as a form of expression. Possessing a reference allows one to perform either a get or a set operation on the underlying assignable. (One may also consider read-only or write-only references, or more complex capabilities for assignables, in a similar manner.) This suggests the following syntax for references:

Typ	τ	$::=$	$\texttt{ref}(\tau)$	$\tau\,\texttt{ref}$	assignable
Exp	e	$::=$	$\texttt{ref}[a]$	$\&\,a$	reference
Cmd	m	$::=$	$\texttt{getref}(e)$	$@\,e$	contents
			$\texttt{setref}(e_1; e_2)$	$e_1 := e_2$	update

The statics of references is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} \texttt{ref}[a] : \texttt{ref}(\tau)} \tag{37.12a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{ref}(\tau)}{\Gamma \vdash_\Sigma \texttt{getref}(e) \sim \tau} \tag{37.12b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{ref}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \texttt{setref}(e_1; e_2) \sim \tau} \tag{37.12c}$$

Rule (37.12a) specifies that the name of any active assignable is an expression of type $\texttt{ref}(\tau)$.

The dynamics is defined to defer to the corresponding operation on the assignable to which a reference refers.

$$\frac{}{\texttt{ref}[a]\ \mathsf{val}_{\Sigma,a}} \tag{37.13a}$$

[1] This can also be achieved using capabilities by using the setter for one to modify the assignable and using the getter for the other to determine whether it changed.

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{getref}(e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{getref}(e') \parallel \mu} \tag{37.13b}$$

$$\frac{}{\texttt{getref}(\texttt{ref}[a]) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{get}[a] \parallel \mu} \tag{37.13c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\texttt{setref}(e_1; e_2) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{setref}(e_1'; e_2) \parallel \mu} \tag{37.13d}$$

$$\frac{}{\texttt{setref}(\texttt{ref}[a]; e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{set}[a](e) \parallel \mu} \tag{37.13e}$$

A reference to an assignable is a value. The `getref` and `setref` operations on references defer to the corresponding operations on assignables once the reference has been determined.

Suprisingly, the addition of references to assignables does not violate the stack discipline, so long as reference types are deemed immobile. This ensures that a reference can never escape the scope of the assignable to which it refers, which is essential to maintaining safety. We leave to the reader the task of proving safety for the extension of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ with reference types.

As an example of programming with references, the double increment procedure given earlier can be coded using references, rather than capabilities, as follows:

$$\lambda\,(r{:}\texttt{nat ref}.\texttt{do}\,\{x \leftarrow \texttt{@}\,r\,;r := \texttt{s}(\texttt{s}(x))\}).$$

Since the argument is a reference to an assignable, rather than a get and set capability, it is assured that the body of the procedure acts on a single assignable when performing the get and set operations on the reference.

37.6 Aliasing

References and capabilities allow assignables to be treated as values that can be passed as arguments to procedures. This allows us to write programs, such as the double increment procedure, that act on assignables that are not in scope within the body of the procedure. Such expressive power, however, comes at a price: we must carefully consider whether two

references refer to the same assignable or not. This phenomenon is called *aliasing*; it greatly complicates reasoning about program correctness.

Consider, for example, the problem of writing a procedure that, when given two references, x and y, adds twice the contents of y to the contents of x. One way to write this code creates no complications:

$$\lambda\,(x{:}\mathtt{nat\ ref}.\,\lambda\,(y{:}\mathtt{nat\ ref}.\,\mathtt{do}\,\{x' \leftarrow \mathtt{@}\,x\,;y' \leftarrow \mathtt{@}\,y\,;x := x' + y' + y'\})).$$

Even if x and y refer to the same assignable, the effect will be to set the contents of the assignable referenced by x to twice the contents of the assignable referenced by y.

But now consider the following apparently equivalent implementation of the "same" procedure:

$$\lambda\,(x{:}\mathtt{nat\ ref}.\,\lambda\,(y{:}\mathtt{nat\ ref}.\,\mathtt{do}\,\{x\,\mathtt{+=}\,y\,;x\,\mathtt{+=}\,y\})),$$

where x += y is the command

$$\{x' \leftarrow \mathtt{@}\,x\,;y' \leftarrow \mathtt{@}\,y\,;x := x' + y'\}$$

that adds the contents of y to the contents of x. The second implementation works properly provided that x and y do not refer to the same assignable. For if they are aliases in that they both refer to the same assignable, a, with contents n_0, the result is that a is to set $4 \times n_0$, instead of the intended $3 \times n_0$.

In this case it is entirely obvious how to avoid the problem: use the first implementation, rather than the second! But the difficulty is not in fixing the problem once it has been uncovered, but rather noticing the problem in the first place. Wherever references (or capabilities) are used, the problems of interference lurk. Avoiding them requires very careful consideration of all possible aliasing relationships among all of the references in play at a given point of a computation. The problem is that the number of possible aliasing relationships among n references grows at least quadratically in n (we must consider all possible pairings) and can even be worse when more subtle relationships among three or more variables must be considered. Aliasing is a prime source of errors in imperative programs, and remains a strong argument against using imperative methods whenever possible.

37.7 Notes

Modernized Algol is essentially a reformulation of Reynolds's Idealized Algol [89] in which we have maintained a clearer separation between computations that depend on the store and those that do not. In Idealized Algol, as in the original Algol language, an assignable may be used as a form

of expression standing for its current contents. While syntactically convenient, this convention introduces an unfortunate dependency of expression evaluation on the store that we avoid here. The concept of mobility of a type was introduced in the ML5 language for distributed computing [102], with the similar meaning that a value of a mobile type cannot depend on local resources. This restriction is used here to ensure that Modernized Algol adheres to the stack discipline. The possibility of admitting references to stack-allocated assignables was implicit in Reynolds's account, essentially as the concept of a capability discussed here. In the literature it is common to associate references with heap allocation; the treatment of references given here shows that this association is inessential.

What are called here *assignables* are regrettably called *variables* in the programming language literature. This clash of terminology is the source of considerable confusion and misunderstanding. It is preferable to retain the well-established meaning of a variable as standing for an unspecified object of a specified type, but to do so requires that we invent a new word for the name of a piece of mutable storage. The word *assignable* seems apt, and equally as convenient as the misappropriated word *variable*.

The modal distinction between expressions and commands was present in the original formulation of Algol 60, more than 50 years ago. The same organization has been popularized by Haskell, under the name "IO monad," without a clear modal distinction between the commands and expressions.

Chapter 38

Mutable Data Structures

In Chapter 37 we considered an imperative programming language that adheres to the stack discipline in that assignables are allocated and deallocated on a last-in, first-out basis. To ensure this we restricted the types of return values from a command, the types of contents of assignables, to be *mobile* types, ones whose values cannot depend on the stack of assignables. Function and command types are not mobile, nor are reference types, because these types may classify values that refer to an assignable.

A major use of references, however, is to implement *mutable* data structures whose structure may be changed at execution time. The classic example is a linked list in which the tail of any initial segment of the list may be changed to refer to another list. Crucially, any such alteration is shared among all uses of that list. (This behavior is in contrast to an *immutable* list, which can never change once created.) The usual way to implement a linked list is to specify that the tail of a list is not another list, but rather a *reference* to an assignable containing the tail of the list. The list structure is altered by setting the target assignable of the reference.

For this strategy to make sense, references must be mobile, and hence that assignables have indefinite extent—they must persist beyond the scope of their declaration. Assignables with indefinite extent are said to be *scope-free*, or simply *free*. In this chapter we consider a variation of Modernized Algol in which all assignables are free, and hence all types are mobile. The dynamics of this variation of Modernized Algol is significantly different from that given in Chapter 37 in that assignables are *heap-allocated*, rather than *stack-allocated*.

We also consider a further variation in which the distinction between commands and expressions is eliminated. This facilitates the use of *benign*

effects to achieve purely functional behavior using references. An example is a self-adjusting data structure that, externally, is a pure dictionary structure, but which internally makes use of mutation to rebalance itself.

38.1 Free Assignables

The statics of the language $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$ of free assignables is essentially the same as that of the language $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$, except that all types are regarded as mobile. To account for this liberalization of the statics, the dynamics must be generalized to relax the stack discipline and allow references to an assignable to escape its scope of declaration. The dynamics of $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$ is given by a transition system between states of the form $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$, in which a command, m, is executed relative to a memory, μ, that assigns values to the assignables declared in Σ. The signature, Σ, is only ever extended by transition; assignables are never deallocated.

The dynamics of $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\texttt{ret}(e) \parallel \mu\,\}\ \mathsf{final}} \tag{38.1a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\texttt{ret}(e) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{ret}(e') \parallel \mu\,\}} \tag{38.1b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\texttt{bnd}(e; x\,.\,m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{bnd}(e'; x\,.\,m) \parallel \mu\,\}} \tag{38.1c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\texttt{bnd}(\texttt{do}(\texttt{ret}(e)); x\,.\,m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,[e/x]m \parallel \mu\,\}} \tag{38.1d}$$

$$\frac{\nu\,\Sigma\,\{\,m_1 \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m_1' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\texttt{bnd}(\texttt{do}(m_1); x\,.\,m_2) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{bnd}(\texttt{do}(m_1'); x\,.\,m_2) \parallel \mu'\,\}} \tag{38.1e}$$

$$\frac{}{\nu\,\Sigma, a : \tau\,\{\,\texttt{get}[a] \parallel \mu \otimes \langle a : e\rangle\,\} \mapsto \nu\,\Sigma, a : \tau\,\{\,\texttt{ret}(e) \parallel \mu \otimes \langle a : e\rangle\,\}} \tag{38.1f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\texttt{set}[a]\,(e) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{set}[a]\,(e') \parallel \mu\,\}} \tag{38.1g}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a:\tau}}{\nu\,\Sigma, a : \tau\,\{\,\texttt{set}[a]\,(e) \parallel \mu \otimes \langle a : \_\rangle\,\} \mapsto \nu\,\Sigma, a : \tau\,\{\,\texttt{ret}(e) \parallel \mu \otimes \langle a : e\rangle\,\}} \tag{38.1h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a\,.\,m)\;\|\;\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{dcl}(e';a\,.\,m)\;\|\;\mu\,\}} \tag{38.1i}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a\,.\,m)\;\|\;\mu\,\}\mapsto\nu\,\Sigma,a:\tau\,\{\,m\;\|\;\mu\otimes\langle a:e\rangle\,\}} \tag{38.1j}$$

The most important difference is expressed by Rule (38.1j), which allows assignables to escape their scope of declaration.

38.2 Free References

References to assignables are values of type $\mathtt{ref}(\tau)$, where τ is the type of the contents of the underlying assignable. When all types are mobile, references may appear in data structures, may be stored assignables of reference type, and may be returned from commands, without restriction. For example, we may define the command $\mathtt{newref}[\tau](e)$ to stand for the command

$$\mathtt{dcl}\,a:=e\,\mathtt{in}\,\mathtt{ret}\,(\&\,a), \tag{38.2}$$

which allocates and initializes an assignable, and immediately returns a reference to it. Obviously the sensibility of this definition relies on the mobility of reference types, and the scope-free allocation of assignables.

The statics and dynamics of this construct may be derived from this definition. The following typing rule is admissible:

$$\frac{\Gamma\vdash_\Sigma e:\tau}{\Gamma\vdash_\Sigma \mathtt{newref}[\tau](e)\sim\mathtt{ref}(\tau)} \tag{38.3}$$

Moreover, the dynamics is given by the following rules:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e)\;\|\;\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e')\;\|\;\mu\,\}} \tag{38.4a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e)\;\|\;\mu\,\}\mapsto\nu\,\Sigma,a:\tau\,\{\,\mathtt{ret}(\mathtt{ref}[a])\;\|\;\mu\otimes\langle a:e\rangle\,\}} \tag{38.4b}$$

The dynamics of the `getref` and `setref` commands is essentially the same as in Chapter 37, but must be adapted to the setting of free assignables.

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{getref}(e)\;\|\;\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{getref}(e')\;\|\;\mu\,\}} \tag{38.5a}$$

$$\nu\,\Sigma\,\{\,\mathtt{getref(ref[}a\mathtt{])}\parallel\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{get[}a\mathtt{]}\parallel\mu\,\} \tag{38.5b}$$

$$\frac{e_1\underset{\Sigma}{\mapsto}e_1'}{\nu\,\Sigma\,\{\,\mathtt{setref(}e_1\mathtt{;}e_2\mathtt{)}\parallel\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{setref(}e_1'\mathtt{;}e_2\mathtt{)}\parallel\mu\,\}} \tag{38.5c}$$

$$\nu\,\Sigma\,\{\,\mathtt{setref(ref[}a\mathtt{]\,;}e_2\mathtt{)}\parallel\mu\,\}\mapsto\nu\,\Sigma\,\{\,\mathtt{set[}a\mathtt{](}e_2\mathtt{)}\parallel\mu\,\} \tag{38.5d}$$

Observe that the evaluation of expressions cannot alter or extend the memory, only commands may do this.

38.3 Safety

The proof of safety for free assignables and references is surprisingly tricky. The main difficulty is to account for the possibility of cyclic dependencies of data structures in the memory. The contents of one assignable may contain a reference to itself, or a reference to another assignable that contains a reference to it, and so forth. For example, consider the following procedure, e, of type $\mathtt{nat}\to\mathtt{nat\ cmd}$:

$$\lambda\,(x\mathtt{:nat.\,ifz}\,x\,\{\mathtt{z}\Rightarrow\mathtt{do}\,\{\mathtt{ret}\,(1)\}\mid\mathtt{s}(x')\Rightarrow\mathtt{do}\,\{f\leftarrow a\,;y\leftarrow\mathtt{call}\,f(x')\,;\mathtt{ret}\,(x*y)\}\}).$$

Let μ be a memory of the form $\mu'\otimes\langle a:e\rangle$ in which the contents of a contains, via the body of the procedure, a reference to a itself. Indeed, if the procedure e is called with a non-zero argument, it will "call itself" by indirect reference through a! (We will see in Section 38.4 on page 370 that such a situation can arise—the memory need not be "preloaded" for such cycles to arise.)

The possibility of cyclic dependencies means that some care in the definition of the judgement $\mu:\Sigma$ is required. The following rule defines the well-formed states:

$$\frac{\vdash_\Sigma m\sim\tau\quad\vdash_\Sigma\mu:\Sigma}{\nu\,\Sigma\,\{\,m\parallel\mu\,\}\ \mathsf{ok}} \tag{38.6}$$

The first premise of the rule states that the command m is well-formed relative to Σ. The second premise states that the memory, μ, conforms to Σ, *relative to the whole of* Σ so that cyclic dependencies are permitted. The judgement $\vdash_{\Sigma'}\mu:\Sigma$ is defined as follows:

$$\frac{\forall a:\rho\in\Sigma\quad\exists e\quad\mu(a)=e\text{ and }\vdash_{\Sigma'}e:\rho}{\vdash_{\Sigma'}\mu:\Sigma} \tag{38.7}$$

Theorem 38.1 (Preservation)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$*, then* $\vdash_\Sigma e' : \tau$*.*
2. *If* $\nu\,\Sigma\,\{\, m \parallel \mu \,\}$ *ok and* $\nu\,\Sigma\,\{\, m \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, m' \parallel \mu' \,\}$*, then* $\nu\,\Sigma'\,\{\, m' \parallel \mu' \,\}$ *ok.*

Proof. Simultaneously, by induction on transition. We prove the following stronger form of the second statement:

> If $\nu\,\Sigma\,\{\, m \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, m' \parallel \mu' \,\}$, where $\vdash_\Sigma m \sim \tau$, $\vdash_\Sigma \mu : \Sigma$, then Σ' extends Σ, and $\vdash_{\Sigma'} m' \sim \tau$, and $\vdash_{\Sigma'} \mu' : \Sigma'$.

Consider, for example, the transition

$$\nu\,\Sigma\,\{\, \mathtt{dcl}(e; a\,.\,m) \parallel \mu \,\} \mapsto \nu\,\Sigma, a : \rho\,\{\, m \parallel \mu \otimes \langle a : e \rangle \,\}$$

where e val_Σ. By assumption and inversion of Rule (37.7d) we have ρ such that $\vdash_\Sigma e : \rho$, $\vdash_{\Sigma, a:\rho} m \sim \tau$, and $\vdash_\Sigma \mu : \Sigma$. But since extension of Σ with a fresh assignable does not affect typing, we also have $\vdash_{\Sigma,a:\rho} \mu : \Sigma$ and $\vdash_{\Sigma,a:\rho} e : \rho$, from which it follows by Rule (38.7) that $\vdash_{\Sigma,a:\rho} \mu \otimes \langle a : e \rangle : \Sigma, a : \rho$.

The other cases follow a similar pattern, and are left as an exercise for the reader.

□

Theorem 38.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* e val_Σ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$*.*
2. *If* $\nu\,\Sigma\,\{\, m \parallel \mu \,\}$ *ok then either* $\nu\,\Sigma\,\{\, m \parallel \mu \,\}$ *final or* $\nu\,\Sigma\,\{\, m \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, m' \parallel \mu' \,\}$ *for some* Σ'*,* μ'*, and* m'*.*

Proof. Simultaneously, by induction on typing. For the second statement we prove the stronger form

> If $\vdash_\Sigma m \sim \tau$ and $\vdash_\Sigma \mu : \Sigma$, then either $\nu\,\Sigma\,\{\, m \parallel \mu \,\}$ final, or $\nu\,\Sigma\,\{\, m \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, m' \parallel \mu' \,\}$ for some Σ', μ', and m'.

Consider, for example, the typing rule

$$\frac{\Gamma \vdash_\Sigma e : \rho \quad \Gamma \vdash_{\Sigma, a:\rho} m \sim \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a\,.\,m) \sim \tau}$$

We have by the first inductive hypothesis that either e val_Σ or $e \underset{\Sigma}{\mapsto} e'$ for some e'. In the latter case we have by Rule (38.1i)

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a\,.\,m)\parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{dcl}(e'; a\,.\,m)\parallel \mu\,\}.$$

In the former case we have by Rule (38.1j) that

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a\,.\,m)\parallel \mu\,\} \mapsto \nu\,\Sigma, a:\rho\,\{\,m\parallel \mu\otimes\langle a:e\rangle\,\}.$$

As another example, consider the typing rule

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \mathtt{get}[a] \sim \tau}$$

By assumption $\vdash_{\Sigma,a:\tau} \mu : \Sigma, a:\tau$, and hence there exists e $\mathsf{val}_{\Sigma,a:\tau}$ such that $\mu = \mu' \otimes \langle a:e\rangle$ and $\vdash_{\Sigma,a:\tau} e:\tau$. By Rule (38.1f)

$$\nu\,\Sigma, a:\tau\,\{\,\mathtt{get}[a]\parallel \mu'\otimes\langle a:e\rangle\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,\mathtt{ret}(e)\parallel \mu'\otimes\langle a:e\rangle\,\},$$

as required. The other cases are handled similarly.

□

38.4 Integrating Commands and Expressions

The integral formulation of free references is given by the language $\mathcal{L}\{\mathtt{nat\ ref} \rightharpoonup\}$, which consolidates expressions and commands. The following rules are illustrative:

$$\frac{\Gamma \vdash_\Sigma e:\tau}{\Gamma \vdash_\Sigma \mathtt{do}(e):\mathtt{cmd}(\tau)} \tag{38.8a}$$

$$\frac{\Gamma \vdash_\Sigma e_1:\mathtt{cmd}(\tau_1) \quad \Gamma, x:\tau_1 \vdash_\Sigma e_2:\tau_2}{\Gamma \vdash_\Sigma \mathtt{bnd}(e_1; x\,.\,e_2):\tau_2} \tag{38.8b}$$

$$\frac{\Gamma \vdash_\Sigma e_1:\tau_1 \quad \Gamma \vdash_{\Sigma,a:\tau_1} e_2:\tau_2}{\Gamma \vdash_\Sigma \mathtt{dcl}(e_1; a\,.\,e_2):\tau_2} \tag{38.8c}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \mathtt{get}[a]:\tau} \tag{38.8d}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} e:\tau}{\Gamma \vdash_{\Sigma,a:\tau} \mathtt{set}[a](e):\tau} \tag{38.8e}$$

The dynamics of the integral formulation of mutation is defined as a transition system between states of the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is an expression involving the assignables declared in Σ, and μ is a memory providing values for each of these assignables. It is a straightforward exercise to reformulate Rules (38.1) to eliminate the mode distinction between commands and expressions. Rules (38.5) may similarly be adapted to the integral setting.

The modal and integral formulations of references have complementary strengths and weaknesses. The chief virtue of the modal formulation is that the use of assignment is confined to commands, leaving expressions as pure computations. One consequence is that typing judgements for expressions retain their force even in the presence of references to free assignables, so that the type $\texttt{unit} \rightharpoonup \texttt{unit}$ contains only the identity and the divergent functions, and the type $\texttt{nat} \rightharpoonup \texttt{nat}$ consists solely of partial functions on the natural numbers. By contrast the integral formulation enjoys none of these properties. Any expression may alter or allocate new assignables, and the semantics of typing assertions is therefore significantly weakened compared to the modal formulation. In particular, the type $\texttt{unit} \rightharpoonup \texttt{unit}$ contains infinitely many distinct functions, and the type $\texttt{nat} \rightharpoonup \texttt{nat}$ contains procedures that in no way represent partial functions because they retain state across calls.

While the modal separation of pure expressions from impure commands may seem like an unalloyed good, the situation is actually more complex. The central problem is that the modal formulation inhibits the use of effects to implement purely functional behavior. For example, a self-adjusting tree, such as a splay tree, uses in-place mutation to provide an efficient implementation of what is otherwise a purely functional dictionary structure mapping keys to values. This is an example of a *benign effect*, one that does not affect the behavior, but only the efficiency, of the implementation.

Many other examples arise in practice. For example, suppose that we wish to instrument an otherwise pure functional program with code to collect execution statistics for profiling. In the integral setting it is a simple matter to allocate free assignables that contain profiling information collected by assignments that update their contents at critical points in the program. In the modal setting, however, we must globally restructure the program to transform it from a pure expression to an impure command. Another example is provided by the technique of *backpatching* for implementing recursion using a free assignable, which we now describe in more detail.

In the integral formulation we may implement the factorial function using backpatching as follows:

```
dcl a := λn:nat.0 in
  { f ← λn:nat.ifz(n, 1, n'.n * a(n'))
  ; _ ← a := f
  ; f
  }
```

wherein we have used the concrete syntax for commands introduced in Chapter 37. Observe that the assignable a is used as an expression standing for its contents (that is, it stands for the abstract syntax `get[`a`]`).

This expression returns a function of type `nat` $\rightharpoonup$ `nat` that is obtained by (a) allocating a free assignable initialized arbitrarily (and immaterially) 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 assignable, (c) assigning this function to the assignable, and (d) returning that function. The result is a value of function type that uses an assignable "under the hood" in a manner not visible to its clients.

In contrast the modal formulation forces us to make explicit the reliance on private state.

```
dcl a := λn:nat.do{ret 0} in
  { f ← ret (λ n:nat. ...)
  ; _ ← a := f
  ; ret f
  }
```

where the elided procedure body is as follows:

```
ifz(n,do{ret(1)},n'.do{f←a; x←run(f(n')); ret (n*x)}).
```

Each branch of the conditional test returns a command. In the case that the argument is zero, the command simply returns the value `1`. Otherwise, it fetches the contents of the assignable, calls it on the predecessor, and returns the result of multiplying this by the argument.

The modal implementation of factorial is a command (not an expression) of type `nat` $\rightarrow$ `(nat cmd)`, which exposes two properties of the backpatching implementation:

1. The command that builds the recursive factorial function is impure, because it allocates and assigns to the assignable used to implement backpatching.

2. The body of the factorial function is impure, because it accesses the assignable to effect the recursive call.

As a result the factorial function (so implemented) may no longer be used as a function, but must instead be called as a procedure. For example, to compute the factorial of n, we must write

$\{$ `f` $\leftarrow$ *fact*`; x` $\leftarrow$ `run (f(`n`)); return x` $\}$

where *fact* stands for the command implementing factorial given above. The factorial procedure is bound to a variable, which is then applied to yield an encapsulated command that, when activated, computes the desired result.

These examples illustrate that exposing the reliance on effects in the type system is both a boon and a bane. Under the integral formulation a "boring" type such as `unit` $\rightharpoonup$ `unit` can have very "interesting" behavior—for example, it may depend on or alter the contents of an assignable, or may allocate new assignables. Under the modal formulation a value of such a boring type is "uninteresting": it can only be the identity or the divergent function. An interesting function must have an interesting type such as `unit` $\rightharpoonup$ `unit cmd`, which makes clear that the body of the function engenders storage effects. On the other hand, as the example of backpatching makes clear, the integral formulation allows one to think of types as descriptions of *behavior*, rather than descriptions of *implementation*. The factorial function, whether implemented using backpatching or not, is a pure function of type `nat` $\rightharpoonup$ `nat`. The reliance on assignment is an implementation detail that remains hidden from the caller. The modal formulation, however, exposes the reliance on effects in both the definition and implementation of the factorial function, and hence forces it to be treated as an imperative procedure, rather than a pure function.

38.5 Notes

The separation of the allocation method (stack *versus* heap) for assignables from the possibility of creating references to them simplifies the semantics and clarifies the relation between the two concepts. Standard ML [67] introduced the concept of references to heap-allocated assignables (called *locations*) as a means of supporting mutable data structures. The formulation considered here is inspired by Wright and Felleisen [105] and the author [40]. The concept of scope extrusion is borrowed from the π-calculus [66] to formalize the distinction between stack- and heap-allocation of assignables.

Part XV

Laziness

Chapter 39

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* evaluation order 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.

Another aspect of laziness is the use of *general recursion* to define self-referential computations, including recursive functions. The role of laziness in this setting is to defer evaluation of any self-reference until it is actually required for a computation.

Traditionally, languages are classified into one of two categories. *Lazy languages* use a call-by-need interpretation of function application, impose a lazy evaluation strategy for data structures, and allow unrestricted use of general recursion. *Strict languages* take the opposite positions: call-by-value for function application, eager evaluation of data structures, and limitations on general recursion (typically, to functions). More recently, however, language designers have come to realize that it is not *whole languages* that

should be lazy or strict, but rather that the type system should distinguish lazy and strict evaluation order. In its most basic form this only requires the introduction of a type whose values are suspended computations that are evaluated by-need. (A more sophisticated approach is the subject of Chapter 40.)

39.1 Need Dynamics

The distinguishing feature of call-by-need is the use of *memoization* to record the value of an expression whenever it is computed so that, should the value of that expression ever be required again, the stored value can be returned without recomputing it. This is achieved by augmenting the computation state with a *memo table* that associates an expression (not necessarily a value) to each of a finite set of symbols. The symbols serve as *names* of the expressions to which they are associated by the memo table. Whenever the value of a name is required, the associated expression is evaluated and its value is both stored in the memo table under the same name and returned as the value of that name. This ensures that any subsequent evaluation of the same name returns the new value without recomputing it.

Another perspective on call-by-need is that it uses names to mediate *sharing* among multiple occurrences of a sub-expression within a larger expression. Ordinary substitution often replicates an expression, generating one copy for each occurrence of the target of the substitution. Under call-by-need each expression is given a name which serves as a proxy for it. In particular, expression names are substituted for variables so that all occurrences have the same name and hence refer to the same copy of the expression to which it is associated. In this way we economize on both the time required to evaluate the expression, which would be needlessly repeated under call-by-name, and the space required to store it during computation, which would be replicated under call-by-name.

The need dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is based on a transition system with states of the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where Σ is a finite set of hypotheses $a_1 : \tau_1, \ldots, a_n : \tau_n$ associating types to names, e is an expression that may involve the names in Σ, and μ maps each name declared in Σ to either an expression or a special symbol, $\bullet$, called the *black hole*. (The role of the black hole will be made clear below.)

The call-by-need dynamics consists of the following two forms of judgement:

1. $e\ \mathsf{val}_\Sigma$, stating that e is a value that may involve the names in Σ.

2. $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$, stating that one step of evaluation of the expression e relative to memo table μ with the names declared in Σ results in the expression e' relative to the memo table μ' with names declared in Σ'.

The dynamics is defined so that the collection of active names grows monotonically, and so that the type of a name never changes. The memo table may be altered destructively during execution to reflect progress in the evaluation of the expression associated with a given name.

The judgement $e\ \mathsf{val}_\Sigma$ is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}_\Sigma} \tag{39.1a}$$

$$\frac{}{\mathtt{s}(a)\ \mathsf{val}_{\Sigma,a:\mathtt{nat}}} \tag{39.1b}$$

$$\frac{}{\mathtt{lam}[\tau]\,(x\,.\,e)\ \mathsf{val}_\Sigma} \tag{39.1c}$$

Rules (39.1a) through (39.1c) specify that $\mathtt{z}$ is a value, any expression of the form $\mathtt{s}(a)$, where a is a name, is a value, and that any λ-abstraction, possibly containing names, is a value. It is important that names themselves are not values, rather they stand for (possibly unevaluated) expressions as specified by the memo table.

The initial and final states of evaluation are defined as follows:

$$\frac{}{\nu\,\varnothing\,\{\,e \parallel \varnothing\,\}\ \mathsf{initial}} \tag{39.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{39.2b}$$

Rule (39.2a) specifies that an initial state consists of an expression evaluated relative to an empty memo table. Rule (39.2b) specifies that a final state has the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is a value relative to Σ.

The transition judgement for the call-by-need dynamics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a:e\rangle\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a:e\rangle\,\}} \tag{39.3a}$$

$$\frac{\nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a:\bullet\rangle\,\} \mapsto \nu\,\Sigma', a:\tau\,\{\,e' \parallel \mu' \otimes \langle a:\bullet\rangle\,\}}{\nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a:e\rangle\,\} \mapsto \nu\,\Sigma', a:\tau\,\{\,a \parallel \mu' \otimes \langle a:e'\rangle\,\}} \tag{39.3b}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{s}(e) \parallel \mu\,\} \mapsto \nu\,\Sigma, a:\mathtt{nat}\,\{\,\mathtt{s}(a) \parallel \mu \otimes \langle a:e\rangle\,\}} \tag{39.3c}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{ifz}(e;e_0;x.e_1) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{ifz}(e';e_0;x.e_1) \parallel \mu'\,\}} \tag{39.3d}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,e_0 \parallel \mu\,\}} \tag{39.3e}$$

$$\frac{}{\left\{\begin{array}{c}\nu\,\Sigma, a:\mathtt{nat}\,\{\,\mathtt{ifz}(\mathtt{s}(a);e_0;x.e_1) \parallel \mu \otimes \langle a:e\rangle\,\} \\ \mapsto \\ \nu\,\Sigma, a:\mathtt{nat}\,\{\,[a/x]e_1 \parallel \mu \otimes \langle a:e\rangle\,\}\end{array}\right\}} \tag{39.3f}$$

$$\frac{\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e_1' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{ap}(e_1;e_2) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{ap}(e_1';e_2) \parallel \mu'\,\}} \tag{39.3g}$$

$$\frac{}{\left\{\begin{array}{c}\nu\,\Sigma\,\{\,\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \parallel \mu\,\} \\ \mapsto \\ \nu\,\Sigma, a:\tau\,\{\,[a/x]e \parallel \mu \otimes \langle a:e_2\rangle\,\}\end{array}\right\}} \tag{39.3h}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{fix}[\tau](x.e) \parallel \mu\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a:[a/x]e\rangle\,\}} \tag{39.3i}$$

Rule (39.3a) governs a name whose associated expression is a value; the value of the name is the value associated to that name in the memo table. Rule (39.3b) specifies that if the expression associated to a name is not a value, then it is evaluated "in place" until such time as Rule (39.3a) applies. This is achieved by switching the focus of evaluation to the associated expression, while at the same time associating the *black hole* to that name. The black hole represents the absence of a value for that name, so that any attempt to access it during evaluation of its associated expression cannot make progress. This signals a circular dependency that, if not caught using

a black hole, would initiate an infinite regress. We may therefore think of the black hole as catching a particular form of non-termination that arises when the value of an expression associated to a name depends on the name itself.

Rule (39.3c) specifies that evaluation of $\mathtt{s}(e)$ allocates a fresh name, a, for the expression e, and yields the value $\mathtt{s}(a)$. The value of e is not determined until such time as the predecessor is required in a subsequent computation. This implements a lazy dynamics for the successor. Rule (39.3f), which governs a conditional branch on a successor, substitutes the name, a, for the variable, x, when computing the predecessor of a non-zero number, ensuring that all occurrences of x share the same predecessor computation.

Rule (39.3g) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (39.3h) specifies that to evaluate an application of a λ-abstraction we allocate a fresh name for the argument, and substitute this name for the parameter of the function. The argument is evaluated only if it is needed in the subsequent computation, and then that value is shared among all occurrences of the parameter in the body of the function.

General recursion is implemented by Rule (39.3i). Recall from Chapter 12 that the expression $\mathtt{fix}[\tau](x.e)$ stands for the solution of the recursion equation $x = e$, where x may occur within e. Rule (39.3i) computes this solution by associating a fresh name, a, with the body, e, substituting a for x within e to effect the self-reference. It is this substitution that permits a named expression to depend on its own name. For example, the expression $\mathtt{fix}\, x{:}\tau\, \mathtt{is}\, x$ associates the expression a to a in the memo table, and returns a. The next step of evaluation is stuck, because it seeks to evaluate a with a bound to the black hole. In contrast an expression such as $\mathtt{fix}\, f{:}\rho \to \tau\, \mathtt{is}\, \lambda\, (x{:}\rho.\, e)$ does not get stuck, because the self-reference is "hidden" within the λ-abstraction, and hence need not be evaluated to determine the value of the binding.

39.2 Safety

We write $\Gamma \vdash_\Sigma e : \tau$ to mean that e has type τ under the assumptions Γ, treating names declared in Σ as expressions of their associated type. The rules as as in Chapter 12, with the addition of the following rule for names:

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} a : \tau} \tag{39.4}$$

This rule is in contrast to the use of symbols in Chapter 37, for example, because the lazy semantics implicitly allocates names, and fetches and stores bindings for them during execution; the programmer is not permitted to perform these operations directly.

The judgement $\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{ok}$ is defined by the following rules:

$$\frac{\vdash_\Sigma e : \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{ok}} \tag{39.5a}$$

$$\frac{\forall a : \tau \in \Sigma \quad \mu(a) = e \neq \bullet \Longrightarrow \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{39.5b}$$

Rule (39.5b) permits self-reference through the memo table by allowing the expression associated to a name, a, to contain a, or, more generally, to contain a name whose associated expression contains a, and so on through any finite chain of such dependencies. Moreover, a name that is bound to the "black hole" is deemed to be of any type.

Theorem 39.1 (Preservation)**.** *Suppose that* $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$*. If* $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ *ok, then* $\nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$ *ok.*

Proof. We prove by induction on Rules (39.3) that if $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$ and $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e' : \tau$.

Consider Rule (39.3b), for which we have $e = e' = a$, $\mu = \mu_0 \otimes \langle a : e_0 \rangle$, $\mu' = \mu_0' \otimes \langle a : e_0' \rangle$, and

$$\nu\,\Sigma, a : \tau\,\{\,e_0 \parallel \mu_0 \otimes \langle a : \bullet \rangle\,\} \mapsto \nu\,\Sigma', a : \tau\,\{\,e_0' \parallel \mu_0' \otimes \langle a : \bullet \rangle\,\}.$$

Assume that $\vdash_{\Sigma, a:\tau} \mu : \Sigma, a : \tau$. It follows that $\vdash_{\Sigma, a:\tau} e_0 : \tau$ and $\vdash_{\Sigma, a:\tau} \mu_0 : \Sigma$, and hence that

$$\vdash_{\Sigma, a:\tau} \mu_0 \otimes \langle a : \bullet \rangle : \Sigma, a : \tau.$$

We have by induction that $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma', a:\tau} e_0' : \tau'$ and

$$\vdash_{\Sigma', a:\tau} \mu_0 \otimes \langle a : \bullet \rangle : \Sigma, a : \tau.$$

But then

$$\vdash_{\Sigma', a:\tau} \mu' : \Sigma', a : \tau,$$

which suffices for the result.

Consider Rule (39.3g), so that e is the application $\mathtt{ap}(e_1; e_2)$ and

$$\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e_1' \parallel \mu'\,\}.$$

Suppose that $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$. By inversion of typing $\vdash_\Sigma e_1 : \tau_2 \to \tau$ for some type τ_2 such that $\vdash_\Sigma e_2 : \tau_2$. By induction $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e_1' : \tau_2 \to \tau$. By weakening we have $\vdash_{\Sigma'} e_2 : \tau_2$, so that $\vdash_{\Sigma'} \mathtt{ap}(e_1'; e_2) : \tau$, which is enough for the result.

