Practical Foundations for Programming Languages

SECOND EDITION

Robert Harper
Carnegie Mellon University

Preface to the Second Edition

Writing the second edition to a text book incurs the same risk as building the second version of a software system. It is difficult to make substantive improvements, while avoiding the temptation to overburden and undermine the foundation on which one is building. With the hope of avoiding the second system effect, I have sought to make corrections, revisions, expansions, and deletions that improve the coherence of the development, remove some topics that distract from the main themes, add new topics that were omitted from the first edition, and include exercises for almost every chapter.

The revision removes a number of typographical errors, corrects a few material errors (especially the formulation of the parallel abstract machine and of concurrency in Algol), and improves the writing throughout. Some chapters have been deleted (general pattern matching and polarization, restricted forms of polymorphism), some have been completely rewritten (the chapter on higher kinds), some have been substantially revised (general and parametric inductive definitions, concurrent and distributed Algol), several have been reorganized (to better distinguish partial from total type theories), and a new chapter has been added (on type refinements). Titular attributions on several chapters have been removed, not to diminish credit, but to avoid confusion between the present and the original formulations of several topics. A new system of (pronounceable!) language names has been introduced throughout. The exercises generally seek to expand on the ideas in the main text, and their solutions often involve significant technical ideas that merit study. Routine exercises of the kind one might include in a homework assignment are deliberately few.

My purpose in writing this book is to establish a comprehensive framework for formulating and analyzing a broad range of ideas in programming languages. If language design and programming methodology are to advance from a trade-craft to a rigorous discipline, it is essential that we first get the definitions right. Then, and only then, can there be meaningful analysis and consolidation of ideas. My hope is that I have helped to build such a foundation.

I am grateful to Stephen Brookes, Evan Cavallo, Karl Crary, Jon Sterling, James R. Wilcox, and Todd Wilson for their help in critiquing drafts of this edition and for their suggestions for revision. I thank my department head, Frank Pfenning, for his support of my work on the completion of this edition. Thanks also to my editors, Ada Brunstein and Lauren Cowles, for their guidance and assistance. And thanks to Evan Cavallo and Andrew Shulaev for corrections to the draft.

any particular standard of merchantability, or that they will meet requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use this material in such a manner, it is at your own risk. The author and publisher disclaim all liability for direct or consequential damage resulting from its use.

Pittsburgh
July, 2015

Contents

Part I

Judgments and Rules

Chapter 1

Abstract Syntax

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

The informal concept of syntax involves 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 can be used. At this level phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

We will not concern ourselves in this book with concrete syntax, but will instead consider pieces of syntax to be finite trees augmented with a means of expressing the binding and scope of identifiers within a syntax tree. To prepare the ground for the rest of the book, we define in this chapter what is a "piece of syntax" in two stages. First, we define abstract syntax trees, or asts, which capture the hierarchical structure of a piece of syntax, while avoiding commitment to their concrete representation as a string. Second, we augment abstract syntax trees with the means of specifying the binding (declaration) and scope (range of significance) of an identifier. Such enriched forms of abstract syntax are called abstract binding trees, or abts for short.

Several functions and relations on abts are defined that give precise meaning to the informal ideas of binding and scope of identifiers. The concepts are infamously difficult to define properly, and are the mother lode of bugs for language implementors. Consequently, precise definitions are essential, but they are also fairly technical and take some getting used to. It is probably best to skim this chapter on first reading to get the main ideas, and return to it for clarification as necessary.

1.1 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. Asts are classified into a variety of *sorts* corresponding to different forms of syntax. A *variable* stands for an unspecified, or generic, piece of syntax of a specified sort. Asts can be combined by an *operator*, which has an *arity* specifying the sort of the operator and the number and sorts of its arguments. An operator of sort s and arity $s_1, \ldots, s_n$ combines $n \geq 0$ asts of sort $s_1, \ldots, s_n$, respectively, into a compound ast of sort s.

The concept of a variable is central, and therefore deserves special emphasis. A variable is an *unknown* object drawn from some domain. The unknown can become known by *substitution* of a particular object for all occurrences of a variable in a formula, thereby specializing a general formula to a particular instance. For example, in school algebra variables range over real numbers, and we may form polynomials, such as $x^2 + 2\,x + 1$, that can be specialized by substitution of, say, 7 for x to obtain $7^2 + (2 \times 7) + 1$, which can be simplified according to the laws of arithmetic to obtain 64, which is $(7 + 1)^2$.

Abstract syntax trees are classified by *sorts* that divide asts into syntactic categories. For example, familiar programming languages often have a syntactic distinction between expressions and commands; these are two sorts of abstract syntax trees. Variables in abstract syntax trees range over sorts in the sense that only asts of the specified sort of the variable can be plugged in for that variable. Thus it would make no sense to replace an expression variable by a command, nor a command variable by an expression, the two being different sorts of things. But the core idea carries over from school mathematics, namely that *a variable is an unknown, or a place-holder, whose meaning is given by substitution*.

As an example, consider a language of arithmetic expressions built from numbers, addition, and multiplication. The abstract syntax of such a language consists of a single sort Exp generated by these operators:

1. An operator `num[`n`]` of sort Exp for each $n \in \mathbb{N}$;
2. Two operators, `plus` and `times`, of sort Exp, each with two arguments of sort Exp.

The expression $2 + (3 \times x)$, which involves a variable, x, would be represented by the ast

$$\texttt{plus(num[2];times(num[3];}x\texttt{))}$$

of sort Exp, under the assumption that x is also of this sort. Because, say, `num[4]`, is an ast of sort Exp, we may plug it in for x in the above ast to obtain the ast

$$\texttt{plus(num[2];times(num[3];num[4]))},$$

which is written informally as $2 + (3 \times 4)$. We may, of course, plug in more complex asts of sort Exp for x to obtain other asts as result.

The tree structure of asts provides a very useful principle of reasoning, called *structural induction*. Suppose that we wish to prove that some property $\mathcal{P}(a)$ holds of all asts a of a given sort. To show this it is enough to consider all the ways in which a can be generated, and show that the property holds in each case under the assumption that it holds for its constituent asts (if any). So, in the case of the sort Exp just described, we must show

1. The property holds for any variable x of sort Exp: prove that $\mathcal{P}(x)$.

2. The property holds for any number, $\texttt{num}[\,n\,]$: for every $n \in \mathbb{N}$, prove that $\mathcal{P}(\texttt{num}[\,n\,])$.

3. Assuming that the property holds for a_1 and a_2, prove that it holds for $\texttt{plus}(\,a_1\,;a_2\,)$ and $\texttt{times}(\,a_1\,;a_2\,)$: if $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}(\texttt{plus}(\,a_1\,;a_2\,))$ and $\mathcal{P}(\texttt{times}(\,a_1\,;a_2\,))$.

Because these cases exhaust all possibilities for the formation of a, we are assured that $\mathcal{P}(a)$ holds for any ast a of sort Exp.

It is common to apply the principle of structural induction in a form that takes account of the interpretation of variables as place-holders for asts of the appropriate sort. Informally, it is often useful to prove a property of an ast involving variables in a form that is conditional on the property holding for the variables. Doing so anticipates that the variables will be replaced with asts that ought to have the property assumed for them, so that the result of the replacement will have the property as well. This amounts to applying the principle of structural induction to properties $\mathcal{P}(a)$ of the form "if a involves variables $x_1, \ldots, x_k$, and $\mathcal{Q}$ holds of each x_i, then $\mathcal{Q}$ holds of a", so that a proof of $\mathcal{P}(a)$ for all asts a by structural induction is just a proof that $\mathcal{Q}(a)$ holds for all asts a under the assumption that $\mathcal{Q}$ holds for its variables. When there are no variables, there are no assumptions, and the proof of $\mathcal{P}$ is a proof that $\mathcal{Q}$ holds for all *closed* asts. On the other hand if x is a variable in a, and we replace it by an ast b for which $\mathcal{Q}$ holds, then $\mathcal{Q}$ will hold for the result of replacing x by b in a.

For the sake of precision, we now give precise definitions of these concepts. Let $\mathcal{S}$ be a finite set of sorts. For a given set $\mathcal{S}$ of sorts, an *arity* has the form $(s_1, \ldots, s_n)s$, which specifies the sort $s \in \mathcal{S}$ of an operator taking $n \geq 0$ arguments, each of sort $s_i \in \mathcal{S}$. Let $\mathcal{O} = \{\,\mathcal{O}_\alpha\,\}$ be an arity-indexed family of disjoint sets of *operators* $\mathcal{O}_\alpha$ of arity α. If o is an operator of arity $(s_1, \ldots, s_n)s$, we say that o has sort s and has n arguments of sorts $s_1, \ldots, s_n$.

Fix a set $\mathcal{S}$ of sorts and an arity-indexed family $\mathcal{O}$ of sets of operators of each arity. Let $\mathcal{X} = \{\,\mathcal{X}_s\,\}_{s \in \mathcal{S}}$ be a sort-indexed family of disjoint finite sets $\mathcal{X}_s$ of *variables* x of sort s. When $\mathcal{X}$ is clear from context, we say that a variable x is of sort s if $x \in \mathcal{X}_s$, and we say that x is *fresh for* $\mathcal{X}$, or just *fresh* when $\mathcal{X}$ is understood, if $x \notin \mathcal{X}_s$ for any sort s. If x is fresh for $\mathcal{X}$ and s is a sort, then $\mathcal{X}, x$ is the family of sets of variables obtained by adding x to $\mathcal{X}_s$. The notation is ambiguous in that the sort s is not explicitly stated, but determined from context.

The family $\mathcal{A}[\mathcal{X}] = \{\,\mathcal{A}[\mathcal{X}]_s\,\}_{s \in \mathcal{S}}$ of *abstract syntax trees*, or *asts*, of sort s is the smallest family satisfying the following conditions:

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 asts: if o is an operator of arity $(s_1, \ldots, s_n)s$, 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* can 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 has arity $(s_1, \ldots, s_n)s$ and $\mathcal{P}_{s_1}(a_1)$ and $\ldots$ and $\mathcal{P}_{s_n}(a_n)$, then $\mathcal{P}_s(o(\,a_1\,;\ldots;a_n\,))$.

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

Variables are given meaning by *substitution*. If $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $\{b/x\}a \in \mathcal{A}[\mathcal{X}]_{s'}$ is 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. $\{b/x\}x = b$ 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 asts is well-defined.

Theorem 1.1. *If $a \in \mathcal{A}[\mathcal{X}, x]$, then for every $b \in \mathcal{A}[\mathcal{X}]$ 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. □

1.2 Abstract Binding Trees

Abstract binding trees, or *abts*, enrich asts 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 can be used, either as a place-holder (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 can 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 identifiers is immaterial, so long as we 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, $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$ 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 $y + y$ in the second case refers to the inner declaration, not the outer one as before.

The concept of an ast can be enriched to account for binding and scope of a variable. These enriched asts are called *abstract binding trees*, or *abts* for short. Abts generalize asts by allowing an operator to bind any finite number (possibly zero) of variables in each argument. 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, operators are assigned *generalized arities* of the form $(v_1, \ldots, v_n)s$, which specifies operators of sort s with n arguments of *valence* $v_1, \ldots, v_n$. In general a valence v has the form $s_1, \ldots, s_k.s$, which specifies the sort of an argument as well as the number and sorts of the variables bound within it. We say that a sequence $\vec{x}$ of variables is of sort $\vec{s}$ to mean that the two sequences have the same length k and that the variable x_i is of sort s_i for each $1 \leq i \leq k$.

Thus, to specify that the operator `let` has arity $(\mathsf{Exp}, \mathsf{Exp}.\mathsf{Exp})\mathsf{Exp}$ indicates that it is of sort Exp whose first argument is of sort Exp and binds no variables, and whose second argument is also of sort Exp, within which is bound one variable of sort Exp. The informal expression $\texttt{let}\ x\ \texttt{be}\ 2 + 2\ \texttt{in}\ x \times x$ may then be written as the abt

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

in which the operator `let` has two arguments, the first of which is an expression, and the second of which is an abstractor that binds one expression variable.

Fix a set $\mathcal{S}$ of sorts, and a family $\mathcal{O}$ of disjoint sets of operators indexed by their generalized arities. For a given family of disjoint sets of variables $\mathcal{X}$, the family of *abstract binding trees*, or *abts* $\mathcal{B}[\mathcal{X}]$ is defined similarly to $\mathcal{A}[\mathcal{X}]$, except that $\mathcal{X}$ is not fixed throughout the definition, but rather changes as we enter the scopes of abstractors.

This simple idea is surprisingly hard to make precise. A first attempt at the definition is as the least family of sets closed under the following conditions:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.

2. For each operator o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if $a_1 \in \mathcal{B}[\mathcal{X}, \vec{x}_1]_{s_1}$, ..., and $a_n \in \mathcal{B}[\mathcal{X}, \vec{x}_n]_{s_n}$, 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 renaming 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 adds x to $\mathcal{X}$, which implies that the second cannot also add x to $\mathcal{X}, x$, because it is not fresh for $\mathcal{X}, x$. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names, which is achieved using *fresh renamings*, which are bijections between sequences of variables. Specifically, a fresh renaming (relative to $\mathcal{X}$) of a finite sequence of variables $\vec{x}$ is a bijection $\rho : \vec{x} \leftrightarrow \vec{x}'$ between $\vec{x}$ and $\vec{x}'$, where $\vec{x}'$ is fresh for $\mathcal{X}$. We write $\widehat{\rho}(a)$ for the result of replacing each occurrence of x_i in a by $\rho(x_i)$, its fresh counterpart.

This is achieved by altering the second clause of the definition of abts using fresh renamings as follows:

> For each operator o of arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if for each $1 \leq i \leq n$ and each fresh renaming $\rho_i : \vec{x}_i \leftrightarrow \vec{x}_i'$, we have $\widehat{\rho}_i(a_i) \in \mathcal{B}[\mathcal{X}, \vec{x}_i']$, then $o(\ \vec{x}_1\ .\ a_1\ ;\ \ldots\ ;\ \vec{x}_n\ .\ a_n\ ) \in \mathcal{B}[\mathcal{X}]_s$.

The renaming, $\widehat{\rho}_i(a_i)$, of each a_i ensures that collisions cannot occur, and that the abt is valid for almost all renamings of any bound variables that occur within it.

The principle of structural induction extends to abts, and is called *structural induction modulo fresh renaming*. It states that to show that $\mathcal{P}[\mathcal{X}](a)$ 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 arity $(\vec{s}_1.s_1, \ldots, \vec{s}_n.s_n)s$, if for each $1 \leq i \leq n$, $\mathcal{P}[\mathcal{X}, \vec{x}_i']_{s_i}(\widehat{\rho}_i(a_i))$ holds for every $\rho_i : \vec{x}_i \leftrightarrow \vec{x}_i'$ with $\vec{x}_i' \notin \mathcal{X}$, then $\mathcal{P}[\mathcal{X}]_s(o(\ \vec{x}_1\ .\ a_1\ ;\ \ldots\ ;\ \vec{x}_n\ .\ a_n\ ))$.

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 judgment $x \in a$, 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 $\rho : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \widehat{\rho}(a_i)$.

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.

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons), means that a and b are identical up to the choice of bound variable names. The α-equivalence relation is 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 $1 \leq i \leq n$, $\widehat{\rho_i}(a_i) =_\alpha \widehat{\rho'_i}(a'_i)$ for all fresh renamings $\rho_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\rho'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

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 *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.

The definition of $\{b/x\}a$ is quite delicate, and merits careful consideration.

One trouble spot for substitution is to notice that if x is bound by an abstractor within a, then x does not occur free within the abstractor, and hence is unchanged by substitution. For example, $\{b/x\}\texttt{let}(a_1 ; x . a_2) = \texttt{let}(\{b/x\}a_1 ; x . a_2)$, there being no free occurrences of x in $x . a_2$. Another trouble spot is the *capture* of a free variable of b during substitution. For example, if $y \in b$, and $x \neq y$, then $\{b/x\}\texttt{let}(a_1 ; y . a_2)$ is undefined, rather than being $\texttt{let}(\{b/x\}a_1 ; y . \{b/x\}a_2)$, as one might at first suspect. For example, provided that $x \neq y$, $\{y/x\}\texttt{let}(\texttt{num}[0] ; y . \texttt{plus}(x ; y))$ is undefined, not $\texttt{let}(\texttt{num}[0] ; y . \texttt{plus}(y ; y))$, which confuses two different variables named y.

Although capture avoidance is an essential characteristic of substitution, it is, in a sense, merely a technical nuisance. If the names of bound variables have no significance, then capture can always be avoided by first renaming the bound variables in a to avoid any free variables in b. In the foregoing example if we rename the bound variable y to y' to obtain $a' \triangleq \texttt{let}(\texttt{num}[0] ; y' . \texttt{plus}(x ; y'))$, then $\{b/x\}a'$ *is* defined, and is equal to $\texttt{let}(\texttt{num}[0] ; y' . \texttt{plus}(b ; y'))$. The price for avoiding capture in this way is that substitution is only determined up to α-equivalence, and so we may no longer think of substitution as a function, but only as a proper relation.

To restore the functional character of substitution, it is sufficient to adopt the *identification convention*, which is stated as follows:

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

That is, α-equivalent abts are regarded as identical. Substitution can be extended to α-equivalence classes of abts to avoid capture by choosing representatives of the equivalence classes of b and a in such a way that substitution is defined, then forming the equivalence class of the result. Any two choices of representatives for which substitution is defined gives α-equivalent results, so that substitution becomes a well-defined total function. *We will adopt the identification convention for abts throughout this book.*

It will often be necessary to consider languages whose abstract syntax cannot be specified by a fixed set of operators, but rather requires that the available operators be sensitive to the context in which they occur. For our purposes it will suffice to consider a set of *symbolic parameters*, or *symbols*, that index families of operators so that as the set of symbols varies, so does the set of

operators. An *indexed operator* o is a family of operators indexed by symbols u, so that $o[u]$ is an operator when u is an available symbol. If $\mathcal{U}$ is a finite set of symbols, then $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is the sort-indexed family of abts that are generated by operators and variables as before, admitting all indexed operator instances by symbols $u \in \mathcal{U}$. Whereas a variable is a place-holder that stands for an unknown abt of its sort, a symbol *does not stand for anything*, and is not, itself, an abt. The only significance of symbol is whether it is the same as or differs from another symbol; the operator instances $o[u]$ and $o[u']$ are the same exactly when u is u', and are the same symbol.

The set of symbols is extended by introducing a *new*, or *fresh*, symbol within a scope using the abstractor $u\,.\,a$, which binds the symbol u within the abt a. An abstracted symbol is "new" in the same sense as for an abstracted variable: the name of the bound symbol can be varied at will provided that no conflicts arise. This renaming property ensures that an abstracted symbol is distinct from all others in scope. The only difference between symbols and variables is that the only operation on symbols is renaming; there is no notion of substitution for a symbol.

Finally, a word about notation: to help improve the readability we often "group" and "stage" the arguments to an operator, using round brackets and braces to show grouping, and generally regarding stages to progress from right to left. All arguments in a group are considered to occur at the same stage, though their order is significant, and successive groups are considered to occur in sequential stages. Staging and grouping is often a helpful mnemonic device, but has no fundamental significance. For example, the abt $o[a_1;a_2](a_3;x\,.\,a_4)$ is the same as the abt $o(a_1;a_2;a_3;x\,.\,a_4)$, as would be any other order-preserving grouping or staging of its arguments.

1.3 Notes

The concept of abstract syntax has its origins in the pioneering work of Church, Turing, and Gödel, who first considered 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 (1952), has a thorough account of such representations. The Lisp language (McCarthy, 1965; Allen, 1978) introduced a much more practical and direct representation of syntax as *symbolic expressions*. These ideas were developed further in the language ML (Gordon et al., 1979), which featured a type system capable of expressing abstract syntax trees. The AUTOMATH project (Nederpelt et al., 1994) introduced the idea of using Church's λ notation (Church, 1941) to account for the binding and scope of variables. These ideas were developed further in LF (Harper et al., 1993).