□

The statement of the progress theorem allows for the possibility of encountering a black hole, representing a checkable form of non-termination. The judgement $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ loops, stating that e diverges by virtue of encountering the black hole, is defined by the following rules:

$$\frac{}{\nu\,\Sigma, a : \tau\,\{\,a \parallel \mu \otimes \langle a : \bullet \rangle\,\}\ \mathsf{loops}} \tag{39.6a}$$

$$\frac{\nu\,\Sigma, a : \tau\,\{\,e \parallel \mu \otimes \langle a : \bullet \rangle\,\}\ \mathsf{loops}}{\nu\,\Sigma, a : \tau\,\{\,a \parallel \mu \otimes \langle a : e \rangle\,\}\ \mathsf{loops}} \tag{39.6b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{ifz}(e; e_0; x.e_1) \parallel \mu\,\}\ \mathsf{loops}} \tag{39.6c}$$

$$\frac{\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{ap}(e_1; e_2) \parallel \mu\,\}\ \mathsf{loops}} \tag{39.6d}$$

Theorem 39.2 (Progress)**.** *If $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ ok, then either $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ final, or $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ loops, or there exists μ' and e' such that $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$.*

Proof. We proceed by induction on the derivations of $\vdash_\Sigma e : \tau$ and $\vdash_\Sigma \mu : \Sigma$ implicit in the derivation of $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ ok.

Consider Rule (12.1a), where the variable, a, is declared in Σ. Thus $\Sigma = \Sigma_0, a : \tau$ and $\vdash_\Sigma \mu : \Sigma$. It follows that $\mu = \mu_0 \otimes \langle a : e_0 \rangle$ with $\vdash_\Sigma \mu_0 : \Sigma_0$ and $\vdash_\Sigma e_0 : \tau$. Note that $\vdash_\Sigma \mu_0 \otimes \langle a : \bullet \rangle : \Sigma$. Applying induction to the derivation of $\vdash_\Sigma e_0 : \tau$, we consider three cases:

1. $\nu\,\Sigma\,\{\,e_0 \parallel \mu \otimes \langle a : \bullet \rangle\,\}$ final. By inversion of Rule (39.2b) we have $e_0\ \mathsf{val}_\Sigma$, and hence by Rule (39.3a) we obtain $\nu\,\Sigma\,\{\,a \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,e_0 \parallel \mu\,\}$.
2. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes \langle a : \bullet \rangle\,\}$ loops. By applying Rule (39.6b) we obtain $\nu\,\Sigma\,\{\,a \parallel \mu\,\}$ loops.
3. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes \langle a : \bullet \rangle\,\} \mapsto \nu\,\Sigma'\,\{\,e_0' \parallel \mu_0' \otimes \langle a : \bullet \rangle\,\}$. By applying Rule (39.3b) we obtain
$$\nu\,\Sigma\,\{\,a \parallel \mu \otimes \langle a : e_0 \rangle\,\} \mapsto \nu\,\Sigma'\,\{\,a \parallel \mu' \otimes \langle a : e_0' \rangle\,\}.$$

□

39.3 Lazy Data Structures

The call-by-need dynamics extends to product, sum, and recursive types in a straightforward manner. For example, the need dynamics of lazy product types is given by the following rules:

$$\frac{}{\mathtt{pair}(a_1;a_2)\ \mathsf{val}_{\Sigma,a_1:\tau_1,a_2:\tau_2}} \tag{39.7a}$$

$$\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\mathtt{pair}(e_1;e_2)\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\mathtt{pair}(a_1;a_2)\parallel\mu\otimes\langle a_1:e_1\rangle\otimes\langle a_2:e_2\rangle\,\} \end{array}\right\} \tag{39.7b}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,e'\parallel\mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{proj}[\mathtt{l}]\,(e)\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,\mathtt{proj}[\mathtt{l}]\,(e')\parallel\mu'\,\}} \tag{39.7c}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{proj}[\mathtt{l}]\,(e)\parallel\mu\,\}\ \mathsf{loops}} \tag{39.7d}$$

$$\left\{\begin{array}{c} \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\mathtt{proj}[\mathtt{l}]\,(\mathtt{pair}(a_1;a_2))\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,a_1\parallel\mu\,\} \end{array}\right\} \tag{39.7e}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,e'\parallel\mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{proj}[\mathtt{r}]\,(e)\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,\mathtt{proj}[\mathtt{r}]\,(e')\parallel\mu'\,\}} \tag{39.7f}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{proj}[\mathtt{r}]\,(e)\parallel\mu\,\}\ \mathsf{loops}} \tag{39.7g}$$

$$\left\{\begin{array}{c} \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\mathtt{proj}[\mathtt{r}]\,(\mathtt{pair}(a_1;a_2))\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,a_2\parallel\mu\,\} \end{array}\right\} \tag{39.7h}$$

A pair is considered a value only if its arguments are names (Rule (39.7a)), which are introduced when the pair is created (Rule (39.7b)). The first and second projections evaluate to one or the other name in the pair, inducing a demand for the value of that component (Rules (39.7e) and (39.7h)).

Using similar techniques we may give a need dynamics to sums and recursive types. We leave the formalization of these as an exercise for the reader.

39.4 Suspensions

Another way to introduce laziness is to consolidate the machinery of the by-need dynamics into a single type whose values are possibly unevaluated, memoized computations. The type of *suspensions* of type τ, written $\tau\,\mathtt{susp}$, has as introductory form $\mathtt{susp}\,x : \tau\,\mathtt{is}\,e$ representing the suspended, possibly self-referential, computation, e, of type τ, and as eliminatory form the operation $\mathtt{force}(e)$ that evaluates the suspended computation presented by e, records the value in a memo table, and returns that value as result.

Using suspension types we may construct other lazy types according to our needs in a particular program. For example, the type of lazy pairs with components of type τ_1 and τ_2 is expressible as the type

$$\tau_1\,\mathtt{susp} \times \tau_2\,\mathtt{susp}$$

and the type of call-by-need functions with domain τ_1 and range τ_2 is expressible as the type

$$\tau_1\,\mathtt{susp} \to \tau_2.$$

We may also express more complex combinations of eagerness and laziness, such as the type of "lazy lists" consisting of computations that, when forced, evaluate either to the empty list, or a non-empty list consisting of a natural number and another lazy list:

$$\mu t.(\mathtt{unit} + (\mathtt{nat} \times t))\,\mathtt{susp}.$$

This type should be contrasted with the type

$$\mu t.(\mathtt{unit} + (\mathtt{nat} \times t\,\mathtt{susp}))$$

whose values are the empty list and a pair consisting of a natural number and a computation of another such value.

The syntax of suspensions is given by the following grammar:

Typ	τ	::=	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$	suspension
Exp	e	::=	$\mathtt{susp}[\tau](x.e)$	$\mathtt{susp}\,x:\tau\,\mathtt{is}\,e$	delay
			$\mathtt{force}(e)$	$\mathtt{force}(e)$	force
			$\mathtt{susp}[a]$	$\mathtt{susp}[a]$	self-reference

Suspensions are self-referential; the bound variable, x, refers to the suspension itself. The expression $\mathtt{susp}[a]$ is a reference to the suspension named a.

The statics of the suspension type is given using a judgement of the form $\Gamma \vdash_\Sigma e : \tau$, where Σ assigns types to the names of suspensions. It is defined by the following rules:

$$\frac{\Gamma, x : \mathtt{susp}(\tau) \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{susp}[\tau](x.e) : \mathtt{susp}(\tau)} \tag{39.8a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{susp}(\tau)}{\Gamma \vdash_\Sigma \mathtt{force}(e) : \tau} \tag{39.8b}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} \mathtt{susp}[a] : \mathtt{susp}(\tau)} \tag{39.8c}$$

Rule (39.8a) checks that the expression, e, has type τ under the assumption that x, which stands for the suspension itself, has type $\mathtt{susp}(\tau)$.

The by-need dynamics of suspensions is defined by the following rules:

$$\frac{}{\mathtt{susp}[a]\ \mathsf{val}_{\Sigma, a:\tau}} \tag{39.9a}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\mathtt{susp}[\tau](x.e) \parallel \mu\,\} \\ \mapsto \\ \nu\,\Sigma, a:\tau\,\{\,\mathtt{susp}[a] \parallel \mu \otimes \langle a : [a/x]e\rangle\,\} \end{array}\right\}} \tag{39.9b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{force}(e) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{force}(e') \parallel \mu'\,\}} \tag{39.9c}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a:\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a:\tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes \langle a : e\rangle\,\} \\ \mapsto \\ \nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a : e\rangle\,\} \end{array}\right\}} \tag{39.9d}$$

$$\frac{\begin{array}{c} \nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a : \bullet\rangle\,\} \\ \mapsto \\ \nu\,\Sigma', a:\tau\,\{\,e' \parallel \mu' \otimes \langle a : \bullet\rangle\,\} \end{array}}{\left\{\begin{array}{c} \nu\,\Sigma, a:\tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes \langle a : e\rangle\,\} \\ \mapsto \\ \nu\,\Sigma', a:\tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu' \otimes \langle a : e'\rangle\,\} \end{array}\right\}} \tag{39.9e}$$

Rule (39.9a) specifies that a reference to a suspension is a value. Rule (39.9b) specifies that evaluation of a delayed computation consists of allocating

a fresh name for it in the memo table, and returning a reference to that suspension. Rules (39.9c) to (39.9e) specify that demanding the value of a suspension forces evaluation of the suspended computation, which is then stored in the memo table and returned as result.

39.5 Notes

The by-need dynamics given here is inspired by Ariola and Felleisen's formulation [7], but with the crucial distinction that by-need cells are *not* variables, but rather assignables (to which there is at most one assignment). Contrarily, variables are given meaning by substitution, the very concept that call-by-need seeks to restrict!

Chapter 40

Polarization

Up to this point we have frequently encountered arbitrary choices in the dynamics of various language constructs. For example, when specifying the dynamics of pairs, we must choose, rather arbitrarily, between the *lazy* dynamics, in which all pairs are values regardless of the value status of their components, and the *eager* dynamics, in which a pair is a value only if its components are both values. We could even consider a *half-eager* (or, equivalently, *half-lazy*) dynamics, in which a pair is a value only if, say, the first component is a value, but without regard to the second.

Similar questions arise with sums (all injections are values, or only injections of values are values), recursive types (all folds are values, or only folds of values are values), and function types (functions should be called by-name or by-value). Whole languages are built around adherence to one policy or another. For example, Haskell decrees that products, sums, and recursive types are to be lazy, and functions are to be called by name, whereas ML decrees the exact opposite policy. Not only are these choices arbitrary, but it is also unclear why they should be linked. For example, one could very sensibly decree that products, sums, and recursive types are lazy, yet impose a call-by-value discipline on functions. Or one could have eager products, sums, and recursive types, yet insist on call-by-name. It is not at all clear which of these points in the space of choices is right; each has its adherents, and each has its detractors.

Are we therefore stuck in a tarpit of subjectivity? No! The way out is to recognize that these distinctions should not be imposed by the language designer, but rather are choices that are to be made by the programmer. This may be achieved by recognizing that differences in dynamics reflect fundamental *type distinctions* that are being obscured by languages that im-

pose one policy or another. We can have both eager and lazy pairs in the same language by simply distinguishing them as two distinct types, and similarly we can have both eager and lazy sums in the same language, and both by-name and by-value function spaces, by providing sufficient type distinctions as to make the choice available to the programmer.

Eager and lazy types are distinguished by their *polarity*, which is either *positive* or *negative* according to whether the type is defined by the values that inhabit the type or the behavior of expressions of that type. Postive types are *eager*, or *inductive*, in that they are defined by their values. Negative types are *lazy*, or *coinductive*, in that they are defined by the behavior of their elements. The polarity of types is made explicit using a technique called called *focusing*, or *focalization*. A focused presentation of a programming language distinguishes three general forms of expression, *(positive and negative) values*, *(positive and negative) continuations*, and *(neutral) computations*.

40.1 Positive and Negative Types

Polarization consists of distinguishing positive from negative types according to the following two principles:

1. A positive type is defined by its introduction rules, which specify the *values* of that type in terms of other values. The elimination rules are *inversions* that specify a computation by pattern matching on values of that type.

2. A negative type is defined by its elimination rules, which specify the *observations* that may be performed on elements of that type. The introduction rules specify the *values* of that type by specifying how they respond to observations.

Based on this characterization we can anticipate that the type of natural numbers would be positive, since it is defined by zero and successor, whereas function types would be negative, since they are characterized by their behavior when applied, and not by their internal structure.

The language $\mathcal{L}^{\pm}\{\mathtt{nat} \rightharpoonup\}$ is a polarized formulation of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The syntax of types in this language is by the following grammar:

PTyp	τ^+	$::=$	$\mathtt{dn}(\tau^-)$	$\downarrow\tau^-$	suspension
			$\mathtt{nat}$	$\mathtt{nat}$	naturals
NTyp	τ^-	$::=$	$\mathtt{up}(\tau^+)$	$\uparrow\tau^+$	inclusion
			$\mathtt{parr}(\tau_1^+;\tau_2^-)$	$\tau_1^+ \rightharpoonup \tau_2^-$	partial function

The types $\downarrow\tau^-$ and $\uparrow\tau^+$ effect a *polarity shift* from negative to positive and positive to negative, respectively. Intuitively, the shifted type $\uparrow\tau^+$ is just the inclusion of positive into negative values, whereas the shifted type $\downarrow\tau^-$ represents the type of suspended computations of negative type.

The domain of the negative function type is required to be positive, but its range is negative. This allows us to form right-iterated function types

$$\tau_1^+ \rightharpoonup (\tau_2^+ \rightharpoonup (\ldots (\tau_{n-1}^+ \rightharpoonup \tau_n^-)))$$

directly, but to form a left-iterated function type requires shifting,

$$\downarrow(\tau_1^+ \rightharpoonup \tau_2^-) \rightharpoonup \tau^-,$$

to turn the negative function type into a positive type. Conversely, shifting is needed to define a function whose range is positive, $\tau_1^+ \rightharpoonup \uparrow\tau_2^+$.

40.2 Focusing

The syntax of $\mathcal{L}^{\pm}\{\mathtt{nat}\rightharpoonup\}$ is motivated by the polarization of its types. For each polarity we have a sort of values and a sort of continuations with which we may create (neutral) computations.

PVal	v^+	::=	$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(v^+)$	$\mathtt{s}(v^+)$	successor
			$\mathtt{del}^-(e)$	$\mathtt{del}^-(e)$	delay
PCont	k^+	::=	$\mathtt{ifz}(e_0; x.e_1)$	$\mathtt{ifz}(e_0; x.e_1)$	conditional
			$\mathtt{force}^-(k^-)$	$\mathtt{force}^-(k^-)$	evaluate
NVal	v^-	::=	$\mathtt{lam}[\tau^+](x.e)$	$\lambda\,(x{:}\tau^+.e)$	abstraction
			$\mathtt{del}^+(v^+)$	$\mathtt{del}^+(v^+)$	inclusion
			$\mathtt{fix}(x.v^-)$	$\mathtt{fix}\,x\,\mathtt{is}\,v^-$	recursion
NCont	k^-	::=	$\mathtt{ap}(v^+; k^-)$	$\mathtt{ap}(v^+; k^-)$	application
			$\mathtt{force}^+(x.e)$	$\mathtt{force}^+(x.e)$	evaluate
Comp	e	::=	$\mathtt{ret}(v^-)$	$\mathtt{ret}(v^-)$	return
			$\mathtt{cut}^+(v^+; k^+)$	$v^+ \triangleright k^+$	cut
			$\mathtt{cut}^-(v^-; k^-)$	$v^- \triangleright k^-$	cut

The positive values include the numerals, and the negative values include functions. In addition we may delay a computation of a negative value to form a positive value using $\mathtt{del}^-(e)$, and we may consider a positive value to be a negative value using $\mathtt{del}^+(v^+)$. The positive continuations include

the conditional branch, sans argument, and the negative continuations include application sites for functions consisting of a positive argument value and a continuation for the negative result. In addition we include positive continuations to force the computation of a suspended negative value, and to extract an included positive value. Computations, which correspond to machine states, consist of returned negative values (these are final states), states passing a positive value to a positive continuation, and states passing a negative value to a negative continuation. General recursion appears as a form of negative value; the recursion is unrolled when it is made the subject of an observation.

40.3 Statics

The statics of $\mathcal{L}^{\pm}\{\mathtt{nat} \rightharpoonup\}$ consists of a collection of rules for deriving judgements of the following forms:

- Positive values: $\Gamma \vdash v^+ : \tau^+$.
- Positive continuations: $\Gamma \vdash k^+ : \tau^+ > \gamma^-$.
- Negative values: $\Gamma \vdash v^- : \tau^-$.
- Negative continuations: $\Gamma \vdash k^- : \tau^- > \gamma^-$.
- Computations: $\Gamma \vdash e : \gamma^-$.

Throughout Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1^+, \ldots, x_n : \tau_n^+,$$

for some $n \geq 0$, and γ^- is any negative type.

The typing rules for continuations specify both an argument type (on which values they act) and a result type (of the computation resulting from the action on a value). The typing rules for computations specify that the outcome of a computation is a negative type. All typing judgements specify that variables range over positive types. (These restrictions may always be met by appropriate use of shifting.)

The statics of positive values consists of the following rules:

$$\frac{}{\Gamma, x : \tau^+ \vdash x : \tau^+} \tag{40.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{40.1b}$$

$$\frac{\Gamma \vdash v^+ : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(v^+) : \mathtt{nat}} \tag{40.1c}$$

$$\frac{\Gamma \vdash e : \tau^-}{\Gamma \vdash \mathtt{del}^-(e) : \downarrow \tau^-} \tag{40.1d}$$

Rule (40.1a) specifies that variables range over positive values. Rules (40.1b) and (40.1c) specify that the values of type `nat` are just the numerals. Rule (40.1d) specifies that a suspended computation (necessarily of negative type) is a positive value.

The statics of positive continuations consists of the following rules:

$$\frac{\Gamma \vdash e_0 : \gamma^- \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \gamma^-}{\Gamma \vdash \mathtt{ifz}(e_0; x.e_1) : \mathtt{nat} > \gamma^-} \tag{40.2a}$$

$$\frac{\Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash \mathtt{force}^-(k^-) : \downarrow \tau^- > \gamma^-} \tag{40.2b}$$

Rule (40.2a) governs the continuation that chooses between two computations according to whether a natural number is zero or non-zero. Rule (40.2b) specifies the continuation that forces a delayed computation with the specified negative continuation.

The statics of negative values is defined by these rules:

$$\frac{\Gamma, x : \tau_1^+ \vdash e : \tau_2^-}{\Gamma \vdash \lambda\,(x{:}\tau_1^+.e) : \tau_1^+ \rightharpoonup \tau_2^-} \tag{40.3a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+}{\Gamma \vdash \mathtt{del}^+(v^+) : \uparrow \tau^+} \tag{40.3b}$$

$$\frac{\Gamma, x : \downarrow \tau^- \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{fix}\, x\, \mathtt{is}\, v^- : \tau^-} \tag{40.3c}$$

Rule (40.3a) specifies the statics of a λ-abstraction whose argument is a positive value, and whose result is a computation of negative type. Rule (40.3b) specifies the inclusion of positive values as negative values. Rule (40.3c) specifies that negative types admit general recursion.

The statics of negative continuations is defined by these rules:

$$\frac{\Gamma \vdash v_1^+ : \tau_1^+ \quad \Gamma \vdash k_2^- : \tau_2^- > \gamma^-}{\Gamma \vdash \mathtt{ap}(v_1^+; k_2^-) : \tau_1^+ \rightharpoonup \tau_2^- > \gamma^-} \tag{40.4a}$$

$$\frac{\Gamma, x : \tau^+ \vdash e : \gamma^-}{\Gamma \vdash \mathtt{force}^+(x.e) : \uparrow \tau^+ > \gamma^-} \tag{40.4b}$$

Rule (40.4a) is the continuation representing the application of a function to the positive argument, v_1^+, and executing the body with negative continuation, k_2^-. Rule (40.4b) specifies the continuation that passes a positive value, viewed as a negative value, to a computation.

The statics of computations is given by these rules:

$$\frac{\Gamma \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{ret}(v^-) : \tau^-} \tag{40.5a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+ \quad \Gamma \vdash k^+ : \tau^+ > \gamma^-}{\Gamma \vdash v^+ \triangleright k^+ : \gamma^-} \tag{40.5b}$$

$$\frac{\Gamma \vdash v^- : \tau^- \quad \Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash v^- \triangleright k^- : \gamma^-} \tag{40.5c}$$

Rule (40.5a) specifies the basic form of computation that simply returns the negative value v^-. Rules (40.5b) and (40.5c) specify computations that pass a value to a contination of appropriate polarity.

40.4 Dynamics

The dynamics of $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ is given by a transition system $e \mapsto e'$ specifying the steps of computation. The rules are all axioms; no premises are required because the continuation is used to manage pending computations.

The dynamics consists of the following rules:

$$\overline{\mathtt{z} \triangleright \mathtt{ifz}(e_0; x.e_1) \mapsto e_0} \tag{40.6a}$$

$$\overline{\mathtt{s}(v^+) \triangleright \mathtt{ifz}(e_0; x.e_1) \mapsto [v^+/x]e_1} \tag{40.6b}$$

$$\overline{\mathtt{del}^-(e) \triangleright \mathtt{force}^-(k^-) \mapsto e\,;k^-} \tag{40.6c}$$

$$\overline{\lambda\,(x{:}\tau^+.e) \triangleright \mathtt{ap}(v^+;k^-) \mapsto [v^+/x]e\,;k^-} \tag{40.6d}$$

$$\overline{\mathtt{del}^+(v^+) \triangleright \mathtt{force}^+(x.e) \mapsto [v^+/x]e} \tag{40.6e}$$

$$\overline{\mathtt{fix}\,x\,\mathtt{is}\,v^- \triangleright k^- \mapsto [\mathtt{del}^-(\mathtt{fix}\,x\,\mathtt{is}\,v^-)/x]v^- \triangleright k^-} \tag{40.6f}$$

These rules specify the interaction between values and continuations.

Rules (40.6) make use of two forms of substitution, $[v^+/x]e$ and $[v^+/x]v^-$, which are defined as in Chapter 1. They also employ a new form of *composition*, written $e\,;k_0^-$, which composes a computation with a continuation

by attaching k_0^- to the end of the computation specified by e. This composition is defined mutually recursive with the compositions $k^+ ; k_0^-$ and $k^- ; k_0^-$, which essentially concatenate continuations (stacks).

$$\overline{\mathtt{ret}(v^-) ; k_0^- = v^- \triangleright k_0^-} \tag{40.7a}$$

$$\frac{k^- ; k_0^- = k_1^-}{(v^- \triangleright k^-) ; k_0^- = v^- \triangleright k_1^-} \tag{40.7b}$$

$$\frac{k^+ ; k_0^- = k_1^+}{(v^+ \triangleright k^+) ; k_0^- = v^+ \triangleright k_1^+} \tag{40.7c}$$

$$\frac{e_0 ; k^- = e_0' \quad x \mid e_1 ; k^- = e_1'}{\mathtt{ifz}(e_0; x.e_1) ; k^- = \mathtt{ifz}(e_0'; x.e_1')} \tag{40.7d}$$

$$\frac{k^- ; k_0^- = k_1^-}{\mathtt{force}^-(k^-) ; k_0^- = \mathtt{force}^-(k_1^-)} \tag{40.7e}$$

$$\frac{k^- ; k_0^- = k_1}{\mathtt{ap}(v^+; k^-) ; k_0^- = \mathtt{ap}(v^+; k_1^-)} \tag{40.7f}$$

$$\frac{x \mid e ; k_0^- = e'}{\mathtt{force}^+(x.e) ; k_0^- = \mathtt{force}^+(x.e')} \tag{40.7g}$$

Rules (40.7d) and (40.7g) make use of the generic hypothetical judgement defined in Chapter 3 to express that the composition is defined uniformly in the bound variable.

40.5 Safety

The proof of preservation for $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ reduces to the proof of the typing properties of substitution and composition.

Lemma 40.1 (Substitution)**.** *Suppose that* $\Gamma \vdash v^+ : \rho^+$.

1. *If* $\Gamma, x : \rho^+ \vdash e : \gamma^-$, *then* $\Gamma \vdash [v^+/x]e : \gamma^-$.
2. *If* $\Gamma, x : \rho^+ \vdash v^- : \tau^-$, *then* $\Gamma \vdash [v^+/x]v^- : \tau^-$.
3. *If* $\Gamma, x : \rho^+ \vdash k^+ : \tau^+ > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^+ : \tau^+ > \gamma^-$.
4. *If* $\Gamma, x : \rho^+ \vdash v_1^+ : \tau^+$, *then* $\Gamma \vdash [v^+/x]v_1^+ : \tau^+$.
5. *If* $\Gamma, x : \rho^+ \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^- : \tau^- > \gamma^-$.

Proof. Simultaneously, by induction on the derivation of the typing of the target of the substitution. □

Lemma 40.2 (Composition)**.**

1. *If* $\Gamma \vdash e : \tau^- $ *and* $\Gamma \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash e \,;\, k^- : \tau^- > \gamma^-$.
2. *If* $\Gamma \vdash k_0^+ : \tau^+ > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^+ \,;\, k_1^- : \tau^+ > \gamma_1^-$.
3. *If* $\Gamma \vdash k_0^- : \tau^- > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^- \,;\, k_1^- : \tau^- > \gamma_1^-$.

Proof. Simultaneously, by induction on the derivations of the first premises of each clause of the lemma. □

Theorem 40.3 (Preservation)**.** *If* $\Gamma \vdash e : \gamma^-$ *and* $e \mapsto e'$, *then* $\Gamma \vdash e' : \gamma^-$.

Proof. By induction on transition, appealing to inversion for typing and Lemmas 40.1 on the preceding page and 40.2. □

The progress theorem reduces to the characterization of the values of each type. Focusing makes the required properties evident, since it defines directly the values of each type.

Theorem 40.4 (Progress)**.** *If* $\Gamma \vdash e : \gamma^-$, *then either* $e = \mathtt{ret}(v^-)$ *for some* v^-, *or there exists* e' *such that* $e \mapsto e'$.

40.6 Polarization

The syntax of $\mathcal{L}^{\pm}\{\mathtt{nat} \rightharpoonup\}$ exposes the symmetries between positive and negative types, and hence between eager and lazy computation. It is not, however, especially convenient for writing programs because it requires that each computation in a program be expressed in the stilted form of a value juxtaposed with a continuation. It would be useful to have a more practical syntax that is translatable into the present language. A natural choice is to use the syntax of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ itself, but since that syntax does not determine the evaluation order, we have to choose a *polarization* that commits to a particular dynamics.

We will consider two polarizations of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, the *eager* and the *lazy*, which correspond to two well-known strategies in language design.

under construction

40.7 Notes

The concept of polarization originates with Andreoli's work on focusing as a technique for proof search in linear logic [6]. The application to programming by Zeilberger [106] is the inspiration for the formulation given here.

Part XVI

Parallelism

Chapter 41

Nested Parallelism

Parallel computation seeks to reduce the running times of programs by allowing many computations to be carried out simultaneously. For example, if one wishes to add two numbers, each given by a complex computation, we may consider evaluating the addends simultaneously, then computing their sum. The ability to exploit parallelism is limited by the dependencies among parts of a program. Obviously, if one computation depends on the result of another, then we have no choice but to execute them sequentially so that we may propagate the result of the first to the second. Consequently, the fewer dependencies among sub-computations, the greater the opportunities for parallelism. This argues for functional models of computation, because the possibility of mutation of shared assignables imposes sequentialization constraints on imperative code.

In this chapter we discuss *nested parallelism* in which we nest parallel computations within one another in a hierarchical manner. Nested parallelism is sometimes called *fork-join* parallelism to emphasize the hierarchical structure arising from *forking* two (or more) parallel computations, then *joining* these computations to combine their results before proceeding. We will consider two forms of dynamics for nested parallelism. The first is a structural dynamics in which a single transition on a compound expression may involve multiple transitions on its constituent expressions. The second is a cost dynamics (introduced in Chapter 9) that focuses attention on the sequential and parallel complexity (also known as the *work* and *depth*) of a parallel program by associating a *series-parallel graph* with each computation.

41.1 Binary Fork-Join

We begin with a parallel language whose sole source of parallelism is the simultaneous evaluation of two variable bindings. This is modelled by a construct of the form $\mathtt{par}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e$, 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. As an example, *parallel pairing* may be defined as the expression

$$\mathtt{par}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ \langle x_1, x_2\rangle,$$

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

$$\mathtt{par}(e_1; e_2; x_1.x_2.e),$$

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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{par}(e_1; e_2; x_1.x_2.e) : \tau} \tag{41.1}$$

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 41.3 on page 408.)

We will give both a *sequential* and a *parallel* dynamics of the parallel `let` construct. The definition of the sequential dynamics as a transition judgement of the form $e \mapsto_{\mathsf{seq}} e'$ is entirely straightforward:

$$\frac{e_1 \mapsto e_1'}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \mathtt{par}(e_1'; e_2; x_1.x_2.e)} \tag{41.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \mathtt{par}(e_1; e_2'; x_1.x_2.e)} \tag{41.2b}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \mapsto_{\mathsf{seq}} [e_1, e_2 / x_1, x_2] e} \tag{41.2c}$$

The parallel dynamics is given by a transition judgement of the form $e \mapsto_{\mathsf{par}} e'$, defined as follows:

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \mapsto_{\mathsf{par}} \mathtt{par}(e_1'; e_2'; x_1 \,.\, x_2 \,.\, e)} \tag{41.3a}$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \text{ val}}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \mapsto_{\mathsf{par}} \mathtt{par}(e_1'; e_2; x_1 \,.\, x_2 \,.\, e)} \tag{41.3b}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \mapsto_{\mathsf{par}} \mathtt{par}(e_1; e_2'; x_1 \,.\, x_2 \,.\, e)} \tag{41.3c}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \mapsto_{\mathsf{par}} [e_1, e_2 / x_1, x_2] e} \tag{41.3d}$$

The parallel dynamics 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.

An important advantage of the present approach is captured by the *implicit parallelism theorem*, which states that the sequential and the parallel dynamics 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 *efficiency*. Since the sequential dynamics is deterministic (every expression has at most one value), the implicit parallelism theorem implies that the parallel dynamics is also deterministic. This clearly distinguishes *parallelism*, which is deterministic, from *concurrency*, which is non-deterministic (see Chapters 43 and 44 for more on concurrency).

A proof of the implicit parallelism theorem may be given by giving an evaluation dynamics, $e \Downarrow v$, in the style of Chapter 9, and showing that

$$e \mapsto_{\mathsf{par}}^* v \quad \text{iff} \quad e \Downarrow v \quad \text{iff} \quad e \mapsto_{\mathsf{seq}}^* v$$

(where v is a closed expression such that v val). The crucial rule of the evaluation dynamics is the one governing the parallel `let` construct:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2 / x_1, x_2] e \Downarrow v}{\mathtt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e) \Downarrow v} \tag{41.4}$$

It is easy to show that the sequential dynamics agrees with the evaluation dynamics by a straightforward extension of the proof of Theorem 9.2 on page 83.

Lemma 41.1. *$e \mapsto^*_{seq} v$ iff $e \Downarrow v$.*

Proof. It suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{seq}} v_1$ and $e_2 \mapsto^*_{\mathsf{seq}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{seq}} v$, then

$$\mathtt{par}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e \mapsto^*_{\mathsf{seq}} v.$$

We leave the details of the proof as an exercise for the reader. □

By a similar argument we may show that the parallel dynamics also agrees with the evaluation dynamics, and hence with the sequential dynamics.

Lemma 41.2. *$e \mapsto^*_{par} v$ iff $e \Downarrow v$.*

Proof. It suffices to show that if $e \mapsto_{\mathsf{par}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{par}} v$, then

$$\mathtt{par}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e \mapsto^*_{\mathsf{par}} v.$$

The proof of the first is by a straightforward induction on the parallel dynamics. The proof of the second proceeds by simultaneous induction on the derivations of $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$. If $e_1 = v_1$ with v_1 val and $e_2 = v_2$ with v_2 val, then the result follows immediately from the third premise. If $e_2 = v_2$ but $e_1 \mapsto_{\mathsf{par}} e_1' \mapsto^*_{\mathsf{par}} v_1$, then by induction we have that $\mathtt{par}\ x_1 = e_1'\ \mathtt{and}\ x_2 = v_2\ \mathtt{in}\ e \mapsto^*_{\mathsf{par}} v$, and hence the result follows by an application of Rule (41.3b). The symmetric case follows similarly by an application of Rule (41.3c), and in case both e_1 and e_2 take a step, the result follows by induction and Rule (41.3a). □

Theorem 41.3 (Implicit Parallelism)**.** *The sequential and parallel dynamics coincide: for all v val, $e \mapsto^*_{seq} v$ iff $e \mapsto^*_{par} v$.*

Proof. By Lemmas 41.1 and 41.2. □

The implicit parallelism theorem states that parallelism does not affect the semantics of a program, only the efficiency of its execution. Correctness concerns are factored out, focusing attention on complexity.

41.2 Cost Dynamics

In this section we define a *parallel cost dynamics* 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
			$\mathbf{1}$	unit cost
			$c_1 \otimes c_2$	parallel combination
			$c_1 \oplus c_2$	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 $\mathbf{1}$ 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:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{41.5}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{41.6}$$

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 *parallel complexity* of a computation. The critical path length is the least number of sequential steps that can be taken, even if we have unlimited parallelism available to us, because of steps that can be taken only after the completion of another.

In Chapter 9 we introduced *cost dynamics* as a means of assigning time complexity to evaluation. The proof of Theorem 9.7 on page 86 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 structural dynamics given in Section 41.1 on page 402.

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 dynamics. By definition we arrange that $e \Downarrow^{\mathbf{0}} e$ when e val. The cost assignment for `let` is given by the following rule:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{41.7}$$

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 dynamics 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.

Two simple facts about the cost dynamics are important to keep in mind. First, the cost assignment does not influence the outcome.

Lemma 41.4. *$e \Downarrow v$ iff $e \Downarrow^c v$ for some c.*

Proof. From right to left, erase the cost assignments to obtain an evaluation derivation. From left to right, decorate the evaluation derivations with costs as determined by the rules defining the cost dynamics. □

Second, the cost of evaluating an expression is uniquely determined.

Lemma 41.5. *If $e \Downarrow^{c} v$ and $e \Downarrow^{c'} v$, then c is c'.*

Proof. A routine induction on the derivation of $e \Downarrow^{c} v$. □

The link between the cost dynamics and the structural dynamics 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 41.6. *If $e \Downarrow^{c} v$, then $e \mapsto^{w}_{\mathsf{seq}} v$ and $e \mapsto^{d}_{\mathsf{par}} v$, where $w = \mathit{wk}(c)$ and $d = \mathit{dp}(c)$. Conversely, if $e \mapsto^{w}_{\mathsf{seq}} v$, then there exists c such that $e \Downarrow^{c} v$ with $\mathit{wk}(c) = w$, and if $e \mapsto^{d}_{\mathsf{par}} v'$, then there exists c' such that $e \Downarrow^{c'} v'$ with $\mathit{dp}(c') = d$.*

Proof. The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being Rule (41.7). By induction we have $e_1 \mapsto^{w_1}_{\mathsf{seq}} v_1$, $e_2 \mapsto^{w_2}_{\mathsf{seq}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto^{w}_{\mathsf{seq}} v$, where $w_1 = \mathit{wk}(c_1)$, $w_2 = \mathit{wk}(c_2)$, and $w = \mathit{wk}(c)$. By pasting together derivations we obtain a derivation

$$\begin{aligned}
\mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e) &\mapsto^{w_1}_{\mathsf{seq}} \mathtt{par}(v_1; e_2; x_1\,.\,x_2\,.\,e)\\
&\mapsto^{w_2}_{\mathsf{seq}} \mathtt{par}(v_1; v_2; x_1\,.\,x_2\,.\,e)\\
&\mapsto_{\mathsf{seq}} [v_1, v_2/x_1, x_2]e\\
&\mapsto^{w}_{\mathsf{seq}} v.
\end{aligned}$$

Noting that $\mathit{wk}((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \mapsto^{d_1}_{\mathsf{par}} v_1$, $e_2 \mapsto^{d_2}_{\mathsf{par}} v_2$, and $e \mapsto^{d}_{\mathsf{par}} v$, where $d_1 = \mathit{dp}(c_1)$, $d_2 = \mathit{dp}(c_2)$, and $d = \mathit{dp}(c)$. 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:

$$\begin{aligned}
\mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e) &\mapsto^{d_1}_{\mathsf{par}} \mathtt{par}(v_1; e_2'; x_1\,.\,x_2\,.\,e)\\
&\mapsto^{d_2 - d_1}_{\mathsf{par}} \mathtt{par}(v_1; v_2; x_1\,.\,x_2\,.\,e)\\
&\mapsto_{\mathsf{par}} [v_1, v_2/x_1, x_2]e\\
&\mapsto^{d}_{\mathsf{par}} v.
\end{aligned}$$

Calculating $dp((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1, d_2) + 1 + d$ completes the proof.

Turning to the second part, it suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $wk(c) = wk(c') + 1$, and if $e \mapsto_{\mathsf{par}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $dp(c) = dp(c') + 1$.

Suppose that $e = \mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e_0)$ with e_1 val and e_2 val. Then $e \mapsto_{\mathsf{seq}} e'$, where $e = [e_1, e_2/x_1, x_2]e_0$ and there exists c' such that $e' \Downarrow^{c'} v$. But then $e \Downarrow^{c} v$, where $c = (\mathbf{0} \otimes \mathbf{0}) \oplus \mathbf{1} \oplus c'$, and a simple calculation shows that $wk(c) = wk(c') + 1$, as required. Similarly, $e \mapsto_{\mathsf{par}} e'$ for e' as above, and hence $e \Downarrow^{c} v$ for some c such that $dp(c) = dp(c') + 1$, as required.

Suppose that $e = \mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e_0)$ and $e \mapsto_{\mathsf{seq}} e'$, where $e' = \mathtt{par}(e_1'; e_2; x_1\,.\,x_2\,.\,e_0)$ and $e_1 \mapsto_{\mathsf{seq}} e_1'$. From the assumption that $e' \Downarrow^{c'} v$, we have by inversion that $e_1' \Downarrow^{c_1'} v_1$, $e_2 \Downarrow^{c_2'} v_2$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0'} v$, with $c' = (c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0'$. By induction there exists c_1 such that $wk(c_1) = 1 + wk(c_1')$ and $e_1 \Downarrow^{c_1} v_1$. But then $e \Downarrow^{c} v$, with $c = (c_1 \otimes c_2') \oplus \mathbf{1} \oplus c_0'$.

By a similar argument, suppose that $e = \mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e_0)$ and $e \mapsto_{\mathsf{par}} e'$, where $e' = \mathtt{par}(e_1'; e_2'; x_1\,.\,x_2\,.\,e_0)$ and $e_1 \mapsto_{\mathsf{par}} e_1'$, $e_2 \mapsto_{\mathsf{par}} e_2'$, and $e' \Downarrow^{c'} v$. Then by inversion $e_1' \Downarrow^{c_1'} v_1$, $e_2' \Downarrow^{c_2'} v_2$, $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. But then $e \Downarrow^{c} v$, where $c = (c_1 \otimes c_2) \oplus \mathbf{1} \oplus c_0$, $e_1 \Downarrow^{c_1} v_1$ with $dp(c_1) = 1 + dp(c_1')$, $e_2 \Downarrow^{c_2} v_2$ with $dp(c_2) = 1 + dp(c_2')$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. Calculating, we obtain

$$\begin{aligned} dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) \\ &= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) \\ &= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 \\ &= dp(c') + 1, \end{aligned}$$

which completes the proof. □

Corollary 41.7. *If $e \mapsto_{seq}^{w} v$ and $e \mapsto_{par}^{d} v'$, then v is v' and $e \Downarrow^{c} v$ for some c such that $wk(c) = w$ and $dp(c) = d$.*

41.3 Multiple Fork-Join

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 *sequence* of values of a specified type. The primitive operations on sequences 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 sequence simultaneously, forming a sequence of the results.

We will consider here a simple language of sequence operations to illustrate the main ideas.

Typ	τ	::=	$\mathtt{seq}(\tau)$	$\tau\,\mathtt{seq}$	sequence
Exp	e	::=	$\mathtt{seq}(e_0,\ldots,e_{n-1})$	$[e_0,\ldots,e_{n-1}]$	sequence
			$\mathtt{len}(e)$	$\lvert e\rvert$	size
			$\mathtt{sub}(e_1;e_2)$	$e_1[e_2]$	element
			$\mathtt{tab}(x.e_1;e_2)$	$\mathtt{tab}(x.e_1;e_2)$	tabulate
			$\mathtt{map}(x.e_1;e_2)$	$[e_1 \mid x \in e_2]$	map
			$\mathtt{cat}(e_1;e_2)$	$\mathtt{cat}(e_1;e_2)$	concatenate

The expression $\mathtt{seq}(e_0,\ldots,e_{n-1})$ evaluates to an n-sequence whose elements are given by the expressions $e_0,\ldots,e_{n-1}$. The operation $\mathtt{len}(e)$ returns the number of elements in the sequence given by e. The operation $\mathtt{sub}(e_1;e_2)$ retrieves the element of the sequence given by e_1 at the index given by e_2. The tabulate operation, $\mathtt{tab}(x.e_1;e_2)$, yields the sequence of length given by e_2 whose ith element is given by $[i/x]e_1$. The operation $\mathtt{map}(x.e_1;e_2)$ computes the sequence whose ith element is given by $[e/x]e_1$, where e is the ith element of the sequence given by e_2. The operation $\mathtt{cat}(e_1;e_2)$ concatenates two sequences of the same type.

The statics of these operations is given by the following typing rules:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \mathtt{seq}(e_0,\ldots,e_{n-1}) : \mathtt{seq}(\tau)} \tag{41.8a}$$

$$\frac{\Gamma \vdash e : \mathtt{seq}(\tau)}{\Gamma \vdash \mathtt{len}(e) : \mathtt{nat}} \tag{41.8b}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{seq}(\tau) \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{sub}(e_1;e_2) : \tau} \tag{41.8c}$$

$$\frac{\Gamma, x : \mathtt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{tab}(x.e_1;e_2) : \mathtt{seq}(\tau)} \tag{41.8d}$$

$$\frac{\Gamma \vdash e_2 : \mathtt{seq}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \mathtt{map}(x.e_1;e_2) : \mathtt{seq}(\tau')} \tag{41.8e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{seq}(\tau) \quad \Gamma \vdash e_2 : \mathtt{seq}(\tau)}{\Gamma \vdash \mathtt{cat}(e_1;e_2) : \mathtt{seq}(\tau)} \tag{41.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{seq}(e_0, \ldots, e_{n-1}) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{41.9a}$$

$$\frac{e \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1})}{\texttt{len}(e) \Downarrow^{c \oplus \mathbf{1}} \texttt{num}[n]} \tag{41.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{n-1}) \quad e_2 \Downarrow^{c_2} \texttt{num}[i] \quad (0 \leq i < n)}{\texttt{sub}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{41.9c}$$

$$\frac{e_2 \Downarrow^{c} \texttt{num}[n] \quad [\texttt{num}[0]/x]e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad [\texttt{num}[n-1]/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{41.9d}$$

$$\frac{\begin{array}{c} e_2 \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1}) \\ [v_0/x]e_1 \Downarrow^{c_0} v_0' \quad \ldots \quad [v_{n-1}/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}' \end{array}}{\texttt{map}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0', \ldots, v_{n-1}')} \tag{41.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{m-1}) \quad e_2 \Downarrow^{c_2} \texttt{seq}(v_0', \ldots, v_{n-1}')}{\texttt{cat}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \bigotimes_{i=0}^{m+n-1} \mathbf{1}} \texttt{seq}(v_0, \ldots, v_{m-1}, v_0', \ldots, v_{n-1}')} \tag{41.9f}$$

The cost dynamics for sequence operations may be validated by introducing a sequential and parallel cost dynamics and extending the proof of Theorem 41.6 on page 407 to cover this extension.

41.4 Provably Efficient Implementations

Theorem 41.6 on page 407 states that the cost dynamics accurately models the dynamics of the parallel `let` construct, whether executed sequentially or in parallel. This validates the cost dynamics from the point of view of the dynamics of the language, 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 restriction to a fixed number, $p > 0$, of processors on which to schedule the workload. In addition to limiting the available parallelism this also imposes some synchronization overhead that must be accounted for in order to make accurate predictions of run-time behavior on a concrete parallel platform. A *provably efficient implementation* is one for which we may establish an asymptotic bound on the actual execution time once these overheads are taken into account.

A provably efficient implementation must take account of the limitations and capabilities of the actual hardware on which the program is to be run. Since we are only interested in asymptotic upper bounds, it is convenient to formulate an abstract machine model, and to show that the primitives of the language can be implemented on this model with guaranteed time (and space) bounds. One popular model is the *SMP*, or *shared-memory multiprocessor*, which consists of $p > 0$ sequential processors coordinated by an interconnect network that provides constant-time access to shared memory by each of the processors.[1] The multiprocessor is assumed to provide a constant-time synchronization primitive with which to control simultaneous access to a memory cell. There are a variety of such primitives, any of which is sufficient to provide a parallel fetch-and-add instruction that allows each processor to obtain the current contents of a memory cell and update it by adding a fixed constant in a single atomic operation—the interconnect serializes any simultaneous accesses by more than one processor.

Building a provably efficient implementation of parallelism involves two majors tasks. First, we must show that each of the primitives of the language may be implemented efficiently on the abstract machine model. Second, we must show how to schedule the workload across the processors so as to minimize execution time by maximizing parallelism. When working with a low-level machine model such as an SMP, both tasks involve a fair bit of technical detail to show how to use low-level machine instructions, including a synchronization primitive, to implement the language primitives and to schedule the workload. Collecting these together, we may then give an asymptotic bound on the time complexity of the implementation that relates the abstract cost of the computation to cost of implementing the workload on a p-way multiprocessor. The prototypical result of this kind is called *Brent's Theorem*.

Theorem 41.8. *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 $O(\max(w/p, d))$.*

The theorem tells us that we can never execute a program in fewer steps than its depth, d, and that, at best, we can divide the work up evenly into w/p rounds of execution by the p processors. Observe that if $p = 1$ then the theorem establishes an upper bound of $O(w)$ steps, the sequential complexity of the computation. Moreover, if the work is proportional to the

[1] A slightly weaker assumption is that each access may require up to $\lg p$ time to account for the overhead of synchronization, but we shall neglect this refinement in the present, simplified account.

depth, then we are unable to exploit parallelism, and the overall time is proportional to the work alone.

This motivates the definition of a useful figure of merit, called the *parallelizability ratio*, which is the ratio, w/d, of work to depth. If $w/d \gg p$, then the program is said to be *parallelizable*, because then $w/p \gg d$, and we may therefore reduce running time by using p processors at each step. If, on the other hand, the parallelizability ratio is a constant, then d will dominate w/p, and we will have little opportunity to exploit parallelism to reduce running time. It is not known, in general, whether a problem admits a parallelizable solution. 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 a open problem in complexity theory to characterize which problems are parallelizable, and which are not.

To illustrate the essential ingredients of the proof of Brent's Theorem we will consider a dynamics that models the scheduling of work onto p parallel processors, each of which implements the dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ as described in Chapter 12. The parallel dynamics is defined on states $\nu\,\Sigma\,\{\,\mu\,\}$ of the form

$$\nu\, a_1 : \tau_1, \ldots a_n : \tau_n \,\{\, \langle a_1 : e_1 \rangle \otimes \ldots \otimes \langle a_n : e_n \rangle \,\},$$

where $n \geq 1$. Such a state represents a computation that has been decomposed into n parallel tasks. Each task is given a *name*. The occurrence of a name within a task represents a dependency of that task on the named task. A task is said to be *blocked* on the tasks on which it depends; a task with no dependencies is said to be *ready*.

There are two forms of transition, the *local* and the *global*. The local transitions represent the steps of the individual processors. These consist of the steps of execution of expressions as defined in Chapter 12, augmented with transitions governing parallelism. The global transitions represent the simultaneous execution of local transitions on up to some fixed number, p, of processors.

Local transitions have the form

$$\nu\,\Sigma\, a : \tau \,\{\, \mu \otimes \langle a : e \rangle \,\} \mapsto_{loc} \nu\,\Sigma'\, a : \tau \,\{\, \mu' \otimes \langle a : e' \rangle \,\},$$

where e is ready, and either (a) Σ and Σ' are empty, and μ and μ' are empty; or (b) Σ and μ are empty, and Σ' and μ' declare the types and bindings of two distinct names; or (c) Σ' and μ' are both empty, and Σ and μ declare the types and bindings of two distinct names. These conditions correspond

to the three possible outcomes of a local step by a task: (a) the task takes a step of computation in the sense of Chapter 12; (b) the task forks two new tasks, and waits for their completion; (c) the task joins two parallel tasks that have completed execution.

$$\frac{e \mapsto e'}{\nu\, a : \tau\, \{\, \langle a : e \rangle \,\} \mapsto_{loc} \nu\, a : \tau\, \{\, \langle a : e' \rangle \,\}} \tag{41.10a}$$

$$\left\{ \begin{array}{c} \nu\, a : \tau\, \{\, \langle a : \mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e) \rangle \,\} \\ \mapsto_{loc} \\ \nu\, a_1 : \tau_1\, a_2 : \tau_2\, a : \tau\, \{\, \langle a_1 : e_1 \rangle \otimes \langle a_2 : e_2 \rangle \otimes \langle a : \mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e) \rangle \,\} \end{array} \right\} \tag{41.10b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\left\{ \begin{array}{c} \nu\, a_1 : \tau_1\, a_2 : \tau_2\, a : \tau\, \{\, \langle a_1 : e_1 \rangle \otimes \langle a_2 : e_2 \rangle \otimes \langle a : \mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e) \rangle \,\} \\ \mapsto_{loc} \\ \nu\, a : \tau\, \{\, \langle a : [e_1, e_2 / x_1, x_2] e \rangle \,\} \end{array} \right\}} \tag{41.10c}$$

Rule (41.10a) represents one step of execution according to the rules of Chapter 12. Rule (41.10b) represents the creation of two parallel tasks on which the executing task depends. The expression $\mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e)$ is blocked on tasks a_1 and a_2, so that no local step applies to it. Rule (41.10c) synchronizes a task with the tasks on which it depends once their execution has completed; those tasks are no longer required, and are therefore eliminated from the state.

Each global transition represents the simultaneous execution of one step of computation on each of up to $p \geq 1$ processors.

$$\frac{\begin{array}{c} \nu\, \Sigma_1 a_1 : \tau_1\, \{\, \mu_1 \otimes \langle a_1 : e_1 \rangle \,\} \mapsto_{loc} \nu\, \Sigma_1' a_1 : \tau_1\, \{\, \mu_1' \otimes \langle a_1 : e_1' \rangle \,\} \\ \cdots \\ \nu\, \Sigma_n a_n : \tau_n\, \{\, \mu_n \otimes \langle a_n : e_n \rangle \,\} \mapsto_{loc} \nu\, \Sigma_n' a_n : \tau_n\, \{\, \mu_n' \otimes \langle a_n : e_n' \rangle \,\} \end{array}}{\left\{ \begin{array}{c} \nu\, \Sigma_0\, \Sigma_1\, a_1 : \tau_1\, \ldots\, \Sigma_n\, a_n : \tau_n\, \{\, \mu_0 \otimes \mu_1 \otimes \langle a_1 : e_1 \rangle \otimes \ldots \otimes \mu_n \otimes \langle a_n : e_n \rangle \,\} \\ \mapsto_{glo} \\ \nu\, \Sigma_0\, \Sigma_1'\, a_1 : \tau_1\, \ldots\, \Sigma_n'\, a_n : \tau_n\, \{\, \mu_0 \otimes \mu_1' \otimes \langle a_1 : e_1' \rangle \otimes \ldots \otimes \mu_n' \otimes \langle a_n : e_n' \rangle \,\} \end{array} \right\}} \tag{41.11}$$

At each global step some number, $1 \leq n \leq p$, of ready tasks are scheduled for execution.[2] Since no two distinct tasks may depend on the same task,

[2] The rule does not require that n be chosen as large as possible. A scheduler that always chooses the largest possible $1 \leq n \leq p$ is said to be *greedy*.

we may partition the n tasks so that each scheduled task is grouped with the tasks on which it depends as necessary for any local join step. Any local fork step introduces two fresh tasks that are added to the state as a result of the global transition; any local join step eliminates two tasks whose execution has completed.

A subtle point is that it is implicit in our name binding conventions that the names of any created tasks are to be *globally unique*, even though they are *locally created*. In implementation terms this requires a synchronization step among the processors to ensure that task names are not acccidentally reused among the parallel processors.

The proof of Brent's Theorem for this high-level dynamics is now obvious, provided only that the global scheduling steps are performed greedily so as to maximize the use of processors at each round. If, at each stage of a computation, there are p ready tasks, then the computation will complete in w/p steps, where w is the work complexity of the program. We may, however, be unable to make full use of all p processors at any given stage. This would only be because the dependencies among computations, which are reflected in the variable occurrences and in the definition of the depth complexity of the computation, inhibits parallelism to the extent that evaluation cannot complete in fewer than d rounds. This limitation is significant only to the extent that d is larger than w/p; otherwise, the overall time is bounded by w/p, making maximal use of all p processors.

41.5 Notes

Parallelism should not be confused with concurrency. Parallelism is about efficiency, not semantics; the meaning of a program is independent of whether it is executed in parallel or not. Concurrency is about composition, not efficiency; the meaning of a concurrent program is very weakly specified so that one may compose it with other programs without altering its meaning. This distinction, and the formulation of it given here, was pioneered by Blelloch [12]. In particular the concepts of a cost semantics and its provably efficient implementation are based on Blelloch's work [13, 15].

Chapter 42

Futures and Speculation

A *future* is a computation whose evaluation is initiated in advance of any demand for its value. Like a suspension, a future represents a value that is to be determined later. Unlike a suspension, a future is always evaluated, regardless of whether its value is actually required. In a sequential setting futures are of little interest; a future of type τ is just an expression of type τ. In a parallel setting, however, futures are of interest because they provide a means of initiating a parallel computation whose result is not needed until (presumably) much later, by which time it will have been completed.

The prototypical example of the use of futures is to implementing *pipelining*, a method for overlapping the stages of a multistage computation to the fullest extent possible. This minimizes the latency caused by one stage waiting for the completion of a previous stage by allowing the two stages to proceed in parallel until such time as an explicit dependency is encountered. Ideally, the computation of the result of an earlier stage is completed by the time a later stage requires it. At worst the later stage must be delayed until the earlier stage completes, incurring what is known as a *pipeline stall*.

A *suspension* is a delayed computation whose result may or may not be needed for the overall computation to finish. *Speculation* is a parallel dynamics for suspensions in which suspended computations are executed in parallel with the main thread of computation without regard to whether the suspension is forced. If the value of the suspension is eventually required, then speculation pays off, but if not, the effort to evaluate it wasted. Speculation is therefore not work-efficient: if the value of the suspension is never needed, more work has been undertaken than is necessary to determine the outcome of the computation. Speculation can be useful in situations where there is an excess of computing resources available, more than can be used

in a guaranteed work-efficient manner. In such situations it cannot hurt to perform extra work as long as resources are used that would otherwise be idle.

Parallel futures, in contrast to speculatively evaluated suspensions, are *work efficient* in that the overall work done by a computation involving futures is no more than the work required by a sequential execution. Speculative suspensions, in contrast, are *work inefficient* in that speculative execution may be in vain—the overall computation may involve more steps than the work required to compute the result. For this reason speculation is a risky strategy for exploiting parallelism. It can make good use of available resources, but perhaps only at the expense of doing more work than necessary!

42.1 Futures

The syntax of futures is given by the following grammar:

Typ	τ	::=	$\texttt{fut}(\tau)$	$\tau\,\texttt{fut}$	future
Exp	e	::=	$\texttt{fut}(e)$	$\texttt{fut}(e)$	future
			$\texttt{syn}(e)$	$\texttt{syn}(e)$	synchronize

The type $\tau\,\texttt{fut}$ is the type of futures of type τ. Futures are introduced by the expression $\texttt{fut}(e)$, which schedules e for evaluation and returns a reference to it. Futures are eliminated by the expression $\texttt{syn}(e)$, which synchronizes with the future referred to by e, returning its value.

42.1.1 Statics

The statics of futures is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{fut}(e) : \texttt{fut}(\tau)} \tag{42.1a}$$

$$\frac{\Gamma \vdash e : \texttt{fut}(\tau)}{\Gamma \vdash \texttt{syn}(e) : \tau} \tag{42.1b}$$

These rules are unsurprising, since futures add no new capabilities to the language beyond providing an opportunity for parallel evaluation.

42.1.2 Sequential Dynamics

The sequential dynamics of futures is easily defined. Futures are evaluated eagerly; synchronization returns the value of the future.

$$\frac{e\ \mathsf{val}}{\mathtt{fut}(e)\ \mathsf{val}} \tag{42.2a}$$

$$\frac{e \mapsto e'}{\mathtt{fut}(e) \mapsto \mathtt{fut}(e')} \tag{42.2b}$$

$$\frac{e \mapsto e'}{\mathtt{syn}(e) \mapsto \mathtt{syn}(e')} \tag{42.2c}$$

$$\frac{e\ \mathsf{val}}{\mathtt{syn}(\mathtt{fut}(e)) \mapsto e} \tag{42.2d}$$

42.2 Suspensions

The syntax of (non-recursive) suspensions is given by the following grammar:[1]

Typ	τ	::=	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$	suspension
Exp	e	::=	$\mathtt{susp}(e)$	$\mathtt{susp}(e)$	delay
			$\mathtt{force}(e)$	$\mathtt{force}(e)$	force

The type $\tau\,\mathtt{susp}$ is the type of suspended computations of type τ. The introductory form, $\mathtt{susp}(e)$, delays the computation of e until forced, and the eliminatory form, $\mathtt{force}(e)$, forces evaluation of a delayed computation.

42.2.1 Statics

The statics of suspensions is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{susp}(e) : \mathtt{susp}(\tau)} \tag{42.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{force}(e) : \tau} \tag{42.3b}$$

Thus, the statics for suspensions as given by Rules (42.3) is essentially equivalent to the statics for futures given by Rules (42.1).

[1] We confine ourselves to the non-recursive case to facilitate the comparison with futures.

42.2.2 Sequential Dynamics

The definition of the sequential dynamics of suspensions is similar to that of futures, except that suspended computations are values.

$$\frac{}{\texttt{susp}(e)\ \texttt{val}} \tag{42.4a}$$

$$\frac{e \mapsto e'}{\texttt{susp}(e) \mapsto \texttt{susp}(e')} \tag{42.4b}$$

$$\frac{}{\texttt{force}(\texttt{susp}(e)) \mapsto e} \tag{42.4c}$$

Compared with futures, the sole difference is that a suspension is only evaluated when forced, whereas a future is always evaluated, regardless of whether its value is needed.

42.3 Parallel Dynamics

Futures are only interesting insofar as they admit a parallel dynamics that allows the computation of the future to proceed concurrently with some other computation. Suspensions are (as we saw in Chapter 39) useful for reasons other than parallelism, but they also admit a parallel, speculative interpretation. In this section we give a parallel dynamics of futures and suspensions in which the creation, execution, and synchronization of tasks is made explicit. Interestingly, the parallel dynamics of futures and suspensions is *identical*, except for the termination condition. Whereas futures require all tasks to be completed before termination, speculatively evaluated suspensions may be abandoned before they are completed. For the sake of concreteness we will give the parallel dynamics of futures, remarking only where alterations must be made for speculative evaluation of suspensions.

The parallel dynamics of futures relies on a modest extension to the language given in Section 42.1 on page 416 to introduce *names* for tasks. Let Σ be a finite mapping assigning types to names. The expression $\texttt{fut}[a]$ is a value referring to the outcome of task a. The statics of this expression is given by the following rule:[2]

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} \texttt{fut}[a] : \texttt{fut}(\tau)} \tag{42.5}$$

[2] A similar rule governs the analogous construct, $\texttt{susp}[a]$, in the case of suspensions.

Rules (42.1) carry over in the obvious way with Σ recording the types of the task names.

States of the parallel dynamics have the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is the *focus* of evaluation, and μ represents the parallel futures (or suspensions) that have been activated thus far in the computation. Formally, μ is a finite mapping assigning expressions to the task names declared in Σ. A state is well-formed according to the following rule:

$$\frac{\vdash_\Sigma e : \tau \quad (\forall a \in dom(\Sigma))\ \vdash_\Sigma \mu(a) : \Sigma(a)}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{ok}} \tag{42.6}$$

As discussed in Chapter 38 this rule admits self-referential and mutually referential futures. A more refined condition could as well be given that avoids circularities; we leave this as an exercise for the reader.

The parallel dynamics is divided into two phases, the *local* phase, which defines the basic steps of evaluation of an expression, and the *global* phase, which executes all possible local steps in parallel. The local dynamics is defined by the following rules:

$$\frac{}{\mathtt{fut}[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{42.7a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{fut}(e) \parallel \mu\,\} \mapsto_{loc} \nu\,\Sigma, a:\tau\,\{\,\mathtt{fut}[a] \parallel \mu \otimes \langle a : e\rangle\,\}} \tag{42.7b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto_{loc} \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{syn}(e) \parallel \mu\,\} \mapsto_{loc} \nu\,\Sigma'\,\{\,\mathtt{syn}(e') \parallel \mu'\,\}} \tag{42.7c}$$

$$\frac{e'\ \mathsf{val}_{\Sigma,a:\tau}}{\left\{\begin{array}{c}\nu\,\Sigma, a:\tau\,\{\,\mathtt{syn}(\mathtt{fut}[a]) \parallel \mu \otimes \langle a : e'\rangle\,\} \\ \mapsto_{loc} \\ \nu\,\Sigma, a:\tau\,\{\,e' \parallel \mu \otimes \langle a : e'\rangle\,\}\end{array}\right\}} \tag{42.7d}$$

Rule (42.7b) activates a future named a executing the expression e and returns a reference to it. Rule (42.7d) synchronizes with a future whose value has been determined. Note that a local transition always has the form

$$\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto_{loc} \nu\,\Sigma\,\Sigma'\,\{\,e' \parallel \mu \otimes \mu'\,\}$$

where Σ' is either empty or declares the type of a single symbol, and μ' is either empty or of the form $\langle a : e'\rangle$ for some expression e'.

A global step of the parallel dynamics consists of at most one local step for the focal expression and one local step for each of up to p futures, where $p > 0$ is a fixed parameter representing the number of processors.

$$\frac{\begin{array}{c}\mu = \mu_0 \otimes \langle a_1 : e_1 \rangle \otimes \cdots \otimes \langle a_n : e_n \rangle \\ \mu'' = \mu_0 \otimes \langle a_1 : e_1' \rangle \otimes \cdots \otimes \langle a_n : e_n' \rangle \\ \nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto^{\mathbf{0,1}}_{loc} \nu\,\Sigma\,\Sigma'\,\{\, e' \parallel \mu \otimes \mu' \,\} \\ (\forall 1 \leq i \leq n) \quad \nu\,\Sigma\,\{\, e_i \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma\,\Sigma_i'\,\{\, e_i' \parallel \mu \otimes \mu_i' \,\}\end{array}}{\left\{\begin{array}{c}\nu\,\Sigma\,\{\, e \parallel \mu \,\} \\ \mapsto_{glo} \\ \nu\,\Sigma\,\Sigma'\,\Sigma_1'\,\ldots\,\Sigma_n'\,\{\, e' \parallel \mu'' \otimes \mu' \otimes \mu_1' \otimes \cdots \otimes \mu_n' \,\}\end{array}\right\}} \tag{42.8a}$$

Rule (42.8a) allows the focus expression to take either zero or one steps since it may be blocked awaiting the completion of evaluation of a parallel future (or forcing a suspension). The futures allocated by the local steps of execution are consolidated in the result of the global step. We assume without loss of generality that the names of the new futures in each local step are pairwise disjoint so that the combination makes sense. In implementation terms satisfying this disjointness assumption means that the processors must synchronize their access to memory.

The initial state of a computation, whether for futures or suspensions, is defined by the rule

$$\frac{}{\nu\,\emptyset\,\{\, e \parallel \emptyset \,\}\ \mathsf{initial}} \tag{42.9}$$

Final states differ according to whether we are considering futures or suspensions. In the case of futures a state is final iff both the focus and all parallel futures have completed evaluation:

$$\frac{e\ \mathsf{val}_\Sigma \quad \mu\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{final}} \tag{42.10a}$$

$$\frac{(\forall a \in dom(\Sigma))\ \mu(a)\ \mathsf{val}_\Sigma}{\mu\ \mathsf{val}_\Sigma} \tag{42.10b}$$

In the case of suspensions a state is final iff the focus is a value:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{final}} \tag{42.11}$$

This corresponds to the speculative nature of the parallel evaluation of suspensions whose outcome may not be needed to determine the final outcome of the program.

42.4 Applications of Futures

Pipelining provides a good example of the use of parallel futures. Consider a situation in which a *producer* builds a list whose elements represent units of work, and a *consumer* that traverses the work list and acts on each element of that list. The elements of the work list can be thought of as "instructions" to the consumer, which maps a function over that list to carry out those instructions. An obvious sequential implementation first builds the work list, then traverses it to perform the work indicated by the list. This is fine as long as the elements of the list can be produced quickly, but if each element requires a substantial amount of computation, it would be preferable to overlap production of the next list element with execution of the previous unit of work. This can be easily programmed using futures.

Let `flist` be the recursive type $\mu t.\,\texttt{unit} + (\texttt{nat} \times t\,\texttt{fut})$, whose elements are `nil`, defined to be $\texttt{fold}(\texttt{l} \cdot \langle\rangle)$, and $\texttt{cons}(e_1, e_2)$, defined to be $\texttt{fold}(\texttt{r} \cdot \langle e_1, \texttt{fut}(e_2)\rangle)$. The producer is a recursive function that generates a value of type `flist`:

```
fix produce : (nat → nat opt) → nat → flist is
  λ f. λ i.
    case f(i) {
      null ⇒ nil
    | just x ⇒ cons(x, fut (produce f (i+1)))
    }
```

On each iteration the producer generates a parallel future to produce the tail. This computation proceeds after the producer returns so that it overlap subsequent computation.

The consumer folds an operation over the work list as follows:

```
fix consume : ((nat×nat)→nat) → nat → flist → nat is
  λ g. λ a. λ xs.
    case xs {
      nil ⇒ a
    | cons (x, xs) ⇒ consume g (g (x, a)) (syn xs)
    }
```

The consumer synchronizes with the tail of the work list just at the point where it makes a recursive call and hence requires the head element of the tail to continue processing. At this point the consumer will block, if necessary, to await computation of the tail before continuing the recursion.

Another application of futures is to provide more control over parallelism in a language with suspensions. Rather than evaluate suspensions speculatively, which is not work efficient, we may instead add futures to the language in addition to suspensions. One application of futures in such a setting is called a *spark*. A spark is a computation that is executed in parallel with another purely for its effect on suspensions. The spark traverses a data structure, forcing the suspensions within so that their values are computed and stored, but otherwise yielding no useful result. The idea is that the spark forces the suspensions that will be needed by the main computation, but taking advantage of parallelism in the hope that their values will have been computed by the time the main computation requires them.

The sequential dynamics of the spark expression $\texttt{spark}(e_1;e_2)$ is simply to evaluate e_1 before evaluating e_2. This is useful in the context of a by-need dynamics for suspensions, since evaluation of e_1 will record the values of some suspensions in the memo table for subsequent use by the computation e_2. The parallel dynamics specifies, in addition, that e_1 and e_2 are to be evaluated in parallel. The behavior of sparks is captured by the definition of $\texttt{spark}(e_1;e_2)$ in terms of futures:

$$\texttt{let}\ \_\ \texttt{be}\ \texttt{fut}(e_1)\ \texttt{in}\ e_2.$$

Evaluation of e_1 commences immediately, but its value, if any, is abandoned. This encoding does not allow for evaluation of e_1 to be abandoned as soon as e_2 reaches a value, but this scenario is not expected to arise for the intended mode of use of sparks. The expression e_1 should be a quick traversal that does nothing other than force the suspensions in some data structure, exiting as soon as this is complete. Presumably this computation takes less time than it takes for e_2 to perform its work before forcing the suspensions that were forced by e_2, otherwise there is little to be gained from the use of sparks in the first place!

As an example, consider the type `strm` of streams of numbers defined by the recursive type $\mu t.(\texttt{unit} + (\texttt{nat} \times t))\ \texttt{susp}$. Elements of this type are suspended computations that, when forced, either signals the end of stream, or produces a number and another such stream. Suppose that s is such a stream, and assume that we know, for reasons of its construction, that it is finite. We wish to compute $\texttt{map}(f)(s)$ for some function f, and to overlap this computation with the production of the stream elements. We will make use of a function `mapforce` that forces successive elements of the input stream, but yields no useful output. The computation $\texttt{spark}(\texttt{mapforce}(s);\texttt{map}(f)(s))$ forces the elements of the stream in parallel with the computation of $\texttt{map}(f)(s)$, with the intention that all

suspensions in s are forced before their values are required by the main computation.

Finally, note that it is easy to encode binary nested parallelism using futures. This may be accomplished by defining $\texttt{par}(e_1; e_2; x_1 \,.\, x_2 \,.\, e)$ to stand for the expression

$$\texttt{let } x_1' \texttt{ be fut}(e_1) \texttt{ in let } x_2 \texttt{ be } e_2 \texttt{ in let } x_1 \texttt{ be syn}(x_1') \texttt{ in } e$$

The order of bindings is important to ensure that evaluation of e_2 proceeds in parallel with evaluation of e_1. Observe that evaluation of e cannot, in any case, proceed until both are complete.

42.5 Notes

Futures were introduced by Halstead in the MultiLisp language [49] for expressing parallel computation. Essentially the same concept was introduced by Arvind under the name "I-structures", assign-once assignables with a parallel execution semantics [8]. The account given here is based on Greiner and Blelloch's work on speculative parallelism [37].

Part XVII

Concurrency

Chapter 43

Process Calculus

So far we have mainly studied the statics and dynamics 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. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to synchronous and asynchronous communication.

Our presentation 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 the arrival of any one of several events. Other forms of process include parallel composition and the declaration of a communication channel. The basic forms of event are *signalling* and *querying* on a channel. Events are combined using a non-deterministic choice operator that signals the arrival any one of a specified collection of events.

43.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an *event*, which specifies the *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 *signal* and a *query* on a

channel. The processes synchronize when one signals on a channel that the other is querying, after which they both 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.

Proc	P	::=	`await(`E`)`	$\$\,E$	synchronize
Evt	E	::=	`null`	$\mathbf{0}$	null
			`or(`E_1`;`E_2`)`	$E_1 + E_2$	choice
			`que[`a`](`P`)`	$?a\,;P$	query
			`sig[`a`](`P`)`	$!a\,;P$	signal

The variables a, b, and c range over symbols serving as *channel names* that mediate communication among the processes.

We will not distinguish between events that differ only up to *structural congruence*, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{E \equiv E'}{\$\,E \equiv \$\,E'} \tag{43.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{43.1b}$$

$$\frac{P \equiv P'}{?a\,;P \equiv ?a\,;P'} \tag{43.1c}$$

$$\frac{P \equiv P'}{!a\,;P \equiv !a\,;P'} \tag{43.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{43.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{43.1f}$$

$$\frac{}{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3} \tag{43.1g}$$

Imposing structural congruence on sequential processes enables us to think of an event as having the form

$$!a\,;P_1 + \ldots\, ?a\,;Q_1 + \ldots$$

consisting of a sum of signal and query events, with the sum of no events being the null event, $\mathbf{0}$.

An illustrative example of 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.

$$V = \$\,(?2p;\$\,(!tea;V + ?2p;\$\,(!cof;V)))$$

As the example indicates, we permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs. (Later we will show how to avoid reliance on recursive definitions.)

Because the computation occurring within a process is suppressed, sequential processes have no dynamics on their own, but only through their interaction with other processes. 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.

43.2 Interaction

Processes become interesting when they are allowed to interact with one another to achieve a common goal. To account for interaction we enrich the language of processes with *concurrent composition*:

Proc	P	::=	$\mathtt{await}(E)$	$\$\,E$	synchronize
			$\mathtt{stop}$	$\mathbf{1}$	inert
			$\mathtt{par}(P_1;P_2)$	$P_1 \parallel P_2$	composition

The process $\mathbf{1}$ 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 $\mathbf{1}$ with $\$\,\mathbf{0}$, the process that awaits the event that will never occur, but we prefer to treat the inert process as a primitive concept.

We will identify processes up to structural congruence, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{}{P \parallel \mathbf{1} \equiv P} \tag{43.2a}$$

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{43.2b}$$

$$\frac{}{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{43.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{43.2d}$$

Up to structural congruence 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 stands for the null process, **1**.

Interaction between processes consists of synchronization of two complementary actions. The dynamics of interaction is defined by two forms of judgement. The transition judgement $P \mapsto P'$ states that the process P evolves to the process P' as a result of a single step of computation. The family of transition judgements, $P \overset{\alpha}{\mapsto} P'$, where α is an *action*, states that the process P may evolve to the process P' provided that the action α is permissible in the context in which the transition occurs (in a sense to be made precise momentarily). The possible actions are given by the following grammar:

Act	α	::=	`que[`a`]`	$a\,?$	query
			`sig[`a`]`	$a\,!$	signal
			`sil`	ε	silent

The *query action*, $a\,?$, and the *signal action*, $a\,!$, are complementary, and the *silent action*, ε, is self-complementary. We define the *complementary* action to α to be the action $\overline{\alpha}$ given by the equations $\overline{a\,?} = a\,!$, $\overline{a\,!} = a\,?$, and $\overline{\varepsilon} = \varepsilon$. As a notational convenience, we often regard the unlabelled transition $P \mapsto P'$ to be the labelled transition $P \overset{\varepsilon}{\mapsto} P'$.

$$\frac{}{\$\,(!a\,;P + E) \overset{a\,!}{\mapsto} P} \tag{43.3a}$$

$$\frac{}{\$\,(?a\,;P + E) \overset{a\,?}{\mapsto} P} \tag{43.3b}$$

$$\frac{P_1 \overset{\alpha}{\mapsto} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\mapsto} P_1' \parallel P_2} \tag{43.3c}$$

$$\frac{P_1 \overset{\alpha}{\mapsto} P_1' \quad P_2 \overset{\overline{\alpha}}{\mapsto} P_2'}{P_1 \parallel P_2 \mapsto P_1' \parallel P_2'} \tag{43.3d}$$

Rules (43.3a) and (43.3b) specify that any of the events on which a process is synchronizing may occur. Rule (43.3d) 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:

$$U = \$\,!2p;\$\,!2p;\$\,?cof;\mathbf{1}.$$

Here is a trace of the interaction between V and U:

$$\begin{aligned} V \parallel U &\mapsto \$\,!tea;V + ?2p;\$\,!cof;V \parallel \$\,!2p;\$\,?cof;\mathbf{1} \\ &\mapsto \$\,!cof;V \parallel \$\,?cof;\mathbf{1} \\ &\mapsto V \end{aligned}$$

These steps are justified, respectively, by the following pairs of labelled transitions:

$$\begin{aligned} U &\xrightarrow{2p!} U' = \$\,!2p;\$\,?cof;\mathbf{1} \\ V &\xrightarrow{2p?} V' = \$\,(!tea;V + ?2p;\$\,!cof;V) \end{aligned}$$

$$\begin{aligned} U' &\xrightarrow{2p!} U'' = \$\,?cof;\mathbf{1} \\ V' &\xrightarrow{2p?} V'' = \$\,!cof;V \end{aligned}$$

$$\begin{aligned} U'' &\xrightarrow{cof?} \mathbf{1} \\ V'' &\xrightarrow{cof!} V \end{aligned}$$

We have suppressed uses of structural congruence in the above derivations to avoid clutter, but it is important to see its role in managing the nondeterministic choice of events by a process.