The concept of abstract binding trees presented here was inspired by the system of notation developed in the NuPRL Project, which is described in Constable (1986) and from Martin-Löf's system of arities, which is described in Nordstrom et al. (1990). Their enrichment with symbol binders is influenced by Pitts and Stark (1993).

Exercises

1.1. Prove by structural induction on abstract syntax trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

1.2. Prove by structural induction modulo renaming on abstract binding trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{B}[\mathcal{X}] \subseteq \mathcal{B}[\mathcal{Y}]$.

1.3. Show that if $a =_\alpha a'$ and $b =_\alpha b'$ and both $\{b/x\}a$ and $\{b'/x\}a'$ are defined, then $\{b/x\}a =_\alpha \{b'/x\}a'$.

1.4. Bound variables can be seen as the formal analogs of pronouns in natural languages. The binding occurrence of a variable at an abstractor fixes a "fresh" pronoun for use within its body that refers unambiguously to that variable (in contrast to English, in which the referent of a pronoun can often be ambiguous). This observation suggests an alternative representation of abts, called *abstract binding graphs*, or *abg's* for short, as directed graphs constructed as follows:

(a) Free variables are atomic nodes with no outgoing edges.

(b) Operators with n arguments are n-ary nodes, with one outgoing edge directed at each of their children.

(c) Abstractors are nodes with one edge directed to the scope of the abstracted variable.

(d) Bound variables are back edges directed at the abstractor that introduced it.

Notice that asts, thought of as abts with no abstractors, are *acyclic* directed graphs (more precisely, variadic trees), whereas general abts can be *cyclic*. Draw a few examples of abg's corresponding to the example abts given in this chapter. Give a precise definition of the sort-indexed family $\mathcal{G}[\mathcal{X}]$ of abstract binding graphs. What representation would you use for bound variables (back edges)?

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 *judgments*, or *assertions*, of a variety of forms. Judgments are statements about one or more abstract binding trees of some sort. The rules specify necessary and sufficient conditions for the validity of a judgment, and hence fully determine its meaning.

2.1 Judgments

We start with the notion of a *judgment*, or *assertion*, about an abstract binding tree. We shall make use of many forms of judgment, including examples such as these:

n nat	n is a natural number
$n_1 + n_2 = n$	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 judgment states that one or more abstract binding trees have a property or stand in some relation to one another. The property or relation itself is called a *judgment form*, and the judgment that an object or objects have that property or stand in that relation is said to be an *instance* of that judgment form. A judgment form is also called a *predicate*, and the objects constituting an instance are its *subjects*. We write a J or J a, for the judgment asserting that J holds of the abt a. Correspondingly, we sometimes notate the judgment form J by $-$ J, or J $-$, using a dash to indicate the absence of an argument to J. When it is not important to stress the subject of the judgment, we write J to stand for an unspecified judgment, that is, an instance of some judgment form. For particular judgment forms, we freely use prefix, infix, or mix-fix notation, as illustrated by the above examples, in order to enhance readability.

2.2 Inference Rules

An *inductive definition* of a judgment 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 judgments of the form being defined. The judgments above the horizontal line are called the *premises* of the rule, and the judgment 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 can 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 form an inductive definition of the judgment form $-$ 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 judgment form $-$ 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 judgment form a is b expressing the equality of two abts a and b such that a nat and b nat is inductively defined by the following rules:

$$\frac{}{\texttt{zero is zero}} \tag{2.4a}$$

$$\frac{a \texttt{ is } b}{\texttt{succ(}\, a \,\texttt{) is succ(}\, b \,\texttt{)}} \tag{2.4b}$$

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* judgment form 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 judgment: J holds *if* there is a way to obtain it using the given rules. To be the *strongest* judgment form 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 judgment holds, it is enough to exhibit a *derivation* of it. A derivation of a judgment is a finite composition of rules, starting with axioms and ending with that judgment. It can 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 judgment J.

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

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

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

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

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

For example, this is a derivation of $\mathtt{succ}(\,\mathtt{succ}(\,\mathtt{succ}(\,\mathtt{zero}\,)\,)\,)\ \mathtt{nat}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{zero\ nat}}}{\mathtt{succ}(\,\mathtt{zero}\,)\ \mathtt{nat}}}{\mathtt{succ}(\,\mathtt{succ}(\,\mathtt{zero}\,)\,)\ \mathtt{nat}}}{\mathtt{succ}(\,\mathtt{succ}(\,\mathtt{succ}(\,\mathtt{zero}\,)\,)\,)\ \mathtt{nat}}\,. \tag{2.5}$$

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{empty tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(empty ; empty) tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(node(empty ; empty) ; empty) tree}}\,. \tag{2.6}$$

To show that an inductively defined judgment 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 judgments, 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 judgment occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgment, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgment is not derivable. We may go on and on adding more judgments 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 judgment 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, judgments whose derivations are to be sought. Initially, this set consists solely of the judgment we wish to derive. At each stage, we remove a judgment from the queue, and consider all rules whose conclusion is that judgment. 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 can be discarded. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgment, 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 judgments to the goal set, never reaching a point at which all goals have been satisfied.

2.4 Rule Induction

Because an inductive definition specifies the *strongest* judgment form 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 $a\ \mathcal{P}$ holds whenever $a\ \mathsf{J}$ is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining the judgment form J. More precisely, the property $\mathcal{P}$ respects the rule

$$\frac{a_1\ \mathsf{J} \quad \ldots \quad a_k\ \mathsf{J}}{a\ \mathsf{J}}$$

if $\mathcal{P}(a)$ holds whenever $\mathcal{P}(a_1), \ldots, \mathcal{P}(a_k)$ do. The assumptions $\mathcal{P}(a_1), \ldots, \mathcal{P}(a_k)$ are called the *inductive hypotheses*, and $\mathcal{P}(a)$ is called the *inductive conclusion* of the inference.

The principle of rule induction is simply the expression of the definition of an inductively defined judgment form as the *strongest* judgment form closed under the rules comprising the definition. Thus, the judgment 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 judgment; the latter means that we may reason about an inductively defined judgment form by rule induction.

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

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

The sufficiency of these conditions is the familiar principle of *mathematical induction*.

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

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

The sufficiency of these conditions is called the principle of *tree induction*.

We may also show by rule induction that the predecessor of a natural number is also a natural number. Although 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. Define $\mathcal{P}(a)$ to mean that if $a = \mathtt{succ}(\,b\,)$, then b nat. It suffices to show that $\mathcal{P}$ is closed under rules (2.2).

Rule (2.2a) Vacuous, because $\mathtt{zero}$ is not of the form $\mathtt{succ}(\,-\,)$.

Rule (2.2b) The premise of the rule ensures that b nat when $a = \mathtt{succ}(\,b\,)$.

□

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

Lemma 2.2. *If* a nat, *then* a is a.

Proof. By rule induction on rules (2.2):

Rule (2.2a) Applying rule (2.4a) we obtain $\mathtt{zero}$ is $\mathtt{zero}$.

Rule (2.2b) Assume that a is a. It follows that $\mathtt{succ}(\,a\,)$ is $\mathtt{succ}(\,a\,)$ by an application of rule (2.4b).

□

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

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

Proof. Similar to the proof of Lemma 2.1. □

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

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

may be instances of either a previously defined judgment form, or the judgment form being defined. For example, the following rules define the judgment form $-$ list, which states that a is a list of natural numbers:

$$\frac{}{\mathtt{nil}\ \mathsf{list}} \tag{2.7a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\mathtt{cons}(\,a\,;b\,)\ \mathsf{list}} \tag{2.7b}$$

The first premise of rule (2.7b) is an instance of the judgment form a nat, which was defined previously, whereas the premise b list is an instance of the judgment form being defined by these rules.

Frequently two or more judgments 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 judgment forms, any of which may appear as the premise of any rule. Because the rules defining each judgment form may involve any of the others, none of the judgment forms can be taken to be defined prior to the others. Instead we must understand that all of the judgment forms are being defined at once by the entire collection of rules. The judgment forms defined by these rules are, as before, the strongest judgment 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 judgment forms simultaneously.

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

$$\frac{}{\mathtt{zero}\ \mathsf{even}} \tag{2.8a}$$

$$\frac{b\ \mathsf{odd}}{\mathtt{succ}(\,b\,)\ \mathsf{even}} \tag{2.8b}$$

$$\frac{a\ \mathsf{even}}{\mathtt{succ}(\,a\,)\ \mathsf{odd}} \tag{2.8c}$$

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

1. $\mathcal{P}(\mathtt{zero})$;
2. if $\mathcal{Q}(b)$, then $\mathcal{P}(\mathtt{succ}(\,b\,))$;
3. if $\mathcal{P}(a)$, then $\mathcal{Q}(\mathtt{succ}(\,a\,))$.

As an example, we may use simultaneous rule induction to prove that (1) if a even, then either a is `zero` or a is $\mathtt{succ}(\,b\,)$ with b odd, and (2) if a odd, then a is $\mathtt{succ}(\,b\,)$ with b even. We define $\mathcal{P}(a)$ to hold iff a is `zero` or a is $\mathtt{succ}(\,b\,)$ for some b with b odd, and define $\mathcal{Q}(b)$ to hold iff b is $\mathtt{succ}(\,a\,)$ for some a with a even. The desired result follows by rule induction, because we can prove the following facts:

1. $\mathcal{P}(\mathtt{zero})$, which holds because `zero` is `zero`.
2. If $\mathcal{Q}(b)$, then $\mathtt{succ}(\,b\,)$ is $\mathtt{succ}(\,b'\,)$ for some b' with $\mathcal{Q}(b')$. Take b' to be b and apply the inductive assumption.
3. If $\mathcal{P}(a)$, then $\mathtt{succ}(\,a\,)$ is $\mathtt{succ}(\,a'\,)$ for some a' with $\mathcal{P}(a')$. Take a' to be a and apply the inductive assumption.

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:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero}\,;b\,;b)} \tag{2.9a}$$

$$\frac{\mathsf{sum}(a\,;b\,;c)}{\mathsf{sum}(\mathtt{succ}(\,a\,)\,;b\,;\mathtt{succ}(\,c\,))} \tag{2.9b}$$

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 is c'.

For existence, let $\mathcal{P}(a)$ 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 $\mathcal{P}(a)$ by rule induction on rules (2.2). We have two cases to consider:

Rule (2.2a) We are to show $\mathcal{P}(\mathtt{zero})$. Assuming b nat and taking c to be b, we obtain $\mathsf{sum}(\mathtt{zero}\,;\, b\,;\, c)$ by rule (2.9a).

Rule (2.2b) Assuming $\mathcal{P}(a)$, we are to show $\mathcal{P}(\mathtt{succ}(\,a\,))$. 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' to be $\mathtt{succ}(\,c\,)$, and applying rule (2.9b), 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 is c_2 by rule induction based on rules (2.9).

Rule (2.9a) We have a is $\mathtt{zero}$ and c_1 is 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 we obtain b is b.

Rule (2.9b) We have that a is $\mathtt{succ}(\,a'\,)$ and c_1 is $\mathtt{succ}(\,c_1'\,)$, 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 is $\mathtt{succ}(\,c_2'\,)$ where $\mathsf{sum}(a'\,;\,b\,;\,c_2')$. By the outer inductive hypothesis c_1' is c_2' and so c_1 is c_2.

□

2.7 Notes

Aczel (1977) provides a thorough account of the theory of inductive definitions on which the present account is based. A significant difference is that we consider inductive definitions of judgments over abts as defined in Chapter 1, rather than with natural numbers. The emphasis on judgments is inspired by Martin-Löf's logic of judgments (Martin-Löf, 1983, 1987).

Exercises

2.1. Give an inductive definition of the judgment $\mathsf{max}(m\,;\,n\,;\,p)$, where m nat, n nat, and p nat, with the meaning that p is the larger of m and n. Prove that every m and n are related to a unique p by this judgment.

2.2. Consider the following rules, which define the judgment $\mathsf{hgt}(t\,;n)$ stating that the binary tree t has *height* n.

$$\frac{}{\mathsf{hgt}(\mathtt{empty}\,;\mathtt{zero})} \tag{2.10a}$$

$$\frac{\mathsf{hgt}(t_1\,;n_1) \quad \mathsf{hgt}(t_2\,;n_2) \quad \mathsf{max}(n_1\,;n_2\,;n)}{\mathsf{hgt}(\mathtt{node}(\,t_1\,;t_2\,)\,;\mathtt{succ}(\,n\,))} \tag{2.10b}$$

Prove that the judgment $\mathtt{hgt}$ defines a function from trees to natural numbers.

2.3. Given an inductive definition of *ordered variadic trees* whose nodes have a finite, but variable, number of children with a specified left-to-right ordering among them. Your solution should consist of a simultaneous definition of two judgments, t tree, stating that t is a variadic tree, and f forest, stating that f is a "forest" (finite sequence) of variadic trees.

2.4. Give an inductive definition of the height of a variadic tree of the kind defined in Exercise 2.3. Your definition should make use of an auxiliary judgment defining the height of a forest of variadic trees, and will be defined simultaneously with the height of a variadic tree. Show that the two judgments so defined each define a function.

2.5. Give an inductive definition of the *binary natural numbers*, which are either zero, twice a binary number, or one more than twice a binary number. The size of such a representation is logarithmic, rather than linear, in the natural number it represents.

2.6. Give an inductive definition of addition of binary natural numbers as defined in Exercise 2.5. *Hint*: Proceed by analyzing both arguments to the addition, and make use of an auxiliary function to compute the successor of a binary number. *Hint*: Alternatively, define both the sum and the sum-plus-one of two binary numbers mutually recursively.

Chapter 3

Hypothetical and General Judgments

A *hypothetical judgment* expresses an entailment between one or more hypotheses and a conclusion. We will consider two notions of entailment, called *derivability* and *admissibility*. Both express a form of entailment, but they differ in that derivability is stable under extension with new rules, admissibility is not. A *general judgment* expresses the universality, or genericity, of a judgment. There are two forms of general judgment, the *generic* and the *parametric*. The generic judgment expresses generality with respect to all substitution instances for variables in a judgment. The parametric judgment expresses generality with respect to renamings of symbols.

3.1 Hypothetical Judgments

The hypothetical judgment codifies the rules for expressing the validity of a conclusion conditional on the validity of one or more hypotheses. There are two forms of hypothetical judgment that differ according to the sense in which the conclusion is conditional on the hypotheses. One is stable under extension with more rules, and the other is not.

3.1.1 Derivability

For a given set $\mathcal{R}$ of rules, we define the *derivability* judgment, written $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$, where each J_i and K are basic judgments, to mean that we may derive K from the *expansion* $\mathcal{R} \cup \{ J_1, \ldots, J_k \}$ of the rules $\mathcal{R}$ with the axioms

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

We treat the *hypotheses*, or *antecedents*, of the judgment, $J_1, \ldots, J_k$ as "temporary axioms", and derive the *conclusion*, or *consequent*, by composing rules in $\mathcal{R}$. Thus, evidence for a hypothetical judgment consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$.

We use capital Greek letters, usually Γ or Δ, to stand for a finite set of basic judgments, and write $\mathcal{R} \cup \Gamma$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgment in Γ. The

judgment $\Gamma \vdash_{\mathcal{R}} K$ means that K is derivable from rules $\mathcal{R} \cup \Gamma$, and the judgment $\vdash_{\mathcal{R}} \Gamma$ 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 judgment

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

relative to rules (2.2). This judgment is valid for *any* choice of object a, as shown by the derivation

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

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

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

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 $\mathcal{R} \cup \Gamma$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}') \cup \Gamma$, because 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 judgment 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 un-exercised 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 judgment stating that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. That is, the conclusion J is derivable from rules $\mathcal{R}$ when 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 judgment is *vacuously* true. An equivalent way to define the judgment $J_1, \ldots, J_n \models_{\mathcal{R}} J$ 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 $\mathcal{R}$. Given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may build a derivation of J using the rules in $\mathcal{R}$.

For example, the admissibility judgment

$$\mathtt{succ}(\, a \,)\ \mathsf{even} \models_{(2.8)} a\ \mathsf{odd} \tag{3.6}$$

is valid, because any derivation of $\mathtt{succ}(\, a \,)\ \mathsf{even}$ from rules (2.8) must contain a sub-derivation of $a\ \mathsf{odd}$ from the same rules, which justifies the conclusion. This fact can be proved by induction on rules (2.8). That judgment (3.6) is valid may also be expressed by saying that the rule

$$\frac{\mathtt{succ}(\, a \,)\ \mathsf{even}}{a\ \mathsf{odd}} \tag{3.7}$$

is *admissible* relative to rules (2.8).

In contrast to derivability the admissibility judgment is *not* stable under extension to the rules. For example, if we enrich rules (2.8) with the axiom

$$\frac{}{\mathtt{succ}(\, \mathtt{zero} \,)\ \mathsf{even}}\ , \tag{3.8}$$

then rule (3.6) is *inadmissible*, because there is no composition of rules deriving $\mathtt{zero}\ \mathsf{odd}$. 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, note that

$$\mathtt{succ}(\, \mathtt{zero} \,)\ \mathsf{even} \not\models_{(2.8)} \mathtt{zero}\ \mathsf{odd},$$

because there is no derivation of the right-hand side when the left-hand side is added as an axiom to rules (2.8). Yet the corresponding admissibility judgment

$$\mathtt{succ}(\, \mathtt{zero} \,)\ \mathsf{even} \models_{(2.8)} \mathtt{zero}\ \mathsf{odd}$$

is valid, because the hypothesis is false: there is no derivation of $\texttt{succ}(\,\texttt{zero}\,)\ \texttt{even}$ from rules (2.8). Even so, the derivability

$$\texttt{succ}(\,\texttt{zero}\,)\ \texttt{even} \vdash_{(2.8)} \texttt{succ}(\,\texttt{succ}(\,\texttt{zero}\,)\,)\ \texttt{odd}$$

is valid, because we may derive the right-hand side from the left-hand side by composing rules (2.8).

Evidence for admissibility can 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 judgment enjoys the same structural properties as derivability, and hence is a form of hypothetical judgment:

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 judgments in Γ are derivable from these rules, then J must also be derivable assuming that Γ and 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 judgments in Γ are derivable, so is K, by assumption, and hence so are the judgments in Γ, K, and hence so is J.

Theorem 3.3. *The admissibility judgment $\Gamma \models_{\mathcal{R}} J$ 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 $\mathcal{R}$. It follows by rule induction on $\mathcal{R}, r$ that every derivation from the expanded set of rules $\mathcal{R}, r$ can be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to prove a property of the judgments derivable from $\mathcal{R}, r$, when r is admissible with respect to $\mathcal{R}$, it suffices show that the property is closed under rules $\mathcal{R}$ alone, because its admissibility states that the consequences of rule r are implicit in those of rules $\mathcal{R}$.

3.2 Hypothetical Inductive Definitions

It is useful to enrich the concept of an inductive definition to allow rules with derivability judgments as premises and conclusions. Doing so lets us 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 set 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 when each J_i is a derivable consequence of Γ, augmented with the 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.

We require that all rules in a hypothetical inductive definition be *uniform* in the sense that they are applicable in *all* global contexts. Uniformity ensures that a rule can be presented in *implicit*, or *local form*,

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

in which the global context has been suppressed with the understanding that the rule applies for any choice of global hypotheses.

A hypothetical inductive definition is to be regarded as an ordinary inductive definition of a *formal derivability judgment* $\Gamma \vdash J$ consisting of a finite set of basic judgments Γ and a basic judgment J. A set of hypothetical rules $\mathcal{R}$ defines the strongest formal derivability judgment that is *structural* and *closed* under uniform rules $\mathcal{R}$. Structurality means that the formal derivability judgment 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 judgment. We write $\Gamma \vdash_{\mathcal{R}} J$ to mean that $\Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

The principle of *hypothetical rule induction* is just the principle of rule induction applied to the formal hypothetical judgment. So to show that $\mathcal{P}(\Gamma \vdash J)$ when $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules of $\mathcal{R}$ and under the structural rules.[1] Thus, for each rule of the form (3.9), whether structural or in $\mathcal{R}$, we must show that

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

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

In practice we usually dispense with the structural rules by the method described in Section 3.1.2. 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 rules of a hypothetical inductive definition are uniform, the structural rules (3.11b) and (3.11c) are clearly admissible. Usually, rule (3.11a) must be postulated explicitly as a rule, rather than shown to be admissible on the basis of the other rules.

[1] Writing $\mathcal{P}(\Gamma \vdash J)$ is a mild abuse of notation in which the turnstile is used to separate the two arguments to $\mathcal{P}$ for the sake of readability.

3.3 General Judgments

General judgments codify the rules for handling variables in a judgment. As in mathematics in general, a variable is treated as an *unknown* ranging over a specified set of objects. A *generic* judgment states that a judgment holds for any choice of objects replacing designated variables in the judgment. Another form of general judgment codifies the handling of symbolic parameters. A *parametric* judgment expresses generality over any choice of fresh renamings of designated symbols of a judgment. To keep track of the active variables and symbols in a derivation, we write $\Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ to say that J is derivable from Γ according to rules $\mathcal{R}$, with objects consisting of abts over symbols $\mathcal{U}$ and variables $\mathcal{X}$.

The concept of uniformity of a rule must be extended to require that rules be *closed under renaming and substitution* for variables and *closed under renaming* for parameters. More precisely, if $\mathcal{R}$ is a set of rules containing a free variable x of sort s then it must also contain all possible substitution instances of abts a of sort s for x, including those that contain other free variables. Similarly, if $\mathcal{R}$ contains rules with a parameter u, then it must contain all instances of that rule obtained by renaming u of a sort to any u' of the same sort. Uniformity rules out stating a rule for a variable, without also stating it all instances of that variable. It also rules out stating a rule for a parameter without stating it for all possible renamings of that parameter.

Generic derivability judgment is defined by

$$\mathcal{Y} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \Gamma \vdash^{\mathcal{X}\,\mathcal{Y}}_{\mathcal{R}} J,$$

where $\mathcal{Y} \cap \mathcal{X} = \emptyset$. Evidence for generic derivability consists of a *generic derivation* ∇ involving the variables $\mathcal{X}\,\mathcal{Y}$. So long as the rules are uniform, the choice of $\mathcal{Y}$ does not matter, in a sense to be explained shortly.

For example, the generic derivation ∇,

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

is evidence for the judgment

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

provided $x \notin \mathcal{X}$. Any other choice of x would work just as well, as long as all rules are uniform.

The generic derivability judgment enjoys the following *structural properties* governing the behavior of variables, provided that $\mathcal{R}$ is uniform.

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

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

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

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 judgment. It is left implicit in the principle of substitution that the substituting abt is of the same sort as the substituted variable.

Parametric derivability is defined analogously to generic derivability, albeit by generalizing over symbols, rather than variables. Parametric derivability is defined by

$$\mathcal{V} \parallel \mathcal{Y} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J \quad \text{iff} \quad \mathcal{Y} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U}\,\mathcal{V};\mathcal{X}} J,$$

where $\mathcal{V} \cap \mathcal{U} = \emptyset$. Evidence for parametric derivability consists of a derivation ∇ involving the symbols $\mathcal{V}$. Uniformity of $\mathcal{R}$ ensures that any choice of parameter names is as good as any other; derivability is stable under renaming.

3.4 Generic Inductive Definitions

A *generic inductive definition* admits generic hypothetical judgments 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

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

The variables $\mathcal{Y}$ are the *global variables* of the inference, and, for each $1 \leq i \leq n$, the variables $\mathcal{Y}_i$ 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 can be given in *implicit form*,

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

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

When specialized to a set of generic rules, the principle of rule induction states that to show $\mathcal{P}(\mathcal{Y} \mid \Gamma \vdash J)$ when $\mathcal{Y} \mid \Gamma \vdash_{\mathcal{R}} J$, 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

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

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

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

$$\frac{}{\mathcal{Y} \mid \Gamma, J \vdash J} \tag{3.14a}$$

$$\frac{\mathcal{Y} \mid \Gamma \vdash J}{\mathcal{Y} \mid \Gamma, J' \vdash J} \tag{3.14b}$$

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

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

$$\frac{\mathcal{Y} \mid \Gamma \vdash J \quad \mathcal{Y} \mid \Gamma, J \vdash J'}{\mathcal{Y} \mid \Gamma \vdash J'} \tag{3.14e}$$

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

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, because we may assimilate the added variable x to the global variables, and the added hypothesis J, to the global hypotheses. The admissibility of rule (3.14d) is ensured by the identification convention for the formal generic judgment. Rule (3.14f) must be verified explicitly for each inductive definition.

The concept of a generic inductive definition extends to parametric judgments as well. Briefly, rules are defined on formal parametric judgments of the form $\mathcal{V} \parallel \mathcal{Y} \mid \Gamma \vdash J$, with symbols $\mathcal{V}$, as well as variables, $\mathcal{Y}$. Such formal judgments are identified up to renaming of its variables and its symbols to ensure that the meaning is independent of the choice of variable and symbol names.

3.5 Notes

The concepts of entailment and generality are fundamental to logic and programming languages. The formulation given here builds on Martin-Löf (1983, 1987) and Avron (1991). Hypothetical and general reasoning are consolidated into a single concept in the AUTOMATH languages (Nederpelt et al., 1994) and in the LF Logical Framework (Harper et al., 1993). These systems allow arbitrarily nested combinations of hypothetical and general judgments, whereas the present account considers only general hypothetical judgments over basic judgment forms. On the other hand we consider here symbols, as well as variables, which are not present in these previous accounts. Parametric judgments are required for specifying languages that admit the dynamic creation of "new" objects (see Chapter 34).

Exercises

3.1. *Combinators* are inductively defined by the rule set $\mathcal{C}$ given as follows:

$$\frac{}{\mathsf{s}\ \mathsf{comb}} \tag{3.15a}$$

$$\frac{}{\mathtt{k}\ \mathsf{comb}} \tag{3.15b}$$

$$\frac{a_1\ \mathsf{comb} \quad a_2\ \mathsf{comb}}{\mathtt{ap}(\, a_1\, ;\, a_2\,)\ \mathsf{comb}} \tag{3.15c}$$

Give an inductive definition of the *length* of a combinator defined as the number of occurrences of `S` and `K` within it.

3.2. The general judgment

$$x_1, \ldots, x_n \mid x_1\ \mathsf{comb}, \ldots, x_n\ \mathsf{comb} \vdash_{\mathcal{C}} A\ \mathsf{comb}$$

states that A is a combinator that may involve the variables $x_1, \ldots, x_n$. Prove that if $x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}} a_2\ \mathsf{comb}$ and $a_1\ \mathsf{comb}$, then $\{a_1/x\}a_2\ \mathsf{comb}$ by induction on the derivation of the first hypothesis of the implication.

3.3. *Conversion*, or *equivalence*, of combinators is expressed by the judgment $A \equiv B$ defined by the rule set $\mathcal{E}$ extending $\mathcal{C}$ as follows:[2]

$$\frac{a\ \mathsf{comb}}{a \equiv a} \tag{3.16a}$$

$$\frac{a_2 \equiv a_1}{a_1 \equiv a_2} \tag{3.16b}$$

$$\frac{a_1 \equiv a_2 \quad a_2 \equiv a_3}{a_1 \equiv a_3} \tag{3.16c}$$

$$\frac{a_1 \equiv a_1' \quad a_2 \equiv a_2'}{a_1\, a_2 \equiv a_1'\, a_2'} \tag{3.16d}$$

$$\frac{a_1\ \mathsf{comb} \quad a_2\ \mathsf{comb}}{\mathtt{k}\, a_1\, a_2 \equiv a_1} \tag{3.16e}$$

$$\frac{a_1\ \mathsf{comb} \quad a_2\ \mathsf{comb} \quad a_3\ \mathsf{comb}}{\mathtt{s}\, a_1\, a_2\, a_3 \equiv (\, a_1\, a_3\,)\,(\, a_2\, a_3\,)} \tag{3.16f}$$

The no-doubt mysterious motivation for the last two equations will become clearer in a moment. For now, show that

$$x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}\cup\mathcal{E}} \mathtt{s}\,\mathtt{k}\,\mathtt{k}\,x \equiv x.$$

3.4. Show that if $x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}} a\ \mathsf{comb}$, then there is a combinator a', written $[\,x\,]\,a$ and called *bracket abstraction*, such that

$$x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}\cup\mathcal{E}} a'\, x \equiv a.$$

Consequently, by Exercise 3.2, if $a''\ \mathsf{comb}$, then

$$(\,[\,x\,]\,a\,)\,a'' \equiv \{a''/x\}a.$$

[2] The combinator $\mathtt{ap}(\, a_1\, ;\, a_2\,)$ is written $a_1\, a_2$ for short, left-associatively when used in succession.

Hint: Inductively define the judgment

$$x \mid x \ \mathsf{comb} \vdash \mathsf{abs}_x\ a\ \mathsf{is}\ a',$$

where $x \mid x\ \mathsf{comb} \vdash a\ \mathsf{comb}$. Then argue that it defines a' as a binary function of x and a. The motivation for the conversion axioms governing `k` and `s` should become clear while developing the proof of the desired equivalence.

3.5. Prove that bracket abstraction, as defined in Exercise 3.4, is *non-compositional* by exhibiting a and b such that $a\ \mathsf{comb}$ and

$$x\,y \mid x\ \mathsf{comb}\, y\ \mathsf{comb} \vdash_{\mathcal{C}} b\ \mathsf{comb}$$

such that $\{a/y\}([x]\,b) \neq [x]\,(\{a/y\}b)$. *Hint*: Consider the case that b is y.

Suggest a modification to the definition of bracket abstraction that is *compositional* by showing under the same conditions given above that

$$\{a/y\}([x]\,b) = [x]\,(\{a/y\}b).$$

3.6. Consider the set $\mathcal{B}[\mathcal{X}]$ of abts generated by the operators ap, with arity $(\mathsf{Exp}, \mathsf{Exp})\mathsf{Exp}$, and λ, with arity $(\mathsf{Exp}.\mathsf{Exp})\mathsf{Exp}$, and possibly involving variables in $\mathcal{X}$, all of which are of sort Exp. Give an inductive definition of the judgment $b\ \mathsf{closed}$, which specifies that b has no free occurrences of the variables in $\mathcal{X}$. *Hint*: it is essential to give an inductive definition of the hypothetical, general judgment

$$x_1, \ldots, x_n \mid x_1\ \mathsf{closed}, \ldots, x_n\ \mathsf{closed} \vdash b\ \mathsf{closed}$$

in order to account for the binding of a variable by the λ operator. The hypothesis that a variable is closed seems self-contradictory in that a variable obviously occurs free in itself. Explain why this is not the case by examining carefully the meaning of the hypothetical and general judgments.

Part II

Statics and Dynamics

Chapter 4

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 set of rules for deriving *typing judgments* 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 correct; 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 a simple expression language, **E**, as an illustration of the method that we will employ throughout this book.

4.1 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 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 illus-

trated by example. The following chart summarizes the abstract and concrete syntax of **E**.

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 set of operators and their arities. For example, it specifies that the operator `let` has arity $(\mathsf{Exp}, \mathsf{Exp}.\mathsf{Exp})\mathsf{Exp}$, which specifies that it has two arguments of sort Exp, and binds a variable of sort Exp in the second argument.

4.2 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 the expression `plus`$(x\,;$`num`$[n])$ is sensible depends on whether the variable x is restricted 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 **E** consists of an inductive definition of generic hypothetical judgments of the form

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

where $\mathcal{X}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of variables, using the letters x and y to stand for them. We write $x \notin \Gamma$ to say 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 **E** are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{4.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{4.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{4.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1\,;e_2) : \texttt{num}} \tag{4.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1\,;e_2) : \texttt{num}} \tag{4.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{str} \quad \Gamma \vdash e_2 : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(\, e_1 \,;\, e_2 \,) : \mathtt{str}} \tag{4.1f}$$

$$\frac{\Gamma \vdash e : \mathtt{str}}{\Gamma \vdash \mathtt{len}(\, e \,) : \mathtt{num}} \tag{4.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(\, e_1 \,;\, x \,.\, e_2 \,) : \tau_2} \tag{4.1h}$$

In rule (4.1h) 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 by *induction on typing*, which is rule induction applied to rules (4.1).

Lemma 4.1 (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 (4.1), making use of the fact that variables have at most one type in any typing context. □

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

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

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

4.3 Structural Properties

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

Lemma 4.3 (Weakening)**.** *If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \notin \Gamma$ and any type τ.*

Proof. By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (4.1h). We have that $e' = \mathtt{let}(\, e_1 \,;\, z \,.\, e_2 \,)$, where by the conventions on variables we may assume z is chosen such that $z \notin \Gamma$ 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 (4.1h). □

Lemma 4.4 (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 (4.1h). As in the preceding case, $e' = \texttt{let}(\, e_1 \,;z\,.\, e_2\,)$, where z is chosen so that $z \neq x$ and $z \notin \Gamma$. We have by induction and Lemma 4.3 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 (4.1h) that $\Gamma \vdash \{e/x\}\texttt{let}(\, e_1 \,;z\,.\, e_2\,) : \tau'$, as desired. □

From a programming point of view, Lemma 4.3 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 introducing new variables beyond those required by an expression e does not invalidate e itself; it remains well-formed, with the same type.[1] More importantly, Lemma 4.4 expresses the important concepts of *modularity* and *linking*. We may think of the expressions e and e' as two *components* of a larger system in which e' is 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 to satisfy the assumptions of the client. If so, then we may link them to form the composite system $\{e/x\}e'$. This implementation 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 4.4 is called *decomposition*. It states that any (large) expression can be decomposed into a client and implementor by introducing a variable to mediate their interaction.

Lemma 4.5 (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. □

Lemma 4.5 tells us that any sub-expression can be isolated as a separate module of a larger system. This property 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 **E** given by rules (4.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introduction* and the *elimination*. The introduction forms for a type determine the *values*, or *canonical forms*, of that type. The elimination forms determine how to manipulate the values of a type to form a computation of another (possibly the same) type.

[1] This point may seem so obvious that it is not worthy of mention, but, surprisingly, there are useful type systems that lack this property. Because they do not necessarily validate the structural principle of weakening, they are called *substructural* type systems.

In the language **E** the introduction forms for the type `num` are the numerals, and those for the type `str` are the literals. The elimination 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 clear once we have defined the dynamics of the language in Chapter 5. Then we will see that the elimination forms are *inverse* to the introduction forms in that they "take apart" what the introduction forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 6.

4.4 Notes

The concept of the statics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. Statics in the sense considered here was introduced in the definition of the Standard ML programming language (Milner et al., 1997), building on earlier work by Church and others on the typed λ-calculus (Barendregt, 1992). The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction (Gentzen, 1969). These principles were applied to the structure of programming languages by Martin-Löf (1984, 1980).

Exercises

4.1. It is sometimes useful to give the typing judgment $\Gamma \vdash e : \tau$ an "operational" reading that specifies more precisely the flow of information required to derive a typing judgment (or determine that it is not derivable). The *analytic* mode corresponds to the context, expression, and type being given, with the goal to determine whether the typing judgment is derivable. The *synthetic* mode corresponds to the context and expression being given, with the goal to find the unique type τ, if any, possessed by the expression in that context. These two readings can be made explicit as judgments of the form $e \downarrow \tau$, corresponding to the analytic mode, and $e \uparrow \tau$, corresponding to the synthetic mode.

Give a simultaneous inductive definition of these two judgments according to the following guidelines:

(a) Variables are introduced in synthetic form.

(b) If we can synthesize a unique type for an expression, then we can analyze it with respect to a given type by checking type equality.

(c) Definitions need care, because the type of the defined expression is not given, even when the type of the result is given.

There is room for variation; the point of the exercise is to explore the possibilities.

4.2. One way to limit the range of possibilities in the solution to Exercise 4.1 is to restrict and extend the syntax of the language so that every expression is either synthetic or analytic according to the following suggestions:

(a) Variables are analytic.

(b) Introduction forms are analytic, elimination forms are synthetic.

(c) An analytic expression can be made synthetic by introducing a *type cast* of the form $\texttt{cast}[\tau](e)$ specifying that e must check against the specified type τ, which is synthesized for the whole expression.

(d) The defining expression of a definition must be synthetic, but the scope of the definition can be either synthetic or analytic.

Reformulate your solution to Exercise 4.1 to take account of these guidelines.

Chapter 5

Dynamics

The *dynamics* of a language describes how programs are 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 way. An *equational dynamics* presents the dynamics of a language by a collection of rules defining when one program is *definitionally equivalent* to another.

5.1 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. $s \longmapsto s'$, 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 $s \longmapsto s'$. A state from which no transition is possible is *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 $s \longmapsto s'$, otherwise it is *non-deterministic*.

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that s_0 initial, and $s_i \longmapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff there is no s such that $s_n \longmapsto s$, and it is *complete* iff it is maximal and s_n final. Thus every complete transition sequence is maximal, but maximal sequences are not necessarily complete.

The *iteration* of transition judgment $s \longmapsto^* s'$ is inductively defined by the following rules:

$$\frac{}{s \longmapsto^* s} \tag{5.1a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^* s''}{s \longmapsto^* s''} \tag{5.1b}$$

When applied to the definition of iterated transition, the principle of rule induction states that to show that $P(s, s')$ holds when $s \longmapsto^* s'$, it is enough to show these two properties of P:

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

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

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

$$\frac{}{s \longmapsto^0 s} \tag{5.2a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^n s''}{s \longmapsto^{n+1} s''} \tag{5.2b}$$

Theorem 5.1. *For all states s and s', $s \longmapsto^* s'$ iff $s \longmapsto^k s'$ for some $k \geq 0$.*

Proof. From left to right, by induction on the definition of multi-step transition. From right to left, by mathematical induction on $k \geq 0$. □

5.2 Structural Dynamics

A *structural dynamics* for the language **E** 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 judgment e val, which states that e is a value, is inductively defined by the following rules:

$$\frac{}{\texttt{num}[\,n\,]\ \mathsf{val}} \tag{5.3a}$$

$$\frac{}{\texttt{str}[\,s\,]\ \mathsf{val}} \tag{5.3b}$$

The transition judgment $e \longmapsto e'$ between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n}{\texttt{plus}(\,\texttt{num}[\,n_1\,]\,;\texttt{num}[\,n_2\,]\,) \longmapsto \texttt{num}[\,n\,]} \tag{5.4a}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(\,e_1\,;e_2\,) \longmapsto \texttt{plus}(\,e_1'\,;e_2\,)} \tag{5.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{plus}(\,e_1\,;e_2\,) \longmapsto \texttt{plus}(\,e_1\,;e_2'\,)} \tag{5.4c}$$

$$\frac{s_1 \hat{}\, s_2 = s}{\texttt{cat}(\,\texttt{str}[\,s_1\,]\,;\texttt{str}[\,s_2\,]\,) \longmapsto \texttt{str}[\,s\,]} \tag{5.4d}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{cat}(\,e_1\,;e_2\,) \longmapsto \texttt{cat}(\,e_1'\,;e_2\,)} \tag{5.4e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{cat}(\,e_1\,;e_2\,) \longmapsto \texttt{cat}(\,e_1\,;e_2'\,)} \tag{5.4f}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\texttt{let}(\,e_1\,;x\,.\,e_2\,) \longmapsto \texttt{let}(\,e_1'\,;x\,.\,e_2\,)}\right] \tag{5.4g}$$

$$\frac{[e_1\ \mathsf{val}]}{\texttt{let}(\,e_1\,;x\,.\,e_2\,) \longmapsto \{e_1/x\}e_2} \tag{5.4h}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (5.4a), (5.4d), and (5.4h) are *instruction transitions*, because they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order of execution of instructions.

The bracketed rule, rule (5.4g), and bracketed premise on rule (5.4h), are included for a *by-value* interpretation of `let`, and omitted for a *by-name* interpretation. The by-value interpretation evaluates an expression before binding it to the defined variable, whereas the by-name interpretation binds it in unevaluated form. The by-value interpretation saves work if the defined variable is used more than once, but wastes work if it is not used at all. Conversely, the by-name interpretation saves work if the defined variable is not used, and wastes work if it is used more than once.