43.3 Replication

Some presentations of process calculi 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

$$*P \equiv P \parallel *P. \tag{43.4}$$

Understood as a principle of structural congruence, this rule hides the steps of process creation, and gives no hint as to how often it can or should be applied. One could alternatively build replication into the dynamics to model the details of replication more closely:

$$*P \mapsto P \parallel *P. \tag{43.5}$$

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:

$$V_0 = \$\,(!v; \mathbf{1}) \tag{43.6}$$

$$V_1 = *\,\$\,(?v; V_2) \tag{43.7}$$

$$V_2 = \$\,(?\mathtt{2p}; \$\,(!\mathtt{tea}; V_0 + ?\mathtt{2p}; \$\,(!\mathtt{cof}; V_0))) \tag{43.8}$$

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 $V_0 \parallel V_1$.

This example motivates a restriction on replication that avoids the indeterminacy inherent in accounting for it either as part of structural congruence (Rule (43.4)) or as a computation step (Rule (43.5)). Rather than take replication as a primitive notion, we may instead take *replicated synchronization* as a primitive notion governed by the following rules:

$$\frac{}{*\,\$\,(!a\,;P+E) \stackrel{a\,!}{\longmapsto} P \parallel *\,\$\,(!a\,;P+E)} \tag{43.9a}$$

$$\frac{}{*\,\$\,(?a\,;P+E) \stackrel{a\,?}{\longmapsto} P \parallel *\,\$\,(?a\,;P+E)} \tag{43.9b}$$

The process $*\,\$\,(E)$ is to be regarded not as a composition of replication and synchronization, but as the inseparable combination of these two constructs. The advantage is that the replication occurs only as needed, precisely when a synchronization with another process is possible. This avoids the need to "guess", either by structural congruence or an explicit step, when to replicate a process.

43.4 Allocating Channels

It is often useful (particularly once we have introduce inter-process communication) to introduce new channels within a process, rather than assume that all channels of interaction are given *a priori*. To allow for this, the syntax of processes is enriched with a channel declaration primitive:

$$\mathsf{Proc} \quad P \quad ::= \quad \texttt{new}(a.P) \quad \nu\, a\,.P \quad \text{new channel}$$

The channel, a, is bound within the process P, and hence may be renamed at will (avoiding conflicts) within P. To simplify notation we sometimes write $\nu\, a_1, \ldots, a_k\,.P$ for the iterated declaration $\nu\, a_1 \ldots \nu\, a_k\,.P$.

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{43.10a}$$

$$\frac{P \equiv P'}{\nu\, a\,.P \equiv \nu\, a\,.P'} \tag{43.10b}$$

$$\frac{a \notin P_2}{(\nu\, a\,.\, P_1) \parallel P_2 \equiv \nu\, a\,.\, (P_1 \parallel P_2)} \tag{43.10c}$$

$$\frac{(a \notin P)}{\nu\, a\,.\, P \equiv P} \tag{43.10d}$$

Rule (43.10c), called *scope extrusion*, will be especially important in Section 43.5 on page 436. Rule (43.10d) states that channels may be de-allocated once they are no longer in use.

To account for the scopes of names (and to prepare for later generalizations) it is useful to introduce a static semantics for processes that ensures that names are properly scoped. A *signature*, Σ, is, for the time being, a finite set of channels. The judgement $\vdash_\Sigma P$ proc states that a process, P, is well-formed relative to the channels declared in the signature, Σ.

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{43.11a}$$

$$\frac{\vdash_\Sigma P_1\ \mathsf{proc} \quad \vdash_\Sigma P_2\ \mathsf{proc}}{\vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{43.11b}$$

$$\frac{\vdash_\Sigma E\ \mathsf{event}}{\vdash_\Sigma \$\, E\ \mathsf{proc}} \tag{43.11c}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_\Sigma \nu\, a\,.\, P\ \mathsf{proc}} \tag{43.11d}$$

The foregoing rules make use of an auxiliary judgement, $\vdash_\Sigma E$ event, stating that E is a well-formed event relative to Σ.

$$\frac{}{\vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{43.12a}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a} ?a\,;P\ \mathsf{event}} \tag{43.12b}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a} !a\,;P\ \mathsf{event}} \tag{43.12c}$$

$$\frac{\vdash_\Sigma E_1\ \mathsf{event} \quad \vdash_\Sigma E_2\ \mathsf{event}}{\vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{43.12d}$$

We shall also have need of the judgement $\vdash_\Sigma \alpha$ action stating that α is a well-formed action relative to Σ:

$$\frac{}{\vdash_{\Sigma,a} a?\ \mathsf{action}} \tag{43.13a}$$

$$\frac{}{\vdash_{\Sigma,a} a\,!\ \mathsf{action}} \tag{43.13b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon\ \mathsf{action}} \tag{43.13c}$$

The dynamics is correspondingly generalized to keep track of the set of active channels. The judgement $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$ states that P transitions to P' with action α relative to channels Σ. The rules defining the dynamics are indexed forms of those given above, augmented by an additional rule governing the declaration of a channel. We give the complete set of rules here for the sake of clarity.

$$\frac{}{\$\,(!a\,;P + E) \overset{a\,!}{\underset{\Sigma,a}{\longmapsto}} P} \tag{43.14a}$$

$$\frac{}{\$\,(?a\,;P + E) \overset{a\,?}{\underset{\Sigma,a}{\longmapsto}} P} \tag{43.14b}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1' \parallel P_2} \tag{43.14c}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1' \quad P_2 \overset{\bar{\alpha}}{\underset{\Sigma}{\longmapsto}} P_2'}{P_1 \parallel P_2 \underset{\Sigma}{\longmapsto} P_1' \parallel P_2'} \tag{43.14d}$$

$$\frac{P \overset{\alpha}{\underset{\Sigma,a}{\longmapsto}} P' \quad \vdash_{\Sigma} \alpha\ \mathsf{action}}{\nu\, a\,.\, P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\, a\,.\, P'} \tag{43.14e}$$

Rule (43.14e) states that no process may interact with $\nu\, a\,.\, P$ along the locally-allocated channel, a, since to do so would require that a already be declared in Σ, which is precluded by the freshness convention on binders.

As an example, let us consider again the definition of the vending machine using replication, rather than recursion. 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:

$$(\nu\, v.(V_0 \parallel V_1)) \parallel U \underset{\Sigma}{\mapsto} (\nu\, v.V_2) \parallel U \equiv \nu\, v.(V_2 \parallel U).$$

(The processes V_0, V_1, and V_2 are those defined earlier.) 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.

43.5 Communication

Synchronization is the coordination of the execution of two processes that are willing to undertake the complementary actions of signalling and querying a common channel. *Synchronous communication* is a natural generalization of synchronization to allow more than one bit of data to be communicated between two coordinating processes, a *sender* and a *receiver*. In principle any type of data may be communicated from one process to another, and we can give a uniform account of communication that is independent of the type of data communicated between processes. Communication becomes more interesting in the presence of a type of *channel references*, which allow access to a communication channel to be propagated from one process to another, allowing alteration of the interconnection topology among processes during execution. (Channel references will be discussed in Section 43.6 on page 439.)

To account for interprocess communication we must enrich the language of processes to include *variables*, as well as *channels*, in the formalism. Variables range, as always, over types, and are given meaning by substitution. Channels, on the other hand, are assigned types that classify the data carried on that channel, and are given meaning by send and receive events that generalize the signal and query events considered earlier. The abstract syntax of communication events is given by the following grammar:

Evt	E	::=	$\mathtt{snd}[a](e;P)$	$!\,a(e;P)$	send
			$\mathtt{rcv}[a](x.P)$	$?\,a(x.P)$	receive

The event $\mathtt{rcv}[a](x.P)$ represents the receipt of a value, x, on the channel a, passing x to the process P. The variable, x, is bound within P, and hence may be chosen freely, subject to the usual restrictions on the choice of names of bound variables. The event $\mathtt{snd}[a](e;P)$ represents the transmission of

(the value of) the expression e on channel a, continuing with the process P only once this value has been received.

To account for the type of data that may be sent on a channel, the syntax of channel declaration is generalized to associate a type with each channel name.

$$\text{Proc} \quad P \quad ::= \quad \texttt{new[}\tau\texttt{](}a\texttt{.}P\texttt{)} \quad \nu\, a{:}\tau\,.\,P \quad \text{typed channel}$$

The process $\texttt{new[}\tau\texttt{](}a\texttt{.}P\texttt{)}$ introduces a new channel name, a, with associated type τ for use within the process P. The name, a, is bound within P, and hence may be chosen at will, subject only to avoidance of confusion of distinct names.

The statics of communication extends that of synchronization by associating types to channels and by considering variables that range over a type. The judgement $\Gamma \vdash_\Sigma P \ \mathsf{proc}$ states that P is a well-formed process involving the channels declared in Σ and the variables declared in Γ. It is inductively defined by the following rules, wherein we assume that the typing judgement $\Gamma \vdash_\Sigma e : \tau$ is given separately.

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1} \ \mathsf{proc}} \tag{43.15a}$$

$$\frac{\Gamma \vdash_\Sigma P_1 \ \mathsf{proc} \quad \Gamma \vdash_\Sigma P_2 \ \mathsf{proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2 \ \mathsf{proc}} \tag{43.15b}$$

$$\frac{\Gamma \vdash_{\Sigma, a:\tau} P \ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a{:}\tau\,.\,P \ \mathsf{proc}} \tag{43.15c}$$

$$\frac{\Gamma \vdash_\Sigma E \ \mathsf{event}}{\Gamma \vdash_\Sigma \$\, E \ \mathsf{proc}} \tag{43.15d}$$

Rules (43.15) make use of the auxiliary judgement $\Gamma \vdash_\Sigma E \ \mathsf{event}$, stating that E is a well-formed event relative to Γ and Σ, which is defined as follows:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0} \ \mathsf{event}} \tag{43.16a}$$

$$\frac{\Gamma \vdash_\Sigma E_1 \ \mathsf{event} \quad \Gamma \vdash_\Sigma E_2 \ \mathsf{event}}{\Gamma \vdash_\Sigma E_1 + E_2 \ \mathsf{event}} \tag{43.16b}$$

$$\frac{\Gamma, x : \tau \vdash_{\Sigma, a:\tau} P \ \mathsf{proc}}{\Gamma \vdash_{\Sigma, a:\tau} ?\, a(x\,.\,P) \ \mathsf{event}} \tag{43.16c}$$

$$\frac{\Gamma \vdash_{\Sigma, a:\tau} e : \tau \quad \Gamma \vdash_{\Sigma, a:\tau} P \ \mathsf{proc}}{\Gamma \vdash_{\Sigma, a:\tau} !\, a(e; P) \ \mathsf{event}} \tag{43.16d}$$

Rule (43.16d) makes use of a typing judgement for expressions that ensures that the type of a channel is respected by communication.

The dynamics of synchronous communication is similarly an extension of the dynamics of synchronization. Actions are generalized to include the transmitted value, as well as the channel and its orientation:

$$\begin{array}{llcllll} \textsf{Act} & \alpha & ::= & \texttt{rcv}[a](e) & a\,?\,e & \text{receive} \\ & & & \texttt{snd}[a](e) & a\,!\,e & \text{send} \\ & & & \texttt{sil} & \varepsilon & \text{silent} \end{array}$$

Complementarity is defined, essentially as before, to switch the orientation of an action: $\overline{a\,?\,e} = a\,!\,e$, $\overline{a\,!\,e} = a\,?\,e$, and $\overline{\varepsilon} = \varepsilon$.

The statics ensures that the expression associated with these actions is a value of a type suitable for the channel:

$$\frac{\vdash_{\Sigma,a:\tau} e : \tau \quad e\ \mathsf{val}_{\Sigma,a:\tau}}{\vdash_{\Sigma,a:\tau} a\,!\,e\ \mathsf{action}} \tag{43.17a}$$

$$\frac{\vdash_{\Sigma,a:\tau} e : \tau \quad e\ \mathsf{val}_{\Sigma,a:\tau}}{\vdash_{\Sigma,a:\tau} a\,?\,e\ \mathsf{action}} \tag{43.17b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon\ \mathsf{action}} \tag{43.17c}$$

The dynamics of synchronous communication is defined by replacing Rules (43.14a) and (43.14b) with the following rules:

$$\frac{e \underset{\Sigma,a:\tau}{\longmapsto} e'}{\$\,(!\,a(e;P) + E) \underset{\Sigma,a:\tau}{\longmapsto} \$\,(!\,a(e';P) + E)} \tag{43.18a}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\$\,(!\,a(e;P) + E) \xrightarrow[\Sigma,a:\tau]{a!e} P} \tag{43.18b}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\$\,(?\,a(x.P) + E) \xrightarrow[\Sigma,a:\tau]{a?e} [e/x]P} \tag{43.18c}$$

Rule (43.18c) is non-deterministic in that it "guesses" the value, e, to be received along channel a. Rules (43.18) make reference to the dynamics of expressions, which is left unspecified here so as to avoid an *a priori* commitment as to the nature of values communicated on a channel.

With synchronous communication both the sender and the of a message are blocked until the interaction is completed. This means that the sender must be notified whenever a message is received, and hence there must be an implicit reply channel from the receiver to the sender for the notification. This suggests that synchronous communication may be decomposed into a simpler *asynchronous send* operation, which transmits a message on a channel without waiting for its receipt, together with *channel passing* to transmit an acknowledgement channel along with the message data.

Asynchronous communication is defined by removing the synchronous send event from the process calculus, and adding a new form of process that simply sends a message on a channel. The syntax of asynchronous send is as follows:

$$\mathsf{Proc} \quad P \quad ::= \quad \texttt{asnd[}a\texttt{](}e\texttt{)} \quad \texttt{!}\,a\texttt{(}e\texttt{)} \quad \text{send}$$

The process `asnd[`a`](`e`)` sends the message e on channel a, and then terminates immediately. Without the synchronous send event, every event is, up to structural congruence, a choice of zero or more read events. The statics of asychronous send is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} e : \tau}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{!}\,a\texttt{(}e\texttt{)}\ \mathsf{proc}} \tag{43.19}$$

The dynamics is similarly straightforward:

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{!}\,a\texttt{(}e\texttt{)} \overset{a!e}{\underset{\Sigma}{\longmapsto}} \mathbf{1}} \tag{43.20}$$

The rule for interprocess communication given earlier remains unchanged, since the action associated with the asychronous send is the same as in the synchronous case. One may regard a pending asynchronous send as a "buffer" in which the message is held until a receiver is selected.

43.6 Channel Passing

An interesting case of interprocess communication arises when one process passes one channel to another along a common channel. The channel passed by the sending process need not have been known *a priori* to the receiving process. This allows for new patterns of communication to be established among processes. For example, two processes, P and Q, may

share a channel, a, along which they may send and receive messages. If the scope of a is limited to these processes, then no other process, R, may communicate on that channel; it is, in effect, a *private* channel between P and Q.

It frequently arises, however, that P and Q wish to include the process R in their conversation in a controlled manner. This may be accomplished by first expanding the scope of the channel a to encompass R, then sending (a reference to) the channel a to R along a pre-arranged channel. Upon receipt of the channel reference, R may communicate with P and Q using send and receive operations that act on channel references. Bearing in mind that channels are not themselves forms of expression, such a scenario can be enacted by introducing a type, $\tau\,\texttt{chan}$, whose values are references to channels carrying values of type τ. The elimination forms for the channel type are send and receive operations that act on references, rather than explicitly given channels.[1]

Such a situation may be described schematically by the process expression

$$(\nu\, a\!:\!\tau\,.\,(P \parallel Q)) \parallel R,$$

in which the process R is initially excluded from the scope of the channel a, whose scope encompasses both the processes P and Q. The type τ represents the type of data communicated along channel a; it may be chosen arbitrarily for the sake of this example. The processes P and Q may communicate with each other by sending and receiving along channel a. If these two processes wish to include R in the conversation, then they must communicate the identity of channel a to the process R along some pre-arranged channel, b. If a is a channel carrying values of type τ, then b will be a channel carrying values of type $\tau\,\texttt{chan}$, which are references to τ-carrying channels. The channel b must be known to at least one of P and Q, and also to channel R. This can be described by the following process expression:

$$\nu\, b\!:\!\tau\,\texttt{chan}\,.\,((\nu\, a\!:\!\tau\,.\,(P \parallel Q)) \parallel R).$$

Suppose that P wishes to include R in the conversation by sending a reference to the channel a along b. The process R correspondingly receives a reference to a channel on b, and commences communication with P and Q along that channel. Thus P has the form $\$\,(!\,b(\&\,a\,;P'))$ and R has the

[1] It may be helpful to compare channel types with reference types as described in Chapters 37 and 38. Channels correspond to assignables, and channel types correspond to reference types.

form $\$\,(?\,b(x\,.\,R'))$. The overall process has the form

$$\nu\,b{:}\tau\,\mathtt{chan}\,.\,(\nu\,a{:}\tau\,.\,(\$\,(!\,b(\&\,a\,;P'))\parallel Q)\parallel \$\,(?\,b(x\,.\,R'))).$$

The process P is prepared to send a reference to the channel a along the channel b, where it may be received by the process R. But the scope of a is limited to processes P and Q, so in order for the communication to succeed, we must first expand its scope to encompass R using the concept of scope extrusion introduced in Section 43.4 on page 433 to obtain the structurally equivalent process

$$\nu\,b{:}\tau\,\mathtt{chan}\,.\,\nu\,a{:}\tau\,.\,(\$\,(!\,b(\&\,a\,;P'))\parallel Q\parallel \$\,(?\,b(x\,.\,R'))).$$

The scope of a has been expanded to encompass R, preparing the ground for communication between P and R, which results in the process

$$\nu\,b{:}\tau\,\mathtt{chan}\,.\,\nu\,a{:}\tau\,.\,(P'\parallel Q\parallel [\&\,a/x]R').$$

The reference to the channel a has been substituted for the variable x within R'.

The process R may now communicate with P and Q by sending and receiving messages along the channel referenced by x. This is accomplished using dynamic forms of send and receive in which the channel on which to communicate is determined by evaluation of an expression, rather than specified statically by an explicit channel name. For example, to send a message e of type τ along the channel referred to by x, the process R' would have the form

$$\$\,(!!\,(x\,;e\,;R'')).$$

Similarly, to receive along the referenced channel, the process R' would have the form

$$\$\,(??\,(x\,;y\,.\,R'')).$$

In both cases the dynamic communication forms evolve to the static communication forms once the referenced channel has been determined.

The syntax of channel types is given by the following grammar:

Typ	τ	::=	$\mathtt{chan}(\tau)$	$\tau\,\mathtt{chan}$	channel type
Exp	e	::=	$\mathtt{ch}[a]$	$\&\,a$	reference
Evt	E	::=	$\mathtt{sndref}(e_1;e_2;P)$	$!!\,(e_1\,;e_2\,;P)$	send
			$\mathtt{rcvref}(e;x\,.\,P)$	$??\,(e\,;x\,.\,P)$	receive

The events $\texttt{sndref}(e_1; e_2; P)$ and $\texttt{rcvref}(e; x.P)$ are dynamic versions of the events $\texttt{snd}[a](e; P)$ and $\texttt{rcv}[a](x.P)$ in which the channel is determined dynamically by evaluation of an expression, rather than statically as a fixed parameter of the event.

The statics of channel references is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a:\tau} \&a : \tau\ \texttt{chan}} \tag{43.21a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\ \texttt{chan} \quad \Gamma \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma P\ \textsf{proc}}{\Gamma \vdash_\Sigma !!\,(e_1; e_2; P)\ \textsf{event}} \tag{43.21b}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau\ \texttt{chan} \quad \Gamma, x : \tau \vdash_\Sigma P\ \textsf{proc}}{\Gamma \vdash_\Sigma ??\,(e; x.P)\ \textsf{event}} \tag{43.21c}$$

The introduction of channel references requires that events be evaluated to determine the referent of a dynamically determined channel. This is accomplished by adding the following rules for evaluation of a synchronizing process:

$$\frac{e\ \textsf{val}_\Sigma}{\$\,(!!\,(\&a; e; P) + E) \underset{\Sigma, a:\tau}{\longmapsto} \$\,(!\,a(e; P) + E)} \tag{43.22a}$$

$$\frac{}{\$\,(??\,(\&a; x.P) + E) \underset{\Sigma, a:\tau}{\longmapsto} \$\,(?\,a(x.P) + E)} \tag{43.22b}$$

In addition we require rules for evaluating each of the constituent expressions of a dynamically determined event; these rules are omitted here for the sake of concision.

43.7 Universality

In the presence of both channel references and recursive types the process calculus with communication is a *universal* programming language. One way to prove this is to show that it is capable of encoding the untyped λ calculus with a call-by-name dynamics (see Chapter 19). The main idea of the encoding is to associate each untyped λ-term, u, a process that represents it. This encoding is defined by induction on the structure of the untyped term, u. For the sake of the induction, the representation is defined relative to a channel reference that represents the context in which the term occurs. Since every term in the untyped λ-calculus is a function, a *context* is

a "call site" for the function consisting of an *argument* and the *return context* for the result of the application. Because of the by-name interpretation of application, variables are represented by references to "servers" that listen on a channel for a channel reference representing a call site, and activate their bindings with that channel reference.

We will write $u\,@\,z$, where u is an untyped λ-term and z is a channel reference representing the context in which u is to be evaluated. The free variables of u will be represented by channels on which we may pass a context. Thus, the channel reference z will be a value of type π, and a free variable, x, will be a value of type $\pi\,\mathtt{chan}$. The type π is chosen to satisfy the isomorphism

$$\pi \cong (\pi\,\mathtt{chan} \times \pi)\,\mathtt{chan}.$$

That is, a context is a channel on which is passed an argument and another context. An argument, in turn, is a channel on which is passed a context.

The encoding of untyped λ-terms as processes is given by the following equations:

$$\begin{aligned}
x\,@\,z &= !!\,(x;z) \\
\lambda\,x.\,u\,@\,z &= \$\,??\,(\mathtt{unfold}(z);\langle x, z'\rangle\,.\,u\,@\,z') \\
u_1(u_2)\,@\,z &= \\
&\qquad \nu\,a_1 : \pi\,\mathtt{chan} \times \pi\,.\,(u_1\,@\,\mathtt{fold}(\&\,a_1)) \parallel \nu\,a : \pi\,.\,*\,\$\,?\,a(z_2\,.\,u_2\,@\,z_2) \parallel !\,a_1(\langle \&\,a, z\rangle)
\end{aligned}$$

Here we have taken a few liberties with the syntax for the sake of readability. We use the asynchronous form of a dynamic send operation, since there is no need to be aware of the receipt of the message. Moreover, we use a product pattern, rather than explicit projections, in the dynamic receive to obtain the components of a pair.

The use of static and dynamic communication operations in the translation merits careful explanation. The call site of a λ-term is determined dynamically; one cannot predict at translation time the context in which the term will be used. In particular, the binding of a variable may be used at many different call sites, corresponding to the multiple possible uses of that variable. On the other hand the channel associated to an argument is determined statically. The server associated to the variable listens on a statically determined channel for a context in which to evaluate its binding, which, as just remarked, is determined dynamically.

As a quick check on the correctness of the representation, consider the

following derivation:

$$
\begin{aligned}
(\lambda\, x.\, x)\,(y)\,@\,z \mapsto^* & \\
& \nu\, a_1 : \tau\,.\,(\$\,?\,a_1(\langle x, z'\rangle\,.!!\,(x; z'))) \parallel \nu\, a : \pi\,.\,*\,\$\,?\,a(z_2\,.!!\,(y; z_2)) \parallel\, !\,a_1(\langle \&\,a, z\rangle) \\
& \mapsto^* \nu\, a : \pi\,.\,*\,\$\,?\,a(z_2\,.!!\,(y; z_2)) \parallel\, !\,a(z) \\
& \mapsto^* \nu\, a : \pi\,.\,*\,\$\,?\,a(z_2\,.!!\,(y; z_2)) \parallel\, !!\,(y; z)
\end{aligned}
$$

Apart from the idle server process listening on channel a, this is just the translation $y\,@\,z$.

43.8 Notes

Process calculi as models of concurrency and interaction were introduced and extensively developed by Tony Hoare [47] and Robin Milner [66]. Milner's original formulation, CCS, was introduced to model pure synchronization, whereas Hoare's, CSP, included value-passing. CCS was subsequently extended to become the π-calculus [66], which include channel-passing. Dozens upon dozens of variations and extensions of CSP, CCS, and the π-calculus have been considered in the literature, and continue to be a subject of intensive study. (See Engberg and Nielsen's survey for an account of some of the critical developments in the area [28].)

The formulation of process calculus given here differs critically from the π-calculus and many of its derivatives (Abadi and Fournet's applied π-calculus [1] being a notable exception) in that it maintains a sharp distinction between variables, which are given meaning by substitution, and channel names, which are symbols that serve as indices of send and receive events. This change is important, because (a) disequality of names is fundamental to synchronization, and (b) substitution does not preserve disequality. The failure to distinguish the two concepts has, for these reasons, led to a morass that is avoided here.

The variable/name distinction places greater importance on types than is customary in process calculus. A type is the range of significance of a variable; whenever a variable is introduced, it necessarily stands for a class of expressions of a specified type. Concomitantly, we must introduce dynamically computed events, which arise by substitution for a variable. This, too, changes the character of the calculus, but in exchange gives rise to a well-behaved calculus of concurrent interaction. Concurrent ML [85], a concurrent programming language inspired by process calculus, also features dynamically generated ("first-class") events; static events arise only within the semantics, and not at the program level.

Chapter 44

Concurrent Algol

In this chapter we integrate concurrency into a full-scale programming language based on the Modernized Algol to obtain Concurrent Algol, $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$. Assignables in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ are replaced by more general primitives for communication among processes. Communication consists of broadcasting a *message* consisting of a *channel* attached to a *payload* of type appropriate to that channel. Such messages are simply dynamically classified values, and channels are therefore just dynamic classes (see Chapter 36 for more on dynamic classification). A broadcast message may be accepted by any process, but only those processes that know its channel (class) may extract the payload from the message; all others must handle it as an inscrutable value of message type.

44.1 Concurrent Algol

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$ is obtained by stripping out assignables from $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$, and adding a syntactic level of *processes*:

Typ	τ	$::=$	$\texttt{cmd}(\tau)$	$\tau\ \texttt{cmd}$	commands
Exp	e	$::=$	$\texttt{do}(m)$	$\texttt{do}\ m$	command
Cmd	m	$::=$	$\texttt{ret}\ e$	$\texttt{ret}\ e$	return
			$\texttt{bnd}(e; x.m)$	$\texttt{bnd}\ x \leftarrow e\ ;\ m$	sequence
Proc	p	$::=$	$\texttt{stop}$	$\mathbf{1}$	idle
			$\texttt{proc}(m)$	$\texttt{proc}(m)$	atomic
			$\texttt{par}(p_1; p_2)$	$p_1 \parallel p_2$	parallel
			$\texttt{new}[\tau](a.p)$	$\nu\, a{:}\tau.\, p$	new channel

The process $\texttt{proc}(m)$ is an atomic process executing the command, m. The other forms of process are adapted from Chapter 43. If Σ has the form $a_1 : \tau_1, \ldots, a_n : \tau_n$, then we sometimes write $\nu\,\Sigma\{p\}$ for the iterated form $\nu\,a_1 : \tau_1 \ldots \nu\,a_n : \tau_n\,.\,p$.

The statics is given by the judgements $\Gamma \vdash_\Sigma e : \tau$ and $\Gamma \vdash_\Sigma m \sim \tau$ introduced in Chapter 37, augmented by the judgement $\vdash_\Sigma p\ \mathsf{proc}$ stating that p is a well-formed process over the signature Σ. The latter judgement is defined by the following rules:

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{44.1a}$$

$$\frac{\vdash_\Sigma m \sim \tau}{\vdash_\Sigma \texttt{proc}(m)\ \mathsf{proc}} \tag{44.1b}$$

$$\frac{\vdash_\Sigma p_1\ \mathsf{proc} \quad \vdash_\Sigma p_2\ \mathsf{proc}}{\vdash_\Sigma p_1 \parallel p_2\ \mathsf{proc}} \tag{44.1c}$$

$$\frac{\vdash_{\Sigma, a:\tau} p\ \mathsf{proc}}{\vdash_\Sigma \nu\,a:\tau\,.\,p\ \mathsf{proc}} \tag{44.1d}$$

Processes are tacitly identified up to structural equivalence, as described in Chapter 43.

The transition judgement $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ states that the process p evolves in one step to the process p' with associated action α. The particular actions are specified when specific commands are introduced in Section 44.2 on page 448. As in Chapter 43 we assume that to each action is associated a complementary action, and that the silent action indexes the unlabelled transition judgement.

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma'\,\{\,\texttt{proc}(m') \parallel p\,\}}{\texttt{proc}(m) \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\,\Sigma'\{\texttt{proc}(m') \parallel p\}} \tag{44.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{proc}(\texttt{ret}\,e) \underset{\Sigma}{\mapsto} \mathbf{1}} \tag{44.2b}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1'}{p_1 \parallel p_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \parallel p_2} \tag{44.2c}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \quad p_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} p_2'}{p_1 \parallel p_2 \underset{\Sigma}{\mapsto} p_1' \parallel p_2'} \tag{44.2d}$$

$$\frac{p \overset{\alpha}{\underset{\Sigma,a:\tau}{\longmapsto}} p' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a:\tau\,.\, p \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a:\tau\,.\, p'} \tag{44.2e}$$

Rule (44.2a) states that a step of execution of the atomic process $\texttt{proc}(m)$ consists of a step of execution of the command m, which may result in the allocation of some set, Σ', of symbols and the creation of a concurrent process, p. This rule implements scope extrusion for classes (channels) by expanding the scope of the declaration of a channel to the context in which the command, m, occurs. Rule (44.2b) states that a completed command evolves to the inert (stopped) process; processes are executed solely for their effect, and not for their value. The remaining rules are those of the process calculus that define the interaction between processes and the allocation of symbols within a process.

The auxiliary judgement $m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\, \Sigma' \{ m' \parallel p' \}$ defines the execution behavior of commands. It states that the command, m, transitions to the command, m', while creating new channels, Σ', and new processes, p'. The action, α, specifies the interactions of which m is capable when executed. As a notational convenience we drop mention of the new channels or processes when either are trivial. It is important that the right-hand side of this judgement be construed as a triple consisting of Σ', m', and p', rather than as a process expression comprising these parts.

The general rules defining this auxiliary judgement are as follows:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \texttt{ret}\, e'} \tag{44.3a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\, \Sigma' \{ m_1' \parallel p' \}}{\texttt{bnd}\, x \leftarrow \texttt{do}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\, \Sigma' \{ \texttt{bnd}\, x \leftarrow \texttt{do}\, m_1' \,;\, m_2 \parallel p' \}} \tag{44.3b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{bnd}\, x \leftarrow \texttt{do}\, \texttt{ret}\, e \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} [e/x]m_2} \tag{44.3c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\texttt{bnd}\, x \leftarrow e_1 \,;\, m_2 \underset{\Sigma}{\overset{\varepsilon}{\hookrightarrow}} \texttt{bnd}\, x \leftarrow e_1' \,;\, m_2} \tag{44.3d}$$

These generic rules are supplemented by rules governing commands for communication and synchronization among processes.

44.2 Broadcast Communication

In this section we consider a very general form of process synchronization called *broadcast*. Processes emit and accept messages of type `clsfd`, the type of dynamically classified values considered in Chapter 36. A message consists of a *channel*, which is its class, and a *payload*, which is a value of the type associated with the channel (class). Recipients may pattern match against a message to determine whether it is of a given class, and, if so, recover the associated payload. No process that lacks access to the class of a message may recover the payload of that message. (See Section 36.4 on page 344 for a discussion of how to enforce confidentiality and integrity restrictions using dynamic classification).

The syntax of the commands pertinent to broadcast communication is given by the following grammar:

Cmd	m	::=	`spawn(`e`)`	`spawn(`e`)`	spawn
			`emit(`e`)`	`emit(`e`)`	emit message
			`acc`	`acc`	accept message
			`newch[`τ`]`	`newch`	new class

The command `spawn(`e`)` spawns a process that executes the encapsulated command given by e. The commands `emit(`e`)` and `acc` emit and accept messages, which are classified values whose class is the channel on which the message is sent. The command `newch[`τ`]` returns a reference to a fresh class carrying values of type τ.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd(unit)}}{\Gamma \vdash_\Sigma \texttt{spawn(}e\texttt{)} \sim \texttt{unit}} \tag{44.4a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma \texttt{emit(}e\texttt{)} \sim \texttt{unit}} \tag{44.4b}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{acc} \sim \texttt{clsfd}} \tag{44.4c}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{newch}[\tau] \sim \texttt{class}(\tau)} \tag{44.4d}$$

The execution of commands for broadcast communication is defined by these rules:

$$\frac{}{\texttt{spawn}(\texttt{do}(m)) \xrightarrow[\Sigma]{\varepsilon} \texttt{ret}\,\langle\rangle \parallel \texttt{proc}(m)} \tag{44.5a}$$

$$\frac{e \xmapsto[\Sigma]{} e'}{\texttt{spawn}(e) \xrightarrow[\Sigma]{\varepsilon} \texttt{spawn}(e')} \tag{44.5b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{emit}(e) \xrightarrow[\Sigma]{e!} \texttt{ret}\,\langle\rangle} \tag{44.5c}$$

$$\frac{e \xmapsto[\Sigma]{} e'}{\texttt{emit}(e) \xrightarrow[\Sigma]{\varepsilon} \texttt{emit}(e')} \tag{44.5d}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{acc} \xrightarrow[\Sigma]{e?} \texttt{ret}\,e} \tag{44.5e}$$

$$\frac{}{\texttt{newch}[\tau] \xrightarrow[\Sigma]{\varepsilon} \nu\, a\!:\!\tau.\texttt{ret}\,(\&\,a)} \tag{44.5f}$$

Rule (44.5c) specifies that `emit(`e`)` has the effect of emitting the message e. Correspondingly, Rule (44.5e) specifies that `acc` may accept (any) message that is being sent.

As usual, the preservation theorem for $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \parallel\}$ ensures that well-typed programs remain well-typed during execution. The proof of preservation requires a lemma governing the execution of commands. First, let us define the judgement $\vdash_\Sigma \alpha$ action by the following rules:

$$\frac{}{\vdash_\Sigma \varepsilon\ \mathsf{action}} \tag{44.6a}$$

$$\frac{\vdash_\Sigma e : \texttt{clsfd}}{\vdash_\Sigma e!\ \mathsf{action}} \tag{44.6b}$$

$$\frac{\vdash_\Sigma e : \texttt{clsfd}}{\vdash_\Sigma e?\ \mathsf{action}} \tag{44.6c}$$

Lemma 44.1. *If* $m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma'\,\{\, m' \parallel p' \,\}$ *and* $\vdash_\Sigma m \sim \tau$, *then* $\vdash_\Sigma \alpha$ *action*, $\vdash_{\Sigma\,\Sigma'} m' \sim \tau$, *and* $\vdash_{\Sigma\,\Sigma'} p'$ *proc.*

Proof. By induction on Rules (44.3). □

With this in hand the proof of preservation is straightforward.

Theorem 44.2 (Preservation)**.** *If* $\vdash_\Sigma p$ *proc and* $p \overset{}{\underset{\Sigma}{\mapsto}} p'$, *then* $\vdash_\Sigma p'$ *proc.*

Proof. By induction on transition, appealing to Lemma 44.1 for the crucial steps. □

Typing does not, however, guarantee progress with respect to unlabelled transition, for the simple reason that there may be no other process with which to communicate. By extending progress to labelled transitions we may state that this is the *only* way for the execution of a process to get stuck.