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.

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

Each step in this sequence of transitions is justified by a derivation according to rules (5.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\dfrac{\dfrac{}{\mathtt{plus}(\mathtt{num}[3];\mathtt{num}[3]) \longmapsto \mathtt{num}[6]}\ (5.4a)}{\mathtt{plus}(\mathtt{plus}(\mathtt{num}[3];\mathtt{num}[3]);\mathtt{num}[4]) \longmapsto \mathtt{plus}(\mathtt{num}[6];\mathtt{num}[4])}\ (5.4b)$$

The other steps are similarly justified by composing rules.

The principle of rule induction for the structural dynamics of **E** states that to show $\mathcal{P}(e \longmapsto e')$ when $e \longmapsto e'$, it is enough to show that $\mathcal{P}$ is closed under rules (5.4). For example, we may show by rule induction that the structural dynamics of **E** is *determinate*, which means that an expression may transition to at most one other expression. The proof requires a simple lemma relating transition to values.

Lemma 5.2 (Finality of Values)**.** *For no expression e do we have both e val and $e \longmapsto e'$ for some e'.*

Proof. By rule induction on rules (5.3) and (5.4). □

Lemma 5.3 (Determinacy)**.** *If $e \longmapsto e'$ and $e \longmapsto e''$, then e' and e'' are α-equivalent.*

Proof. By rule induction on the premises $e \longmapsto e'$ and $e \longmapsto e''$, carried out either simultaneously or in either order. The primitive operators, such as addition, are assumed to have a unique value when applied to values. □

Rules (5.4) exemplify the *inversion principle* of language design, which states that the elimination forms are *inverse* to the introduction forms of a language. The search rules determine the *principal arguments* of each elimination form, and the instruction rules specify how to evaluate an elimination form when all of its principal arguments are in introduction form. For example, rules (5.4) specify that both arguments 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 6.

5.3 Contextual Dynamics

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

The instruction transition judgment $e_1 \longrightarrow e_2$ for **E** is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p}{\texttt{plus}(\,\texttt{num}[\,m\,]\,;\texttt{num}[\,n\,]\,) \longrightarrow \texttt{num}[\,p\,]} \tag{5.5a}$$

$$\frac{s \hat{} t = u}{\texttt{cat}(\,\texttt{str}[\,s\,]\,;\texttt{str}[\,t\,]\,) \longrightarrow \texttt{str}[\,u\,]} \tag{5.5b}$$

$$\frac{}{\texttt{let}(\,e_1\,;x\,.\,e_2\,) \longrightarrow \{e_1/x\}e_2} \tag{5.5c}$$

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

$$\frac{}{\circ\ \mathsf{ectx}} \tag{5.6a}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectx}}{\texttt{plus}(\,\mathcal{E}_1\,;e_2\,)\ \mathsf{ectx}} \tag{5.6b}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectx}}{\texttt{plus}(\,e_1\,;\mathcal{E}_2\,)\ \mathsf{ectx}} \tag{5.6c}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here", at the occurrence of the hole. The remaining rules correspond one-for-one to the search rules of the structural dynamics. For example, rule (5.6c) 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 a template that is instantiated by replacing the hole with an instruction to be executed. The judgment $e' = \mathcal{E}\{e\}$ 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:

$$\frac{}{e = \circ\{e\}} \tag{5.7a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(\,e_1\,;e_2\,) = \texttt{plus}(\,\mathcal{E}_1\,;e_2\,)\{e\}} \tag{5.7b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(\,e_1\,;e_2\,) = \texttt{plus}(\,e_1\,;\mathcal{E}_2\,)\{e\}} \tag{5.7c}$$

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 **E** is defined by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \longrightarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \longmapsto e'} \tag{5.8}$$

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 $e \underset{\text{str}}{\longmapsto} e'$ for the transition relation defined by the structural dynamics (rules (5.4)), and $e \underset{\text{ctx}}{\longmapsto} e'$ for the transition relation defined by the contextual dynamics (rules (5.8)).

Theorem 5.4. *$e \underset{str}{\longmapsto} e'$ if, and only if, $e \underset{ctx}{\longmapsto} e'$.*

Proof. From left to right, proceed by rule induction on rules (5.4). 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 $e_0 \longrightarrow e'_0$. For example, for rule (5.4a), take $\mathcal{E} = \circ$, and note that $e \longrightarrow e'$. For rule (5.4b), 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 $e_0 \longrightarrow e'_0$. Take $\mathcal{E} = \mathtt{plus}(\,\mathcal{E}_1\,;e_2\,)$, and note that $e = \mathtt{plus}(\,\mathcal{E}_1\,;e_2\,)\{e_0\}$ and $e' = \mathtt{plus}(\,\mathcal{E}_1\,;e_2\,)\{e'_0\}$ with $e_0 \longrightarrow e'_0$.

From right to left, note that if $e \underset{\text{ctx}}{\longmapsto} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e'_0\}$, and $e_0 \longrightarrow e'_0$. We prove by induction on rules (5.7) that $e \underset{\text{str}}{\longmapsto} e'$. For example, for rule (5.7a), e_0 is e, e'_0 is e', and $e \longrightarrow e'$. Hence $e \underset{\text{str}}{\longmapsto} e'$. For rule (5.7b), we have that $\mathcal{E} = \mathtt{plus}(\,\mathcal{E}_1\,;e_2\,)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e'_1 = \mathcal{E}_1\{e'_0\}$, and $e_1 \underset{\text{str}}{\longmapsto} e'_1$. Therefore e is $\mathtt{plus}(\,e_1\,;e_2\,)$, e' is $\mathtt{plus}(\,e'_1\,;e_2\,)$, and therefore by rule (5.4b), $e \underset{\text{str}}{\longmapsto} e'$.

□

Because the two transition judgments coincide, contextual dynamics can be considered an alternative presentation of a structural dynamics. It has two advantages over structural dynamics, one relatively superficial, one rather less so. The superficial advantage stems from writing rule (5.8) in the simpler form

$$\frac{e_0 \longrightarrow e'_0}{\mathcal{E}\{e_0\} \longmapsto \mathcal{E}\{e'_0\}}\ . \tag{5.9}$$

This formulation is superficially simpler in that it does not make explicit how an expression is decomposed into an evaluation context and a reducible expression. The deeper advantage of contextual dynamics is that all transitions are between complete programs. One need never consider a transition between expressions of any type other than the observable type, which simplifies certain arguments, such as the proof of Lemma 47.16.

5.4 Equational Dynamics

Another formulation of the dynamics of a language regards 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 enough 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 $2^2 + 2 \times 2 + 1 = 9$, which is indeed $(2+1)^2$. We thus obtain a model of computation in which the value of a polynomial for a given value of its variable is determined by substitution and simplification.

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in **E**, 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 judgment $\Gamma \vdash e \equiv e' : \tau$, we tacitly assume that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Here, as usual, we omit explicit mention of the variables $\mathcal{X}$ when they can be determined from the forms of the assumptions Γ.

Definitional equality of expressions in **E** under the by-name interpretation of `let` is inductively defined by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \equiv e : \tau} \tag{5.10a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{5.10b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{5.10c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1\,;e_2) \equiv \mathtt{plus}(e_1'\,;e_2') : \mathtt{num}} \tag{5.10d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(e_1\,;e_2) \equiv \mathtt{cat}(e_1'\,;e_2') : \mathtt{str}} \tag{5.10e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{let}(e_1\,;x\,.\,e_2) \equiv \mathtt{let}(e_1'\,;x\,.\,e_2') : \tau_2} \tag{5.10f}$$

$$\frac{n_1 + n_2 = n}{\Gamma \vdash \mathtt{plus}(\mathtt{num}[n_1]\,;\mathtt{num}[n_2]) \equiv \mathtt{num}[n] : \mathtt{num}} \tag{5.10g}$$

$$\frac{s_1 \hat{} s_2 = s}{\Gamma \vdash \mathtt{cat}(\mathtt{str}[s_1]\,;\mathtt{str}[s_2]) \equiv \mathtt{str}[s] : \mathtt{str}} \tag{5.10h}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{let}(e_1\,;x\,.\,e_2) \equiv \{e_1/x\}e_2 : \tau_2} \tag{5.10i}$$

Rules (5.10a) through (5.10c) state that definitional equality is an *equivalence relation*. Rules (5.10d) through (5.10f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (5.10g) through (5.10i) specify the meanings of the primitive constructs of **E**. We say that rules (5.10) define the *strongest congruence* closed under rules (5.10g), (5.10h), and (5.10i).

Rules (5.10) suffice to calculate the value of an expression by a deduction similar to that used in high school algebra. For example, we may derive the equation

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

by applying rules (5.10). 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 equality is rather weak in that many equivalences that we might intuitively think are true are not derivable from rules (5.10). A prototypical example is the putative equivalence

$$x_1 : \mathtt{num}, x_2 : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{5.11}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from rules (5.10). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{5.12}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (5.11), and all of its instances, given by Equation (5.12), 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 equivalence*, because 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 46.)

Theorem 5.5. *For the expression language* **E**, *the relation* $e \equiv e' : \tau$ *holds iff there exists* e_0 val *such that* $e \longmapsto^* e_0$ *and* $e' \longmapsto^* e_0$.

Proof. The proof from right to left is direct, because every transition step is a valid equation. The converse follows from the following, more general, proposition, which is proved by induction on rules (5.10): if $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then when $e_1 : \tau_1, e'_1 : \tau_1, \ldots, e_n : \tau_n, e'_n : \tau_n$, if for each $1 \leq i \leq n$ the expressions e_i and e'_i evaluate to a common value v_i, then there exists e_0 val such that

$$\{e_1, \ldots, e_n / x_1, \ldots, x_n\} e \longmapsto^* e_0$$

and

$$\{e'_1, \ldots, e'_n / x_1, \ldots, x_n\} e' \longmapsto^* e_0.$$

□

5.5 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 the SECD machine (Landin, 1965), 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 (Plotkin, 1981), which we use heavily throughout this book, was inspired by Church's

and Landin's ideas (Plotkin, 2004). Contextual semantics, which was introduced by Felleisen and Hieb (1992), 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, Gödel, and Church.

Exercises

5.1. Prove that if $s \longmapsto^* s'$ and $s' \longmapsto^* s''$, then $s \longmapsto^* s''$.

5.2. Complete the proof of Theorem 5.1 along the lines suggested there.

5.3. Complete the proof of Theorem 5.5 along the lines suggested there.

5.4. Prove that if $\Gamma \vdash e \equiv e' : \tau$ according to Rules (5.10), then $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ according to Rules (4.1).

Chapter 6

Type Safety

Most 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 **E** states that it will never arise that a number is added to a string, or that two numbers are 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. Safety 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.

Type safety for the language **E** is stated precisely as follows:

Theorem 6.1 (Type Safety)**.**

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

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

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 $e \longmapsto e'$. 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.

6.1 Preservation

The preservation theorem for **E** defined in Chapters 4 and 5 is proved by rule induction on the transition system (rules (5.4)).

Theorem 6.2 (Preservation). *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof. We will give the proof in two cases, leaving the rest to the reader. Consider rule (5.4b),

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

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

Now consider rule (5.4h),

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

Assume that $\texttt{let}(\, e_1 \,;\, x \,.\, e_2 \,) : \tau_2$. By the inversion lemma 4.2, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 4.4 $\{e_1/x\}e_2 : \tau_2$, as desired.

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

□

The proof of preservation is naturally structured as an induction on the transition judgment, because the argument hinges on examining all possible transitions from a given expression. In some cases we 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, sometimes, cannot be made to work at all.

6.2 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 6.3 (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 (4.1) and (5.3). □

Progress is proved by rule induction on rules (4.1) defining the statics of the language.

Theorem 6.4 (Progress). *If $e : \tau$, then either e val, or there exists e' such that $e \longmapsto e'$.*

Proof. The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (4.1d),

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

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

By induction we have that either e_1 val, or there exists e_1' such that $e_1 \longmapsto e_1'$. In the latter case it follows that $\mathtt{plus}(\, e_1 \,;\, e_2 \,) \longmapsto \mathtt{plus}(\, e_1' \,;\, e_2 \,)$, as required. In the former we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \longmapsto e_2'$. In the latter case we have that $\mathtt{plus}(\, e_1 \,;\, e_2 \,) \longmapsto \mathtt{plus}(\, e_1 \,;\, e_2' \,)$, as required. In the former, we have, by the Canonical Forms Lemma 6.3, $e_1 = \mathtt{num}[\, n_1 \,]$ and $e_2 = \mathtt{num}[\, n_2 \,]$, and hence

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

□

Because 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 is more than one rule for a given expression form. Such rules present no difficulites, so long as the proof proceeds by induction on the typing rules, and not on the structure of the expression.

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 together 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.

6.3 Run-Time Errors

Suppose that we wish to extend **E** with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

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

But the expression $\mathtt{div}(\, \mathtt{num}[\, 3 \,] \,;\, \mathtt{num}[\, 0 \,] \,)$ 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, practical, 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. We cannot predict statically whether an expression will be non-zero when evaluated, so the second approach is most often used in practice.

The overall 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 modeling 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 be present in a full inductive definition of this judgment:

$$\frac{e_1 \ \mathsf{val}}{\mathtt{div}(e_1\,;\mathtt{num}[0])\ \mathsf{err}} \tag{6.1a}$$

$$\frac{e_1 \ \mathsf{err}}{\mathtt{div}(e_1\,;e_2)\ \mathsf{err}} \tag{6.1b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{err}}{\mathtt{div}(e_1\,;e_2)\ \mathsf{err}} \tag{6.1c}$$

Rule (6.1a) 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 judgment is available, we may also consider an expression, `error`, which forcibly induces an error, with the following statics and dynamics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{6.2a}$$

$$\frac{}{\mathtt{error}\ \mathsf{err}} \tag{6.2b}$$

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

Theorem 6.5 (Progress With Error)**.** *If $e : \tau$, then either e err, or e val, or there exists e' such that $e \longmapsto e'$.*

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

6.4 Notes

The concept of type safety was first formulated by Milner (1978), who invented the slogan "well-typed programs do not go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen (1994) observed that we can instead show that ill-defined states cannot arise in a well-typed program, giving rise to the slogan "well-typed programs do not get stuck." However, their formulation relied on an analysis showing that no stuck state is well-typed. The progress theorem, which relies on the characterization of canonical forms in the style of Martin-Löf (1980), eliminates this analysis.

Exercises

6.1. Complete the proof of Theorem 6.2 in full detail.

6.2. Complete the proof of Theorem 6.4 in full detail.

6.3. Give several cases of the proof of Theorem 6.5 to illustrate how checked errors are handled in type safety proofs.

Part III

Total Functions

Chapter 9

System T of Higher-Order Recursion

System **T**, well-known as *Gödel's T*, is the combination of function types with the type of natural numbers. In contrast to **E**, which equips the naturals with some arbitrarily chosen arithmetic operations, the language **T** 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 **T** "comes equipped" with a proof of its termination. Although 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 **T**. To do so would demand a master termination proof for every possible program in the language, something that we shall prove does not exist.

9.1 Statics

The syntax of **T** 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
			$\mathtt{rec}[\tau](e;e_0;x.y.e_1)$	$\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow e_1\}$	recursion
			$\lambda[\tau](x.e)$	$\lambda(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z})\ldots)$, in which the successor is applied $n \geq 0$ times to zero. The expression $\mathtt{rec}[\tau](e;e_0;x.y.e_1)$ is called the *recursor*. 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 clear shortly.

Sometimes the *iterator*, $\mathtt{iter}[\tau](e\,;e_0\,;y\,.\,e_1)$, is considered as an alternative to the recursor. It has essentially the same meaning as the recursor, 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 the iterator is a special case of the recursor, because we can always ignore the predecessor binding. Conversely, the recursor is definable from the iterator, provided that we have product types (Chapter 10) at our disposal. To define the recursor from the iterator, we simultaneously compute the predecessor while iterating the specified computation.

The statics of **T** is given by the following typing rules:

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

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{9.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{9.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\tau](e\,;e_0\,;x\,.\,y\,.\,e_1) : \tau} \tag{9.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda[\tau_1](x\,.\,e) : \mathtt{arr}(\tau_1\,;\tau_2)} \tag{9.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2\,;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1\,;e_2) : \tau} \tag{9.1f}$$

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

Lemma 9.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash \{e/x\}e' : \tau'$.

9.2 Dynamics

The closed values of **T** are defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{9.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{9.2b}$$

$$\frac{}{\lambda[\tau](x\,.\,e)\ \mathsf{val}} \tag{9.2c}$$

The premise of rule (9.2b) is included for an *eager* interpretation of successor, and excluded for a *lazy* interpretation.

The transition rules for the dynamics of **T** are as follows:

$$\frac{[e \longmapsto e']}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')} \tag{9.3a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{9.3b}$$

$$\left[\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{9.3c}$$

$$\frac{[e_2 \text{ val}]}{\mathtt{ap}(\lambda[\tau](x.e); e_2) \longmapsto \{e_2/x\}e} \tag{9.3d}$$

$$\frac{e \longmapsto e'}{\mathtt{rec}[\tau](e; e_0; x.y.e_1) \longmapsto \mathtt{rec}[\tau](e'; e_0; x.y.e_1)} \tag{9.3e}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{z}; e_0; x.y.e_1) \longmapsto e_0} \tag{9.3f}$$

$$\frac{\mathtt{s}(e) \text{ val}}{\mathtt{rec}[\tau](\mathtt{s}(e); e_0; x.y.e_1) \longmapsto \{e, \mathtt{rec}[\tau](e; e_0; x.y.e_1)/x, y\}e_1} \tag{9.3g}$$

The bracketed rules and premises are included for an eager successor and call-by-value application, and omitted for a lazy successor and call-by-name application. Rules (9.3f) and (9.3g) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case the recursor reduces to 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 is not evaluated.

Lemma 9.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{nat}$, then either $e = \mathtt{z}$ or $e = \mathtt{s}(e')$ for some e'.*
2. *If $\tau = \tau_1 \to \tau_2$, then $e = \lambda(x : \tau_1)\, e_2$ for some e_2.*

Theorem 9.3 (Safety)**.** 1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

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

9.3 Definability

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

$$e_f(\,\overline{n}\,) \equiv \overline{f(n)} : \mathtt{nat}. \tag{9.4}$$

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

Definitional equality for **T**, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{ap}(\,\lambda[\,\tau_1\,](\,x\,.\,e_2\,)\,;e_1\,) \equiv \{e_1/x\}e_2 : \tau_2} \tag{9.5a}$$

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\,\tau\,](\,\mathtt{z}\,;e_0\,;x\,.\,y\,.\,e_1\,) \equiv e_0 : \tau} \tag{9.5b}$$

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\,\tau\,](\,\mathtt{s}(\,e\,)\,;e_0\,;x\,.\,y\,.\,e_1\,) \equiv \{e, \mathtt{rec}[\,\tau\,](\,e\,;e_0\,;x\,.\,y\,.\,e_1\,)/x, y\}e_1 : \tau} \tag{9.5c}$$

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

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

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

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

For the induction, assume that

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

Then calculate using the rules of definitional equality:

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

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

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

The Ackermann function grows very quickly. For example, $A(4,2) \approx 2^{65,536}$, which is often cited as being 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 arguments (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 the recursor, 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 to showing that it is definable in **T** is to note that $A(m+1,n)$ iterates n times the function $A(m,-)$, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

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

to be the λ-abstraction

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

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

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

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

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

to be the expression

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

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

$$e_a(\,\overline{0}\,)(\,\overline{n}\,) \equiv \mathtt{s}(\,\overline{n}\,) \tag{9.6}$$

$$e_a(\,\overline{m+1}\,)(\,\overline{0}\,) \equiv e_a(\,\overline{m}\,)(\,\overline{1}\,) \tag{9.7}$$

$$e_a(\,\overline{m+1}\,)(\,\overline{n+1}\,) \equiv e_a(\,\overline{m}\,)(\,e_a(\,\mathtt{s}(\,\overline{m}\,)\,)(\,\overline{n}\,)\,). \tag{9.8}$$

That is, the Ackermann function is definable in **T**.

9.4 Undefinability

It is impossible to define an infinite loop in **T**.

Theorem 9.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 46.15. □

Consequently, values of function type in **T** behave like mathematical functions: if $e : \tau_1 \to \tau_2$ and $e_1 : \tau_1$, then $e(\,e_1\,)$ evaluates to a value of type τ_2. 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 **T**. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of **T**. By assigning a unique number to each expression, we may manipulate expressions as data values in **T** so that **T** is able 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 that all 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. Enumerate 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 procedure 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 right 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 9.4, ensure that f_{univ} is a well-defined function. It is called the *universal function* for **T** 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* $\delta : \mathbb{N} \to \mathbb{N}$, by the equation $\delta(m) = f_{univ}(m)(m)$. The δ function is chosen so that $\delta(\ulcorner e \urcorner) = n$ iff $e(\,\overline{\ulcorner e \urcorner}\,) \equiv \overline{n} : \mathtt{nat}$. (The motivation for its definition will become clear in a moment.)

The function f_{univ} is not definable in **T**. Suppose that it were definable by the expression e_{univ}, then the diagonal function δ would be definable by the expression

$$e_\delta = \lambda\,(\,m : \mathtt{nat}\,)\, e_{univ}(\,m\,)(\,m\,).$$

But in that case we would have the equations

$$\begin{aligned} e_\delta(\,\overline{\ulcorner e \urcorner}\,) &\equiv e_{univ}(\,\overline{\ulcorner e \urcorner}\,)(\,\overline{\ulcorner e \urcorner}\,) \\ &\equiv e(\,\overline{\ulcorner e \urcorner}\,). \end{aligned}$$

Now let e_Δ be the function expression

$$\lambda\,(\,x : \mathtt{nat}\,)\,\mathtt{s}(\,e_\delta(\,x\,)\,),$$

[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 **T** may be seen as a form of incompleteness like that considered by Gödel.

[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.

so that we may deduce

$$\begin{aligned} e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) &\equiv \mathtt{s}(\, e_\delta(\overline{\ulcorner e_\Delta \urcorner})\,) \\ &\equiv \mathtt{s}(\, e_\Delta(\overline{\ulcorner e_\Delta \urcorner})\,). \end{aligned}$$

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

We say that a language $\mathcal{L}$ is *universal* if it is possible to write an interpreter for $\mathcal{L}$ in $\mathcal{L}$ itself. It is intuitively clear that f_{univ} is computable in the sense that we can define it in *some* sufficiently powerful programming language. But the preceding argument shows that **T** is not up to the task; it is not a universal programming language. Examination of the foregoing proof reveals an inescapable tradeoff: by insisting that all expressions terminate, we necessarily lose universality—there are computable functions that are not definable in the language.

9.5 Notes

System **T** was introduced by Gödel in his study of the consistency of arithmetic (Gödel, 1980). He showed how to "compile" proofs in arithmetic into well-typed terms of system **T**, and to reduce the consistency problem for arithmetic to the termination of programs in **T**. It was perhaps the first programming language whose design was directly influenced by the verification (of termination) of its programs.

Exercises

9.1. Prove Lemma 9.2.

9.2. Prove Theorem 9.3.

9.3. Attempt to prove that if $e : \mathtt{nat}$ is closed, then there exists n such that $e \longmapsto^* \overline{n}$ under the eager dynamics. Where does the proof break down?

9.4. Attempt to prove termination for all well-typed closed terms: if $e : \tau$, then there exists e' val such that $e \longmapsto^* e'$. You are free to appeal to Lemma 9.2 and Theorem 9.3 as necessary. Where does the attempt break down? Can you think of a stronger inductive hypothesis that might evade the difficulty?

9.5. Define a closed term e of type τ in **T** to be *hereditarily terminating at type* τ by induction on the structure of τ as follows:

(a) If $\tau = \mathtt{nat}$, then e is hereditarily terminating at type τ iff e is terminating (that is, iff $e \longmapsto^* \overline{n}$ for some n.)

(b) If $\tau = \tau_1 \to \tau_2$, then e is hereditarily terminating iff when e_1 is hereditarily terminating at type τ_1, then $e(\,e_1\,)$ is hereditarily terminating at type τ_2.

Attempt to prove hereditary termination for well-typed terms: if $e : \tau$, then e is hereditarily terminating at type τ. The stronger inductive hypothesis bypasses the difficulty that arose in Exercise 9.4, but introduces another obstacle. What is the complication? Can you think of an even stronger inductive hypothesis that would suffice for the proof?

9.6. Show that if e is hereditarily terminating at type τ, $e' : \tau$, and $e' \longmapsto e$, then e' is also hereditarily terminating at type τ. (The need for this result will become clear in the solution to Exercise 9.5.)

9.7. Define an open, well-typed term

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

to be *open hereditarily terminating* iff every substitution instance

$$\{e_1, \ldots, e_n / x_1, \ldots, x_n\}e$$

is closed hereditarily terminating at type τ when each e_i is closed hereditarily terminating at type τ_i for each $1 \leq i \leq n$. Derive Exercise 9.3 from this result.

Part IV

Finite Data Types

Chapter 10

Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated elimination 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 elimination 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*, $\langle \tau_i \rangle_{i \in I}$, 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 *labeled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

10.1 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
			`pr[l]`(e)	$e \cdot$ `l`	left projection
			`pr[r]`(e)	$e \cdot$ `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.

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \tag{10.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2} \tag{10.1d}$$

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

$$\frac{}{\langle\rangle \ \mathsf{val}} \tag{10.2a}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \ \mathsf{val}} \tag{10.2b}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\langle e_1, e_2 \rangle \longmapsto \langle e_1', e_2 \rangle}\right] \tag{10.2c}$$

$$\left[\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto e_2'}{\langle e_1, e_2 \rangle \longmapsto \langle e_1, e_2' \rangle}\right] \tag{10.2d}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{l} \longmapsto e' \cdot \mathtt{l}} \tag{10.2e}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{r} \longmapsto e' \cdot \mathtt{r}} \tag{10.2f}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \cdot \mathtt{l} \longmapsto e_1} \tag{10.2g}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2 \rangle \cdot \mathtt{r} \longmapsto e_2} \tag{10.2h}$$

The bracketed rules and premises are 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 10.1 (Safety). *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

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

Proof. Preservation is proved by induction on transition defined by rules (10.2). Progress is proved by induction on typing defined by rules (10.1). □

10.2 Finite Products

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

Typ	τ	::=	$\mathtt{prod}(\, i \hookrightarrow \tau_i \mid i \in I \,)$	$\langle \tau_i \rangle_{i \in I}$	product
Exp	e	::=	$\mathtt{tpl}(\, i \hookrightarrow e_i \mid i \in I \,)$	$\langle i \hookrightarrow e_i \mid i \in I \rangle$	tuple
			$\mathtt{pr}[\, i \,](\, e \,)$	$e \cdot i$	projection

The variable I stands for a finite *index set* over which products are formed. The type $\mathtt{prod}(\, i \hookrightarrow \tau_i \mid i \in I \,)$, or $\langle \tau_i \rangle_{i \in I}$ for short, is the type of I-*tuples* of expressions e_i of type τ_i, one for each $i \in I$. An I-tuple has the form $\mathtt{tpl}(\, i \hookrightarrow e_i \mid i \in I \,)$, or $\langle i \hookrightarrow e_i \mid i \in I \rangle$ for short, and for each $i \in I$ the ith projection from an I-tuple e is written $\mathtt{pr}[\, i \,](\, e \,)$, or $e \cdot i$ for short.

When $I = \{ i_1, \ldots, i_n \}$, the I-tuple type may be written in the form

$$\langle i_1 \hookrightarrow \tau_1 , \ldots , i_n \hookrightarrow \tau_n \rangle$$

where we make explicit the association of a type to each index $i \in I$. Similarly, we may write

$$\langle i_1 \hookrightarrow e_1 , \ldots , i_n \hookrightarrow e_n \rangle$$

for the I-tuple whose ith component is e_i.

Finite products generalize empty and binary products by choosing I to be empty or the two-element set $\{ \mathtt{l}, \mathtt{r} \}$, respectively. In practice I is often chosen to be a finite set of symbols that serve as labels for the components of the tuple to enhance readability.

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle i_1 \hookrightarrow e_1 , \ldots , i_n \hookrightarrow e_n \rangle : \langle i_1 \hookrightarrow \tau_1 , \ldots , i_n \hookrightarrow \tau_n \rangle} \tag{10.3a}$$

$$\frac{\Gamma \vdash e : \langle i_1 \hookrightarrow \tau_1 , \ldots , i_n \hookrightarrow \tau_n \rangle \quad (1 \leq k \leq n)}{\Gamma \vdash e \cdot i_k : \tau_k} \tag{10.3b}$$

In rule (10.3b) the index $i_k \in I$ is a *particular* element of the index set I, whereas in rule (10.3a), the indices $i_1, \ldots, i_n$ range over the entire index set I.

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

$$\frac{[e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1 , \ldots , i_n \hookrightarrow e_n \rangle \ \mathsf{val}} \tag{10.4a}$$

$$\left[\frac{\left\{\begin{array}{ccccccc} e_1\ \mathsf{val} & \ldots & e_{j-1}\ \mathsf{val} & e'_1 = e_1 & \ldots & e'_{j-1} = e_{j-1} \\ & & e_j \longmapsto e'_j & e'_{j+1} = e_{j+1} & \ldots & e'_n = e_n \end{array}\right\}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \longmapsto \langle i_1 \hookrightarrow e'_1, \ldots, i_n \hookrightarrow e'_n\rangle}\right] \tag{10.4b}$$

$$\frac{e \longmapsto e'}{e \cdot i \longmapsto e' \cdot i} \tag{10.4c}$$

$$\frac{[\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle\ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\rangle \cdot i_k \longmapsto e_k} \tag{10.4d}$$

As formulated, rule (10.4b) specifies that the components of a tuple are evaluated in *some* sequential order, without specifying the order in which the components are considered. It is not hard, but a bit technically complicated, to impose an evaluation order by imposing a total ordering on the index set and evaluating components according to this ordering.

Theorem 10.2 (Safety)**.** *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \longmapsto e'$.*

Proof. The safety theorem is decomposed into progress and preservation lemmas, which are proved as in Section 10.1. □

10.3 Primitive Mutual Recursion

Using products we may simplify the primitive recursion construct of **T** so that only the recursive result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\mathtt{iter}[\tau](e\,;e_0\,;x\,.\,e_1)$, we may define $\mathtt{rec}[\tau](e\,;e_0\,;x\,.\,y\,.\,e_1)$ to be $e' \cdot \mathtt{r}$, where e' is the expression

$$\mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow \langle \mathtt{z}, e_0\rangle \mid \mathtt{s}(x') \hookrightarrow \langle \mathtt{s}(x' \cdot \mathtt{l}), \{x' \cdot \mathtt{l}, x' \cdot \mathtt{r}/x, y\} e_1\rangle\}.$$

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 another 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 primitive recursion*, in which we define two functions simultaneously by primitive recursion. 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 if and only if n is even, and $o(n)$ is non-zero if and only if n is odd.

To define these functions in **T** enriched with products, we first define an auxiliary function $e_{\mathtt{eo}}$ of type

$$\mathtt{nat} \to (\,\mathtt{nat} \times \mathtt{nat}\,)$$

that computes both results simultaneously by swapping back and forth on recursive calls:

$$\lambda\,(\,n : \mathtt{nat}\,)\,\mathtt{iter}\,n\,\{\mathtt{z} \hookrightarrow \langle 1, 0\rangle \mid \mathtt{s}(\,b\,) \hookrightarrow \langle b \cdot \mathtt{r}, b \cdot \mathtt{l}\rangle\}.$$

We may then define $e_{\mathtt{ev}}$ and $e_{\mathtt{od}}$ as follows:

$$\begin{aligned} e_{\mathtt{ev}} &\triangleq \lambda\,(\,n : \mathtt{nat}\,)\,e_{\mathtt{eo}}(\,n\,) \cdot \mathtt{l} \\ e_{\mathtt{od}} &\triangleq \lambda\,(\,n : \mathtt{nat}\,)\,e_{\mathtt{eo}}(\,n\,) \cdot \mathtt{r}. \end{aligned}$$

10.4 Notes

Product types are the most basic form of structured data. All languages have some form of product type, but often in a form that is combined with other, separable, concepts. Common manifestations of products include: (1) functions with "multiple arguments" or "multiple results"; (2) "objects" represented as tuples of mutually recursive functions; (3) "structures," which are tuples with mutable components. There are many papers on finite product types, which include record types as a special case. Pierce (2002) provides a thorough account of record types, and their subtyping properties (for which, see Chapter 24). Allen et al. (2006) analyzes many of the key ideas in the framework of dependent type theory.

Exercises

10.1. A *database schema* may be thought of as a finite product type $\langle \tau \rangle_{i \in I}$, in which the *columns*, or *attributes*. are labeled by the indices I whose values are restricted to *atomic* types, such as `nat` and `str`. A value of a schema type is called a *tuple*, or *instance*, of that schema. A *database* may be thought of as a finite sequence of such tuples, called the *rows* of the database. Give a representation of a database using function, product, and natural numbers types, and define the *project* operation that sends a database with columns I to a database with columns $I' \subseteq I$ by restricting each row to the specified columns.

10.2. Rather than choose between a lazy and an eager dynamics for products, we can instead distinguish two forms of product type, called the *positive* and the *negative*. The statics of the negative product is given by rules (10.1), and the dynamics is lazy. The statics of the positive product, written $\tau_1 \otimes \tau_2$, is given by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \otimes(\,e_1\,;e_2\,) : \tau_1 \otimes \tau_2} \tag{10.5a}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 \otimes \tau_2 \quad \Gamma\, x_1 : \tau_1\, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{split}(\,e_0\,;x_1, x_2\,.\,e\,) : \tau} \tag{10.5b}$$

The dynamics of `fuse`, the introduction form for the positive pair, is eager, essentially because the elimination form, `split`, extracts both components simultaneously.

Show that the negative product is definable in terms of the positive product using the unit and function types to express the lazy dynamics of negative pairing. Show that the positive product is definable in terms of the negative product, provided that we have at our disposal a `let` expression with a by-value dynamics so that we may enforce eager evaluation of positive pairs.

10.3. Specializing Exercise **10.2** to nullary products, we obtain a positive and a negative unit type. The negative unit type is given by rules (10.1), with no elimination forms and one introduction form. Give the statics and dynamics for a positive unit type, and show that the positive and negative unit types are inter-definable without any further assumptions.

Chapter 11

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 allow 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.

11.1 Nullary and Binary 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	$::=$	$\mathtt{case}[\tau](e)$	$\mathtt{case}\, e\, \{\,\}$	null case
			$\mathtt{in}[\mathtt{l}][\tau_1;\tau_2](e)$	$\mathtt{l} \cdot e$	left injection
			$\mathtt{in}[\mathtt{r}][\tau_1;\tau_2](e)$	$\mathtt{r} \cdot e$	right injection
			$\mathtt{case}(e; x_1.e_1; x_2.e_2)$	$\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introduction form. The elimination form, $\mathtt{case}\, e\, \{\,\}$, expresses the contradiction that e is a value of type $\mathtt{void}$. The elements of the binary sum type are labeled to show whether they are drawn from the left or the right summand, either $\mathtt{l} \cdot e$ or $\mathtt{r} \cdot e$. A value of the sum type is eliminated by case analysis.

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

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{case}\, e\, \{\,\} : \tau} \tag{11.1a}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l} \cdot e : \tau_1 + \tau_2} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r} \cdot e : \tau_1 + \tau_2} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\, e \,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} : \tau} \tag{11.1d}$$

For the sake of readability, in rules (11.1b) and (11.1c) we have written $\mathtt{l} \cdot e$ and $\mathtt{r} \cdot e$ in place of the abstract syntax $\mathtt{in}[\mathtt{l}][\tau_1 ; \tau_2](e)$ and $\mathtt{in}[\mathtt{r}][\tau_1 ; \tau_2](e)$, which includes the types τ_1 and τ_2 explicitly. In rule (11.1d) both branches of the case analysis must have the same type. Because a type expresses a static "prediction" on the form of the value of an expression, and because an expression 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:

$$\frac{e \longmapsto e'}{\mathtt{case}\, e \,\{\,\} \longmapsto \mathtt{case}\, e' \,\{\,\}} \tag{11.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{l} \cdot e\ \mathsf{val}} \tag{11.2b}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{r} \cdot e\ \mathsf{val}} \tag{11.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{l} \cdot e \longmapsto \mathtt{l} \cdot e'}\right] \tag{11.2d}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{r} \cdot e \longmapsto \mathtt{r} \cdot e'}\right] \tag{11.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e \,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \mathtt{case}\, e' \,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}} \tag{11.2f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{l} \cdot e \,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \{e/x_1\}e_1} \tag{11.2g}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{r} \cdot e \,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \{e/x_2\}e_2} \tag{11.2h}$$

The bracketed premises and rules are 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 11.1 (Safety). *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

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

Proof. The proof proceeds by induction on rules (11.2) for preservation, and by induction on rules (11.1) for progress. □

11.2 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	::=	$\mathtt{sum}(\, i \hookrightarrow \tau_i \mid i \in I \,)$	$[\tau_i]_{i \in I}$	sum
Exp	e	::=	$\mathtt{in}[\, i \,][\, \vec{\tau} \,](\, e \,)$	$i \cdot e$	injection
			$\mathtt{case}(\, e \,; i \hookrightarrow x_i \,.\, e_i \mid i \in I \,)$	$\mathtt{case}\, e \,\{ i \cdot x_i \hookrightarrow e_i \mid i \in I \}$	case analysis

The variable I stands for a finite index set over which sums are formed. The notation $\vec{\tau}$ stands for a finite function $(i \hookrightarrow \tau_i)_{i \in I}$ for some index set I. The type $\mathtt{sum}(\, i \hookrightarrow \tau_i \mid i \in I \,)$, or $[\tau_i]_{i \in I}$ for short, is the type of I-classified values of the form $\mathtt{in}[\, i \,][\, I \,](\, e_i \,)$, or $i \cdot e_i$ for short, where $i \in I$ and e_i is an expression of type τ_i. An I-classified value is analyzed by an I-way case analysis of the form $\mathtt{case}(\, e \,; i \hookrightarrow x_i \,.\, e_i \mid i \in I \,)$.

When $I = \{\, i_1, \ldots, i_n \,\}$, the type of I-classified values may be written

$$[\, i_1 \hookrightarrow \tau_1 \,, \ldots, i_n \hookrightarrow \tau_n \,]$$

specifying the type associated with each class $l_i \in I$. Correspondingly, the I-way case analysis has the form

$$\mathtt{case}\, e \,\{ i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n \}.$$

Finite sums generalize empty and binary sums by choosing I to be empty or the two-element set $\{\, \mathtt{l}, \mathtt{r} \,\}$, respectively. In practice I is often chosen to be a finite set of symbols that serve as names for the classes so as to enhance readability.

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

$$\frac{\Gamma \vdash e : \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e : [\, i_1 \hookrightarrow \tau_1 \,, \ldots, i_n \hookrightarrow \tau_n \,]} \tag{11.3a}$$

$$\frac{\Gamma \vdash e : [\, i_1 \hookrightarrow \tau_1 \,, \ldots, i_n \hookrightarrow \tau_n \,] \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}\, e \,\{ i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n \} : \tau} \tag{11.3b}$$

These rules generalize the statics for nullary and binary sums given in Section 11.1.

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

$$\frac{[\, e \ \mathsf{val} \,]}{i \cdot e \ \mathsf{val}} \tag{11.4a}$$

$$\left[\frac{e \longmapsto e'}{i \cdot e \longmapsto i \cdot e'}\right] \tag{11.4b}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i \mid i \in I\} \longmapsto \mathtt{case}\, e'\, \{i \cdot x_i \hookrightarrow e_i \mid i \in I\}} \tag{11.4c}$$

$$\frac{i \cdot e\ \mathsf{val}}{\mathtt{case}\, i \cdot e\, \{i \cdot x_i \hookrightarrow e_i \mid i \in I\} \longmapsto \{e/x_i\}e_i} \tag{11.4d}$$

These again generalize the dynamics of binary sums given in Section 11.1.