Theorem 44.3 (Progress)**.** *Suppose that* e val_Σ *for some* e *such that* $\vdash_\Sigma e : \mathtt{clsfd}$. *If* $\vdash_\Sigma p$ *proc, then either* $p \equiv \mathbf{1}$, *or there exists* p' *and* α *such that* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$.

Proof. By induction on Rules (44.1) and (44.4). □

The assumption that there exists a message rules out degenerate situations in which there are no channels, or all channels carry values of an empty type.

44.3 Selective Communication

Broadcast communication provides no means of restricting acceptance to messages of a particular class (that is, of messages on a particular channel). Using broadcast communication we may restrict attention to a particular channel, a, of type, τ, by running the following command:

$$\mathtt{fix}\,\mathit{loop} : \tau\ \mathtt{cmd}\ \mathtt{is}\ \{x \leftarrow \mathtt{acc}\,;\mathtt{match}\,x\,\mathtt{as}\,a \cdot y \Rightarrow \mathtt{ret}\,y\,\mathtt{ow} \Rightarrow \mathtt{emit}(x)\,;\mathtt{run}\,\mathit{loop}\}$$

This command is always capable of receiving a broadcast message. When one arrives, it is examined to determine whether it is classified by the class, a. If so, the underlying value is returned; otherwise the message is rebroadcast to make it available to another process that may be executing a

similar command. *Polling* consists of repeatedly executing the above command until such time as a message of channel a is successfully accepted, if ever. But polling is evidently wasteful of computing resources.

An alternative is to change the language to allow for *selective communication*. Rather than accept any broadcast message, we may confine attention to messages that are sent on any of several possible channels. This may be accomplished by introducing a type, $\texttt{event}(\tau)$, of *events* consisting of a finite choice of accepts all of whose associated payload has the type τ.

Typ	τ	$::=$	$\texttt{event}(\tau)$	$\tau\,\texttt{event}$	events
Exp	e	$::=$	$\texttt{rcv}[a]$	$?\,a$	select
			$\texttt{never}[\tau]$	$\texttt{never}$	null
			$\texttt{or}(e_1;e_2)$	$e_1\,\texttt{or}\,e_2$	choice
Cmd	m	$::=$	$\texttt{sync}(e)$	$\texttt{sync}(e)$	synchronize

Events in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\|\}$ correspond directly to those of the asynchronous process calculus described in Chapter 43. One difference is that the accept event need not carry with it a continuation, as it does in the process calculus; this is handled by the ambient monadic structure on commands. Note, however, that all events in a choice share the same continuation, whereas in process calculus a separate continuation is associated to each event in a choice. Another difference between the two formalisms is that in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\|\}$ events are values of the type $\tau\,\texttt{event}$, whereas in the process calculus events are a special syntactic class, rather than a form of value.

The statics of event expressions is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{rcv}[a] : \texttt{event}(\tau)} \tag{44.7a}$$

$$\frac{}{\Gamma \vdash_{\Sigma} \texttt{never}[\tau] : \texttt{event}(\tau)} \tag{44.7b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \texttt{event}(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \texttt{event}(\tau)}{\Gamma \vdash_{\Sigma} \texttt{or}(e_1;e_2) : \texttt{event}(\tau)} \tag{44.7c}$$

The corresponding dynamics is defined by these rules:

$$\frac{}{\texttt{rcv}[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{44.8a}$$

$$\frac{}{\texttt{never}[\tau]\ \mathsf{val}_{\Sigma}} \tag{44.8b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{val}_\Sigma}{\mathtt{or}(e_1;e_2)\ \mathsf{val}_\Sigma} \tag{44.8c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{or}(e_1;e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1';e_2)} \tag{44.8d}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2 \underset{\Sigma}{\mapsto} e_2'}{\mathtt{or}(e_1;e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1;e_2')} \tag{44.8e}$$

Event values are identified up to structural congruence as described in Chapter 43. This ensures that the ordering of events in a choice is immaterial.

Channel references (see Section 36.2 on page 342) give rise to an additional form of event, $\mathtt{rcvref}(e)$, in which the argument, e, is a reference to the channel on which to accept a message. Its statics is given by the rule

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{class}(\tau)}{\Gamma \vdash_\Sigma \mathtt{rcvref}(e) : \mathtt{event}(\tau)} \tag{44.9a}$$

Its dynamics is defined to dereference its argument and evaluate to an accept event for the referenced channel:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{rcvref}(e) \underset{\Sigma}{\mapsto} \mathtt{rcvref}(e')} \tag{44.10a}$$

$$\frac{}{\mathtt{rcvref}(\mathtt{cls}[a]) \underset{\Sigma,a:\tau}{\longmapsto} \mathtt{rcv}[a]} \tag{44.10b}$$

Turning now to the synchronization command, the statics is given by the following rule:

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \sim \tau} \tag{44.11a}$$

Its execution is defined by these rules in terms of an execution judgement for events to be defined shortly:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{sync}(e) \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \mathtt{sync}(e')} \tag{44.12a}$$

$$\frac{e \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} m}{\texttt{sync}(e) \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} m} \tag{44.12b}$$

Rule (44.12b) specifies that synchronization may take any action engendered by the event given as argument.

The possible actions engendered by an event value are defined by the judgement $e \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} m$, which states that the event value e engenders action α and activates command m. It is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau} \quad \vdash_{\Sigma,a:\tau} e : \tau}{\texttt{rcv}[a] \overset{a\cdot e\,?}{\underset{\Sigma,a:\tau}{\hookrightarrow}} \texttt{ret}(e)} \tag{44.13a}$$

$$\frac{e_1 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} m_1}{\texttt{or}(e_1; e_2) \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} m_1} \tag{44.13b}$$

Rule (44.13a) states that an acceptance on a channel a may synchronize only with messages classified by a. In conjunction with the identification of event values up to structural congruence Rule (44.13b) states that any event among a set of choices may be engender an action.

Selective communication and dynamic events may be used together to implement a communication protocol in which a channel reference is passed on a channel in order to establish a communication path with the recipient. Let a be a channel carrying values of type $\texttt{class}(\tau)$, and let b be a channel carrying values of type τ, so that $\texttt{\&}\,b$ may be passed as a message along channel a. A process that wishes to accept a channel reference on a and then accept on that channel has the form

$$\{x \leftarrow \texttt{sync}(?\,a)\,;y \leftarrow \texttt{sync}(??\,x)\,;\ldots\}.$$

The event $?\,a$ specifies a selective receipt on channel a. Once the value, x, has been accepted, the event $??\,x$ specifies a selective receipt on the channel referenced by x. So, if $\texttt{\&}\,b$ is sent along a, then the event $??\,\texttt{\&}\,b$ evaluates to $?\,b$, which accepts selectively on channel b, even though the receiving process may have no direct access to the channel b itself.

Selective communication may be seen as a simple form of pattern matching in which patterns are restricted to $a \cdot x$, where a is a channel carrying values of some type τ, and x is a variable of type τ. The idea is that selective communication filters for messages that match a pattern of this form,

and proceed by returning the associated value, x. From this point of view it is natural to generalize selective communication to allow arbitrary patterns of type `clsfd`. Since different patterns may bind different variables, it is then natural to associate a separate continuation with each pattern, as in Chapter 15. Basic events are of the form $p \Rightarrow m$, where p is a pattern of type `clsfd`, $x_1, \ldots, x_k$ are its variables, and m is a command involving these variables. Compound events are compositions of such rules, written $r_1 \mid \ldots \mid r_n$, quotiented by structural congruence to ensure that the order of rules is insignificant.

The statics of pattern-driven events may be readily derived from the statics of pattern matching given in Chapter 15. The dynamics is defined by the following rule defining the action engendered by an event:

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \mathtt{clsfd} \quad \theta \Vdash p \lhd e}{p \Rightarrow m \mid rs \xrightarrow[\Sigma]{e?} \hat{\theta}(m)} \tag{44.14}$$

This rule states that we may choose any accept action by a value matching the pattern p, continuing with the corresponding instance of the continuation of the rule.

44.4 Free Assignables as Processes

Scope-free assignables are definable in $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \|\}$ by associating to each assignable a server process that sets and gets the contents of the assignable. To each assignable, a, of type ρ is associated a server that selectively accepts a message on channel a with one of two forms:

1. $\mathtt{get} \cdot (\mathtt{\&}\, b)$, where b is a channel of type ρ. This message requests that the contents of a be sent on channel b.

2. $\mathtt{set} \cdot (\langle e, \mathtt{\&}\, b\rangle)$, where e is a value of type ρ, and b is a channel of type ρ. This message requests that the contents of a be set to e, and that the new contents be transmitted on channel b.

In other words, a is a channel of type τ_{srvr} given by

$$[\mathtt{get} : \rho\ \mathtt{class}, \mathtt{set} : \rho \times \rho\ \mathtt{class}].$$

The server selectively accepts on channel a, then dispatches on the class of the message to satisfy the request.

The server associated with the assignable, a, of type ρ maintains the contents of a using recursion. When called with the current contents of the assignable, the server selectively accepts on channel a, dispatching on the associated request, and calling itself recursively with the (updated, if necessary) contents:

$$\lambda\,(u:\tau_{srvr}\,\texttt{class}.\,\texttt{fix}\,srvr:\rho\to\texttt{void}\,\texttt{cmd}\,\texttt{is}\,\lambda\,(x:\rho.\,\texttt{do}\,\{y\leftarrow\texttt{sync}\,(??\,u)\,;e_{(44.16)}\}))\,. \tag{44.15}$$

The server is a procedure that takes an argument of type ρ, the current contents of the assignable, and yields a command that never terminates, because it restarts the server loop after each request. The server selectively accepts a message on channel a, and dispatches on it as follows:

$$\texttt{case}\,y\,\{\texttt{get}\cdot z\Rightarrow e_{(44.17)}\mid\texttt{set}\cdot\langle x',z\rangle\Rightarrow e_{(44.18)}\}. \tag{44.16}$$

A request to get the contents of the assignable a is served as follows:

$$\{\_\leftarrow\texttt{emit}\,(\texttt{mk}\,(z;x))\,;\texttt{run}\,srvr\,(x)\} \tag{44.17}$$

A request to set the contents of the assignable a is served as follows:

$$\{\_\leftarrow\texttt{emit}\,(\texttt{mk}\,(z;x'))\,;\texttt{run}\,srvr\,(x')\} \tag{44.18}$$

The type $\tau\,\texttt{ref}$ is defined to be $\tau\,\texttt{class}$, the type of channels (classes) carrying a value of type τ. A new free assignable is created by the command $\texttt{ref}\,e_0$, which is defined to be

$$\{x\leftarrow\texttt{newch}\,;\_\leftarrow\texttt{spawn}\,(e_{(44.15)}\,(x)\,(e_0))\,;\texttt{ret}\,x\}. \tag{44.19}$$

A channel carrying a value of type τ_{srvr} is allocated to server as the name of the assignable, and a new server is spawned that accepts requests on that channel, with initial value e_0.

The commands $@\,e_0$ and $e_0:=e_1$ send a message to the server to get and set the contents of an assignable. The code for $@\,e_0$ is as follows:

$$\{x\leftarrow\texttt{newch}\,;\_\leftarrow\texttt{emit}\,(\texttt{mk}\,(e_0;\texttt{get}\cdot x))\,;\texttt{sync}\,(??\,(x))\} \tag{44.20}$$

A channel is allocated for the return value, the server is contacted with a `get` message specifying this channel, and the result of receiving on this channel is returned. Similarly, the code for $e_0:=e_1$ is as follows:

$$\{x\leftarrow\texttt{newch}\,;\_\leftarrow\texttt{emit}\,(\texttt{mk}\,(e_0;\texttt{set}\cdot\langle e_1,x\rangle))\,;\texttt{sync}\,(??\,(x))\} \tag{44.21}$$

44.5 Notes

Concurrent Algol is a synthesis of process calculus and Modernized Algol, and may be seen as a "Algol-like" reformulation of Concurrent ML [85] in which interaction is confined to the command modality. The design is influenced by Brookes's Parallel Algol [16], though the term "parallel" in that work is, as argued in Chapter 41, misleading. The reduction of channels to dynamic classification appears to be new. Most work on concurrent interaction seems to take the notion of communication channel as a central concept (but see Gerlernter's Linda [30] for an alternative viewpoint, albeit in a unityped setting).

Chapter 45

Distributed Algol

A *distributed* computation is one that takes place at many different *sites*, each of which controls some *resources* located at that site. For example, the sites might be nodes on a network, and a resource might be a device or sensor located at that site, or a database controlled by that site. Only programs that execute at a particular site may access the resources situated at that site. Consequently, command execution always takes place at a particular site, called the *locus of execution*. Access to resources at a remote site from a local site is achieved by moving the locus of execution to the remote site, running code to access the local resource, and returning a value to the local site.

In this chapter we consider the language $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \| \ @\}$, an extension of Concurrent Algol with a *spatial* type system that mediates access to located resources on a network. The type safety theorem ensures that all accesses to a resource controlled by a site are through a program executing at that site, even though references to local resources may be freely passed around to other sites on the network. The key idea is that channels and events are *located* at a particular site, and that synchronization on an event may only occur at the site appropriate to that event. Issues of concurrency, which are to do with non-deterministic composition, are thereby cleanly separated from those of distribution, which are to do with the locality of resources on a network.

The concept of location in $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \| \ @\}$ is sufficiently abstract that it admits another useful interpretation that can be useful in computer security settings. The "location" of a computation may also be thought of as the *principal* on whose behalf the computation is executing. From this point of view, a local resource is one that is accessible to a particular principal,

and a mobile computation is one that may be executed by any principal. Movement from one location to another may then be interpreted as executing a piece of code on behalf of another principal, returning its result to the principal that initiated the transfer.

45.1 Statics

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \, @\}$ is inspired by the *possible worlds* interpretation of modal logic. Under that interpretation the truth of a proposition is relative to a *world*, which determines the state of affairs described by that proposition. A proposition may be true in one world, and false in another. For example, one may use possible worlds to model counterfactual reasoning, in which one postulates that certain facts that happen to be true in this, the *actual*, world, might be otherwise in some other, *possible*, world. For instance, in the actual world you, the reader, are reading this book, but in a possible world you may never have taken up the study of programming languages at all. Of course not everything is possible: there is no possible world in which $2 + 2$ is other than 4, for example. Moreover, once a commitment has been made to one counterfactual, others are ruled out. We say that one world is *accessible* from another when the first is a sensible counterfactual relative to the first. So, for example, one may consider that relative to a possible world in which you are the king, there is no further possible world in which someone else is also the king (there being only one sovereign).

In $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \, @\}$ we shall interpret possible worlds as sites on a network, with accessibility between worlds expressing network connectivity. We postulate that every site is connected to itself (reflexivity); that if one site is reachable from another, then the second is also reachable from the first (symmetry); and that if a site is reachable from a reachable site, then this site is itself reachable from the first (transitivity). From the point of view of modal logics, the type system of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \, @\}$ is derived from the logic **S5**, for which accessibility is an equivalence relation.

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \, @\}$ is a modification of that of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$. The following grammar summarizes the key changes:

Typ	τ	::=	$\texttt{cmd}[w](\tau)$	$\tau\,\texttt{cmd}[w]$	commands
			$\texttt{chan}[w](\tau)$	$\tau\,\texttt{chan}[w]$	channels
			$\texttt{event}[w](\tau)$	$\tau\,\texttt{event}[w]$	events
Cmd	m	::=	$\texttt{at}[w](m)$	$\texttt{at}\,w\,\{m\}$	change site

The command, channel, and event types are indexed by the site, w, to which they pertain. There is a new form of command, $\texttt{at}[w](m)$, that changes the locus of execution from one site to another.

A signature, Σ, in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| @\}$, consists of a finite set of declarations of the form $a : \rho @ w$, where ρ is a type and w is a site. Such a declaration specifies that a is a channel carrying a payload of type ρ located at the site w. We may think of a signature, Σ, as a family of signatures, Σ_w, one for each world w, containing the decalarations of the channels located at that world. This partitioning corresponds to the idea that channels are *located* resources in that they are uniquely associated with a site. They may be handled passively at other sites, but their only active role is at the site at which they are declared.

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| @\}$ is given by the following two judgement forms:

$\Gamma \vdash_\Sigma e : \tau$	expression typing
$\Gamma \vdash_\Sigma m \sim \tau @ w$	command typing

The expression typing judgement is independent of the site. This corresponds to the idea that the values of a type have a site-independent meaning: the number 3 is the number 3, regardless of where it is used. On the other hand commands can only be executed at a particular site, since they depend on the state located at that site.

A representative selection of the rules defining the statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| @\}$ is given below:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau @ w}{\Gamma \vdash_\Sigma \texttt{do}(m) : \texttt{cmd}[w](\tau)} \tag{45.1a}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a:\rho@w} \texttt{ch}[a] : \texttt{chan}[w](\rho)} \tag{45.1b}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{never}[\tau] : \texttt{event}[w](\tau)} \tag{45.1c}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a:\rho@w} \texttt{rcv}[a] : \texttt{event}[w](\rho)} \tag{45.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{chan}[w](\tau)}{\Gamma \vdash_\Sigma \texttt{rcvref}(e) : \texttt{event}[w](\tau)} \tag{45.1e}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{event}[w](\tau) \quad \Gamma \vdash_\Sigma e_2 : \texttt{event}[w](\tau)}{\Gamma \vdash_\Sigma \texttt{or}(e_1; e_2) : \texttt{event}[w](\tau)} \tag{45.1f}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{event}[w](\tau)}{\Gamma \vdash_\Sigma \texttt{sync}(e) \sim \tau @ w} \tag{45.1g}$$

$$\frac{\Gamma \vdash_\Sigma m' \sim \tau' \mathbin{@} w'}{\Gamma \vdash_\Sigma \mathtt{at}[w']\,(m') \sim \tau' \mathbin{@} w} \tag{45.1h}$$

Rule (45.1a) states that the type of an encapsulated command records the site at which the command is to be executed. Rules (45.1d) and (45.1e) specify that the type of a (static or dynamic) receive event records the site at which the channel resides. Rules (45.1c) and (45.1f) state that a choice can only be made between events at the same site; there are no cross-site choices. Rule (45.1g) states that the `sync` command returns a value of the same type as that of the event, and may be executed only at the site to which the given event pertains. Finally, Rule (45.1h) states that to execute a command at a site, w', requires that the command pertain to that site. The returned value is then passed to the original site.

45.2 Dynamics

The dynamics is given by a labelled transition judgement between processes, much as in Chapter 44. The principal difference is that the atomic process consisting of a single command has the form $\mathtt{proc}[w]\,(m)$, which specifies the site, w, at which the command, m, is to be executed. The dynamics of processes remains much as in Chapter 44, except for the following rules governing the atomic process:

$$\frac{m \xrightarrow[\Sigma,w]{\alpha} \nu\,\Sigma'\,\{\,m' \parallel p\,\}}{\mathtt{proc}[w]\,(m) \xmapsto[\Sigma]{\alpha} \nu\,\Sigma'\,\{\,\mathtt{proc}[w]\,(m') \parallel p\,\}} \tag{45.2a}$$

$$\frac{}{\mathtt{proc}[w]\,(\mathtt{ret}(\langle\rangle)) \xmapsto[\Sigma]{\varepsilon} \mathtt{stop}} \tag{45.2b}$$

The command execution judgement

$$m \xrightarrow[\Sigma,w]{\alpha} \nu\,\Sigma'\,\{\,m' \parallel p\,\}$$

states that the command, m, when executed at site, w, may undertake the action, α, and in the process create new channels, Σ', and a new process, p. (The result of the transition is not a process expression, but rather should be construed as a structure having three parts, the newly allocated channels, the newly created processes, and a new command; we omit any part when it is trivial.) This may be understood as a family of judgements indexed

by signatures, Σ, and sites, w. At each site there is an associated labelled transition system defining concurrent interaction of processes *at that site*.

The command execution judgement is defined by the following rules:

$$\frac{}{\texttt{spawn}(m) \xrightarrow[\Sigma,w]{\varepsilon} \texttt{ret}(\langle\rangle) \parallel \texttt{proc}[w](m)} \tag{45.3a}$$

$$\frac{}{\texttt{newch}[\tau] \xrightarrow[\Sigma,w]{\varepsilon} \nu\, a{:}\tau@w.\,\texttt{ret}\,(\&\,a)} \tag{45.3b}$$

$$\frac{m \xrightarrow[\Sigma,w']{\alpha} \nu\,\Sigma'\,\{\, m' \parallel p' \,\}}{\texttt{at}[w'](m) \xrightarrow[\Sigma,w]{\alpha} \nu\,\Sigma'\,\{\, \texttt{at}[w'](m') \parallel p' \,\}} \tag{45.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{at}[w'](\texttt{ret}(e)) \xrightarrow[\Sigma,w]{\varepsilon} \texttt{ret}(e)} \tag{45.3d}$$

$$\frac{e \xrightarrow[\Sigma]{\alpha} m}{\texttt{sync}(e) \xrightarrow[\Sigma,w]{\alpha} m} \tag{45.3e}$$

Rule (45.3a) states that new processes created at a site remain at that site—the new process executes the given command at the current site. Rules (45.3c) and (45.3d) state that the command $\texttt{at}[w'](m)$ is executed at site w by executing m at site w', and returning the result to the site w. Rule (45.3e) states that an action may be undertaken at site w if the given event engenders that action. Notice that no cross-site synchronization is possible. Movement between sites is handled separately from synchronization among the processes at a site.

45.3 Safety

The safety theorem for $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \parallel @\}$ ensures that synchronization on a channel may only occur at the site on which the channel resides, even though channel references may be propagated from one site to another during a computation. By the time the reference is resolved and synchronization is attempted the computation will, as a consequence of typing, be located at the appropriate site.

The key to the safety proof is the definition of a well-formed process. The judgement $\vdash_\Sigma p$ proc, which states that the process p is well-formed. Most importantly, the following rule governs the formation of atomic processes:

$$\frac{\vdash_\Sigma m \sim \texttt{unit} \mathbin{@} w}{\vdash_\Sigma \texttt{proc}[w]\,(m)\ \texttt{proc}} \tag{45.4}$$

That is, an atomic process is well-formed if and only if the command it is executing is well-formed at the site at which the process is located.

The proof of preservation relies on a lemma stating the typing properties of the execution judgement.

Lemma 45.1 (Execution)**.** *Suppose that* $m \xrightarrow[\Sigma, w]{\alpha} \nu\,\Sigma'\,\{\, m' \parallel p \,\}$. *If* $\vdash_\Sigma m \sim \tau \mathbin{@} w$, *then* $\vdash_\Sigma \alpha$ *action and* $\vdash_\Sigma \nu\,\Sigma\{\texttt{proc}\,[w]\,(m')\parallel p\}$ *proc.*

Proof. By a straightforward induction on Rules (45.3). □

Theorem 45.2 (Preservation)**.** *If* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ *and* $\vdash_\Sigma p$ *proc, then* $\vdash_\Sigma p'$ *proc.*

Proof. By induction on Rules (45.1), appealing to Lemma 45.1 for atomic processes. □

The progress theorem states that the only impediment to execution of a well-typed program is the possiblity of synchronizing on an event that will never arise.

Theorem 45.3 (Progress)**.** *If* $\vdash_\Sigma p$ *proc, then either* $p \equiv \mathbf{1}$ *or there exists* α *and* p' *such that* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$.

45.4 Situated Types

The foregoing formulation of $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \parallel @\}$ relies on indexing command, channel, and event types by the site to which they pertain so that values of these types may be passed around at will without fear of misinterpretation. The price to pay, however, is that the command, channel, and event types are indexed by the site to which they pertain, leading to repetition and redundancy. One way to mitigate this cost is to separate out the *skeleton*, ϕ, of a type from its *specialization* to a particular site, w, which is written $\phi\langle w\rangle$.

We will now reformulate the statics of $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \parallel @\}$ using judgements of the form $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$ and $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, where Φ consists of

hypotheses of the form $x_i : \phi_i @ w_i$. The type of an expression or command is factored into two parts, the skeleton, ϕ, and a site, w, at which to specialize it. The meaning of the factored judgements is captured by the following conditions:

1. If $\Phi \vdash_\Sigma e : \phi @ w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.
2. If $\Phi \vdash_\Sigma m \sim \phi @ w$, then $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w\rangle @ w$.

If Φ is a context of the form $x_1 : \phi_1 @ w_1, \ldots, x_n : \phi_n @ w_n$, then $\widehat{\Phi}$ is the context $x_1 : \phi_1\langle w_1\rangle, \ldots, x_n : \phi_n\langle w_n\rangle$.

The syntax of skeletons is similar to that for types, with the addition of a means of specializing a skeleton to a particular site.

Fam	ϕ	::=			
			$\mathtt{nat}$	$\mathtt{nat}$	numbers
			$\mathtt{arr}(\phi_1;\phi_2)$	$\phi_1 \to \phi_2$	functions
			$\mathtt{cmd}(\phi)$	$\phi\,\mathtt{cmd}$	computations
			$\mathtt{chan}(\phi)$	$\phi\,\mathtt{chan}$	channels
			$\mathtt{event}(\phi)$	$\phi\,\mathtt{event}$	events
			$\mathtt{at}[w](\phi)$	$\phi\,\mathtt{at}\,w$	situated

The situated type, $\phi\,\mathtt{at}\,w$, which fixes the interpretation of ϕ at the site w.

The instantiation of a family, ϕ, at a site, w, is written $\phi\langle w\rangle$, and is inductively defined by the following rules:

$$\frac{}{\mathtt{nat}\langle w\rangle = \mathtt{nat}} \tag{45.5a}$$

$$\frac{\phi_1\langle w\rangle = \tau_1 \quad \phi_2\langle w\rangle = \tau_2}{(\phi_1 \to \phi_2)\langle w\rangle = \tau_1 \to \tau_2} \tag{45.5b}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{cmd}\langle w\rangle = \tau\,\mathtt{cmd}[w]} \tag{45.5c}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{chan}\langle w\rangle = \tau\,\mathtt{chan}[w]} \tag{45.5d}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{event}\langle w\rangle = \tau\,\mathtt{event}[w]} \tag{45.5e}$$

$$\frac{\phi\langle w'\rangle = \tau'}{(\phi\,\mathtt{at}\,w')\langle w\rangle = \tau'} \tag{45.5f}$$

Crucially, Rule (45.5f) states that the situated family $\phi\,\mathtt{at}\,w'$ is to be interpreted at w by the interpretation of ϕ at w'. Otherwise instantiation serves

merely to decorate the constituent command, channel, and event skeletons with the site at which they are being interpreted.

Any type, τ, of $\mathcal{L}\{\mathtt{nat\ cmd} \rightharpoonup \| \ @\}$ may be embedded as a constant family, $\mathtt{exactly}(\tau)$, such that $\mathtt{exactly}(\tau)\langle w\rangle = \tau$ for any site w. The constant family is inductively defined by the following rules:

$$\frac{}{\mathtt{exactly(nat)} = \mathtt{nat}} \tag{45.6a}$$

$$\frac{\mathtt{exactly}(\tau_1) = \phi_1 \quad \mathtt{exactly}(\tau_2) = \phi_2}{\mathtt{exactly}(\tau_1 \to \tau_2) = \phi_1 \to \phi_2} \tag{45.6b}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\ \mathtt{cmd}[w]) = \phi\ \mathtt{cmd\ at}\ w} \tag{45.6c}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\ \mathtt{chan}[w]) = \phi\ \mathtt{chan\ at}\ w} \tag{45.6d}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\ \mathtt{event}[w]) = \phi\ \mathtt{event\ at}\ w} \tag{45.6e}$$

It is easy to check that $\mathtt{exactly}(\tau)$ is a constant family:

Lemma 45.4. *For any site w, $\mathtt{exactly}(\tau)\langle w\rangle = \tau$.*

The statics of $\mathcal{L}\{\mathtt{nat\ cmd} \rightharpoonup \| \ @\}$ may be given in factored form, as is illustrated by the following selection of typing rules:

$$\frac{\Phi \vdash_\Sigma e : \phi\ @\ w}{\Phi \vdash_\Sigma \mathtt{ret}\ e \sim \phi\ @\ w} \tag{45.7a}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi_1\ @\ w \quad \Phi, x : \phi_1\ @\ w \vdash_\Sigma m_2 \sim \phi_2\ @\ w}{\Phi \vdash_\Sigma \mathtt{bnd}\ x \leftarrow e_1\ ;\ m_2 \sim \phi_2\ @\ w} \tag{45.7b}$$

$$\frac{\Phi \vdash_\Sigma m \sim \phi\ @\ w}{\Phi \vdash_\Sigma \mathtt{do}\ m : \phi\ \mathtt{cmd}\ @\ w} \tag{45.7c}$$

$$\frac{\mathtt{exactly}(\rho) = \phi}{\Phi \vdash_{\Sigma, a:\rho@w} \mathtt{\&}\ a : \phi\ \mathtt{chan}\ @\ w} \tag{45.7d}$$

$$\frac{}{\Phi \vdash_\Sigma \mathtt{never} : \phi\ \mathtt{event}\ @\ w} \tag{45.7e}$$

$$\frac{\mathtt{exactly}(\rho) = \phi}{\Phi \vdash_{\Sigma, a:\rho@w} \mathtt{?}\ a : \phi\ \mathtt{event}\ @\ w} \tag{45.7f}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \mathtt{chan} \mathbin{@} w}{\Phi \vdash_\Sigma ??\, e : \phi\ \mathtt{event} \mathbin{@} w} \tag{45.7g}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi\ \mathtt{event} \mathbin{@} w \quad \Phi \vdash_\Sigma e_2 : \phi\ \mathtt{event} \mathbin{@} w}{\Phi \vdash_\Sigma e_1\ \mathtt{or}\ e_2 : \phi\ \mathtt{event} \mathbin{@} w} \tag{45.7h}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \mathtt{event} \mathbin{@} w}{\Phi \vdash_\Sigma \mathtt{sync}(e) \sim \phi \mathbin{@} w} \tag{45.7i}$$

$$\frac{\Phi \vdash_\Sigma m' \sim \phi' \mathbin{@} w' \quad \phi'\ \mathsf{mobile}}{\Phi \vdash_\Sigma \mathtt{at}\, w' \{m'\} \sim \phi' \mathbin{@} w} \tag{45.7j}$$

Rule (45.7d) specifies that a reference to a channel carrying a value of type ρ is classified by the *constant* family yielding the type ρ at each site. Rule (45.7j) is the most interesting rule, since it include a restriction on the family ϕ'. To see how this arises, inductively we have that $\widehat{\Phi} \vdash_\Sigma m' \sim \phi'\langle w' \rangle \mathbin{@} w'$, which is enough to ensure that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\, w' \{m'\} \sim \phi'\langle w' \rangle \mathbin{@} w$. But we are required to show that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\, w' \{m'\} \sim \phi'\langle w \rangle \mathbin{@} w$! This will only be the case if $\phi'\langle w \rangle = \phi'\langle w' \rangle$, which is to say that ϕ' is a constant family, whose meaning does not depend on the site at which it is instantiated.

The judgement ϕ mobile states that ϕ is a mobile family. It is inductively defined by the following rules:

$$\frac{}{\mathtt{nat}\ \mathsf{mobile}} \tag{45.8a}$$

$$\frac{\phi_1\ \mathsf{mobile} \quad \phi_2\ \mathsf{mobile}}{\phi_1 \to \phi_2\ \mathsf{mobile}} \tag{45.8b}$$

$$\frac{}{\phi\ \mathtt{at}\ w\ \mathsf{mobile}} \tag{45.8c}$$

The remaining families are not mobile, precisely because their instantiation specifies the site of their instances; these do not determine constant families.

Lemma 45.5.

1. *If ϕ mobile, then for every w and w', $\phi\langle w \rangle = \phi\langle w' \rangle$.*
2. *For any type τ, $\mathtt{exactly}(\tau)$ mobile.*

We may then verify that the intended interpretation is valid:

Theorem 45.6.

1. *If* $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, *then* $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w \rangle$.
2. *If* $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, *then* $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w \rangle \mathbin{@} w$.

Proof. By induction on Rules (45.7). □

45.5 Notes

The use of a spatial modality to express locality and mobility constraints in a distributed program was inspired by ML5 [102]. The separation of locality concerns from concurrency concerns is expressed here by supporting communication and synchronization within a site, and treating movement between sites separately. The formulation of situated types is based on Licata and Harper [57].

Part XVIII

Modularity

Chapter 46

Components and Linking

Modularity is the most important technique for controlling the complexity of programs. Programs are decomposed into separate *components* with precisely specified, and tightly controlled, interactions. The pathways for interaction among components determine dependencies that constrain the process by which the components are integrated, or *linked*, to form a complete system. Different systems may use the same components, and one system may use a given component more than once. Sharing of components amortizes the cost of their development across systems, and helps limit errors by limiting coding effort.

Modularity is not limited to programming languages. In mathematics the proof of a theorem is decomposed into a collection of definitions and lemmas. Cross-references among lemmas determine a dependency structure that constrains their integration to form a complete proof of the main theorem. Of course, one man's theorem is another man's lemma; there is no instrinsic limit on the depth and complexity of the hierarchies of results in mathematics. Mathematical structures are themselves composed of separable parts, as, for example a Lie group is a group structure on a manifold.

Modularity arises from the structural properties of the hypothetical and general judgements. Dependencies among components are expressed a by free variables with typing assumptions stating the presumed properties of the component. Linking consists of substitution and discharge of the hypothesis.

46.1 Simple Units and Linking

To decompose a program into units amounts to exploiting the transitivity of the hypothetical judgement (see Chapter 3). The decomposition may be expressed as an interaction between two parties, the *client* and the *implementor*, that is mediated by an agreed-upon contract, called an *interface*. The client *assumes* that the implementor upholds the contract, and the implementor *guarantees* that the contract will be upheld. The assumption made by the client amounts to a declaration of its dependence on the implementor that is discharged by *linking* the two parties in accordance with their agreed-upon contract.

The interface that mediates the interaction between a client and an implementor is a *type*. Linking is nothing other than the implementation of the composite structural rules of substitution and transitivity:

$$\frac{\Gamma \vdash e_{impl} : \tau_{intf} \quad \Gamma, x : \tau_{intf} \vdash e_{client} : \tau_{client}}{\Gamma \vdash [e_{impl}/x]e_{client} : \tau_{client}} \tag{46.1}$$

The type τ_{intf} is the interface type. It defines the capabilities to be provided by the implementor, e_{impl}, that are relied upon by the client, e_{client}. The free variable, x, expresses the dependency of e_{client} on e_{impl}. Use of the implementation by the client is made by using the variable, x.

The interface type, τ_{intf}, is the contract between the client and the implementor. It determines the properties of the implementation on which the client may depend and, at the same time, determines the obligations that the implementor must fulfill. The simplest form of interface type is a labelled tuple type (see Chapter 13). Such a type has the form $\langle f_1 : \tau_1, \ldots, f_n : \tau_n \rangle$, specifying a component with "operations" f_i of type τ_i, which are typically function types. Such a type is commonly called an *application program interface*, or *api*, since it determines the operations that the client (application) may expect from the implementor. A more sophisticated form of interface is one that defines an abstract type (as described in Chapter 23). Such an interface type has the form $\exists(t.\langle f_1 : \tau_1, \ldots, f_n : \tau_n \rangle)$, which defines an abstract type, t, representing the internal state of an "abstract machine" whose "instruction set" consists of the operations $f_1, \ldots, f_n$ whose types may involve t. Being abstract, the type t is not revealed to the client, but is known only to the implementor.[1]

Conceptually, linking is just substitution, but practically this can be implemented in a variety of ways. One method is called *separate compilation*.

[1] See Chapters 23 and 51 for a discussion of type abstraction.

The expressions e_{client} and e_{impl}, called in this context *source modules*, are translated (compiled) into another, lower-level, language, resulting in *object modules*. Linking consists of performing the required substitution at the level of the object language in such a way that the result corresponds to the translation of $[e_{impl}/x]e_{client}$.[2] Another method, called *separate checking*, shifts the requirement for translation to the linker. The client and implementor units are ensured to be type-correct with respect to the interface requirements, but are not translated into lower-level form. Linking then consists of translating the composite program as a whole, often resulting in a more efficient outcome than would be possible when compiling separately.

A more sophisticated, and widely used, implementation of substitution is called *dynamic linking*. Informally, this means that execution of the client commences before the implementation of the components on which it depends are provided. Rather than link prior to execution, we instead execute and link "on the fly." At first blush this might seem to be a radical departure from the methodology developed in this book, since we have consistently required that execution be defined only on expressions with no free variables. But looks can be deceiving. What is really going on with dynamic linking is that the client is implemented by a *stub* that forwards accesses to a stored implementation (typically, in a "file system" or similar data structure). The actual implementation code is not accessed until the client requests it, which may not happen at all. This tends to reduce latency and makes it possible to replace the implementation without recompiling the client.

What is important is not how linking is implemented, but rather that the linking principle enables *separate development*. Once the common interface has been agreed upon, the client and implementor are free to proceed with their work independently of one another. All that is required is that both parties complete their work before the system as a whole can be built.

46.2 Initialization and Effects

Linking resolves the dependencies among the components of a program by substitution. This view is valid so long as the components are given by pure expressions, those that evaluate to a value without inducing any

[2] The correspondence need not be exact, but must be equivalent for all practical purposes, in the sense discussed in Chapter 50.

effects. For in such cases there is no problem with the replication, or complete omission, of a component arising from repeated, or absent, uses of a variable representing it. But what if the expression defining the implementation of a component has an effect when evaluated? At a minimum replication of the component implies replication of its effects. Worse, effects introduce *implicit dependencies* among components that are not apparent from their types. For example, if each of two components mutates a shared assignable, the order in which they are linked with a client program effects the behavior of the whole.

This may raise doubts about the treatment of linking as substitution, but on closer inspection it becomes clear that implicit dependencies are naturally managed by paying attention to the modal distinction between expressions and commands introduced in Chapter 37. Specifically, a component that may have an effect when executed does not have type τ_{intf} of implementations of the interface type, but rather the type $\tau_{intf}\,\texttt{cmd}$ of encapsulated commands that, when executed, have effects and yield such an implementation. Being encapsulated, a value of this type is itself free of effects, but it may have effects when evaluated.

The distinction between the types τ_{intf} and $\tau_{intf}\,\texttt{cmd}$ is mediated by the sequentialization command introduced in Chapter 37. For the sake of generality, let us assume that the client is itself an encapsulated command of type $\tau_{client}\,\texttt{cmd}$, so that it may itself have effects when executed, and may serve as a component of a yet larger system. Assuming that the client refers to the encapsulated implementation by the variable x, the command

$$\texttt{bnd}\,x \leftarrow x\,;\texttt{run}\,e_{client}$$

first determines the implementation of the interface by running the encapsulated command, x, then running the client code with the result bound to x. The implicit dependencies of the client on the implementor are made explicit by the sequentialization command, which ensures that the implementor's effects occur prior to those of the client, precisely because the client depends on the implementor for its execution.

More generally, to manage such interactions in a large program it is common to isolate an *initialization procedure* whose role is to stage the effects engendered by the various components according to some policy or convention. Rather than attempt to survey all possible policies, which are numerous and complex, let us simply observe that the upshot of such conventions is that the initialization procedure is a command of the form

$$\{x_1 \leftarrow x_1\,;\ldots x_n \leftarrow x_n\,;m_{main}\},$$

where $x_1, \ldots, x_n$ represent the components of the system and m_{main} is the main (startup) routine. After linking the initialization procedure has the form

$$\{x_1 \leftarrow e_1 \,; \ldots x_n \leftarrow e_n \,; m_{main}\},$$

where $e_1, \ldots, e_n$ are the encapsulated implementations of the linked components. When the initialization procedure is executed, it is results in the substitution

$$[v_1, \ldots, v_n / x_1, \ldots, x_n] m_{main},$$

where the expressions $v_1, \ldots, v_n$ represent the values resulting from executing $e_1, \ldots, e_n$, respectively, and the implicit effects have occurred in the order specified by the initializer.

46.3 Notes

The relationship between the structural properties of entailment and the practical problem of separate development was implicit in much early work on programming languages, but became explicit once the correspondence between propositions and types was developed. There are many indications of this correspondence, for example in Girard's *Proofs and Types* [33] and Martin-Löf's *Intuitionistic Type Theory* [61], which was first made explicit by Cardelli [19].

Chapter 47

Type Abstractions and Type Classes

An interface is a contract that specifies the rights of a client and the responsibilities of an implementor. Being a specification of behavior, an interface is a type. In principle any type may serve as an interface, but in practice it is usual to structure code into *modules* consisting of separable and reusable components. An interface specifies the behavior of a module expected by a client and imposed on the implementor. It is the fulcrum on which is balanced the tension between separability and integration. As a rule, a module should have a well-defined behavior that can be understood separately, but it is equally important that it be easy to combine modules form an integrated whole.

A fundamental question is, what is the type of a module? That is, what form should an interface take? One long-standing idea is for an interface to be a labelled tuple of functions and procedures with specified types. The types of the fields of the tuple are traditionally called *function headers*, since they summarize the call and return types of each function. Using interfaces of this form is called *procedural abstraction*, because it limits the dependencies between modules to a specified set of procedures. One may think of the fields of the tuple as being the instruction set of an abstract machine. The client makes use of these instructions in its code, and the implementor agrees to provide their implementations.

The problem with procedural abstraction is that it does not provide as much insulation as one might like. For example, a module that implements a dictionary must expose in the types of its operations the exact representation of the tree as, say, a recursive type (or, in more rudimentary languages,

a pointer to a structure that itself may contain such pointers). Yet the client really should not depend on this representation: the whole point of abstraction is to eliminate such dependencies! The solution, as discussed in Chapter 23, is to extend the abstract machine metaphor to allow the internal state of the machine to be hidden from the client. In the case of a dictionary the representation of the dictionary as a binary search tree is hidden by existential quantification. This is called *type abstraction*, because the type of the underlying data (state of the abstract machine) is hidden.

Type abstraction is a powerful method for limiting the dependencies among the modules that constitute a program. It is very useful in many circumstances, but is not universally applicable. It is not always appropriate to use abstract types; often it is useful to expose, rather than obscure, type information across a module boundary. A typical example is the implementation of a dictionary, which is a mapping from keys to values. To use, say, a binary search tree to implement a dictionary, we require that the key type admit a total ordering with which keys can be compared. The dictionary abstraction does not depend on the exact type of the keys, but only requires that the key type be constrained to provide a comparison operation. A *type class* is a specification of such a requirement. The class of comparable types, for example, specifies a type, t, together with an operation, `leq`, of type $(t \times t) \to$ `bool` with which to compare them. Superficially, such a specification looks like a type abstraction, because it specifies a type and one or more operations on it, but with the important difference that the type, t, is not hidden from the client. For if it were, the client would only be able to compare keys using `leq`, but would have no means of obtaining keys to compare! A type class, in contrast to a type abstraction, is not intended to be an exhaustive specification of the operations on a type, but rather as a constraint on its behavior expressed by demanding that certain operations, such as comparison, be available, without limiting the other operations that might be defined on it.

Type abstractions and type classes are two extremal cases of a general concept of module type that we shall discuss in detail in this chapter. The crucial idea is the *controlled revelation* of type information across module boundaries. Type abstractions are opaque; type classes are transparent. These are both instances of *translucency*, which arises from the combination of existential types (Chapter 23), subtyping (Chapter 25), and singleton kinds and subkinding (Chapter 26). Unlike in Chapter 23, however, we will distinguish the types of modules, which we will called *signatures*, from the types of ordinary values. The distinction is not necessary, and can be relaxed, but it will be helpful to keep the two concepts separate at the

outset.

47.1 Type Abstraction

Type abstraction is captured by a form of existential type quantification similar to that described in Chapter 23. For example, a dictionary with keys of type τ_{key} and values of type τ_{val} implements the signature, σ_{dict}, defined as follows:

$$\mathtt{sig}\, t :: \mathtt{T}\, \{\langle \mathtt{empty} : t, \mathtt{insert} : \tau_{\mathsf{key}} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{find} : \tau_{\mathsf{key}} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt} \rangle\}.$$

The type variable, t, of kind `T` is the abstract type of dictionaries on which are defined three operations, `empty`, `insert`, and `find`, with the specified types. It is not essential to fix the type τ_{val}, because the dictionary operations impose no restrictions on it; we will do so only for the sake of simplicity.[1] However, it is essential, at this stage, that the key type, τ_{key}, be fixed, for reasons that will become clearer as we proceed.

An implementation of the signature σ_{dict} is a module, M_{dict}, of the form

$$\mathtt{mod}\, \tau_{\mathsf{dict}}\, \{\langle \mathtt{empty} = \ldots, \mathtt{insert} = \ldots, \mathtt{find} = \ldots \rangle\},$$

where the elided parts implement the dictionary operations in terms of the chosen representation typre, τ_{dict}. For example, τ_{dict} might be a recursive type defining a balanced binary search tree, such as a red-black tree. The dictionary operations work on the underlying representation of the dictionary as such a tree, just as would a packages of existential type discussed in Chapter 23.

To ensure that the representation of the dictionary is hidden from a client, the module M_{dict} is *sealed* with the signature σ_{dict}, which is written

$$M_{\mathsf{dict}} \upharpoonright \sigma_{\mathsf{dict}}.$$

The effect of sealing is to ensure that the *only* information about M_{dict} that is propagated to the client is given by σ_{dict}. In particular, since σ_{dict} only specifies that the type, t, have kind `T`, no information about the choice of t as τ_{dict} in M_{dict} is made available to the client.

A module is a *two-phase* object consisting of a *static part* and a *dynamic part*. The static part is a constructor of a specified kind; the dynamic part

[1] Alternatively, one could treat a dictionary as a constructor of higher kind $\mathtt{T} \to \mathtt{T}$ whose argument represents the value type. The types of the operations would then be polymorphic in the value type of the dictionary.

is a value of a specified type. There are two elimination forms that extract the static and dynamic parts of a module. These are, respectively, a form of constructor and a form of expression. More precisely, the constructor $M \cdot \mathsf{S}$ stands for the static part of M, and the expression $M \cdot \mathsf{D}$ stands for its dynamic part. According to the inversion principle, if a module, M, has introductory form, then $M \cdot \mathsf{S}$ should be equivalent to the static part of M. So, for example, $M_{\mathsf{dict}} \cdot \mathsf{S}$ should be equivalent to τ_{dict}.

But what about a combination such as $(M_{\mathsf{dict}} \upharpoonright \sigma_{\mathsf{dict}}) \cdot \mathsf{S}$? Since the intention of sealing is to hide the representation type, this constructor should *not* be equivalent to τ_{dict}. But what should it be equivalent to? Suppose that M'_{dict} is another implementation of σ_{dict}. Under what conditions should $(M_{\mathsf{dict}} \upharpoonright \sigma_{\mathsf{dict}}) \cdot \mathsf{S}$ be equivalent to $(M'_{\mathsf{dict}} \upharpoonright \sigma_{\mathsf{dict}}) \cdot \mathsf{S}$? Since constructor equivalence is an equivalence relation, this should surely hold if M and M' are equivalent implementations of σ_{dict}. But this condition violates the principle of representation independence for abstract types discussed in Chapters 23 and 51, for then type equivalence depends directly on equivalence of the underlying representations of the abstraction, exactly the thing that we are trying to hide!

The solution is to insist that $M \cdot \mathsf{S}$ is a well-formed constructor only if M is a module *value*, and to ensure that sealed modules are *not* values, but that variables *are*. This ensures that awkward questions about the equivalence of types do not arise, because they are ill-formed. Practically speaking, this restriction has the effect of requiring that sealed modules be bound to variables before they can be used. This mimics the behavior of the elimination form for existentials discussed in Chapter 23 in the sense that if M is a module with signature $\mathtt{sig}\, t :: \mathtt{T}\{\tau\}$, then the module expression

$$\mathtt{open}\, M\, \mathtt{as}\, t\, \mathtt{with}\, x{:}\sigma\, \mathtt{in}\, M' : \rho$$

may be regarded as an abbreviation for the module expression

$$(\mathtt{let}\, X\, \mathtt{be}\, M\, \mathtt{in}\, [X \cdot \mathsf{S}, X \cdot \mathsf{D}/t, x]M') : \rho.$$

Even if M is sealed, its value is bound to the module variable, X, whose static part is $X \cdot \mathsf{S}$ and whose dynamic part is $X \cdot \mathsf{D}$. The requirement that a module be bound to a variable before it may be used is reminiscent of the segregation of commands from expressions in Chapter 37. Indeed, sealing may be seen as a *pro forma* effect similar to a storage effect. Indeed, a sealed module may well have an effect when evaluated, and the role of sealing is to ensure that the reliance on effects by an implementation is not observable by any client.

47.2 Type Classes

Type abstraction is an essential tool for limiting dependencies among modules in a program. The signature of a type abstraction determines all that is known about a module by a client; no other uses of values of an abstract type are permissible. A complementary tool is to use a signature to partially specify the capabilities of a module. Such a mechanism is called a *type class*, or a *view*, in which case an implementation is called an *instance* of the type class or view. Since the signature of a type class serves only as a constraint specifying the minimum capabilities of an unknown module, some other means of working with values of that type must be available. The key to achieving this is to expose, rather than hide, the identity of the static part of a module. In this sense type classes are the "opposite" of type abstractions, but we shall see below that there is a smooth progression between them, mediated by a sub-signature judgement.

Let us consider the implementation of dictionaries as a client of the implementation of its keys.[2] To implement a dictionary using a binary search tree, for example, the only requirement is that keys come equipped with a total ordering given by a comparison operation. This can be expressed by a signature, σ_{ord}, given by

$$\mathtt{sig}\, t :: \mathtt{T}\,\{\langle \mathtt{leq} : (t \times t) \to \mathtt{bool}\rangle\}.$$

Since a given type may be ordered in many ways, it is essential that the ordering be packaged with the type to determine a type of keys.

The implementation of dictionaries as binary search trees takes the form

$$X : \sigma_{\mathsf{ord}} \vdash M_{\mathsf{bstdict}}(X) : \sigma_{\mathsf{dict}}(X),$$

where $\sigma_{\mathsf{dict}}(X)$ is the signature

$$\mathtt{sig}\, t :: \mathtt{T}\,\{\langle \mathtt{empty} : t, \mathtt{insert} : X \cdot \mathsf{S} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{find} : X \cdot \mathsf{S} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt}\rangle\}$$

and $M_{\mathsf{dict}}(X)$ is the implementation of the dictionary operations using binary search trees. Within the module $M_{\mathsf{dict}}(X)$, the static and dynamic parts of X are accessed by $X \cdot \mathsf{S}$ and $X \cdot \mathsf{D}$, respectively. In particular, the comparison operation on keys is accessed by writing $X \cdot \mathsf{D} \cdot \mathtt{leq}$ to select the `leq` field of the dynamic part of X.

[2] The client/implementor distinction is relative, not absolute. A dictionary may serve as the implementor relative to some application code as client, but it is here being regarded as a client of the implementation of the key type.

The signature given to the module variable, X, is intended to express a constraint, or upper bound, on the capabilities of a key type. That is, the module X must provide a key type and a comparison operation on it, but nothing else is assumed or allowed. The dictionary implementation is, in effect, constrained to rely only on these capabilities of keys; no other capabilities are available. In this sense the signature σ_{ord} resembles the signature σ_{dict} given in the previous section in that it fully determines the operations that the client may use on values of this type. On the other hand, when linking with a key instance, it is not intended that the instance be sealed with this signature, for to do so would render it useless! For suppose that $M_{\mathsf{natord}} : \sigma_{\mathsf{ord}}$ is an instance of the class of ordered types under the usual ordering. If we seal M_{natord} with the type class of ordered types, which is written

$$M_{\mathsf{natord}} \upharpoonright \sigma_{\mathsf{ord}},$$

then the *only* operation available on the type key type is comparison, and we have no way to create or otherwise manipulate a value of that type.

A type class is a categorization of pre-existing types, not a means of introducing a new abstract type. Rather than obscure the identity of the static part of M_{natord}, we wish to propagate its identity as `nat` while specifying a comparison with which to order them. This may be achieved using singleton kinds (Chapter 26). Specifically, the most precise, or *principal*, signature of a module is the one that exposes the static part of the module using a singleton kind. In the case of the module M_{natord}, the principal signature is the signature, σ_{natord}, given by

$$\mathtt{sig}\, t :: \mathtt{S(nat)}\, \{\mathtt{leq} : (t \times t) \to \mathtt{bool}\},$$

which, by the rules of equivalence to be detailed in Section 47.3 on page 482, is equivalent to the signature

$$\mathtt{sig}\ \_ :: \mathtt{S(nat)}\, \{\mathtt{leq} : (\mathtt{nat} \times \mathtt{nat}) \to \mathtt{bool}\}.$$

The dictionary implementation, $M_{\mathsf{dict}}(X)$ expects a module, X, with signature σ_{ord}, but the module M_{natord} provides the signature σ_{natord}. Applying the rules of subkinding given in Chapter 26, together with the evident covariance principle for signatures, we obtain the sub-signature relationship

$$\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}.$$

By the subsumption principle, a module of signature σ_{natord} may be provided whenever a module of signature σ_{ord} is required.[3] Therefore M_{natord}

[3] Thus, the signature σ_{ord} is to be viewed as an upper, not a lower, bound on the signature of the unknown module, X, in the implementation of dictionaries.

may be linked to X in $M_{\mathsf{dict}}(X)$.

The combination of subtyping and sealing provides a smooth gradation between type classes and type abstractions. The principal signature for $M_{\mathsf{dict}}(X)$ is the signature $\sigma'_{\mathsf{dict}}(X)$ given by

$$\mathtt{sig}\, t :: \mathtt{S}(\tau_{\mathsf{bst}}(X))\ \{\langle \mathtt{empty} : t, \mathtt{insert} : X \cdot \mathsf{S} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{find} : X \cdot \mathsf{S} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt}\rangle\},$$

where $\tau_{\mathsf{bst}}(X)$ is the type of binary search trees with keys given by the module X of signature σ_{ord}. This is a sub-signature of $\sigma_{\mathsf{dict}}(X)$ given earlier, so that the sealed module

$$M_{\mathsf{dict}}(X) \upharpoonright \sigma_{\mathsf{dict}}(X)$$

is well-formed, and has type $\sigma_{\mathsf{dict}}(X)$, which hides the representation type of the dictionary abstraction.

After linking X to M_{natord}, the signature of the dictionary is specialized by propagating the identity of the static part of M_{natord}. This, too, is achieved by using the sub-signature judgement. As remarked earlier, the dictionary implementation satisfies the typing

$$X : \sigma_{\mathsf{ord}} \vdash M_{\mathsf{bstdict}}(X) : \sigma_{\mathsf{dict}}(X).$$

But since $\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}$, we have, by contravariance, that

$$X : \sigma_{\mathsf{natord}} \vdash M_{\mathsf{bstdict}}(X) : \sigma_{\mathsf{dict}}(X).$$

is also a valid typing judgement. But if $X : \sigma_{\mathsf{natord}}$, then $X \cdot \mathsf{S}$ is equivalent to `nat`, since it has kind `S(nat)`, and hence the following typing is also valid:

$$X : \sigma_{\mathsf{natord}} \vdash M_{\mathsf{bstdict}}(X) : \sigma_{\mathsf{natdict}}.$$

Here $\sigma_{\mathsf{natdict}}$ is the signature

$$\mathtt{sig}\, t :: \mathtt{T}\,\{\langle \mathtt{empty} : t, \mathtt{insert} : \mathtt{nat} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{find} : \mathtt{nat} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt}\rangle\}$$

in which the representation of dictionaries is held abstract, but the representation of keys as natural numbers is publicized. The dependency on X has been eliminated by replacing all occurrences of $X \cdot \mathsf{S}$ within $\sigma_{\mathsf{dict}}(X)$ by `nat`; this is called *sharing propagation*. Having derived this typing we may link X with M_{natord} as described in Chapter 46 to obtain a composite module, M_{natdict}, of signature $\sigma_{\mathsf{natdict}}$, in which keys are natural numbers ordered as specified by M_{natord}.

In some situations it is convenient to exploit subtyping for labelled tuple types to avoid creating an *ad hoc* module specifying the standard ordering on the natural numbers. Instead we can extract the required module directly from the module that implements the abstract type of natural numbers itself. Let X_{nat} be a module variable of signature σ_{nat}, which has the general form

$$\mathtt{sig}\, t :: \mathtt{T}\,\{\langle \mathtt{zero} : t, \mathtt{succ} : t \to t, \mathtt{leq} : (t \times t) \to \mathtt{bool}, \ldots \rangle\}$$

The elided fields of the tuple are *all* of the operations that we may perform to introduce and eliminate natural numbers. Among them is the comparison operation, `leq`, required by the dictionary. Applying the subtyping rules for labelled tuples given in Chapter 25, together with the covariance of signatures, we obtain the subsignature relationship

$$\sigma_{\mathsf{nat}} <: \sigma_{\mathsf{ord}},$$

so that by subsumption the variable, X_{nat}, may be linked to the variable, X, postulated by the dictionary implementation. Subtyping takes care of extracting the required `leq` field from the abstract type of natural numbers, demonstrating that the natural numbers are an instance of the class of ordered types. Of course, the natural numbers may be ordered in many ways, not just according to the usual ordering, in which case we may construct suitable instances of σ_{ord} to suit a given situation.

47.3 A Module Language

The language $\mathcal{L}\{\mathtt{sig}\}$ is a codification of the ideas outlined in the preceding section. The syntax is divided into four levels: expressions classified by types, constructors classified by kinds, and modules classified by signatures. The expression and type level consists of various language mechanisms (that we shall not pin down here) described earlier in this book, including at least product, sum, and partial function types. The constructor and kind level is as described in Chapters 24 and 26, with singleton and dependent kinds. The syntax of $\mathcal{L}\{\mathtt{sig}\}$ is summarized by the following

grammar:

Sig	σ	$::=$	$\mathtt{sig}[\kappa](t.\tau)$	$\mathtt{sig}\, t :: \kappa\, \{\tau\}$	signature
Mod	M	$::=$	X	X	variable
			$\mathtt{mod}(c;e)$	$\mathtt{mod}\, c\, \{e\}$	module
			$\mathtt{seal}[\sigma](M)$	$M \upharpoonright \sigma$	seal
			$\mathtt{let}[\sigma](M_1; X.M_2)$	$(\mathtt{let}\, X\, \mathtt{be}\, M_1\, \mathtt{in}\, M_2) : \sigma$	definition
Con	c	$::=$	$\mathtt{stat}(M)$	$M \cdot \mathsf{S}$	static part
Exp	e	$::=$	$\mathtt{dyn}(M)$	$M \cdot \mathsf{D}$	dynamic part

The statics of $\mathcal{L}\{\mathtt{sig}\}$ consists of the following forms of judgement, in addition to those governing the kind and type levels:

$\Gamma \vdash \sigma\ \mathsf{sig}$	well-formed signature
$\Gamma \vdash \sigma_1 \equiv \sigma_2$	equivalent signatures
$\Gamma \vdash \sigma_1 <: \sigma_2$	sub-signature
$\Gamma \vdash M : \sigma$	well-formed module
$\Gamma \vdash M\ \mathsf{val}$	module value
$\Gamma \vdash e\ \mathsf{val}$	expression value

Rather than segregate hypotheses into zones, we instead admit the following three forms of hypothesis groups:

$X : \sigma, X\ \mathsf{val}$	module value variable
$u :: \kappa$	constructor variable
$x : \tau, x\ \mathsf{val}$	expression value variable

It is important that module and expression variables are always regarded as values to ensure that type abstraction is properly enforced. Correspondingly, each module and expression variable appears in Γ paired with the hypothesis that it is a value. As a notational convenience we will not explicitly state the value hypotheses associated with module and expression variables, under the convention that all such variables implicitly come paired with such an assumption.

The formation, equivalence, and subsignature judgements are defined by the following rules:

$$\frac{\Gamma \vdash \kappa\ \mathsf{kind} \quad \Gamma, u :: \kappa \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathtt{sig}\, u :: \kappa\, \{\tau\}\ \mathsf{sig}} \tag{47.1a}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \mathtt{sig}\, u :: \kappa_1\, \{\tau_1\} \equiv \mathtt{sig}\, u :: \kappa_2\, \{\tau_2\}} \tag{47.1b}$$

$$\frac{\Gamma \vdash \kappa_1 :<: \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Gamma \vdash \mathtt{sig}\, u :: \kappa_1\,\{\tau_1\} <: \mathtt{sig}\, u :: \kappa_2\,\{\tau_2\}} \tag{47.1c}$$

Most importantly, signatures are covariant in both the kind and type positions: subkinding and subtyping are preserved by the formation of a signature. It is a consequence of Rule (47.1b) that

$$\mathtt{sig}\, u :: \mathtt{S}(c)\,\{\tau\} \equiv \mathtt{sig}\ \_ :: \mathtt{S}(c)\,\{[c/u]\tau\}$$

and, further, it is a consequence of Rule (47.1c) that

$$\mathtt{sig}\ \_ :: \mathtt{S}(c)\,\{[c/u]\tau\} <: \mathtt{sig}\ \_ :: \mathtt{T}\,\{[c/u]\tau\}$$

and therefore

$$\mathtt{sig}\, u :: \mathtt{S}(c)\,\{\tau\} <: \mathtt{sig}\ \_ :: \mathtt{T}\,\{[c/u]\tau\}.$$

It is also the case that

$$\mathtt{sig}\, u :: \mathtt{S}(c)\,\{\tau\} <: \mathtt{sig}\, u :: \mathtt{T}\,\{\tau\}.$$

But the two supersignatures of $\mathtt{sig}\, u :: \mathtt{S}(c)\,\{\tau\}$ are *incomparable* with respect to the subsignature judgement. This fact is important in the statics of module definitions, as will detailed shortly.

The statics of module expressions is given by the following rules:

$$\frac{}{\Gamma, X : \sigma \vdash X : \sigma} \tag{47.2a}$$

$$\frac{\Gamma \vdash c :: \kappa \quad \Gamma \vdash e :: [c/u]\tau}{\Gamma \vdash \mathtt{mod}\, c\,\{e\} : \mathtt{sig}\, u :: \kappa\,\{\tau\}} \tag{47.2b}$$

$$\frac{\Gamma \vdash \sigma\ \mathsf{sig} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash M \upharpoonright \sigma : \sigma} \tag{47.2c}$$

$$\frac{\Gamma \vdash \sigma\ \mathsf{sig} \quad \Gamma \vdash M_1 : \sigma_1 \quad \Gamma, X : \sigma_1 \vdash M_2 : \sigma}{\Gamma \vdash (\mathtt{let}\, X\, \mathtt{be}\, M_1\, \mathtt{in}\, M_2) : \sigma : \sigma} \tag{47.2d}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash M : \sigma'} \tag{47.2e}$$

In Rule (47.2b) it is always possible to choose κ to be the principal kind of c, which uniquely determines c up to constructor equivalence. For such a choice, the signature $\mathtt{sig}\, u :: \kappa\,\{\tau\}$ is equivalent to $\mathtt{sig}\ \_ :: \kappa\,\{[c/u]\tau\}$, which propagates the identity of the static part of the module expression into the type of its dynamic part. Rule (47.2c) is to be used in conjunction with subsumption (Rule (47.2e)) to ensure that M has the specified signature.

The need for a signature annotation on a module definition is a manifestation of the *avoidance problem*. Rule (47.2d) would be perfectly sensible were the signature, σ, omitted from the syntax of the definition. However, omitting this information greatly complicates type checking. If σ were omitted from the syntax of the definition, the type checker would be required to find a signature, σ, for the body of the definition that *avoids* the module variable, X. Inductively, we may suppose that we have found a signature, σ_1, for the module M_1, and a signature, σ_2, for the module M_2, under the assumption that X has signature σ_1. To find a signature for an unadorned definition, we must find a supersignature, σ, of σ_2 that avoids X. To ensure that all possible choices of σ are accounted for, we seek to find the least (most precise) such signature with respect to the subsignature relation; this is called the *principal signature* of a module. The problem is that there is no least supersignature of a given signature, σ_2, that avoids a specified variable, X! (To see this, consider the example given earlier of a signature with two incomparable supersignatures, each of which may be taken to avoid a variable, X, that occurs in the subsignature.) Consequently, modules do not have principal signatures, a significant complication for type checking. To avoid this problem, we insist that the avoiding supersignature, σ, be given by the programmer so that the type checker need not try to find it.

In the presence of modules we have a new form of constructor expression, $M \cdot \mathsf{S}$, and a new form of value expression, $M \cdot \mathsf{D}$. These operations, respectively, extract the static and dynamic parts of the module M. Their formation rules are as follows:

$$\frac{\Gamma \vdash M \ \mathsf{val} \quad \Gamma \vdash M : \mathtt{sig}\, u :: \kappa\,\{\tau\}}{\Gamma \vdash M \cdot \mathsf{S} :: \kappa} \tag{47.3a}$$

$$\frac{\Gamma \vdash M : \mathtt{sig}\, \_ :: \kappa\,\{\tau\}}{\Gamma \vdash M \cdot \mathsf{D} : \tau} \tag{47.3b}$$

Rule (47.3a) requires that the module expression, M, be a value in accordance with the following rules:

$$\frac{}{\Gamma, X : \sigma, X \ \mathsf{val} \vdash X \ \mathsf{val}} \tag{47.4a}$$

$$\frac{\Gamma \vdash e \ \mathsf{val}}{\Gamma \vdash \mathtt{mod}\, c\, \{e\} \ \mathsf{val}} \tag{47.4b}$$

Rule (47.3a) specifies that only module values have well-defined static parts, and hence precludes reference to the static part of a sealed module

(which is not a value). This ensures representation independence for abstract types, as discussed in Section 47.1 on page 477. For if $M \cdot \mathsf{S}$ were admissible when M is a sealed module, it would be a type whose identity depends on the underlying implementation, in violation of the abstraction principle. Module variables are, on the other hand, values, so that if $X : \mathtt{sig}\, t :: \mathtt{T}\,\{\tau\}$ is a module variable, then $X \cdot \mathsf{S}$ is a well-formed type. What this means in practice is that sealed modules must be bound to variables before they can be used. It is for this reason that we include definitions among module expressions.

Rule (47.3b) requires that the signature of the module, M, be non-dependent, so that the result type, τ, does not depend on the static part of the module. This may not always be the case. For example, if M is a sealed module, say $N \upharpoonright \mathtt{sig}\, t :: \mathtt{T}\,\{t\}$ for some module N, then projection $M \cdot \mathsf{D}$ is ill-formed. If it were to be well-formed, then its type would be $M \cdot \mathsf{S}$, which would project a type from a sealed module, in violation of representation independence for abstract types. But if M is a module value, then it is always possible to derive a non-dependent signature for it, provided that we include the following rule of *self-recognition*:

$$\frac{\Gamma \vdash M : \mathtt{sig}\, u :: \kappa\,\{\tau\} \quad \Gamma \vdash M\ \mathsf{val}}{\Gamma \vdash M : \mathtt{sig}\, u :: \mathtt{S}(M \cdot \mathsf{S} :: \kappa)\,\{\tau\}} \tag{47.5}$$

This rule propagates the identity of the static part of a module value into its signature. By sharing propagation the dependency of the type of the dynamic part on the static part is then eliminable.

The following rule of constructor equivalence ensures that a type projection from a module value is eliminable:

$$\frac{\Gamma \vdash \mathtt{mod}\, c\,\{e\} : \mathtt{sig}\, t :: \kappa\,\{\tau\} \quad \Gamma \vdash \mathtt{mod}\, c\,\{e\}\ \mathsf{val}}{\Gamma \vdash \mathtt{mod}\, c\,\{e\} \cdot \mathsf{S} \equiv c :: \kappa} \tag{47.6}$$

The requirement that the expression, e, be a value, which is implicit in the second premise of the rule, is not strictly necessary, but does no harm. A consequence of this rule is that apparent dependencies of closed constructors (or kinds) on modules may always be eliminated. In particular the identity of the constructor $\mathtt{mod}\, c\,\{e\} \cdot \mathsf{S}$ is independent of e, as would be expected if representation independence is to be assured.[4]

[4] It is possible to extend the concept of hereditary substitution to singletons and modules in such a way that even these apparent dependencies are avoided entirely, so that constructors and kinds are completely independent of modules and expressions. We will not develop this extension here.

The dynamics of modules is entirely straightforward:

$$\frac{e \mapsto e'}{\mathtt{mod}\, c\, \{e\} \mapsto \mathtt{mod}\, c\, \{e'\}} \tag{47.7a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{mod}\, c\, \{e\} \cdot \mathsf{D} \mapsto e} \tag{47.7b}$$

There is no need to evaluate constructors at run-time, because the dynamics of expressions does not depend on the their types. It is straightforward to prove type safety for this dynamics relative to the foregoing statics. We leave the precise formulation and proof as an exercise for the reader.[5]

47.4 Notes

The use of dependent types to express modularity was first proposed by MacQueen [59]. Subsequent studies extended this proposal to account for the *phase distinction* between static and dynamic phases of processing [44] and for type abstraction, as well as type classes [43, 56]. The avoidance problem was first isolated by Castagna and Pierce [20] and Harper and Lillibridge [43], and has come to play a central role in subsequent work on modules [27]. The self-recognition rule was introduced by Harper and Lillibridge [43] and Leroy [56], and was subsequently identified as a manifestation of higher-order singletons [99]. A consolidation of these ideas was used as the foundation for a mechanization of the metatheory of the modules [54].

The semantic formulation of modules considered here focuses attention on the type structure required to support modularity. An alternative formulation is based on *elaboration*, a translation of modularity constructs into more primitive notions such as polymorphism and higher-order functions. *The Definition of Standard ML* [67] pioneered the elaboration approach. Building on earlier work of Russo, a more type-theoretic formulation was given by Rossberg, *et al.* [92] that avoids the need for the extensions considered here at the expense of a less direct explanation of the semantics of modularity.

The module system given here is sometimes described as a *second-class module system*, since modules cannot be stored in data structures or passed to or from ordinary functions. But this description is misleading, because

[5] To account for exceptions or other checked errors, one must define a judgement M err stating that evaluation of M results in an error, and adjust the statement and proof of safety accordingly.

the formulation considered here is *strictly more general* than one that restricts attention to *first-class* modules, which enjoy these capabilities. Specifically, Dreyer [27] shows that it is easy to extend $\mathcal{L}\{\texttt{sig}\}$ with constructs that allow modules to be packaged as elements of existential type, thereby allowing them to be used as run-time values, and to extract modules from existentials.

Chapter 48

Module Hierarchies and Transformations

Part XIX

Equivalence

Chapter 49

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 11 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* for two expressions to be equal in this sense? It is intuitively obvious that these two expressions are not *definitionally* equivalent, because they cannot be shown equivalent by symbolic execution. 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. Two functions are *logically 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 logically 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, these functions are said to be *observationally equivalent*. The

main result of this chapter is that observational and logical equivalence coincide for a variant of $\mathcal{L}\{\texttt{nat} \to\}$ in which the successor is evaluated eagerly, so that a value of type `nat` is a numeral.

49.1 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 dynamics 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

$$\mathcal{C}\{\circ\} \triangleq \mathcal{C}_1\{\mathcal{C}_2\{\circ\}\},$$

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 statics 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 statics of $\mathcal{L}\{\mathtt{nat}\to\}$ (see Rules (11.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \tag{49.1a}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})}{\mathtt{s}(\mathcal{C}) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})} \tag{49.1b}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{natrec}\,\mathcal{C}\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{49.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau') \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{natrec}\,e\,\{\mathtt{z} \Rightarrow \mathcal{C}_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{49.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \rhd \tau')}{\mathtt{natrec}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow \mathcal{C}_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{49.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \tau_1 \rhd \tau_2)}{\lambda\,(x{:}\tau_1.\,\mathcal{C}_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2)} \tag{49.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1\,(e_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{49.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}{e_1\,(\mathcal{C}_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{49.1h}$$

Lemma 49.1. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, *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 49.2. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, *and* $\mathcal{C}' : (\Gamma' \rhd \tau') \rightsquigarrow (\Gamma'' \rhd \tau'')$, *then* $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma'' \rhd \tau'')$.

Lemma 49.3. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ *and* $x \notin dom(\Gamma)$, *then* $\mathcal{C} : (\Gamma, x : \rho \rhd \tau) \rightsquigarrow (\Gamma', x : \rho \rhd \tau')$.

Proof. By induction on Rules (49.1). □

A *complete program* is a closed expression of type `nat`.

Definition 49.1. *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 49.2. *Suppose that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$ *are two expressions of the same type. Two such expressions are* observationally equivalent, *written* $\Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *for every program context* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\emptyset \rhd \mathtt{nat})$.

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 $\Gamma \vdash e_1 \mathrel{\mathcal{E}} e_2 : \tau$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } \Gamma \vdash e \mathrel{\mathcal{E}} e' : \tau, \text{ then } \Gamma' \vdash \mathcal{C}\{e\} \mathrel{\mathcal{E}} \mathcal{C}\{e'\} : \tau'$$

for every expression context $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$. Such a family of relations is *consistent* iff $\emptyset \vdash e \mathrel{\mathcal{E}} e' : \mathtt{nat}$ implies $e \simeq e'$.

Theorem 49.4. *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 context with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $\Gamma \vdash e \mathrel{\mathcal{E}} e' : \tau$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$, then by congruence $\varnothing \vdash \mathcal{C}\{e\} \mathrel{\mathcal{E}} \mathcal{C}\{e'\} : \mathtt{nat}$, and hence by consistency $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$. □

A *closing substitution*, γ, for the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning closed 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 $[e_1, \ldots, e_n / x_1, \ldots, x_n]e$, and write $\gamma : \Gamma$ to mean that if $x : \tau$ occurs in Γ, then there exists a closed expression, e, such that $\gamma(x) = e$ and $e : \tau$. We write $\gamma \cong_\Gamma \gamma'$, where $\gamma : \Gamma$ and $\gamma' : \Gamma$, to express that $\gamma(x) \cong_{\Gamma(x)} \gamma'(x)$ for each x declared in Γ.