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

Proof. The proof is like that for the binary case, as described in Section 11.1. □

11.3 Applications of Sum Types

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

11.3.1 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, $\langle\rangle$, whereas the type `void` has no elements at all. Consequently, if e : `unit`, then if e evaluates to a value, that value is $\langle\rangle$ — in other words, e has *no interesting value*. 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. Thus 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!

11.3.2 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	::=	`true`	`true`	truth
			`false`	`false`	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`.

The statics of Booleans is given by the following typing rules:

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \tag{11.5b}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : \tau} \tag{11.5c}$$

The dynamics is given by the following value and transition rules:

$$\frac{}{\mathtt{true}\ \mathsf{val}} \tag{11.6a}$$

$$\frac{}{\mathtt{false}\ \mathsf{val}} \tag{11.6b}$$

$$\frac{}{\mathtt{if}\, \mathtt{true}\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \longmapsto e_1} \tag{11.6c}$$

$$\frac{}{\mathtt{if}\, \mathtt{false}\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \longmapsto e_2} \tag{11.6d}$$

$$\frac{e \longmapsto e'}{\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \longmapsto \mathtt{if}\, e'\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2} \tag{11.6e}$$

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

$$\mathtt{bool} = \mathtt{unit} + \mathtt{unit} \tag{11.7a}$$

$$\mathtt{true} = \mathtt{l} \cdot \langle\rangle \tag{11.7b}$$

$$\mathtt{false} = \mathtt{r} \cdot \langle\rangle \tag{11.7c}$$

$$\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 = \mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.7d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. It is a simple matter to check that the readily-defined statics and dynamics of the type `bool` are engendered by these definitions.

11.3.3 Enumerations

More generally, sum types can 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

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

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\mathtt{suit} = [\mathtt{unit}]_{\_ \in I}$, where $I = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and the type family is constant over this set. The case analysis form for a labeled 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.

Other examples of enumeration types abound. For example, most languages have a type `char` of characters, which is a large enumeration type containing all possible `Unicode` (or other such standard classification) characters. Each character is assigned a *code* (such as `UTF-8`) used for interchange among programs. The type `char` is equipped with operations such as $\texttt{chcode}(\,n\,)$ that yield the `char` associated to the code n, and $\texttt{codech}(\,c\,)$ that yield the code of character c. Using the linear ordering on codes we may define a total ordering of characters, called the *collating sequence* determined by that code.

11.3.4 Options

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

Typ	τ	::=	$\texttt{opt}(\,\tau\,)$	$\tau\,\texttt{opt}$	option
Exp	e	::=	$\texttt{null}$	$\texttt{null}$	nothing
			$\texttt{just}(\,e\,)$	$\texttt{just}(\,e\,)$	something
			$\texttt{ifnull}[\,\tau\,](\,e\,;e_1\,;x\,.\,e_2\,)$	$\texttt{ifnull}\,e\,\{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(\,x\,) \hookrightarrow e_2\}$	null test

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

The option type is definable from sums and nullary products according to the following equations:[1]

$$\tau\,\texttt{opt} = \texttt{unit} + \tau \tag{11.8a}$$

$$\texttt{null} = \texttt{l} \cdot \langle\rangle \tag{11.8b}$$

$$\texttt{just}(\,e\,) = \texttt{r} \cdot e \tag{11.8c}$$

$$\texttt{ifnull}\,e\,\{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(\,x_2\,) \hookrightarrow e_2\} = \texttt{case}\,e\,\{\texttt{l} \cdot \_ \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.8d}$$

We leave it to the reader to check 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 arises from two related errors. The first error is to deem values of certain types to be mysterious entities called *pointers*. This terminology arises from suppositions about how these values might be represented at run-time, rather than on their semantic role in the language. 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 stand for a value of that type at all, but rather rejects all attempts to use it.

To help avoid such failures, such languages usually include a function, say $\texttt{null} : \tau \to \texttt{bool}$, that yields `true` if its argument is null, and `false` otherwise. Such a test allows the programmer to

[1] We often write an underscore in place of a bound variable that is not used within its scope.

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

$$\texttt{if null}(\,e\,)\,\texttt{then}\ldots \mathit{error}\ldots\,\texttt{else}\ldots \mathit{proceed}\ldots. \tag{11.9}$$

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 is the failure to distinguish the type τ from the type $\tau\ \texttt{opt}$. Rather than think of the elements of type τ as pointers, and thereby have to worry about the null pointer, we instead distinguish 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,

$$\texttt{ifnull}\,e\,\{\texttt{null} \hookrightarrow e_{error} \mid \texttt{just}(\,x\,) \hookrightarrow e_{ok}\}, \tag{11.10}$$

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 necessary. Note that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (11.9); the advantage of option types is precisely that they do so.

11.4 Notes

Heterogeneous data structures are ubiquitous. Sums codify heterogeneity, yet few languages support them in the form given here. The best approximation in commercial languages is the concept of a class in object-oriented programming. A class is an injection into a sum type, and dispatch is case analysis on the class of the data object. (See Chapter 26 for more on this correspondence.) The absence of sums is the origin of C.A.R. Hoare's self-described "billion dollar mistake," the null pointer (Hoare, 2009). Bad language designs put the burden of managing "null" values entirely at run-time, instead of making the possibility or the impossibility of "null" apparent at compile time.

Exercises

11.1. Complete the definition of a finite enumeration type sketched in Section 11.3.3. Derive enumeration types from finite sum types.

11.2. The essence of Hoare's mistake is the misidentification of the type $\tau\ \texttt{opt}$ with the type $\texttt{bool} \times \tau$. Values of the latter type are pairs consisting of a boolean "flag" and a value of type τ. The idea is that the flag indicates whether the associated value is "present". When the flag is `true`, the second component is present, and, when the flag is `false`, the second component is absent.

Analyze Hoare's mistake by attempting to define τ `opt` to be the type `bool` $\times \tau$ by filling in the following chart:

$$\texttt{null} \triangleq \textbf{?}$$
$$\texttt{just}(\,e\,) \triangleq \textbf{?}$$
$$\texttt{ifnull}\, e\, \{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(\,x\,) \hookrightarrow e_2\} \triangleq \textbf{?}$$

Argue that *even if* we adopt Hoare's convention of admitting a "null" value of every type, the chart cannot be properly filled.

11.3. Databases have a version of the "null pointer" problem that arises when not every tuple provides a value for every attribute (such as a person's middle name). More generally, many commercial databases are limited to a single atomic type for each attribute, presenting problems when the value of that attribute may have several types (for example, one may have different sorts of postal codes depending on the country). Consider how to address these problems using the methods discussed in Exercise 10.1. Suggest how to handle null values and heterogeneous values that avoids some of the complications that arise in traditional formulations of databases.

11.4. A *combinational circuit* is an open expression of type

$$x_1 : \texttt{bool}, \ldots, x_n : \texttt{bool} \vdash e : \texttt{bool},$$

which computes a boolean value from n boolean inputs. Define a NOR and a NAND gate as boolean circuits with two inputs and one output. There is no reason to restrict to a single output. For example, define an HALF-ADDER that takes two boolean inputs, but produces two boolean outputs, the sum and the carry outputs of the HALF-ADDER. Then define a FULL-ADDER that takes three inputs, the addends and an incoming carry, and produces two outputs, the sum and the outgoing carry. Define the type NYBBLE to be the product `bool` $\times$ `bool` $\times$ `bool` $\times$ `bool`. Define the combinational circuit NYBBLE-ADDER that takes two nybbles as input and produces a nybble and a carry-out bit as output.

11.5. A *signal* is a time-varying sequence of booleans, representing the status of the signal at each time instant. An RS latch is a fundamental digital circuit with two input signals and two output signals. Define the type `signal` of signals to be the function type `nat` $\to$ `bool` of infinite sequences of booleans. Define an RS latch as a function of type

$$(\,\texttt{signal} \times \texttt{signal}\,) \to (\,\texttt{signal} \times \texttt{signal}\,).$$

Part V

Types and Propositions

Part VI

Infinite Data Types

Part VII

Variable Types

Chapter 16

System F of Polymorphic Types

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 **T** 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 have the same behavior when executed.

Obviously it would be useful to capture the 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 *polymorphic*. In this chapter we will study the language **F**, which was 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 17), 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.)

16.1 Polymorphic Abstraction

The language **F** is a variant of **T** in which we eliminate the type of natural numbers, but add, in compensation, polymorphic types:[1]

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	
			$\lambda[\tau](x.e)$	$\lambda(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\Lambda(t.e)$	$\Lambda(t)\,e$	type abstraction
			$\mathtt{App}[\tau](e)$	$e[\tau]$	type application

A *type abstraction* $\Lambda(t.e)$ defines a *generic*, or *polymorphic*, function with *type variable* 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 plugged in for the type variable to obtain the result. The *universal type*, $\mathtt{all}(t.\tau)$, classifies polymorphic functions.

The statics of **F** consists of two judgment forms, the *type formation* judgment,

$$\Delta \vdash \tau \ \mathsf{type},$$

and the *typing judgment*,

$$\Delta\ \Gamma \vdash e : \tau.$$

The hypotheses Δ have the form t type, where t is a variable of sort Typ, and the hypotheses Γ have the form $x : \tau$, where x is a variable of sort Exp.

The rules defining the type formation judgment are as follows:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{16.1a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{16.1b}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{type}} \tag{16.1c}$$

The rules defining the typing judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{16.2a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \lambda[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{16.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{16.2c}$$

[1] Girard's original version of System F included the natural numbers as a basic type.

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(t\,.\,e) : \mathtt{all}(t\,.\,\tau)} \tag{16.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t\,.\,\tau') \quad \Delta \vdash \tau\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : \{\tau/t\}\tau'} \tag{16.2e}$$

Lemma 16.1 (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 (16.2). □

The statics admits the structural rules for a general hypothetical judgment. In particular, we have the following critical substitution property for type formation and expression typing.

Lemma 16.2 (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, written $C[\tau_1][\tau_2][\tau_3]$. 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 **F** is given as follows:

$$\frac{}{\lambda[\,\tau\,](\,x\,.\,e\,)\ \mathsf{val}} \tag{16.3a}$$

$$\frac{}{\Lambda(\,t\,.\,e\,)\ \mathsf{val}} \tag{16.3b}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\,\lambda[\,\tau_1\,](\,x\,.\,e\,)\,;e_2\,) \longmapsto \{e_2/x\}e} \tag{16.3c}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(\,e_1\,;e_2\,) \longmapsto \mathtt{ap}(\,e_1'\,;e_2\,)} \tag{16.3d}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(\,e_1\,;e_2\,) \longmapsto \mathtt{ap}(\,e_1\,;e_2'\,)}\right] \tag{16.3e}$$

$$\frac{}{\mathtt{App}[\,\tau\,](\,\Lambda(\,t\,.\,e\,)\,) \longmapsto \{\tau/t\}e} \tag{16.3f}$$

$$\frac{e \longmapsto e'}{\mathtt{App}[\,\tau\,](\,e\,) \longmapsto \mathtt{App}[\,\tau\,](\,e'\,)} \tag{16.3g}$$

The bracketed premises and rule are included for a call-by-value interpretation, and omitted for a call-by-name interpretation of **F**.

It is a simple matter to prove safety for **F**, using familiar methods.

Lemma 16.3 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{arr}(\,\tau_1\,;\tau_2\,)$, then $e = \lambda[\,\tau_1\,](\,x\,.\,e_2\,)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \mathtt{all}(\,t\,.\,\tau'\,)$, then $e = \Lambda(\,t\,.\,e'\,)$ with t type $\vdash e' : \tau'$.*

Proof. By rule induction on the statics. □

Theorem 16.4 (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof. By rule induction on the dynamics. □

Theorem 16.5 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \longmapsto e'$.*

Proof. By rule induction on the statics. □

16.2 Polymorphic Definability

The language **F** is astonishingly expressive. Not only are all (lazy) finite products and sums definable in the language, but so are all (lazy) inductive and coinductive types. Their definability is most naturally expressed using definitional equality, which is the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash (\ \lambda\ (\ x : \tau_1\ )\ e_2\ )(\ e_1\ ) \equiv \{e_1/x\}e_2 : \tau_2} \tag{16.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash (\ \Lambda(\ t\ )\ e\ )[\ \rho\ ] \equiv \{\rho/t\}e : \{\rho/t\}\tau} \tag{16.4b}$$

In addition there are rules omitted here specifying that definitional equality is a congruence relation (that is, an equivalence relation respected by all expression-forming operations).

16.2.1 Products and Sums

The nullary product, or unit, type is definable in **F** as follows:

$$\begin{aligned} \mathtt{unit} &\triangleq \forall(\ r\ .\ r \to r\ ) \\ \langle\rangle &\triangleq \Lambda(\ r\ )\ \lambda\ (\ x : r\ )\ x \end{aligned}$$

The identity function plays the role of the null tuple, because it is the only closed value of this type.

Binary products are definable in **F** by using encoding tricks similar to those described in Chapter 21 for the untyped λ-calculus:

$$\begin{aligned} \tau_1 \times \tau_2 &\triangleq \forall(\ r\ .\ (\tau_1 \to \tau_2 \to r) \to r\ ) \\ \langle e_1, e_2\rangle &\triangleq \Lambda(\ r\ )\ \lambda\ (\ x : \tau_1 \to \tau_2 \to r\ )\ x(\ e_1\ )(\ e_2\ ) \\ e \cdot \mathtt{l} &\triangleq e[\ \tau_1\ ](\ \lambda\ (\ x : \tau_1\ )\ \lambda\ (\ y : \tau_2\ )\ x\ ) \\ e \cdot \mathtt{r} &\triangleq e[\ \tau_2\ ](\ \lambda\ (\ x : \tau_1\ )\ \lambda\ (\ y : \tau_2\ )\ y\ ) \end{aligned}$$

The statics given in Chapter 10 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in **F** 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 **F**:

$$\begin{aligned} \mathtt{void} &\triangleq \forall(\ r\ .\ r\ ) \\ \mathtt{case}\ e\ \{\ \} &\triangleq e[\ \rho\ ] \end{aligned}$$

Binary sums are also definable in **F**:

$$\begin{aligned}
\tau_1 + \tau_2 &\triangleq \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r) \\
\mathtt{l} \cdot e &\triangleq \Lambda(r)\,\lambda(x:\tau_1 \to r)\,\lambda(y:\tau_2 \to r)\,x(e) \\
\mathtt{r} \cdot e &\triangleq \Lambda(r)\,\lambda(x:\tau_1 \to r)\,\lambda(y:\tau_2 \to r)\,y(e) \\
\mathtt{case}\,e\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} &\triangleq \\
e[\rho](\lambda(x_1:\tau_1)\,e_1)(\lambda(x_2:\tau_2)\,e_2)&
\end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in **F**:

$$\mathtt{case}\,\mathtt{l} \cdot d_1\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \equiv \{d_1/x_1\}e_1 : \rho$$

and

$$\mathtt{case}\,\mathtt{r} \cdot d_2\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \equiv \{d_2/x_2\}e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 11 is correctly implemented by these definitions.

16.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in **F**. The key is the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}[\tau](e_0; e_1; x.e_2) : \tau}.$$

Because the result type τ is arbitrary, this means that if we have an iterator, then we can use it 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, given the initial result for $\mathtt{z}$ and a transformation from the result for x into the result for $\mathtt{s}(x)$, yields the result of iterating the transformation n times, starting with the initial result.

Because the *only* operation we can perform on a natural number is to iterate up to it, we may simply *identify* a natural number, n, with the polymorphic iterate-up-to-n function just described. Thus we may define the type of natural numbers in **F** by the following equations:

$$\begin{aligned}
\mathtt{nat} &\triangleq \forall(t.t \to (t \to t) \to t) \\
\mathtt{z} &\triangleq \Lambda(t)\,\lambda(z:t)\,\lambda(s:t \to t)\,z \\
\mathtt{s}(e) &\triangleq \Lambda(t)\,\lambda(z:t)\,\lambda(s:t \to t)\,s(e[t](z)(s)) \\
\mathtt{iter}[\tau](e_0; e_1; x.e_2) &\triangleq e_0[\tau](e_1)(\lambda(x:\tau)\,e_2)
\end{aligned}$$

It is easy to check that the statics and dynamics of the natural numbers type given in Chapter 9 are derivable in **F** under these definitions. The representations of the numerals in **F** are called the *polymorphic Church numerals*.

The encodability of the natural numbers shows that **F** is *at least as expressive* as **T**. But is it *more* expressive? Yes! It is possible to show that the evaluation function for **T** is definable in **F**, even though it is not definable in **T** itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for **F** is not definable in **F**. We may enrich **F** a bit more to define the evaluator for **F**, 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.

16.3 Parametricity Overview

A remarkable property of **F** is that polymorphic types severely constrain the behavior of their elements. We 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 is the identity function. Informally, when i is applied to a type, τ, and an argument of type τ, it returns a value of type τ. But because 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 is equivalent to either $\Lambda(\,t\,)\,\lambda\,(\,x:t\,)\,\lambda\,(\,y:t\,)\,x$ or $\Lambda(\,t\,)\,\lambda\,(\,x:t\,)\,\lambda\,(\,y:t\,)\,y$. Intuitively, when b is applied to two arguments of a given type, the only value it can return is one of the givens.

Properties of a program in **F** that can be proved knowing only its type are called *parametricity properties*. The facts about the functions i and b stated above are examples of parametricity properties. Such properties are sometimes called "free theorems," because they come from typing "for free", without any knowledge of the code itself. It bears repeating that in **F** we prove non-trivial behavioral properties of programs without ever examining the program text. The key to this incredible fact is that we are able to prove a deep property, called *parametricity*, about the language **F**, that then applies to every program written in **F**. One may say that the type system "pre-verifies" programs with respect to a broad range of useful properties, eliminating the need to prove those properties about every program separately. The parametricity theorem for **F** explains the remarkable experience that if a piece of code type checks, then it "just works." Parametricity narrows the space of well-typed programs sufficiently that the opportunities for programmer error are reduced to almost nothing.

So how does the parametricity theorem work? Without getting into too many technical details (but see Chapter 48 for a full treatment), we can give a brief summary of the main idea. Any function $i : \forall(\,t\,.\,t \to t\,)$ in **F** enjoys the following property:

> *For any type τ and any property $\mathcal{P}$ of the type τ, then if $\mathcal{P}$ holds of $x : \tau$, then $\mathcal{P}$ holds of $i[\,\tau\,](\,x\,)$.*

To show that for any type τ, and any x of type τ, the expression $i[\,\tau\,](\,x\,)$ is equivalent to x, it suffices to fix $x_0 : \tau$, and consider the property $\mathcal{P}_{x_0}$ that holds of $y : \tau$ iff y is equivalent to x_0. Obviously $\mathcal{P}$ holds of x_0 itself, and hence by the above-displayed property of i, it sends any argument satisfying $\mathcal{P}_{x_0}$ to a result satisfying $\mathcal{P}_{x_0}$, which is to say that it sends x_0 to x_0. Because x_0 is an arbitrary element of τ, it follows that $i[\,\tau\,]$ is the identity function, $\lambda\,(\,x:\tau\,)\,x$, on the type τ, and because τ is itself arbitrary, i is the polymorphic identity function, $\Lambda(\,t\,)\,\lambda\,(\,x:t\,)\,x$.