Lemma 49.5. *If $\Gamma \vdash e \cong e' : \tau$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. Moreover, if $\gamma \cong_\Gamma \gamma'$, then $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e)$ and $\hat{\gamma}(e') \cong_\tau \widehat{\gamma'}(e')$.*

Proof. Let $\mathcal{C} : (\varnothing \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$ 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 49.3 on the preceding page we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows directly that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$. Since $\Gamma \vdash e \cong e' : \tau$, 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 $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$.

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 $\widehat{\gamma'}(e) \cong_\tau \widehat{\gamma'}(e')$. Now if $\gamma \cong_\Gamma \gamma'$, then by congruence we have $\mathcal{D}\{e\} \cong_{\mathtt{nat}} \mathcal{D}'\{e\}$, and $\mathcal{D}\{e'\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$. It follows that $\mathcal{D}\{e'\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$, and so, by consistency of observational equivalence, we have $\mathcal{D}\{e'\} \simeq \mathcal{D}'\{e'\}$, which is to say that $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. □

Theorem 49.4 on the preceding page licenses the principle of *proof by coinduction*: to show that $\Gamma \vdash e \cong e' : \tau$, it is enough to exhibit a consistent congruence, $\mathcal{E}$, such that $\Gamma \vdash e \mathrel{\mathcal{E}} e' : \tau$. 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.

49.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of expressions of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that merely manipulate expressions without actually inspecting them. For example, we may pass an expression of type τ to a function that merely returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate expressions at arm's length, affording no opportunities to distinguish one from another.

This leads to the definition of logical equivalence alluded to in the introduction.

Definition 49.3. Logical equivalence *is a family of relations $e \sim_\tau e'$ between closed expressions of type τ. It is defined by induction on τ as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad \textit{if } e_1 \sim_{\tau_1} e_1', \textit{ then } e\,(e_1) \sim_{\tau_2} e'\,(e_1')$$

The definition of logical equivalence at type `nat` licenses the following principle of *proof by `nat`-induction*. To show that $\mathcal{E}\ (e, e')$ whenever $e \sim_{\mathtt{nat}} e'$, 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.

Lemma 49.6. *Logical equivalence is symmetric and transitive: if $e \sim_\tau e'$, then $e' \sim_\tau e$, and if $e \sim_\tau e'$ and $e' \sim_\tau e''$, then $e \sim_\tau e''$.*

Proof. Simultaneously, by induction on the structure of τ. If $\tau = \mathtt{nat}$, the result is immediate. If $\tau = \tau_1 \to \tau_2$, then we may assume that logical equivalence is symmetric and transitive at types τ_1 and τ_2. For symmetry, assume that $e \sim_\tau e'$; we wish to show $e' \sim_\tau e$. Assume that $e_1' \sim_{\tau_1} e_1$; it suffices to show that $e'(e_1') \sim_{\tau_2} e(e_1)$. By induction we have that $e_1 \sim_{\tau_1} e_1'$. Therefore by assumption $e(e_1) \sim_{\tau_2} e'(e_1')$, and hence by induction $e'(e_1') \sim_{\tau_2} e(e_1)$. For transitivity, assume that $e \sim_\tau e'$ and $e' \sim_\tau e''$; we are to show $e \sim_\tau e''$. Suppose that $e_1 \sim_{\tau_1} e_1''$; it is enough to show that $e(e_1) \sim_\tau e''(e_1'')$. By symmetry and transitivity we have $e_1 \sim_{\tau_1} e_1$, so by assumption $e(e_1) \sim_{\tau_2} e'(e_1)$. We also have by assumption $e'(e_1) \sim_{\tau_2} e''(e_1'')$. By transitivity we have $e'(e_1) \sim_{\tau_2} e''(e_1'')$, which suffices for the result. □

Logical 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 $\Gamma \vdash x : \tau$. *Open logical equivalence*, written $\Gamma \vdash e \approx e' : \tau$, is defined to mean that $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

Lemma 49.7. *Open logical equivalence is symmetric and transitive.*

Proof. Follows immediately from Lemma 49.6 on the preceding page and the definition of open logical equivalene. □

49.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

Lemma 49.8 (Converse Evaluation)**.** *Suppose that $e \sim_\tau e'$. If $d \mapsto e$, then $d \sim_\tau e'$, and if $d' \mapsto e'$, then $e \sim_\tau d'$.*

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 $e \sim_\tau e'$, and $d \mapsto e$. To show that $d \sim_\tau e'$, we assume $e_1 \sim_{\tau_1} e_1'$ and show $d(e_1) \sim_{\tau_2} e'(e_1')$. It follows from the assumption that $e(e_1) \sim_{\tau_2} e'(e_1')$. Noting that $d(e_1) \mapsto e(e_1)$, the result follows by induction. □

Lemma 49.9 (Consistency)**.** *If $e \sim_{\mathtt{nat}} e'$, then $e \simeq e'$.*

Proof. Immediate, from Definition 49.3 on the facing page. □

Theorem 49.10 (Reflexivity). *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e \approx e : \tau$.

Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e)$. The proof proceeds by induction on typing derivations; we consider a few representative cases.

Consider the case of Rule (10.4a), in which $\tau = \tau_1 \to \tau_2$ and $e = \lambda\,(x\!:\!\tau_1.\, e_2)$. Since e is a value, we are to show that

$$\lambda\,(x\!:\!\tau_1.\, \hat{\gamma}(e_2)) \sim_{\tau_1\to\tau_2} \lambda\,(x\!:\!\tau_1.\, \widehat{\gamma'}(e_2)).$$

Assume that $e_1 \sim_{\tau_1} e_1'$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim_{\tau_2} [e_1'/x]\widehat{\gamma'}(e_2)$. Let $\gamma_2 = \gamma[x \mapsto e_1]$ and $\gamma_2' = \gamma'[x \mapsto e_1']$, and observe that $\gamma_2 \sim_{\Gamma,x:\tau_1} \gamma_2'$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim_{\tau_2} \hat{\gamma}_2'(e_2)$, from which the result follows directly.

Now consider the case of Rule (11.1d), for which we are to show that

$$\mathtt{natrec}(\hat{\gamma}(e); \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{natrec}(\widehat{\gamma'}(e); \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)).$$

By the induction hypothesis applied to the first premise of Rule (11.1d), we have

$$\hat{\gamma}(e) \sim_{\mathtt{nat}} \widehat{\gamma'}(e).$$

We proceed by `nat`-induction. It suffices to show that

$$\mathtt{natrec}(\mathtt{z}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{natrec}(\mathtt{z}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)), \tag{49.2}$$

and that

$$\mathtt{natrec}(\mathtt{s}(\overline{n}); \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{natrec}(\mathtt{s}(\overline{n}); \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)), \tag{49.3}$$

assuming

$$\mathtt{natrec}(\overline{n}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{natrec}(\overline{n}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)). \tag{49.4}$$

To show (49.2), by Lemma 49.8 on the previous page it is enough to show that $\hat{\gamma}(e_0) \sim_\tau \widehat{\gamma'}(e_0)$. This is assured by the outer inductive hypothesis applied to the second premise of Rule (11.1d).

To show (49.3), define

$$\delta = \gamma[x \mapsto \overline{n}][y \mapsto \mathtt{natrec}(\overline{n}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1))]$$

and

$$\delta' = \gamma'[x \mapsto \overline{n}][y \mapsto \mathtt{natrec}(\overline{n}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1))].$$

By (49.4) we have $\delta \sim_{\Gamma,x:\mathtt{nat},y:\tau} \delta'$. Consequently, by the outer inductive hypothesis applied to the third premise of Rule (11.1d), and Lemma 49.8 on the preceding page, the required follows. □

Corollary 49.11 (Termination). *If $e : \tau$, then $e \mapsto^* e'$ for some e' val.*

Corollary 49.12 (Equivalence). *Open logical equivalence is an equivalence relation.*

Lemma 49.13 (Congruence). *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $\Gamma \vdash e \approx e' : \tau$, then $\Gamma_0 \vdash \mathcal{C}_0\{e\} \approx \mathcal{C}_0\{e'\} : \tau_0$.*

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 $\Gamma \vdash e \approx e' : \tau$, we are to show that

$$\Gamma_0 \vdash \mathcal{C}_0\{e\} \approx \mathcal{C}_0\{e'\} : \tau_1 \to \tau_2,$$

which is to say

$$\Gamma_0 \vdash \lambda\,(x\!:\!\tau_1.\,\mathcal{C}_2\{e\}) \approx \lambda\,(x\!:\!\tau_1.\,\mathcal{C}_2\{e'\}) : \tau_1 \to \tau_2.$$

We know, by induction, that

$$\Gamma_0, x : \tau_1 \vdash \mathcal{C}_2\{e\} \approx \mathcal{C}_2\{e'\} : \tau_2.$$

Suppose that $\gamma_0 \sim_{\Gamma_0} \gamma_0'$, and that $e_1 \sim_{\tau_1} e_1'$. Let $\gamma_1 = \gamma_0[x \mapsto e_1]$, $\gamma_1' = \gamma_0'[x \mapsto e_1']$, and observe that $\gamma_1 \sim_{\Gamma_0, x:\tau_1} \gamma_1'$. By Definition 49.3 on page 498 it is enough to show that

$$\hat{\gamma_1}(\mathcal{C}_2\{e\}) \sim_{\tau_2} \hat{\gamma_1'}(\mathcal{C}_2\{e'\}),$$

which follows immediately from the inductive hypothesis.

□

Theorem 49.14. *If $\Gamma \vdash e \approx e' : \tau$, then $\Gamma \vdash e \cong e' : \tau$.*

Proof. By Lemmas 49.9 on page 499 and 49.13, and Theorem 49.4 on page 497.

□

Corollary 49.15. *If $e : \mathtt{nat}$, then $e \cong_{\mathtt{nat}} \overline{n}$, for some $n \geq 0$.*

Proof. By Theorem 49.10 on the preceding page we have $e \sim_\tau e$. Hence for some $n \geq 0$, we have $e \sim_{\mathtt{nat}} \overline{n}$, and so by Theorem 49.14, $e \cong_{\mathtt{nat}} \overline{n}$. □

Lemma 49.16. *For closed expressions $e : \tau$ and $e' : \tau$, if $e \cong_\tau e'$, then $e \sim_\tau e'$.*

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 $e \sim_{\mathtt{nat}} e'$. If $\tau = \tau_1 \to \tau_2$, then we are to show that whenever $e_1 \sim_{\tau_1} e_1'$, we have $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 49.14 on the previous page we have $e_1 \cong_{\tau_1} e_1'$, and hence by congruence of observational equivalence it follows that $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

Theorem 49.17. *If* $\Gamma \vdash e \cong e' : \tau$, *then* $\Gamma \vdash e \approx e' : \tau$.

Proof. Assume that $\Gamma \vdash e \cong e' : \tau$, and that $\gamma \sim_\Gamma \gamma'$. By Theorem 49.14 on the preceding page we have $\gamma \cong_\Gamma \gamma'$, so by Lemma 49.5 on page 497 $\hat{\gamma}(e) \cong_\tau \hat{\gamma'}(e')$. Therefore, by Lemma 49.16 on the preceding page, $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. □

Corollary 49.18. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \approx e' : \tau$.

Theorem 49.19. *If* $\Gamma \vdash e \equiv e' : \tau$, *then* $\Gamma \vdash e \approx e' : \tau$, *and hence* $\Gamma \vdash e \cong e' : \tau$.

Proof. By an argument similar to that used in the proof of Theorem 49.10 on page 500 and Lemma 49.13 on the preceding page, then appealing to Theorem 49.14 on the previous page. □

Corollary 49.20. *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 49.19 we have $e \sim_{\mathtt{nat}} e'$ and hence $e \simeq e'$. □

49.4 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 logical equivalence, and then transferred to observational equivalence by appeal to Corollary 49.18. The laws are presented as inference rules with the meaning that if all of the premises are true judgements about observational equivalence, then so are the conclusions. In other words each rule is admissible as a principle of observational equivalence.

49.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau} \tag{49.5a}$$

$$\frac{\Gamma \vdash e' \cong e : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{49.5b}$$

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau} \tag{49.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences.

$$\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{49.6a}$$

This is called the *principle of symbolic evaluation*.

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\Gamma' \vdash \mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'} \tag{49.7a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e/x]e_2 \cong [e/x]e_2' : \tau'} \tag{49.8a}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'} \tag{49.8b}$$

49.4.2 Equivalence Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{\Gamma, x : \tau_1 \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2} \tag{49.9}$$

Consequently, every expression of function type is equivalent to a λ-abstraction:

$$\frac{}{\Gamma \vdash e \cong \lambda\,(x{:}\tau_1.\, e(x)) : \tau_1 \to \tau_2} \tag{49.10}$$

49.4.3 Induction Law

An equation involving a free variable, x, of type `nat` can be proved by induction on x.

$$\frac{\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \text{ (for every } n \in \mathbb{N})}{\Gamma, x : \mathtt{nat} \vdash e \cong e' : \tau} \tag{49.11a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $\Gamma \vdash [\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau$, and
2. $\Gamma \vdash [\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau$, if $\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau$.

49.5 Notes

The technique of *logical relations* interprets types as relations (here, equivalence relations) by associating with each type constructor a relational action that transforms the relation interpreting its arguments to the relation interpreting the constructed type. Logical relations are a fundamental tool in proof theory [97] and form the foundation for the semantics of the NuPRL type theory [23, 5, 39]. The use of logical relations to characterize observational equivalence is essentially an adaptation of the NuPRL semantics to the simpler setting of Gödel's System **T**.

Chapter 50

Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, with an eager interpretation of the type of natural numbers. The development proceeds long lines similar to those in Chapter 49, 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*.

50.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 49, 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 49. Specifically, we define the judgement $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ by rules similar to Rules (49.1), 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 50.1. *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 49.

Definition 50.2. *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 50.1. *Observational equivalence is the coarsest consistent congruence.*

Proof. See the proof of Theorem 49.4 on page 497. □

Lemma 50.2 (Substitution and Functionality)**.** *If* $\Gamma \vdash e \cong e' : \tau$ *and* $\gamma : \Gamma$, *then* $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. *Moreover, if* $\gamma \cong_\Gamma \gamma'$, *then* $\hat{\gamma}(e) \cong_\tau \hat{\gamma}'(e)$ *and* $\hat{\gamma}(e') \cong_\tau \hat{\gamma}'(e')$.

Proof. See Lemma 49.5 on page 497. □

50.2 Logical Equivalence

Definition 50.3. Logical equivalence, $e \sim_\tau e'$, *between closed expressions of type* τ *is defined by induction on* τ *as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad e_1 \sim_{\tau_1} e_1' \textit{ implies } e\,(e_1) \sim_{\tau_2} e'\,(e_1')$$

Formally, logical equivalence is defined as in Chapter 49, except that the definition of Kleene equivalence is altered to account for non-termination. Logical equivalence is extended to open terms by substitution. Specifically, we define $\Gamma \vdash e \approx e' : \tau$ to mean that $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

By the same argument as given in the proof of Lemma 49.6 on page 498 logical equivalence is symmetric and transitive, as is its open extension.

Lemma 50.3 (Strictness)**.** *If* $e : \tau$ *and* $e' : \tau$ *are both divergent, then* $e \sim_\tau e'$.

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 $e(e_1) \sim_{\tau_2} e'(e_1')$, as required. □

Lemma 50.4 (Converse Evaluation)**.** *Suppose that* $e \sim_\tau e'$. *If* $d \mapsto e$, *then* $d \sim_\tau e'$, *and if* $d' \mapsto e'$, *then* $e \sim_\tau d'$.

50.3 Logical and Observational Equivalence Coincide

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 $\texttt{fix}^m\,x{:}\tau\,\texttt{is}\,e$, where $m \geq 0$. The statics of bounded recursion is the same as for general recursion:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}^m\,[\tau]\,(x.e) : \tau}\ . \tag{50.1a}$$

The dynamics of bounded recursion is defined as follows:

$$\frac{}{\texttt{fix}^0\,[\tau]\,(x.e) \mapsto \texttt{fix}^0\,[\tau]\,(x.e)} \tag{50.2a}$$

$$\frac{}{\texttt{fix}^{m+1}\,[\tau]\,(x.e) \mapsto [\texttt{fix}^m\,[\tau]\,(x.e)\,/\,x]e} \tag{50.2b}$$

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 key property of bounded recursion is the principle of fixed point induction, which permits reasoning about a recursive computation by induction on the number of unrollings required to reach a value. The proof relies on *compactness*, which will be stated and proved in Section 50.4 on page 510 below.

Theorem 50.5 (Fixed Point Induction)**.** *Suppose that* $x : \tau \vdash e : \tau$. *If*

$$(\forall m \geq 0)\ \mathit{fix}^m\,x{:}\tau\,\texttt{is}\,e \sim_\tau \mathit{fix}^m\,x{:}\tau\,\texttt{is}\,e',$$

then $\texttt{fix}\,x{:}\tau\,\texttt{is}\,e \sim_\tau \texttt{fix}\,x{:}\tau\,\texttt{is}\,e'$.

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 logical 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 $e_2 \sim_{\tau_2} e_2'$, 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

$$\text{for every } m \geq 0,\ \mathcal{A}\{\texttt{fix}^m\,x{:}\rho\,\texttt{is}\,e\} \sim_\tau \mathcal{A}'\{\texttt{fix}^m\,x{:}\rho\,\texttt{is}\,e'\}, \tag{50.3}$$

then

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}. \tag{50.4}$$

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 (50.3). By Definition 50.3 on page 506, 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 50.14 on page 513 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 (50.3) 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 50.14 on page 513

$$\mathcal{A}'\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e'\} \simeq \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 50.3 on page 506, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\}(e_1) \sim_{\tau_2} \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}(e_1')$$

whenever $e_1 \sim_{\tau_1} e_1'$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (50.3) that for every $m \geq 0$

$$\mathcal{A}_2\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e'\}.$$

Noting that $\mathcal{A}_2 \approx \mathcal{A}_2' : \rho \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\},$$

as required. □

Lemma 50.6 (Reflexivity)**.** *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e \approx e : \tau$.

Proof. The proof proceeds along the same lines as the proof of Theorem 49.10 on page 500. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider Rule (12.1g). Assuming $\gamma \sim_\Gamma \gamma'$, we are to show that

$$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,\hat{\gamma}(e) \sim_\tau \mathtt{fix}\,x{:}\tau\,\mathtt{is}\,\widehat{\gamma'}(e).$$

By Theorem 50.5 on page 507 it is enough to show that, for every $m \geq 0$,

$$\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e).$$

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 50.4 on page 506, it is enough to show that $\hat{\gamma}(e_1) \sim_\tau \widehat{\gamma'}(e_1)$, where

$$e_1 = [\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{50.5}$$
$$e_1' = [\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e). \tag{50.6}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e),$$

then

$$[\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e) \sim_\tau [\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e).$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows. □

Symmetry and transitivity of eager logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager logical equivalence is therefore an equivalence relation.

Lemma 50.7 (Congruence). *If* $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, *and* $\Gamma \vdash e \approx e' : \tau$, *then* $\Gamma_0 \vdash \mathcal{C}_0\{e\} \approx \mathcal{C}_0\{e'\} : \tau_0$.

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$, following along similar lines to the proof of Lemma 50.6 on the preceding page. □

Logical equivalence is consistent, by definition. Consequently, it is contained in observational equivalence.

Theorem 50.8. *If* $\Gamma \vdash e \approx e' : \tau$, *then* $\Gamma \vdash e \cong e' : \tau$.

Proof. By consistency and congruence of logical equivalence. □

Lemma 50.9. *If* $e \cong_\tau e'$, *then* $e \sim_\tau e'$.

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 $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 50.8 on the preceding page $e_1 \cong_{\tau_1} e_1'$, and hence by Lemma 50.2 on page 506 $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

Theorem 50.10. *If* $\Gamma \vdash e \cong e' : \tau$, *then* $\Gamma \vdash e \approx e' : \tau$.

Proof. Assume that $\Gamma \vdash e \cong e' : \tau$. Suppose that $\gamma \sim_\Gamma \gamma'$. By Theorem 50.8 on the preceding page we have $\gamma \cong_\Gamma \gamma'$, and so by Lemma 50.2 on page 506 we have

$$\hat{\gamma}(e) \cong_\tau \hat{\gamma}'(e').$$

Therefore by Lemma 50.9 on the preceding page we have

$$\hat{\gamma}(e) \sim_\tau \hat{\gamma}'(e').$$

□

Corollary 50.11. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \approx e' : \tau$.

50.4 Compactness

The principle of fixed point induction is derived from a critical property of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, 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.

The proof of compactness (Theorem 50.13 on page 512) makes use of the stack machine for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ defined in Chapter 29, augmented with the following transitions for bounded recursive expressions:

$$\frac{}{k \rhd \mathtt{fix}^0\, x{:}\tau\, \mathtt{is}\, e \mapsto k \rhd \mathtt{fix}^0\, x{:}\tau\, \mathtt{is}\, e} \tag{50.7a}$$

$$\frac{}{k \rhd \mathtt{fix}^{m+1}\, x{:}\tau\, \mathtt{is}\, e \mapsto k \rhd [\mathtt{fix}^m\, x{:}\tau\, \mathtt{is}\, e/x]e} \tag{50.7b}$$

It is straightforward to extend the proof of correctness of the stack machine (Corollary 29.4 on page 277) to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function, f, in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$:

$$\mathtt{fix}\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda\, (x\mathtt{:nat}.\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}).$$

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 of compactness, 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 50.12. *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 50.13 (Compactness). *Suppose that $y : \tau \vdash e : \mathtt{nat}$ where $y \notin f^{(\omega)}$. 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 $m \geq 0$,

$$[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

and

$$[f^{(\omega)}/y]k \triangleleft [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleleft [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}.$$

(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 29.4 on page 277).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)},$$

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)} \mapsto [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

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

$$[f^{(m)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

By Lemma 50.12 on the preceding page

$$[f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}$$

and we need only observe that

$$[f^{(m+1)}/y]k \triangleright f^{(m+1)} \mapsto [f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x$$

to complete the proof. If, on the other hand, the initial step is an unrolling, but $e \neq y$, then we have for some $z \notin f^{(\omega)}$ and $z \neq y$

$$[f^{(\omega)}/y]k \triangleright \mathtt{fix}\, z{:}\tau\, \mathtt{is}\, d_\omega \mapsto [f^{(\omega)}/y]k \triangleright [\mathtt{fix}\, z{:}\tau\, \mathtt{is}\, d_\omega/z]d_\omega \mapsto^* \epsilon \triangleleft \overline{n}.$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [\mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_m/z]d_m \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 50.12 on page 511 we have

$$[f^{(m+1)}/y]k \triangleright [\mathtt{fix}\, z\mathtt{:}\tau\, \mathtt{is}\, d_{m+1}/z]d_{m+1} \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

Corollary 50.14. *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 50.13 on the facing page to obtain m, and note that the required Kleene equivalence follows. □

50.5 Co-Natural Numbers

In Chapter 12 we considered a variation of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with the co-natural numbers, `conat`, as base type. This is achieved by specifying that $\mathtt{s}(e)$ 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 $\mathtt{fix}\, x\mathtt{:conat}\, \mathtt{is}\, \mathtt{s}(x)$, which consists of an infinite stack of successors. Since the successor is intepreted lazily, ω evaluates to a value, namely $\mathtt{s}(\omega)$, its own successor. It follows that the principle of mathematical induction is not valid for the co-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 co-natural numbers. The main difference is that the definition of logical equivalence at type `conat` 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 `conat` 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 $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and *vice versa*; and (b) if $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$, 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 50.14 on the previous page can be proved for the co-natural numbers by essentially the same argument.

The definition of logical equivalence at type `conat` is defined to be the *weakest* equivalence relation, $\mathcal{E}$, between closed terms of type `conat` satisfying the following *`conat`-consistency conditions*: if $e\ \mathcal{E}\ e' : \mathtt{conat}$, then

1. If $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and *vice versa*.
2. If $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$ with $e_1\ \mathcal{E}\ e_1' : \mathtt{conat}$, and *vice versa*.

It is immediate that if $e \sim_{\mathtt{conat}} e'$, then $e \simeq e'$, and so logical equivalence is consistent. It is also strict in that if e and e' are both divergent expressions of type `conat`, then $e \sim_{\mathtt{conat}} e'$—simply because the preceding two conditions are vacuously true in this case.

This is an example of the more general principle of *proof by `conat`-coinduction*. To show that $e \sim_{\mathtt{conat}} e'$, it suffices to exhibit a relation, $\mathcal{E}$, such that

1. $e\ \mathcal{E}\ e' : \mathtt{conat}$, and
2. $\mathcal{E}$ satisfies the `conat`-consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in logical equivalence at type `conat`, and hence $e \sim_{\mathtt{conat}} e'$, as required.

As an application of `conat`-coinduction, let us consider the proof of Theorem 50.5 on page 507. The overall argument remains as before, but the proof for the type `conat` must be altered as follows. Suppose that $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \mathtt{conat}$, 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

$$\text{for every } m \geq 0,\ a^{(m)} \sim_{\mathtt{conat}} a'^{(m)}.$$

We are to show that

$$a \sim_{\mathtt{conat}} a'.$$

Define the functions p_n for $n \geq 0$ on closed terms of type `conat` 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 $a_n \mathrel{\mathcal{E}} a'_n : \mathtt{conat}$ for all $n \geq 0$. We will show that the relation $\mathcal{E}$ satisfies the `conat`-consistency conditions, and so it is contained in logical equivalence. Since $a \mathrel{\mathcal{E}} a' : \mathtt{conat}$ (by construction), the result follows immediately.

To show that $\mathcal{E}$ is `conat`-consistent, suppose that $a_n \mathrel{\mathcal{E}} a'_n : \mathtt{conat}$ for some $n \geq 0$. We have by Corollary 50.14 on page 513 $a_n \simeq a_n^{(m)}$, for some $m \geq 0$, and hence, by the assumption, $a_n \simeq {a'_n}^{(m)}$, and so by Corollary 50.14 on page 513 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 $b_n \mathrel{\mathcal{E}} b'_n : \mathtt{conat}$ by construction, as required.

50.6 Notes

The use of logical relations to characterize observational equivalence for **PCF** is inspired by Constable and Smith's treatment of partiality in type theory [24] and by Pitts's studies observational equivalence [80]. Although the technical details differ, the proof of compactness here is inspired by Pitts's structurally inductive characterization of termination using an abstract machine. It is critical to restrict attention to transition systems whose states are complete programs (closed expressions of observable type); structural operational semantics usually does not fulfill this requirement, thereby requiring a considerably more complex argument than given here.

Chapter 51

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 22. 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 collection of programs we may write, and also to limit the collection 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.

51.1 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 22, 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.

51.2 Observational Equivalence

The definition of observational equivalence given in Chapters 49 and 50 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 51.1. *Two expressions of the same type are* observationally equivalent, *written* $\Delta; \Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$.

Lemma 51.1. *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 51.2.

1. *If* $\Delta, t; \Gamma \vdash e \cong e' : \tau$ *and* ρ *type, then* $\Delta; [\rho/t]\Gamma \vdash [\rho/t]e \cong [\rho/t]e' : [\rho/t]\tau$.
2. *If* $\emptyset; \Gamma, x : \rho \vdash e \cong e' : \tau$ *and* $d : \rho$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d/x]e' : \tau$. *Moreover, if* $d \cong_\rho d'$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d'/x]e : \tau$ *and* $\emptyset; \Gamma \vdash [d/x]e' \cong [d'/x]e' : \tau$.

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 49.5 on page 497. □

51.3 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 $\mathcal{L}\{\to\forall\}$ is somewhat more complex than for $\mathcal{L}\{\mathtt{nat}\to\}$. 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 49 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 some form) 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 collection of "admissible" binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

Definition 51.2 (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 $d \cong_\rho e$ and $d' \cong_{\rho'} e'$, 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 $R(e, d')$.*

Closure under converse evaluation with turn out to be a consequence of respect for observational equivalence, 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. An *admissible 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 an admissible relation assignment between δ and δ'.

Logical equivalence is defined in terms of its generalization, called *parametric logical equivalence*, written $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$, defined as follows.

Definition 51.3 (Parametric Logical Equivalence)**.** *The relation $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta']$ is defined by induction on the structure of τ by the following conditions:*

$$
\begin{array}{lll}
e \sim_t e'\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e, e') \\
e \sim_{\mathbf{2}} e'\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e \simeq e' \\
e \sim_{\tau_1 \to \tau_2} e'\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim_{\tau_1} e_1'\ [\eta : \delta \leftrightarrow \delta'] \textit{ implies} \\
 & & e\,(e_1) \sim_{\tau_2} e'\,(e_1')\ [\eta : \delta \leftrightarrow \delta'] \\
e \sim_{\forall\,(t\,.\,\tau)} e'\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every } \rho,\ \rho', \textit{ and every admissible } R : \rho \leftrightarrow \rho', \\
 & & e\,[\rho] \sim_\tau e'\,[\rho']\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']]
\end{array}
$$

Logical equivalence is defined in terms of parametric 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 51.4 (Logical Equivalence)**.** *The expressions $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$ are* logically equivalent*, written $\Delta; \Gamma \vdash e \approx e' : \tau$ iff for every assigment δ and δ' of closed types to type variables in Δ, and every admissible relation assignment $\eta : \delta \leftrightarrow \delta'$, if $\gamma \sim_\Gamma \gamma'\ [\eta : \delta \leftrightarrow \delta']$, then $\hat{\gamma}(\hat{\delta}(e)) \sim_\tau \widehat{\gamma'}(\widehat{\delta'}(e'))\ [\eta : \delta \leftrightarrow \delta']$.*

When e, e', and τ are closed, then this definition states that $e \sim_\tau e'$ iff $e \sim_\tau e'\ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

Lemma 51.3 (Closure under Converse Evaluation)**.** *Suppose that $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta']$. If $d \mapsto e$, then $d \sim_\tau e'$, and if $d' \mapsto e'$, then $e \sim_\tau d'$.*

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 51.4 (Respect for Observational Equivalence)**.** *Suppose that* $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$*. If* $d \cong_{\hat{\delta}(\tau)} e$ *and* $d' \cong_{\hat{\delta}'(\tau)} e'$*, then* $d \sim_\tau d' \ [\eta : \delta \leftrightarrow \delta']$*.*

Proof. By induction on the structure of τ, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall(t.\tau_2)$, then we are to show that for every admissible $R : \rho \leftrightarrow \rho'$,

$$d\,[\rho] \sim_{\tau_2} d'\,[\rho'] \ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']].$$

Since observational equivalence is a congruence, $d\,[\rho] \cong_{[\rho/t]\hat{\delta}(\tau_2)} e\,[\rho]$, $d'\,[\rho] \cong_{[\rho'/t]\hat{\delta}'(\tau_2)} e'\,[\rho]$. From the assumption it follows that

$$e\,[\rho] \sim_{\tau_2} e'\,[\rho'] \ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']],$$

from which the result follows by induction. □

Corollary 51.5. *The relation* $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$ *is an admissible relation between closed types* $\hat{\delta}(\tau)$ *and* $\hat{\delta}'(\tau)$*.*

Proof. By Lemmas 51.3 on the facing page and 51.4. □

Corollary 51.6. *If* $\Delta; \Gamma \vdash e \approx e' : \tau$*, and* $\Delta; \Gamma \vdash d \cong e : \tau$ *and* $\Delta; \Gamma \vdash d' \cong e' : \tau$*, then* $\Delta; \Gamma \vdash d \approx d' : \tau$*.*

Proof. By Lemma 51.2 on page 519 and Corollary 51.5. □

Lemma 51.7 (Compositionality)**.** *Suppose that*

$$e \sim_\tau e' \ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\delta}(\rho)] \leftrightarrow \delta'[t \mapsto \hat{\delta}'(\rho)]],$$

where $R : \hat{\delta}(\rho) \leftrightarrow \hat{\delta}'(\rho)$ *is such that* $R(d, d')$ *holds iff* $d \sim_\rho d' \ [\eta : \delta \leftrightarrow \delta']$*. Then* $e \sim_{[\rho/t]\tau} e' \ [\eta : \delta \leftrightarrow \delta']$*.*

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 $u \notin \rho$, 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 51.8 (Parametricity)**.** *If* $\Delta; \Gamma \vdash e : \tau$*, then* $\Delta; \Gamma \vdash e \approx e : \tau$*.*

Proof. By rule induction on the statics of $\mathcal{L}\{\to\forall\}$ given by Rules (22.2). We consider two representative cases here.

Rule (22.2d) 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 admissible $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}(\hat{\delta}(e)) \sim_\tau [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e))\ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

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 51.3 on page 522.

Rule (22.2e) Suppose $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and $\gamma \sim_\Gamma \gamma'$ $[\eta : \delta \leftrightarrow \delta']$. By induction we have

$$\hat{\gamma}(\hat{\delta}(e)) \sim_{\forall(t.\tau)} \widehat{\gamma'}(\widehat{\delta'}(e))\ [\eta : \delta \leftrightarrow \delta']$$

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 $d \sim_\rho d'$ $[\eta : \delta \leftrightarrow \delta']$. By Corollary 51.5 on the previous page, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim_\tau \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}]\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \hat{\rho'}]].$$

By Lemma 51.7 on the preceding page

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim_{[\rho/t]\tau} \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}]\ [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))\,[\hat{\delta}(\rho)] && (51.1)\\ &= \hat{\gamma}(\hat{\delta}(e\,[\rho])), && (51.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] &= \widehat{\gamma'}(\widehat{\delta'}(e))\,[\widehat{\delta'}(\rho)] && (51.3)\\ &= \widehat{\gamma'}(\widehat{\delta'}(e\,[\rho])), && (51.4) \end{aligned}$$

from which the result follows.

□

Corollary 51.9. *If* $\Delta;\Gamma \vdash e \cong e' : \tau$, *then* $\Delta;\Gamma \vdash e \approx e' : \tau$.

Proof. By Theorem 51.8 on page 523 $\Delta;\Gamma \vdash e \approx e : \tau$, and hence by Corollary 51.6 on page 523, $\Delta;\Gamma \vdash e \approx e' : \tau$. □

Lemma 51.10 (Congruence)**.** *If* $\Delta;\Gamma \vdash e \approx e' : \tau$ *and* $\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\Delta';\Gamma' \triangleright \tau')$, *then* $\Delta';\Gamma' \vdash \mathcal{C}\{e\} \approx \mathcal{C}\{e'\} : \tau$.

Proof. By induction on the structure of $\mathcal{C}$, following along very similar lines to the proof of Theorem 51.8 on page 523. □

Lemma 51.11 (Consistency)**.** *Logical equivalence is consistent.*

Proof. Follows immediately from the definition of logical equivalence. □

Corollary 51.12. *If* $\Delta;\Gamma \vdash e \approx e' : \tau$, *then* $\Delta;\Gamma \vdash e \cong e' : \tau$.

Proof. By Lemma 51.11 Logical equivalence is consistent, and by Lemma 51.10, it is a congruence, and hence is contained in observational equivalence. □

Corollary 51.13. *Logical and observational equivalence coincide.*

Proof. By Corollaries 51.9 and 51.12. □

If $d : \tau$ and $d \mapsto e$, then $d \sim_\tau e$, and hence by Corollary 51.12, $d \cong_\tau e$. 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, now that we have established the coincidence of logical and observational equivalence.

Corollary 51.14 (Extensionality)**.**

1. $e \cong_{\tau_1 \to \tau_2} e'$ *iff for all* $e_1 : \tau_1$, $e\,(e_1) \cong_{\tau_2} e'\,(e_1)$.
2. $e \cong_{\forall\,(t\,.\,\tau)} e'$ *iff for all* ρ, $e\,[\rho] \cong_{[\rho/t]\tau} e'\,[\rho]$.

Proof. The forward direction is immediate in both cases, since observational equivalence is a congruence, by definition. The backward direction is proved similarly in both cases, by appeal to Theorem 51.8 on page 523. In the first case, by Corollary 51.13 it suffices to show that $e \sim_{\tau_1 \to \tau_2} e'$. To this end suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By the assumption we have $e(e_1') \cong_{\tau_2} e'(e_1')$. By parametricity we have $e \sim_{\tau_1 \to \tau_2} e$, and hence $e(e_1) \sim_{\tau_2} e(e_1')$. The result then follows by Lemma 51.4 on

page 523. In the second case, by Corollary 51.13 on the preceding page it is sufficient to show that $e \sim_{\forall(t.\tau)} e'$. Suppose that $R : \rho \leftrightarrow \rho'$ for some closed types ρ and ρ'. It suffices to show that $e[\rho] \sim_\tau e'[\rho'] \ [\eta : \delta \leftrightarrow \delta']$, where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$. By the assumption we have $e[\rho'] \cong_{[\rho/t]\tau} e'[\rho']$. By parametricity $e \sim_{\forall(t.\tau)} e$, and hence $e[\rho] \sim_\tau e'[\rho'] \ [\eta : \delta \leftrightarrow \delta']$. The result then follows by Lemma 51.4 on page 523. □

Lemma 51.15 (Identity Extension)**.** *Let $\eta : \delta \leftrightarrow \delta$ be such that $\eta(t)$ is observational equivalence at type $\delta(t)$ for each $t \in dom(\delta)$. Then $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta]$ iff $e \cong_{\hat{\delta}(\tau)} e'$.*

Proof. The backward direction follows immediately from Theorem 51.8 on page 523 and respect for observational equivalence. The forward direction is proved by induction on the structure of τ, appealing to Corollary 51.14 on the previous page to establish observational equivalence at function and polymorphic types. □

51.4 Parametricity Properties

The parametricity theorem enables us to deduce properties of expressions of $\mathcal{L}\{\to\forall\}$ 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 $\forall(t.t \to t)$ behaves like the identity function.

Theorem 51.16. *Let $e : \forall(t.t \to t)$ be arbitrary, and let id be $\Lambda(t.\lambda(x:t.x))$. Then $e \cong_{\forall(t.t\to t)} id$.*

Proof. By Corollary 51.13 on the preceding page it is sufficient to show that $e \sim_{\forall(t.t\to t)} id$. 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 $e[\rho](e_0) \cong_\rho e_0$, for then the result follows by the admissibility of R and the assumption $e_0 \ R \ e_0'$.

By Theorem 51.8 on page 523 we have $e \sim_{\forall(t.t\to t)} e$. Let the relation $S : \rho \leftrightarrow \rho$ be defined by $d \; S \; d'$ iff $d \cong_\rho e_0$ and $d' \cong_\rho e_0$. 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, $e[\rho](e_0) \cong_\rho e_0$.

□

In Chapter 22 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 dynamics of these constructs as given in the earlier chapters is derivable from the dynamics 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 22 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 51.16 on the facing 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 22:

$$\begin{aligned}\tau_1 \times \tau_2 &= \forall(r.(\tau_1 \to \tau_2 \to r) \to r)\\ \langle e_1, e_2\rangle &= \Lambda(r.\lambda\,(x{:}\tau_1 \to \tau_2 \to r.\,x(e_1)(e_2)))\\ e \cdot \texttt{l} &= e[\tau_1](\lambda\,(x{:}\tau_1.\lambda\,(y{:}\tau_2.\,x)))\\ e \cdot \texttt{r} &= e[\tau_2](\lambda\,(x{:}\tau_1.\lambda\,(y{:}\tau_2.\,y)))\end{aligned}$$

It is easy to check that $\langle e_1, e_2\rangle \cdot \texttt{l} \cong_{\tau_1} e_1$ and $\langle e_1, e_2\rangle \cdot \texttt{r} \cong_{\tau_2} e_2$ 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 51.8 on page 523.

Lemma 51.17. *If $e : \tau_1 \times \tau_2$, then $e \cong_{\tau_1\times\tau_2} \langle e_1, e_2\rangle$ for some $e_1 : \tau_1$ and $e_2 : \tau_2$.*

Proof. Expanding the definitions of pairing and the product type, and applying Corollary 51.13 on page 525, we let ρ and ρ' be arbitrary closed types, and let $R : \rho \leftrightarrow \rho'$ be an admissible relation between them. Suppose further that

$$h \sim_{\tau_1\to\tau_2\to t} h' \ [\eta : \delta \leftrightarrow \delta'],$$

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 $e_2 : \tau_2$,

$$e\,[\rho]\,(h) \sim_t h'\,(e_1)\,(e_2) \ [\eta : \delta \leftrightarrow \delta'],$$

which is to say

$$e\,[\rho]\,(h) \ R \ h'\,(e_1)\,(e_2).$$

Now by Theorem 51.8 on page 523 we have $e \sim_{\tau_1\times\tau_2} e$. Define the relation $S : \rho \leftrightarrow \rho'$ by $d \ S \ d'$ iff the following conditions are satisfied:

1. $d \cong_\rho h\,(d_1)\,(d_2)$ for some $d_1 : \tau_1$ and $d_2 : \tau_2$;
2. $d' \cong_{\rho'} h'\,(d_1')\,(d_2')$ for some $d_1' : \tau_1$ and $d_2' : \tau_2$;
3. $d \ R \ d'$.

This is clearly an admissible relation. Noting that

$$h \sim_{\tau_1\to\tau_2\to t} h' \ [\eta' : \delta \leftrightarrow \delta'],$$

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 $e \cdot \mathtt{l} \cong_{\tau_1} e_1$ and $e \cdot \mathtt{r} \cong_{\tau_2} e_2$. We wish to show that $e \cong_{\tau_1\times\tau_2} \langle e_1, e_2\rangle$. From Lemma 51.17 it is easy to deduce that $e \cong_{\tau_1\times\tau_2} \langle e \cdot \mathtt{l}, e \cdot \mathtt{r}\rangle$ by congruence and direct calculation. Hence, by congruence we have $e \cong_{\tau_1\times\tau_2} \langle e_1, e_2\rangle$.

By a similar line of reasoning we may show that the Church encoding of the natural numbers given in Chapter 22 strongly defines the natural numbers in that the following properties hold:

1. $\texttt{natiter}\,\texttt{z}\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\} \cong_\rho e_0$.
2. $\texttt{natiter}\,\texttt{s}(e)\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\} \cong_\rho [\texttt{natiter}\,e\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}/x]e_1$.
3. Suppose that $x : \texttt{nat} \vdash r(x) : \rho$. If
   (a) $r(\texttt{z}) \cong_\rho e_0$, and
   (b) $r(\texttt{s}(e)) \cong_\rho [r(e)/x]e_1$,

   then for every $e : \texttt{nat}$, $r(e) \cong_\rho \texttt{natiter}\,e\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}$.

The first two equations, which constitute weak definability, are easily established by calculation, using the definitions given in Chapter 22. 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 51.18. *If $e : \texttt{nat}$, then either $e \cong_{\texttt{nat}} \texttt{z}$, or there exists $e' : \texttt{nat}$ such that $e \cong_{\texttt{nat}} \texttt{s}(e')$. Consequently, there exists $n \geq 0$ such that $e \cong_{\texttt{nat}} \overline{n}$.*

Proof. By Theorem 51.8 on page 523 we have $e \sim_{\texttt{nat}} e$. Define the relation $R : \texttt{nat} \leftrightarrow \texttt{nat}$ to be the strongest relation such that $d\ R\ d'$ iff either $d \cong_{\texttt{nat}} \texttt{z}$ and $d' \cong_{\texttt{nat}} \texttt{z}$, or $d \cong_{\texttt{nat}} \texttt{s}(d_1)$ and $d' \cong_{\texttt{nat}} \texttt{s}(d'_1)$ and $d_1\ R\ d'_1$. It is easy to see that $\texttt{z}\ R\ \texttt{z}$, and if $e\ R\ e'$, then $\texttt{s}(e)\ R\ \texttt{s}(e')$. Letting $\texttt{zero} = \texttt{z}$ and $\texttt{succ} = \lambda\,(x{:}\texttt{nat}.\,\texttt{s}(x))$, we have

$$e\texttt{[nat](zero)(succ)}\ R\ e\texttt{[nat](zero)(succ)}.$$

The result follows by the induction principle arising from the definition of R as the strongest relation satisfying its defining conditions. □

51.5 Representation Independence, Revisited

In Section 23.4 on page 215 we discussed the property of *representation independence* for abstract types. This property states that if two implementations of an abstract type are "similar", then the client behavior is not affected by replacing one for the other. The crux of the matter is the definition of similarity of two implementations. Informally, two implementations of an abstract type are similar if there is a relation, E, between their representation types that is *preserved* by the operations of the type. The relation E

may be thought of as expressing the "equivalence" of the two representations; checking that each operation preserves E amounts to checking that the result of performing that operation on equivalent representations yields equivalent results.

As an example, we argued in Section 23.4 on page 215 that two implementations of a queue abstraction are similar. In the one the queue is represented by a list of elements in reverse arrival order (the latest element to arise is the head of the list). Enqueueing an element is easy; simply add it to the front of the list. Dequeueing an element requires reversing the list, removing the first element, and reversing the rest to obtain the new queue. In the other the queue is represented by a pair of lists, with the "back half" representing the latest arrivals in reverse order of arrival time, and the "front half" representing the oldest arrivals in order of arrival (the next one to depart the queue is at the head of the list). Enqueueing remains easy; the element is added to the "back half" as the first element. Dequeueing breaks into two cases. If the "front half" is non-empty, simply remove the head element and return the queue consisting of the "back half" as-is together with the tail of the "front half". If, on the other hand, the "front half" is empty, then the queue is reorganizing by reversing the "back half" and making it the new "front half", leaving the new "back half" empty. These two representations of queues are related by the relation E such that $q\,E\,(b,f)$ iff q is b followed by the reversal of f. It is easy to check that the operations of the queue preserve this relationship.

In Chapter 23 we asserted without proof that the existence of such a relation was sufficient to ensure that the behavior of any client is insensitive to the choice of either implementation. The proof of this intuitively plausible result relies on parametricity. One way to explain this is via the definition of existential types in $\mathcal{L}\{\rightarrow\forall\}$ described in Section 23.3 on page 214. According to that definition, the client, e, of an abstract type $\exists(t.\tau)$ is a polymorphic function of type $\forall(t.\tau \rightarrow \tau_2)$, where τ_2, the result type of the computation, does not involve the type variable t. Being polymorphic, the client enjoys the parametricity property given by Theorem 51.8 on page 523. Specifically, suppose that ρ_1 and ρ_2 are two closed representation types and that $R : \rho_1 \leftrightarrow \rho_2$ is an admissible relation between them. For example, in the case of the queue abstraction, ρ_1 is the type of lists of elements of the queue, ρ_2 is the type of a pair of lists of elements, and R is the relation E given above. Suppose further that $e_1 : [\rho_1/t]\tau$ and $e_2 : [\rho_2/t]\tau$ are two implementations of the operations such that

$$e_1 \sim_\tau e_2\ [\eta : \delta_1 \leftrightarrow \delta_2], \tag{51.5}$$

where $\eta(t) = R$, $\delta_1(t) = \rho_1$, and $\delta_2(t) = \rho_2$. In the case of the queues example the expression e_1 is the implementation of the queue operations in terms of lists, and the e_2 is the implementation in terms of pairs of lists described earlier. Condition (51.5) states that the two implementations are similar in that they preserve the relation R between the representation types. By Theorem 51.8 on page 523 it follows that the client, e, satisfies

$$e \sim_{\tau_2} e\,[\eta : \delta_1 \leftrightarrow \delta_2].$$

But since τ_2 is a closed type (in particular, does not involve t), this is equivalent to

$$e \sim_{\tau_2} e\,[\emptyset : \emptyset \leftrightarrow \emptyset].$$

But then by Lemma 51.15 on page 526 we have

$$e\,[\rho_1]\,(e_1) \cong_{\tau_2} e\,[\rho_2]\,(e_2).$$

That is, the client behavior is not affected by the change of representation.

51.6 Notes

The concept of parametricity is latent in Girard's proof of normalization for System **F** [32], because only predicates, rather than binary relations, are considered. Reynolds's analysis [86], though technically flawed because of its reliance on a (non-existent) set-theoretic model of System **F**, emphasizes the centrality of logical equivalence for characterizing equality of polymorphic programs. The application of parametricity to representation independence was suggested by Reynolds, and developed for existential types by Mitchell [69] and Pitts [79]. The extension of System **F** with an observable type appears to be essential to even define observational equivalence, but this point seems not to have been made elsewhere in the literature.

Chapter 52

Process Equivalence

As the name implies a process is an ongoing computation that may interact with other processes by sending and receiving messages. From this point of view a concurrent computation has no definite "final outcome" but rather affords an opportunities for interaction that may well continue indefinitely. The notion of equivalence of processes must therefore be based on their potential for interaction, rather than on the "answer" that they may compute. Let P and Q be such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc. We say that P and Q are *equivalent*, written $P \approx_\Sigma Q$, iff there is a bisimulation, $\mathcal{R}$, such that $P\,\mathcal{R}_\Sigma\,Q$. A family of relations $\mathcal{R} = \{\mathcal{R}_\Sigma\}_\Sigma$ is a *bisimulation* iff whenever P may evolve to P' taking the action α, then Q may also evolve to some process Q' taking the same action such that $P'\ \mathcal{R}_\Sigma\ Q'$, and, conversely, if Q may evolve to Q' taking action α, then P may evolve to P' taking the same action, and $P'\ \mathcal{R}_\Sigma\ Q'$. This captures the idea that the two processes afford the same opportunities for interaction in that they each simulate each other's behavior with respect to their ability to interact with their environment.

52.1 Process Calculus

We will consider a process calculus that consolidates the main ideas explored in Chapters 43 and 44. We will assume given an ambient language of expressions that includes the type `clsfd` of classified values (see Chapter 36). Channels are treated as dynamically generated classes with which to build messages, as described in Chapter 44.

The syntax of the process calculus is given by the following grammar:

Proc	P	::=	`stop`	$\mathbf{1}$	inert
			`par`$(P_1; P_2)$	$P_1 \parallel P_2$	composition
			`await`(E)	$\$\, E$	synchronize
			`new[`τ`](`$a.P$`)`	$\nu\, a{:}\tau.\, P$	allocation
			`emit`(e)	$!\, e$	broadcast
Evt	E	::=	`null`	$\mathbf{0}$	null
			`or`$(E_1; E_2)$	$E_1 + E_2$	choice
			`acc`$(x.P)$	$?\,(x.P)$	acceptance

The statics is given by the judgements $\Gamma \vdash_\Sigma P$ proc and $\Gamma \vdash_\Sigma E$ event defined by the following rules. We assume given a judgement $\Gamma \vdash_\Sigma e : \tau$ for τ a type including the type `clsfd` of classified values.

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{52.1a}$$

$$\frac{\Gamma \vdash_\Sigma P_1\ \mathsf{proc} \quad \Gamma \vdash_\Sigma P_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{52.1b}$$

$$\frac{\Gamma \vdash_\Sigma E\ \mathsf{event}}{\Gamma \vdash_\Sigma \$\, E\ \mathsf{proc}} \tag{52.1c}$$

$$\frac{\Gamma \vdash_{\Sigma, a:\tau} P\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a{:}\tau.\, P\ \mathsf{proc}} \tag{52.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma !\, e\ \mathsf{proc}} \tag{52.1e}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{52.1f}$$

$$\frac{\Gamma \vdash_\Sigma E_1\ \mathsf{event} \quad \Gamma \vdash_\Sigma E_2\ \mathsf{event}}{\Gamma \vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{52.1g}$$

$$\frac{\Gamma, x : \mathtt{clsfd} \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma ?\,(x.P)\ \mathsf{event}} \tag{52.1h}$$

The dynamics is given by the judgements $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ and $E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$, defined as in Chapter 43. We assume given the judgements $e \underset{\Sigma}{\mapsto} e'$ and $e\ \mathsf{val}_\Sigma$ for expressions. Processes and events are identified up to structural congruence, as described in Chapter 43.

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2} \tag{52.2a}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \quad P_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} P_2'}{P_1 \parallel P_2 \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2'} \tag{52.2b}$$

$$\frac{E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P}{\$\, E \overset{\alpha}{\underset{\Sigma}{\mapsto}} P} \tag{52.2c}$$

$$\frac{P \overset{\alpha}{\underset{\Sigma, a:\tau}{\longmapsto}} P' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a:\tau\,.\, P \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a:\tau\,.\, P'} \tag{52.2d}$$

$$\frac{e \text{ val}_\Sigma \quad \vdash_\Sigma e : \texttt{clsfd}}{!\, e \overset{e\,!}{\underset{\Sigma}{\mapsto}} \mathbf{1}} \tag{52.2e}$$

$$\frac{E_1 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P}{E_1 + E_2 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P} \tag{52.2f}$$

$$\frac{e \text{ val}_\Sigma}{?\,(x\,.\,P) \overset{e\,?}{\underset{\Sigma}{\hookrightarrow}} [e/x]P} \tag{52.2g}$$

Assuming that substitution is valid for expressions, it is also valid for processes and events.

Lemma 52.1.

1. *If* $\Gamma, x : \tau \vdash_\Sigma P$ *proc and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]P$ *proc.*

2. *If* $\Gamma, x : \tau \vdash_\Sigma E$ *event and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]E$ *event.*

Transitions preserve well-formedness of processes and events.

Lemma 52.2.

1. *If* $\vdash_\Sigma P$ *proc and* $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$, *then* $\vdash_\Sigma P'$ *proc.*

2. *If* $\vdash_\Sigma E$ *event and* $P \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$, *then* $\vdash_\Sigma P$ *proc.*

52.2 Strong Equivalence

Bisimilarity makes precise the informal idea that two processes are equivalent if they each can take the same actions and, in doing so, evolve into equivalent processes. A *process relation*, $\mathcal{P}$, is a family $\{ \mathcal{P}_\Sigma \}$ of binary relations between processes P and Q such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc, and an *event relation*, $\mathcal{E}$, is a family $\{ \mathcal{E}_\Sigma \}$ of binary relations between events E and F such that $\vdash_\Sigma E$ event and $\vdash_\Sigma F$ event. A *(strong) bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation, $\mathcal{P}$, and an event relation, $\mathcal{E}$, satisfying the following conditions:

1. If $P \; \mathcal{P}_\Sigma \; Q$, then
   (a) if $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$, then there exists Q' such that $Q \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$ with $P' \; \mathcal{P}_\Sigma \; Q'$, and
   (b) if $Q \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$, then there exists P' such that $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$ with $P' \; \mathcal{P}_\Sigma \; Q'$.
2. If $E \; \mathcal{E}_\Sigma \; F$, then
   (a) if $E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$, then there exists Q such that $F \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} Q$ with $P \; \mathcal{P}_\Sigma \; Q$, and
   (b) if $F \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} Q$, then there exists P such that $E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$ with $P \; \mathcal{P}_\Sigma \; Q$.

The qualifier "strong" refers to the fact that the action, α, in the conditions on being a bisimulation include the silent action, ε. (In Section 52.3 on page 539 we discuss another notion of bisimulation in which the silent actions are treated specially.)

(Strong) equivalence is the pair $(\approx, \approx)$ of process and event relations such that $P \approx_\Sigma Q$ and $E \approx_\Sigma F$ iff there exists a strong bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P \; \mathcal{P}_\Sigma \; Q$, and $E \; \mathcal{E}_\Sigma \; F$.

Lemma 52.3. *Strong equivalence is a strong bisimulation.*

Proof. Follows immediately from the definition. □

The definition of strong equivalence gives rise to the principle of *proof by coinduction*. To show that $P \approx_\Sigma Q$, it is enough to give a bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P \; \mathcal{P}_\Sigma \; Q$ (and similarly for events). An instance of coinduction that arises fairly frequently is to choose $(\mathcal{P}, \mathcal{E})$ to be $(\approx \cup \mathcal{P}_0, \approx \cup \mathcal{E}_0)$ for some $\mathcal{P}_0$ and $\mathcal{E}_0$ such that $P \; \mathcal{P}_0 \; Q$, and show that this expansion is a bisimulation. Since strong equivalence is itself a bisimulation, this reduces to show that if

$P'\ \mathcal{P}_0\ Q'$ and $P' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P''$, then $Q' \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q''$ for some Q'' such that either $P'' \approx_\Sigma Q''$ or $P''\ \mathcal{P}_0\ Q''$ (and analogously for transitions from Q', and similarly for event transitions). This proof method amounts to *assuming what we are trying to prove* and showing that this assumption is tenable. The proof that the expanded relation is a bisimulation may make use of the assumptions $\mathcal{P}_0$ and $\mathcal{E}_0$; in this sense "circular reasoning" is a valid method of proof!

Lemma 52.4. *Strong equivalence is an equivalence relation.*

Proof. For reflexivity and symmetry, it suffices to note that the identity relation is a bisimulation, as is the converse of a bisimulation. For transitivity we need that the composition of two bisimulations is again a bisimulation, which follows directly from the definition. □

It remains to verify that strong equivalence is a congruence, which means that each of the process- and event-forming constructs respects strong equivalence. To show this we require the *open extension* of strong equivalence to processes and events with free variables. The relation $\Gamma \vdash_\Sigma P \approx Q$ is defined for processes P and Q such that $\Gamma \vdash_\Sigma P$ proc and $\Gamma \vdash_\Sigma Q$ proc to mean that $\hat{\gamma}(P) \approx_\Sigma \hat{\gamma}(Q)$ for every substitution, γ, of closed values of appropriate type for the variables Γ.

Lemma 52.5. *If* $\Gamma, x : \mathtt{clsfd} \vdash_\Sigma P \approx Q$, *then* $\Gamma \vdash_\Sigma ?\,(x.P) \approx ?\,(x.Q)$.

Proof. Fix a closing substitution, γ, for Γ, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. By assumption we have $x : \mathtt{clsfd} \vdash_\Sigma \hat{P} \approx \hat{Q}$. We are to show that $?\,(x.\hat{P}) \approx_\Sigma ?\,(x.\hat{Q})$. The proof is by coinduction, taking $\mathcal{P} = \approx$ and $\mathcal{E} = \approx \cup\, \mathcal{E}_0$, where

$$\mathcal{E}_0 = \{\, (?\,(x.P'), ?\,(x.Q')) \mid x : \mathtt{clsfd} \vdash_\Sigma P' \approx Q' \,\}.$$

Clearly $?\,(x.\hat{P})\ \mathcal{E}_0\ ?\,(x.\hat{Q})$. Suppose that $?\,(x.P')\ \mathcal{E}_0\ ?\,(x.Q')$. By inspection of Rules (52.2), if $?\,(x.P') \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P''$, then $\alpha = v?$ and $P'' = [v/x]P'$ for some v val_Σ such that $\vdash_\Sigma v : \mathtt{clsfd}$. But $?\,(x.Q') \overset{v?}{\underset{\Sigma}{\hookrightarrow}} [v/x]Q'$, and we have that $[v/x]P' \approx_\Sigma [v/x]Q'$ by the definition of $\mathcal{E}_0$, and hence $[v/x]P'\ \mathcal{E}_0\ [v/x]Q'$, as required. The symmetric case follows symmetrically, completing the proof. □

Lemma 52.6. *If* $\Gamma \vdash_{\Sigma, a:\tau} P \approx Q$, *then* $\Gamma \vdash_\Sigma \nu\, a:\tau.P \approx \nu\, a:\tau.Q$.

Proof. Fix a closing value substitution, γ, for Γ, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. Assuming that $\hat{P} \approx_{\Sigma,a:\tau} \hat{Q}$, we are to show that $\nu\, a{:}\tau\,.\,\hat{P} \approx_{\Sigma} \nu\, a{:}\tau\,.\,\hat{Q}$. The proof is by coinduction, taking $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\, (\nu\, a{:}\tau\,.\,P', \nu\, a{:}\tau\,.\,Q') \mid P' \approx_{\Sigma,a:\tau} Q' \,\}.$$

Clearly $\nu\, a{:}\tau\,.\,\hat{P}\ \mathcal{P}_0\ \nu\, a{:}\tau\,.\,\hat{Q}$. Suppose that $\nu\, a{:}\tau\,.\,P'\ \mathcal{P}_0\ \nu\, a{:}\tau\,.\,Q'$, and that $\nu\, a{:}\tau\,.\,P' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P''$. By inspection of Rules (52.2), we see that $\vdash_\Sigma \alpha$ action and that $P'' = \nu\, a{:}\tau\,.\,P'''$ for some P''' such that $P' \overset{\alpha}{\underset{\Sigma,a:\tau}{\longmapsto}} P'''$. But by definition of $\mathcal{P}_0$ we have $P' \approx_{\Sigma,a:\tau} Q'$, and hence $Q' \overset{\alpha}{\underset{\Sigma,a:\tau}{\longmapsto}} Q'''$ with $P''' \approx_{\Sigma,a:\tau} Q'''$. Letting $Q'' = \nu\, a{:}\tau\,.\,Q'$, we have that $\nu\, a{:}\tau\,.\,Q' \overset{\alpha}{\underset{\Sigma,a:\tau}{\longmapsto}} Q''$ and by definition of $\mathcal{P}_0$ we have $P''\ \mathcal{P}_0\ Q''$, as required. The symmetric case is proved symmetrically, completing the proof. □

Lemmas 52.5 on the previous page and 52.6 on the preceding page capture two different cases of binding, the former of variables, and the latter of classes. The hypothesis of Lemma 52.5 on the previous page relates all substitutions for the variable x in the recipient processes, whereas the hypothesis of Lemma 52.6 on the preceding page relates the consituent processes schematically in the class name, a. This makes all the difference, for if we were to consider all substitution instances of a class name by another class name, then it would no longer be ensured to be "new" within its scope—we could identify the "new" class name with an "old" one by substitution. On the other hand we must consider substitution instances for variables, since the semantics of communication is given by substitution of the accepted value into the accepting process. It is important that these distinct concepts be kept distinct! (See Chapter 36 for an example of what goes wrong when the two concepts are confused.)

Lemma 52.7. *If* $\Gamma \vdash_\Sigma P_1 \approx Q_1$ *and* $\Gamma \vdash_\Sigma P_2 \approx Q_2$, *then* $\Gamma \vdash_\Sigma P_1 \parallel P_2 \approx Q_1 \parallel Q_2$.

Proof. Let γ be a closing value substitution for Γ, and let $\hat{P}_i = \hat{\gamma}(P_i)$ and $\hat{Q}_i = \hat{\gamma}(Q_i)$ for $i = 1, 2$. The proof is by coinduction, considering the relation $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\, (P_1' \parallel P_2', Q_1' \parallel Q_2') \mid P_1' \approx_\Sigma Q_1' \text{ and } P_2' \approx_\Sigma Q_2' \,\}.$$

Suppose that $P_1' \parallel P_2'\ \mathcal{P}_0\ Q_1' \parallel Q_2'$, and that $P_1' \parallel P_2' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P''$. There are two cases to consider, the interesting one being Rule (52.2b). In this case we

have $P'' = P_1'' \parallel P_2''$ with $P_1' \xmapsto[\Sigma]{\alpha} P_1''$ and $P_2' \xmapsto[\Sigma]{\overline{\alpha}} P_2''$. By definition of $\mathcal{P}_0$ we have that $Q_1' \xmapsto[\Sigma]{\alpha} Q_1''$ and $Q_2' \xmapsto[\Sigma]{\overline{\alpha}} Q_2''$ with $P_1'' \approx_\Sigma Q_1''$ and $P_2'' \approx_\Sigma Q_2''$. Letting $Q'' = Q_1'' \parallel Q_2''$, we have that $P'' \; \mathcal{P}_0 \; Q''$, as required. The symmetric case is handled symmetrically, and Rule (52.2a) is handled similarly. □

Lemma 52.8. *If* $\Gamma \vdash_\Sigma E_1 \approx F_1$ *and* $\Gamma \vdash_\Sigma E_2 \approx F_2$, *then* $\Gamma \vdash_\Sigma E_1 + E_2 \approx F_1 + F_2$.

Proof. Follows immediately from Rules (52.2) and the definition of bisimulation. □

Lemma 52.9. *If* $\Gamma \vdash_\Sigma E \approx F$, *then* $\Gamma \vdash_\Sigma \$\,E \approx \$\,F$.

Proof. Follows immediately from Rules (52.2) and the definition of bisimulation. □

Lemma 52.10. *If* $\Gamma \vdash_\Sigma d \cong e : \mathtt{clsfd}$, *then* $\Gamma \vdash_\Sigma !\,d \approx !\,e$.

Proof. The process calculus introduces no new observations on expressions, so that d and e remain indistinguishable as actions. □

Theorem 52.11. *Strong equivalence is a congruence.*

Proof. Follows immediately from the preceding lemmas, which cover each case separately. □

52.3 Weak Equivalence

Strong equivalence expresses the idea that two processes are equivalent if they simulate each other step-by-step. Every action taken by one process is matched by a corresponding action taken by the other. This seems natural for the non-trivial actions $e\,!$ and $e\,?$, but is arguably overly restrictive for the silent action, ε. Silent actions correspond to the actual steps of computation, whereas the send and receive actions express the potential to interact with another process. Silent steps are therefore of a very different flavor than the other forms of action, and therefore might usefully be treated differently from them. Weak equivalence seeks to do just that.

Silent actions arise within the process calculus itself (when two processes communicate), but they play an even more important role when the dynamics of expressions is considered explicitly (as in Chapter 44). For then each step $e \xmapsto[\Sigma]{} e'$ of evaluation of an expression corresponds to a silent

transition for any process in which it is embedded. In particular, $!\,e \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} !\,e'$ whenever $e \underset{\Sigma}{\mapsto} e'$. One may also consider atomic processes of the form `proc(`m`)` consisting of a command to be executed in accordance with the rules of some underlying dynamics. Here again one would expect that each step of command execution induces a silent transition from one atomic process to another.

From the point of view of equivalence, it therefore seems sensible to allow that a silent action by one process may be mimicked by one or more silent actions by another. For example, there appears to be little to be gained by distinguishing, say, `proc(ret 3+4)` from `proc(ret (1+2)+(2+2))` merely because the latter takes more steps to compute the same value than the former! The purpose of weak equivalence is precisely to disregard such trivial distinctions by allowing a transition to be matched by a matching transition, possibly preceded by any number of silent transitions.

A *weak bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation, $\mathcal{P}$, and an event relation, $\mathcal{E}$, satisfying the following conditions:

1. If $P\ \mathcal{P}_\Sigma\ Q$, then
   (a) if $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$, where $\alpha \neq \varepsilon$, then there exists Q'' and Q' such that $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}^{*} Q'' \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} P'$, then $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}^{*} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;
   (b) if $Q \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$, where $\alpha \neq \varepsilon$, then there exists P'' and P' such that $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}^{*} P'' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} Q'$, then $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}^{*} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;
2. If $E\ \mathcal{E}_\Sigma\ F$, then
   (a) if $E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$, then there exists Q such that $F \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} Q$ with $P\ \mathcal{P}_\Sigma\ Q$, and
   (b) if $F \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} Q$, then there exists P such that $E \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} P$ with $P\ \mathcal{P}_\Sigma\ Q$.

(The conditions on the event relation are the same as for strong bisimilarity because there are, in this calculus, no silent actions on events.)

Weak equivalence is the pair $(\sim, \sim)$ of process and event relations defined by $P \sim_\Sigma Q$ and $E \sim_\Sigma F$ iff there exists a weak bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P\ \mathcal{P}_\Sigma\ Q$, and $E\ \mathcal{E}_\Sigma\ F$. The open extension of weak equivalence, written

$\Gamma \vdash_\Sigma P \sim Q$ and $\Gamma \vdash_\Sigma E \sim F$, is defined exactly as is the open extension of strong equivalence.

Theorem 52.12. *Weak equivalence is an equivalence relation and a congruence.*

Proof. Very similar to the proofs of the same properties for strong equivalence. □

52.4 Notes

The literature on process equivalence is extensive. Numerous variations have been considered for an equally numerous array of formalisms. Milner recounts the history and development of the concept of bisimilarity in his monograph on the π-calculus [66], crediting David Park with its original conception [76]. A recurring problem is the failure of bisimilarity to be a congruence, a problem that is avoided here by adhering to the distinction between symbols and variables. The development in this chapter is inspired by Milner, and by a proof of congruence of strong bisimilarity given by Bernardo Toninho for the process calculus considered in Chapter 43.

Part XX

Appendices

Appendix A

Mathematical Preliminaries

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[2] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 783–818. North-Holland, 1977.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[4] John Allen. *The Anatomy of LISP*. Computer Science Series. McGraw-Hill, 1978.

[5] Stuart Allen. A non-type-theoretic definition of martin-löf's types. In *LICS*, pages 215–221, 1987.

[6] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[7] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[8] Arvind, Rishiyur S. Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer, 1986.

[9] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991.

[10] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[11] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, *Computational Structures*. Oxford University Press, 1992.

[12] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[13] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.

[14] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ICFP*, pages 213–225, 1996.

[15] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ICFP*, pages 213–225, 1996.

[16] Stephen D. Brookes. The essence of parallel algol. *Inf. Comput.*, 179(1):118–149, 2002.

[17] Rod M. Burstall, David B. MacQueen, and Donald Sannella. Hope: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980.

[18] Luca Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.

[19] Luca Cardelli. Program fragments, linking, and modularization. In *POPL*, pages 266–277, 1997.

[20] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *POPL*, pages 151–162, 1994.

[21] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[22] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[23] R. L. Constable. *Implementing Mathematics with the Nupri Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986. $21.95.

[24] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

[25] Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[26] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 21–29, New York, NY, USA, 2009. ACM.

[27] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[28] Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing - ten years after. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 599–622. The MIT Press, 2000.

[29] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS: Theoretical Computer Science*, 103, 1992.

[30] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[31] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213. North-Holland, Amsterdam, 1969.

[32] J.-Y. Girard. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*. These d'etat, Universite Paris VII, 1972.

[33] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. (Translated by Paul Taylor and Yves Lafont).

[34] Kurt Gödel. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosphical Logic*, 9:133–142, 1980. (Translated by Wilfrid Hodges and Bruce Watson).

[35] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[36] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *POPL*, pages 309–321, 1996.

[37] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, 1999.

[38] Timothy Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[39] Robert Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1):71–84, 1992.

[40] Robert Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 51(4):201–206, 1994.

[41] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1993.

[42] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, pages 123–137, 1994.

[43] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, pages 123–137, 1994.

[44] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL*, pages 341–354, 1990.

[45] Arend Heyting. *Intuitionism: An Introduction*. North-Holland, second revised edition edition, 1966.

[46] Ralf Hinze and Johan Jeuring. Generic haskell: Practice and theory. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003.

[47] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[48] Simon L. Peyton Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003.

[49] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[50] S. C. Kleene. *Introduction to Metamathematics*. van Nostrand, 1952.

[51] Andrei Kolmogorov. On the principle of the excluded middle. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 414–437. Harvard University Press, 1967.

[52] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31:38–43, January 1988.

[53] P. J. Landin. A correspondence between algol 60 and church's lambda notation. *CACM*, 8:89–101; 158–165, 1965.

[54] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 173–184. ACM, 2007.

[55] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL*, pages 109–122, 1994.

[56] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL*, pages 109–122, 1994.

[57] Daniel R. Licata and Robert Harper. A monadic formalization of ml5. In Karl Crary and Marino Miculan, editors, *LFMTP*, volume 34 of *EPTCS*, pages 69–83, 2010.

[58] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, second edition, 1998.

[59] David B. MacQueen. Using dependent types to express modular structure. In *POPL*, pages 277–286, 1986.

[60] P. Martin Löf. On the meanings of the logical constants and the justifications of the logical laws. Unpublished Lecture Notes, 1983.

[61] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Naples, Italy, 1984.

[62] Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420, 1987.

[63] John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1965.

[64] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.

[65] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[66] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[67] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[68] John C. Mitchell. Coercion and type inference. In *POPL*, pages 175–185, 1984.

[69] John C. Mitchell. Representation independence and data abstraction. In *POPL*, pages 263–276, 1986.

[70] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[71] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[72] Chetan R. Murthy. An evaluation semantics for classical proofs. In *LICS*, pages 96–107. IEEE Computer Society, 1991.

[73] Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP*, pages 207–218. ACM, 2003.

[74] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[75] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science IV*, pages 153–175. North-Holland, 1980.

[76] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[77] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[78] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[79] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 1998.

[80] Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000.

[81] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 1993.

[82] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, 1981.

[83] Gordon D. Plotkin. Lcf considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[84] Gordon D. Plotkin. The origins of structural operational semantics. *J. of Logic and Algebraic Programming*, 60:3–15, 2004.

[85] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[86] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[87] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[88] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer, 1980.

[89] John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[90] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[91] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.

[92] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In Andrew Kennedy and Nick Benton, editors, *TLDI*, pages 89–102. ACM, 2010.

[93] Dana Scott. Lambda calculus: Some models, some philosophy. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 223–265. North Holland, Amsterdam, 1980.

[94] Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

[95] Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer, 1982.

[96] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

[97] Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3):85–97, 1985.

[98] Guy L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition edition, 1990.

[99] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722, 2006.

[100] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999.

[101] David Turner. An overview of miranda. *Bulletin of the EATCS*, 33:103–114, 1987.

[102] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295. IEEE Computer Society, 2004.

[103] Philip Wadler. Call-by-value is dual to call-by-name. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 189–201. ACM, 2003.

[104] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in clf. *Electr. Notes Theor. Comput. Sci.*, 199:67–87, 2008.

[105] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[106] Noam Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3):66–96, 2008.

Revision History

Revision	Date	Author(s)	Description
1.0	21.01.11	RH	Created
1.1	21.01.11	RH	Concurrent and Distributed Algol
1.2	03.02.11	RH	Clarified discussion of rule induction.
1.3	06.23.11	RH	Revamped notation in equational reasoning chapter; miscellaneous corrections and improvements throughout.
1.4	06.28.2011	RH	Minor corrections to inductive definitions and type safety chapters.
1.5	07.19.2011	RH	Revamped treatment of syntactic objects.
1.6	07.20.2011	RH	Reorganized Part I.
1.7	07.21.2011	RH	Moved strings to Part I from Part II.
1.8	07.22.2011	RH	Remove exercises, add end notes.
1.9	08.09.2011	RH	Revise formulation and presentation of Concurrent and Distributed Algol.
1.10	08.12.2011	RH	Added chapter on process equivalence.
1.11	08.15.2011	RH	Disequality of classes for dynamic classification.
1.12	08.18.2011	RH	Revised distributed Algol chapter.
1.13	08.22.2011	RH	Units and linking.
1.14	08.23.2011	RH	Canonization of constructors.
1.15	08.24.2011	RH	Singleton kinds, bounded quantification.
1.16	08.27.2011	RH	Modules.