A similar argument suffices to show that the function b, defined above, is either $\Lambda(t)\,\lambda(x:t)\,\lambda(y:t)\,x$ or $\Lambda(t)\,\lambda(x:t)\,\lambda(y:t)\,y$. By virtue of its type the function b enjoys the parametricity property

> *For any type τ and any property $\mathcal{P}$ of τ, if $\mathcal{P}$ holds of $x : \tau$ and of $y : \tau$, then $\mathcal{P}$ holds of $b[\tau](x)(y)$.*

Choose an arbitrary type τ and two arbitrary elements x_0 and y_0 of type τ. Define $\mathcal{Q}_{x_0,y_0}$ to hold of $z : \tau$ iff either z is equivalent to x_0 or z is equivalent to y_0. Clearly $\mathcal{Q}_{x_0,y_0}$ holds of both x_0 and y_0 themselves, so by the quoted parametricity property of b, it follows that $\mathcal{Q}_{x_0,y_0}$ holds of $b[\tau](x_0)(y_0)$, which is to say that it is equivalent to either x_0 or y_0. Since τ, x_0, and y_0 are arbitrary, it follows that b is equivalent to either $\Lambda(t)\,\lambda(x:t)\,\lambda(y:t)\,x$ or $\Lambda(t)\,\lambda(x:t)\,\lambda(y:t)\,y$.

The parametricity theorem for **F** implies even stronger properties of functions such as i and b considered above. For example, the function i of type $\forall(t\,.\,t \to t)$ also satisfies the following condition:

> *If τ and τ' are any two types, and $\mathcal{R}$ is a binary relation between τ and τ', then for any $x : \tau$ and $x' : \tau'$, if $\mathcal{R}$ relates x to x', then $\mathcal{R}$ relates $i[\tau](x)$ to $i[\tau'](x')$.*

Using this property we may again prove that i is equivalent to the polymorphic identity function. Specifically, if τ is any type and $g : \tau \to \tau$ is any function on that type, then it follows from the type of i alone that $i[\tau](g(x))$ is equivalent to $g(i[\tau](x))$ for any $x : \tau$. To prove this, simply choose $\mathcal{R}$ to the be graph of the function g, the relation $\mathcal{R}_g$ that holds of x and x' iff x' is equivalent to $g(x)$. The parametricity property of i, when specialized to $\mathcal{R}_g$, states that if x' is equivalent to $g(x)$, then $i[\tau](x')$ is equivalent to $g(i[\tau](x))$, which is to say that $i[\tau](g(x))$ is equivalent to $g(i[\tau](x))$. To show that i is equivalent to the identity function, choose $x_0 : \tau$ arbitrarily, and consider the constant function g_0 on τ that always returns x_0. Because x_0 is equivalent to $g_0(x_0)$, it follows that $i[\tau](x_0)$ is equivalent to x_0, which is to say that i behaves like the polymorphic identity function.

16.4 Notes

System F was introduced by Girard (1972) in the context of proof theory and by Reynolds (1974) in the context of programming languages. The concept of parametricity was originally isolated by Strachey, but was not fully developed until the work of Reynolds (1983). The phrase "free theorems" for parametricity theorems was introduced by Wadler (1989).

Exercises

16.1. Give polymorphic definitions and types to the `s` and `k` combinators defined in Exercise 3.1.

16.2. Define in **F** the type `bool` of *Church booleans*. Define the type `bool`, and define `true` and `false` of this type, and the conditional `if` e `then` e_0 `else` e_1, where e is of this type.

16.3. Define in **F** the inductive type of lists of natural numbers as defined in Chapter 15. *Hint*: Define the representation in terms of the recursor (elimination form) for lists, following the pattern for defining the type of natural numbers.

16.4. Define in **F** an arbitrary inductive type, $\mu(t\,.\,\tau)$. *Hint*: generalize your answer to Exercise **16.3**.

16.5. Define the type $t\,\texttt{list}$ as in Exercise **16.3**, with the element type, t, unspecified. Define the finite set of *elements* of a list l to be those x given by the head of some number of tails of l. Now suppose that $f : \forall(t\,.\,t\,\texttt{list} \to t\,\texttt{list})$ is an arbitrary function of the stated type. Show that the elements of $f[\tau](l)$ are a subset of those of l. Thus f may only permute, replicate, or drop elements from its input list to obtain its output list.

Part VIII

Partiality and Recursive Types

Chapter 19

System PCF of Recursive Functions

We introduced the language **T** as a basis for discussing total computations, those for which the type system guarantees termination. The language **M** generalizes **T** to admit inductive and coinductive types, while preserving totality. In this chapter we introduce **PCF** as a basis for discussing partial computations, those that may not terminate when evaluated, even when they are well-typed. At first blush this may seem like a disadvantage, but as we shall see in Chapter 20 it admits greater expressive power than is possible in **T**.

The source of partiality in **PCF** is the concept of *general recursion*, which permits the solution of equations between expressions. The price for admitting solutions to all such equations is that computations may not terminate—the solution to some equations might be undefined (divergent). In **PCF** the programmer must make sure that a computation terminates; the type system does not guarantee it. The advantage is that the termination proof need not be embedded into the code itself, resulting in shorter programs.

For example, consider the equations

$$\begin{aligned} f(0) &\triangleq 1 \\ f(n+1) &\triangleq (n+1) \times f(n). \end{aligned}$$

Intuitively, these equations define the factorial function. They form a system of simultaneous equations in the unknown f which ranges over functions on the natural numbers. The function we seek is a *solution* to these equations—a specific function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied.

A solution to such a system of equations is a fixed point of an associated functional (higher-order function). To see this, let us re-write these equations in another form:

$$f(n) \triangleq \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Re-writing yet again, we seek f given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Now define the *functional* F by the equation $F(f) = f'$, where f' is given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Note well that the condition on f' is expressed in terms of f, the argument to the functional F, and not in terms of f' itself! The function f we seek is a *fixed point* of F, a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words f is defined to be $\mathit{fix}(F)$, where *fix* is a higher-order operator on functionals F that computes a fixed point for it.

Why should an operator such as F have a fixed point? The key is that functions in **PCF** are *partial*, which means that they may diverge on some (or even all) inputs. Consequently, a fixed point of a functional F is the limit of a series of approximations of the desired solution obtained by iterating F. Let us say that a partial function ϕ on the natural numbers, is an *approximation* to a total function f if $\phi(m) = n$ implies that $f(m) = n$. Let $\perp\colon \mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\perp(n)$ is undefined for every $n \in \mathbb{N}$. 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 to $\phi' = F(\phi)$. The partial function ϕ' is defined on 0 and on $m + 1$ for every $m \geq 0$ on which ϕ is defined. Continuing, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ', and hence a further improvement on ϕ. If we start with $\perp$ as the initial approximation to f, then pass to the limit

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

we will obtain the least approximation to f that is defined for every $m \in \mathbb{N}$, and hence is the function f itself. Turning this around, if the limit exists, it is the solution we seek.

Because this construction works for any functional F, we conclude that *all* such operators have fixed points, and hence that *all* systems of equations such as the one given above have solutions. The solution is given by general recursion, but there is no guarantee that it is a total function (defined on all elements of its domain). For the above example it happens to be true, because we can prove by induction that this is so, but in general the solution is a partial function that may diverge on some inputs. It is our task as programmers to ensure that the functions defined by general recursion are total, or at least that we have a grasp of those inputs for which it is well-defined.

19.1 Statics

The syntax of **PCF** 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
			$\mathtt{ifz}[e_0;x.e_1](e)$	$\mathtt{ifz}\,e\,\{\mathtt{z}\hookrightarrow e_0 \mid \mathtt{s}(x)\hookrightarrow e_1\}$	zero test
			$\lambda[\tau](x.e)$	$\lambda(x:\tau)\,e$	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 *general recursion*; it is discussed in more detail below. The expression $\mathtt{ifz}[e_0;x.e_1](e)$ 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 **PCF** is inductively defined by the following rules:

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \tag{19.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z}:\mathtt{nat}} \tag{19.1b}$$

$$\frac{\Gamma \vdash e:\mathtt{nat}}{\Gamma \vdash \mathtt{s}(e):\mathtt{nat}} \tag{19.1c}$$

$$\frac{\Gamma \vdash e:\mathtt{nat} \quad \Gamma \vdash e_0:\tau \quad \Gamma, x:\mathtt{nat} \vdash e_1:\tau}{\Gamma \vdash \mathtt{ifz}[e_0;x.e_1](e):\tau} \tag{19.1d}$$

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda[\tau_1](x.e):\mathtt{parr}(\tau_1;\tau_2)} \tag{19.1e}$$

$$\frac{\Gamma \vdash e_1:\mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2):\tau} \tag{19.1f}$$

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e):\tau} \tag{19.1g}$$

Rule (19.1g) 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 statics.

Lemma 19.1. *If* $\Gamma, x:\tau \vdash e':\tau'$, $\Gamma \vdash e:\tau$, *then* $\Gamma \vdash \{e/x\}e':\tau'$.

19.2 Dynamics

The dynamics of **PCF** is defined by the judgments $e\ \mathsf{val}$, specifying the closed values, and $e \longmapsto e'$, specifying the steps of evaluation.

The judgment $e\ \mathsf{val}$ is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{19.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{19.2b}$$

$$\frac{}{\lambda[\tau](x.e)\ \mathsf{val}} \tag{19.2c}$$

The bracketed premise on rule (19.2b) is included for the *eager* interpretation of the successor operation, and omitted for the *lazy* interpretation. (See Chapter 36 for a further discussion of laziness.)

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

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{19.3a}$$

$$\frac{e \longmapsto e'}{\mathtt{ifz}[e_0; x.e_1](e) \longmapsto \mathtt{ifz}[e_0; x.e_1](e')} \tag{19.3b}$$

$$\frac{}{\mathtt{ifz}[e_0; x.e_1](\mathtt{z}) \longmapsto e_0} \tag{19.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}[e_0; x.e_1](\mathtt{s}(e)) \longmapsto \{e/x\}e_1} \tag{19.3d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{19.3e}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{19.3f}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\lambda[\tau](x.e); e_2) \longmapsto \{e_2/x\}e} \tag{19.3g}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \longmapsto \{\mathtt{fix}[\tau](x.e)/x\}e} \tag{19.3h}$$

The bracketed rule (19.3a) is included for an eager interpretation of the successor, and omitted otherwise. Bracketed rule (19.3f) and the bracketed premise on rule (19.3g) are included for a call-by-value interpretation, and omitted for a call-by-name interpretation, of function application. Rule (19.3h) implements self-reference by substituting the recursive expression itself for the variable x in its body; this is called *unwinding* the recursion.

Theorem 19.2 (Safety)**.**

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

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

Proof. The proof of preservation is by induction on the derivation of the transition judgment. Consider rule (19.3h). Suppose that $\mathtt{fix}[\tau](x.e) : \tau$. By inversion and substitution we have $\{\mathtt{fix}[\tau](x.e)/x\}e : \tau$, as required. The proof of progress proceeds by induction on the derivation of the typing judgment. For example, for rule (19.1g) the result follows because we may make progress by unwinding the recursion. □

It is easy to check that if e val, then e is irreducible in that there is no e' such that $e \longmapsto e'$. The safety theorem implies the converse, that an irreducible expression is a value, provided that it is closed and well-typed.

Definitional equality for the call-by-name variant of **PCF**, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{ifz}[e_0; x.e_1](\mathtt{z}) \equiv e_0 : \tau} \tag{19.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}[e_0; x.e_1](\mathtt{s}(e)) \equiv \{e/x\}e_1 : \tau} \tag{19.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}[\tau](x.e) \equiv \{\mathtt{fix}[\tau](x.e)/x\}e : \tau} \tag{19.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\lambda[\tau_1](x.e_2); e_1) \equiv \{e_1/x\}e_2 : \tau} \tag{19.4d}$$

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

19.3 Definability

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 $x : \tau_1 \rightharpoonup \tau_2$ standing for the function itself. The

dynamics of this construct is given by the axiom

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

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 are defined in **PCF** using fixed points, writing

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

for $\mathtt{fun}\, x(\ y{:}\tau_1\ ){:}\tau_2\ \mathtt{is}\, e$. We may easily check that the static and dynamics of recursive functions are derivable from this definition.

The primitive recursion construct of **T** is defined in **PCF** using recursive functions by taking the expression

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

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

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

The static and dynamics of primitive recursion are derivable in **PCF** using this expansion.

In general, functions definable in **PCF** are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in **PCF** 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 when it is applied.

It is informative to classify those partial functions ϕ that are definable in **PCF**. The *partial recursive functions* are defined to be the primitive recursive functions extended with 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 19.3. *A partial function ϕ on the natural numbers is definable in* **PCF** *iff it is partial recursive.*

Proof sketch. Minimization is definable in **PCF**, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with some tedium, define an evaluator for expressions of **PCF** as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Therefore **PCF** 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 that is or will ever be defined.[1] Therefore **PCF** is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

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

The *evaluator*, or *universal*, function ϕ_{univ} for **PCF** 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 **T**, the universal function ϕ_{univ} for **PCF** is partial (might be undefined for some inputs). It is, in essence, an interpreter that, given the code $\ulcorner e \urcorner$ of a closed expression of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, simulates the dynamics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. Because this process may fail to terminate, the universal function is not defined for all inputs.

By Church's Law the universal function is definable in **PCF**. In contrast, we proved in Chapter 9 that the analogous function is *not* definable in **T** using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 9.4, we may derive the equivalence

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

for **PCF**. But now, instead of concluding that the universal function, e_{univ}, does not exist as we did for **T**, we instead conclude for **PCF** that e_{univ} diverges on the code for e_Δ applied to its own code.

19.4 Finite and Infinite Data Structures

Finite data types (products and sums), including their use in pattern matching and generic programming, carry over verbatim to **PCF**. However, the distinction between the eager and lazy dynamics for these constructs becomes more important. Rather than being a matter of preference, the decision to use an eager or lazy dynamics affects the meaning of a program: the "same" types mean different things in a lazy dynamics than in an eager dynamics. For example, the elements of a product type in an eager language are pairs of values of the component types. In a lazy language they are instead pairs of unevaluated, possibly divergent, computations of the component types, a very different thing indeed. And similarly for sums.

The situation grows more acute for infinite types such as the type `nat` of "natural numbers." The scare quotes are warranted, because the "same" type has a very different meaning under an eager dynamics than under a lazy dynamics. In the former case the type `nat` is, indeed, the authentic type of natural numbers—the least type containing zero and closed under successor. The principle of mathematical induction is valid for reasoning about the type `nat` in an eager dynamics. It corresponds to the inductive type $\mu(t.\,\mathtt{unit} + t)$ in the sense of Chapter 15.

On the other hand, under a lazy dynamics the type `nat` is no longer the type of natural numbers at all. For example, it includes the value

$$\omega \triangleq \mathtt{fix}\, x : \mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x),$$

which has itself as predecessor! It is, intuitively, an "infinite stack of successors", growing without end. It is clearly not a natural number (it is larger than all of them), so the principle of mathematical induction does not apply. In a lazy setting `nat` could be renamed `lnat` to remind us of the distinction. It corresponds to the coinductive type $\nu(t.\,\mathtt{unit} + t)$ in the sense of Chapter 15.

19.5 Totality and Partiality

The advantage of a total programming language such as **T** is that it ensures, by type checking, that every program terminates, and that every function is total. There is no way to have a well-typed program that goes into an infinite loop. This prohibition may seem appealing, until one considers that the upper bound on the time to termination may be large, so large that it might as well diverge for all practical purposes. But let us grant for the moment that it is a virtue of **T** that it precludes divergence. Why, then, bother with a language such as **PCF** that does not rule out divergence? After all, infinite loops are invariably bugs, so why not rule them out by type checking? The notion seems appealing until one tries to write a program in a language such as **T**.

Consider the computation of the greatest common divisor (gcd) of two natural numbers. It can be programmed in **PCF** by solving the following equations using general recursion:

$$\begin{aligned} gcd(m,0) &= m \\ gcd(0,n) &= n \\ gcd(m,n) &= gcd(m-n,n) \quad \text{if } m > n \\ gcd(m,n) &= gcd(m,n-m) \quad \text{if } m < n \end{aligned}$$

The type of *gcd* defined this way is $(\,\mathtt{nat} \times \mathtt{nat}\,) \rightharpoonup \mathtt{nat}$, which suggests that it may not terminate for some inputs. But we may prove by induction on the sum of the pair of arguments that it is, in fact, a total function.

Now consider programming this function in **T**. It is, in fact, programmable using only primitive recursion, but the code to do it is rather painful (try it!). One way to see the problem is that in **T** the only form of looping is one that reduces a natural number by one on each recursive call; it is not (directly) possible to make a recursive call on a smaller number other than the immediate predecessor. In fact one may code up more general patterns of terminating recursion using only primitive recursion as a primitive, but if you check the details, you will see that doing so comes at a price in performance and program complexity. Program complexity can be mitigated by building libraries that codify standard patterns of reasoning whose cost of development should be amortized over all programs, not just one in particular. But there is still the problem of performance. Indeed, the encoding of more general forms of recursion into primitive recursion means that, deep within the encoding, there must be a "timer" that goes down by ones to ensure that the program terminates. The result will be that programs written with such libraries will be slower than necessary.

But, one may argue, **T** is simply not a serious language. A more serious total programming language would admit sophisticated patterns of control without performance penalty. Indeed, one could easily envision representing the natural numbers in binary, rather than unary, and allowing recursive calls by halving to get logarithmic complexity. Such a formulation is possible, as would be quite a number of analogous ideas that avoid the awkwardness of programming in **T**. Could we not then have a practical language that rules out divergence?

We can, but at a cost. We have already seen one limitation of total programming languages: they are not universal. You cannot write an interpreter for **T** within **T**, and this limitation extends to any total language whatever. If this does not seem important, then consider the *Blum Size Theorem (BST)*, which places another limitation on total languages. Fix *any* total language $\mathcal{L}$ that

permits writing functions on the natural numbers. Pick any blowup factor, say 2^{2^n}. The BST states that there is a total function on the natural numbers that is programmable in $\mathcal{L}$, but whose shortest program in $\mathcal{L}$ is larger by the given blowup factor than its shortest program in **PCF**!

The underlying idea of the proof is that *in a total language the proof of termination of a program must be baked into the code itself*, whereas *in a partial language the termination proof is an external verification condition left to the programmer*. There are, and always will be, programs whose termination proof is rather complicated to express, if you fix in advance the means of proving it total. (In **T** it was primitive recursion, but one can be more ambitious, yet still get caught by the BST.) But if you leave room for ingenuity, then programs can be short, because they do not have to embed the proof of their termination in their own running code.

19.6 Notes

The solution to recursion equations described here is based on Kleene's fixed point theorem for complete partial orders, specialized to the approximation ordering of partial functions. The language **PCF** is derived from Plotkin (1977) as a laboratory for the study of semantics of programming languages. Many authors have used PCF as the subject of study of many problems in semantics. It has thereby become the *E. coli* of programming languages.

Exercises

19.1. Consider the problem considered in Section 10.3 of how to define the mutually recursive "even" and "odd" functions. There we gave a solution in terms of primitive recursion. You are, instead, to give a solution in terms of general recursion. *Hint*: consider that a pair of mutually recursive functions is a recursive pair of functions.

19.2. Show that minimization, as explained before the statement of Theorem 19.3, is definable in **PCF**.

19.3. Consider the partial function ϕ_{halts} such that if $e : \mathtt{nat} \rightharpoonup \mathtt{nat}$, then $\phi_{halts}(\ulcorner e \urcorner)$ evaluates to zero iff $e(\ulcorner e \urcorner)$ converges, and evaluates to one otherwise. Prove that ϕ_{halts} is not definable in **PCF**.

19.4. Suppose that we changed the specification of minimization given prior to Theorem 19.3 so that $\psi(n)$ is the least m such that $\phi(m, n) = 0$, and is undefined if no such m exists. Is this "simplified" form of minimization definable in **PCF**?

19.5. Suppose that we wished to define, in the lazy variant of **PCF**, a version of the *parallel or* function specified as a function of two arguments that returns `z` if either of its arguments is `z`, and `s(z)` if both are non-zero. That is, we wish to find an expression e satisfying the

following properties:

$$e(e_1)(e_2) \longmapsto^* \mathtt{z} \text{ if } e_1 \longmapsto^* \mathtt{z}$$
$$e(e_1)(e_2) \longmapsto^* \mathtt{z} \text{ if } e_2 \longmapsto^* \mathtt{z}$$
$$e(e_1)(e_2) \longmapsto^* \mathtt{s}(\mathtt{z}) \text{ if } e_1 \longmapsto^* \mathtt{s}(\_) \text{ and } e_2 \longmapsto^* \mathtt{s}(\_).$$

Thus, e defines a total function of its two arguments, *even if* one of the arguments diverges, as long as the other is $\mathtt{z}$. Clearly such a function cannot be defined in the call-by-value variant of **PCF**, but can it be defined in the call-by-name variant? If so, show how; if not, prove that it cannot be, and suggest an extension of **PCF** that would allow it to be defined.

19.6. We appealed to Church's Law to argue that the universal function for **PCF** is definable in **PCF**. See what is behind this claim by considering two aspects of the problem: (1) Gödel-numbering, the representation of abstract syntax by a number; (2) evaluation, the process of interpreting a function on its inputs. Part (1) is a technical issue arising from the limited data structures available in **PCF**. Part (2) is the heart of the matter; explore its implementation in terms of a solution to Part (1).

Part IX

Dynamic Types

Part X

Subtyping

Part XI

Dynamic Dispatch

Part XII

Control Flow

Chapter 28

Control Stacks

Structural dynamics is convenient for proving properties of languages, such as a type safety theorem, but is less convenient as a guide for implementation. A structural dynamics defines a transition relation using rules that determine where to apply the next instruction without spelling out how to find where the instruction lies within an expression. To make this process explicit we introduce a mechanism, called a *control stack*, that records the work that remains to be done after an instruction is executed. Using a stack eliminates the need for premises on the transition rules so that the transition system defines an *abstract machine* whose steps are determined by information explicit in its state, much as a concrete computer does.

In this chapter we develop an abstract machine **K** for evaluating expressions in **PCF**. The machine makes explicit the context in which primitive instruction steps are executed, and the process by which the results are propagated to determine the next step of execution. We prove that **K** and **PCF** are equivalent in the sense that both achieve the same outcomes for the same expressions.

28.1 Machine Definition

A state s of the stack machine **K** for **PCF** consists of a *control stack* k and a closed expression e. States take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression e on a control stack k.

2. A *return* state of the form $k \triangleleft e$, where e val, corresponds to the evaluation of a stack k on 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 returned. Formally, a control

stack is a list of *frames*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{28.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k\,;f\ \mathsf{stack}} \tag{28.1b}$$

The frames of the **K** machine are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{28.2a}$$

$$\frac{}{\mathtt{ifz}[e_0\,;x\,.\,e_1\,](-)\ \mathsf{frame}} \tag{28.2b}$$

$$\frac{}{\mathtt{ap}(-\,;e_2\,)\ \mathsf{frame}} \tag{28.2c}$$

The frames correspond to search rules in the dynamics of **PCF**. Thus, instead of relying on the structure of the transition derivation to keep a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgment between states of the **PCF** machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers, using an eager dynamics for the successor.

$$\frac{}{k \triangleright \mathtt{z} \longmapsto k \triangleleft \mathtt{z}} \tag{28.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e\,) \longmapsto k\,;\mathtt{s}(-) \triangleright e} \tag{28.3b}$$

$$\frac{}{k\,;\mathtt{s}(-) \triangleleft e \longmapsto k \triangleleft \mathtt{s}(e\,)} \tag{28.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e\,)$, we push a frame on the stack to record the pending successor, and evaluate e; when that returns with e', we return $\mathtt{s}(e'\,)$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}[e_0\,;x\,.\,e_1\,](e\,) \longmapsto k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1\,](-) \triangleright e} \tag{28.4a}$$

$$\frac{}{k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1\,](-) \triangleleft \mathtt{z} \longmapsto k \triangleright e_0} \tag{28.4b}$$

$$\frac{}{k\,;\mathtt{ifz}[e_0\,;x\,.\,e_1\,](-) \triangleleft \mathtt{s}(e\,) \longmapsto k \triangleright \{e/x\}e_1} \tag{28.4c}$$

The test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression is determined, the zero or non-zero branch of the condition is evaluated, substituting the predecessor in the latter case.

Finally, we give the rules for functions, which are evaluated by-name, and the rule for general recursion.

$$\overline{k \rhd \lambda[\tau](x.e) \longmapsto k \lhd \lambda[\tau](x.e)} \tag{28.5a}$$

$$\overline{k \rhd \mathtt{ap}(e_1; e_2) \longmapsto k; \mathtt{ap}(-; e_2) \rhd e_1} \tag{28.5b}$$

$$\overline{k; \mathtt{ap}(-; e_2) \lhd \lambda[\tau](x.e) \longmapsto k \rhd \{e_2/x\}e} \tag{28.5c}$$

$$\overline{k \rhd \mathtt{fix}[\tau](x.e) \longmapsto k \rhd \{\mathtt{fix}[\tau](x.e)/x\}e} \tag{28.5d}$$

It is important that evaluation of a general recursion requires no stack space.

The initial and final states of the **K** machine are defined by the following rules:

$$\frac{}{\epsilon \rhd e \ \mathsf{initial}} \tag{28.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \lhd e \ \mathsf{final}} \tag{28.6b}$$

28.2 Safety

To define and prove safety for the **PCF** machine requires that we introduce a new typing judgment, $k \div \tau$, which states that the stack k expects a value of type τ. This judgment is inductively defined by the following rules:

$$\frac{}{\epsilon \div \tau} \tag{28.7a}$$

$$\frac{k \div \tau' \quad f : \tau \rightsquigarrow \tau'}{k; f \div \tau} \tag{28.7b}$$

This definition makes use of an auxiliary judgment, $f : \tau \rightsquigarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \rightsquigarrow \mathtt{nat}} \tag{28.8a}$$

$$\frac{e_0 : \tau \quad x : \mathtt{nat} \vdash e_1 : \tau}{\mathtt{ifz}[e_0; x.e_1](-) : \mathtt{nat} \rightsquigarrow \tau} \tag{28.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(\,-\,;e_2\,) : \mathtt{parr}(\,\tau_2\,;\tau\,) \rightsquigarrow \tau} \tag{28.8c}$$

The states of the **PCF** machine are well-formed if their stack and expression components match:

$$\frac{k \div \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{28.9a}$$

$$\frac{k \div \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{28.9b}$$

We leave the proof of safety of the **PCF** machine as an exercise.

Theorem 28.1 (Safety)**.** *1. If s ok and $s \longmapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \longmapsto s'$.

28.3 Correctness of the Stack Machine

Does evaluation of an expression e using the **K** machine yield the same result as does the structural dynamics of **PCF**? The answer to this question can be derived from the following facts.

Completeness If $e \longmapsto^* e'$, where e' val, then $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft e'$.

Soundness If $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft e'$, then $e \longmapsto^* e'$ with e' val.

To prove completeness a plausible first step is to consider a proof by induction on the definition of multi-step transition, which reduces the theorem to the following two lemmas:

1. If e val, then $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft e$.
2. If $e \longmapsto e'$, then, for every v val, if $\epsilon \triangleright e' \longmapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft v$.

The first can be proved easily by induction on the structure of e. The second requires an inductive analysis of the derivation of $e \longmapsto e'$ that gives rise to two complications. 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 **K** machine is

$$\epsilon \triangleright \mathtt{ap}(\,e_1\,;e_2\,) \longmapsto \epsilon\,;\mathtt{ap}(\,-\,;e_2\,) \triangleright e_1.$$

To handle such situations we consider the evaluation of e_1 on any stack, not just the empty stack.

Specifically, we prove that if $e \longmapsto e'$ and $k \triangleright e' \longmapsto^* k \triangleleft v$, then $k \triangleright e \longmapsto^* k \triangleleft v$. Reconsider the case $e = \mathtt{ap}(\,e_1\,;e_2\,)$, $e' = \mathtt{ap}(\,e_1'\,;e_2\,)$, with $e_1 \longmapsto e_1'$. We are given that $k \triangleright \mathtt{ap}(\,e_1'\,;e_2\,) \longmapsto^* k \triangleleft v$, and we are to show that $k \triangleright \mathtt{ap}(\,e_1\,;e_2\,) \longmapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \mathtt{ap}(\,e_1'\,;e_2\,) \longmapsto k\,;\mathtt{ap}(\,-\,;e_2\,) \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \longmapsto e_1'$, but to do so we need a value v_1 such that $e_1' \longmapsto^* v_1$, which is not at hand.

We therefore consider the value of each sub-expression of an expression. This information is given by the evaluation dynamics described in Chapter 7, which has the property that $e \Downarrow e'$ iff $e \longmapsto^* e'$ and e' val.

Lemma 28.2. *If $e \Downarrow v$, then for every k stack, $k \triangleright e \longmapsto^* k \triangleleft v$.*

The desired result follows by the analog of Theorem 7.2 for **PCF**, which states that $e \Downarrow v$ iff $e \longmapsto^* v$.

To prove soundness, we note that it is awkward to reason inductively about a multi-step transition from $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft v$. The intermediate steps could involve alternations of evaluation and return states. Instead we consider a **K** machine state to encode an expression, and show that the machine transitions are simulated by the transitions of the structural dynamics.

To do so we define a judgment, $s \leftarrow\!\!\shortmid e$, 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 $s \longmapsto^* s'$, where s' final, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then e' val and $e \longmapsto^* e'$. 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 $s \longmapsto s'$, $s \leftarrow\!\!\shortmid e$, $s' \leftarrow\!\!\shortmid e'$, and $e' \longmapsto^* v$, where v val, then $e \longmapsto^* v$.

The first is quite simple, we need only note that the unraveling of a final state is a value. For the second, it is enough to prove the following lemma.

Lemma 28.3. *If $s \longmapsto s'$, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then $e \longmapsto^* e'$.*

Corollary 28.4. *$e \longmapsto^* \overline{n}$ iff $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft \overline{n}$.*

28.3.1 Completeness

Proof of Lemma 28.2. The proof is by induction on an evaluation dynamics for **PCF**.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \lambda[\tau_2](x\,.\,e) \quad \{e_2/x\}e \Downarrow v}{\mathtt{ap}(e_1\,;e_2) \Downarrow v} \tag{28.10}$$

For an arbitrary control stack k we are to show that $k \triangleright \mathtt{ap}(e_1\,;e_2) \longmapsto^* k \triangleleft v$. Applying both of

the inductive hypotheses in succession, interleaved with steps of the **K** machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(\, e_1 \,;\, e_2 \,) &\longmapsto k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \triangleright e_1 \\ &\longmapsto^* k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \triangleleft \lambda[\, \tau_2 \,](\, x \,.\, e \,) \\ &\longmapsto k \triangleright \{e_2/x\}e \\ &\longmapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

28.3.2 Soundness

The judgment $s \mathrel{\rightarrow\!\!\!\!\!\!+}\; e'$, where s is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgment $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \mathrel{\rightarrow\!\!\!\!\!\!+}\; e'} \tag{28.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \mathrel{\rightarrow\!\!\!\!\!\!+}\; e'} \tag{28.11b}$$

In words, to unravel a state we wrap the stack around the expression to form a complete program. The unraveling relation is inductively defined by the following rules:

$$\frac{}{e \bowtie e = e} \tag{28.12a}$$

$$\frac{k \bowtie \mathtt{s}(\, e \,) = e'}{k \,;\, \mathtt{s}(\, - \,) \bowtie e = e'} \tag{28.12b}$$

$$\frac{k \bowtie \mathtt{ifz}[\, e_0 \,;\, x \,.\, e_1 \,](\, e \,) = e'}{k \,;\, \mathtt{ifz}[\, e_0 \,;\, x \,.\, e_1 \,](\, - \,) \bowtie e = e'} \tag{28.12c}$$

$$\frac{k \bowtie \mathtt{ap}(\, e_1 \,;\, e_2 \,) = e}{k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \bowtie e_1 = e} \tag{28.12d}$$

These judgments both define total functions.

Lemma 28.5. *The judgment $s \mathrel{\rightarrow\!\!\!\!\!\!+}\; e$ relates every state s to a unique expression e, and the judgment $k \bowtie e = e'$ relates every stack k and expression e to a unique expression e'.*

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 unraveling preserves the transition relation.

Lemma 28.6. *If $e \longmapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \longmapsto d'$.*

Proof. The proof is by rule induction on the transition $e \longmapsto e'$. The inductive cases, where the transition rule has a premise, follow easily by induction. The base cases, where 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 $e_1 \longmapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from rules (28.12) that $k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \bowtie e_1 = d$ and $k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \bowtie e_1' = d'$. So by induction $d \longmapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\, \lambda[\, \tau_2 \,](\, x \,.\, e \,) \,;\, e_2 \,)$ and $e' = \{e_2/x\}e$ with $e \longmapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \longmapsto d'$. 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 (28.12) that $k' \bowtie \mathtt{ap}(\, e \,;\, c_2 \,) = d$ and $k' \bowtie \mathtt{ap}(\, e' \,;\, c_2 \,) = d'$. But by the structural dynamics $\mathtt{ap}(\, e \,;\, c_2 \,) \longmapsto \mathtt{ap}(\, e' \,;\, c_2 \,)$, so by the inner inductive hypothesis we have $d \longmapsto d'$, as desired. □

We may now complete the proof of Lemma 28.3.

Proof of Lemma 28.3. The proof is by case analysis on the transitions of the **K** machine. In each case, after unraveling, the transition will correspond to zero or one transitions of the **PCF** structural dynamics.

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 $s = k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \triangleleft \lambda[\, \tau \,](\, x \,.\, e_1 \,)$, and $s' = k \triangleright \{e_2/x\}e_1$. Let e' be such that $k \,;\, \mathtt{ap}(\, - \,;\, e_2 \,) \bowtie \lambda[\, \tau \,](\, x \,.\, e_1 \,) = e'$ and let e'' be such that $k \bowtie \{e_2/x\}e_1 = e''$. Noting that $k \bowtie \mathtt{ap}(\, \lambda[\, \tau \,](\, x \,.\, e_1 \,) \,;\, e_2 \,) = e'$, the result follows from Lemma 28.6.

□

28.4 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 the SECD machine (Landin, 1965), which is a linearization of a structural operational semantics (Plotkin, 1981). The advantage of a machine model is that the explicit treatment of control is needed for languages that allow the control state to be manipulated (see Chapter 30 for a prime example). The disadvantage is that the control state of the computation must be made explicit, necessitating rules for manipulating it that are left implicit in a structural dynamics.

Exercises

28.1. Give the proof of Theorem 28.1 for conditional expressions.

28.2. Formulate a call-by-value variant of the **PCF** machine.

28.3. Analyze the worst-case asymptotic complexity of executing each instruction of the **K** machine.

28.4. Refine the proof of Lemma 28.2 by bounding the number of machine steps taken for each step of the **PCF** dynamics.

Chapter 29

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 signal the need for special handling in unusual circumstances. We could use conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly because the transfer to the handler is conceptually direct and immediate, rather than indirect via explicit checks.

In this chapter we will consider two extensions of **PCF** with exceptions. The first, **FPCF**, enriches **PCF** with the simplest form of exception, called a *failure*, with no associated data. A failure can be intercepted, and turned into a success (or another failure!) by transferring control to another expression. The second, **XPCF**, enriches **PCF** with *exceptions*, with associated data that is passed to an exception handler that intercepts it. The handler may analyze the associated data to determine how to recover from the exceptional condition. A key choice is to decide on the type of the data associated to an exception.

29.1 Failures

The syntax of **FPCF** is defined by the following extension of the grammar of **PCF**:

Exp	e	::=	`fail`	`fail`	signal a failure
			`catch`$(\, e_1 \,;\, e_2 \,)$	`catch` e_1 `ow` e_2	catch a failure

The expression `fail` aborts the current evaluation, and the expression `catch`$(\, e_1 \,;\, e_2 \,)$ catches any failure in e_1 by evaluating e_2 instead. Either e_1 or e_2 may themselves abort, or they may diverge or return a value as usual in **PCF**.

The statics of **FPCF** is given by these rules:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(\, e_1 \,;\, e_2 \,) : \tau} \tag{29.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, because either might determine the value of that expression.

The dynamics of **FPCF** is given using a technique called *stack unwinding*. Evaluation of a `catch` pushes a frame of the form $\texttt{catch}(\, - \,;\, e \,)$ onto the control stack that awaits the arrival of a failure. Evaluation of a `fail` expression pops frames from the control stack until it reaches a frame of the form $\texttt{catch}(\, - \,;\, e \,)$, at which point the frame is removed from the stack and the expression e is evaluated. Failure propagation is expressed by a state of the form $k \blacktriangleleft$, which extends the two forms of state considered in Chapter 28 to express failure propagation.

The **FPCF** machine extends the **PCF** machine with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail} \longmapsto k \blacktriangleleft} \tag{29.2a}$$

$$\frac{}{k \triangleright \texttt{catch}(\, e_1 \,;\, e_2 \,) \longmapsto k \,;\, \texttt{catch}(\, - \,;\, e_2 \,) \triangleright e_1} \tag{29.2b}$$

$$\frac{}{k \,;\, \texttt{catch}(\, - \,;\, e_2 \,) \triangleleft v \longmapsto k \triangleleft v} \tag{29.2c}$$

$$\frac{}{k \,;\, \texttt{catch}(\, - \,;\, e_2 \,) \blacktriangleleft \longmapsto k \triangleright e_2} \tag{29.2d}$$

$$\frac{(f \neq \texttt{catch}(\, - \,;\, e \,))}{k \,;\, f \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{29.2e}$$

Evaluating `fail` propagates a failure up the stack. The act of failing itself, `fail`, will, of course, give rise to a failure. Evaluating $\texttt{catch}(\, e_1 \,;\, e_2 \,)$ consists of pushing the handler on the control stack and evaluating e_1. If a value reaches to the handler, the handler is removed and the value is passed to the surrounding frame. If a failure reaches the handler, the stored expression is evaluated with the handler removed from the control stack. Failures propagate through all frames other than the `catch` frame.

The initial and final states of the **FPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \tag{29.3a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{29.3b}$$

$$\frac{}{\epsilon \blacktriangleleft \ \mathsf{final}} \tag{29.3c}$$

The definition of stack typing given in Chapter 28 can be extended to account for the new forms of frame so that safety can be proved in the same way as before. The only difference is that the statement of progress must be weakened to take account of failure: a well-typed expression is either a value, or may take a step, or may signal failure.

Theorem 29.1 (Safety for **FPCF**). *1. If s ok and $s \longmapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \longmapsto s'$.

29.2 Exceptions

The language **XPCF** enriches **FPCF** with *exceptions*, failures to which a value is attached. The syntax of **XPCF** extends that of **PCF** with the following forms of expression:

Exp	e	::=	$\mathtt{raise}(e)$	$\mathtt{raise}(e)$	raise an exception
			$\mathtt{try}(e_1\,;x\,.\,e_2)$	$\mathtt{try}\,e_1\,\mathtt{ow}\,x \hookrightarrow e_2$	handle an exception

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{try}(e_1\,;x\,.\,e_2)$ binds a variable x in the handler e_2. The associated value of the exception is bound to that variable within e_2, should an exception be raised when e_1 is evaluated.

The statics of exceptions extends the statics of failures to account for the type of the value carried with the exception:

$$\frac{\Gamma \vdash e : \mathtt{exn}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \tag{29.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \mathtt{exn} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}(e_1\,;x\,.\,e_2) : \tau} \tag{29.4b}$$

The type `exn` is some fixed, but as yet unspecified, type of exception values. (The choice of `exn` is discussed in Section 29.3.)

The dynamics of **XPCF** is similar to that of **FPCF**, except that the failure state $k \blacktriangleleft$ is replaced by the exception state $k \blacktriangleleft e$ which passes an exception value e to the stack k. There is only one notion of exception, but the associated value can be used to identify the source of the exception.

The stack frames of the **PCF** machine are extended to include $\mathtt{raise}(-)$ and $\mathtt{try}(-\,;x\,.\,e_2)$. These are used in the following rules:

$$\overline{k \triangleright \mathtt{raise}(e) \longmapsto k\,;\mathtt{raise}(-) \triangleright e} \tag{29.5a}$$

$$\overline{k\,;\mathtt{raise}(-) \triangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5b}$$

$$\overline{k \triangleright \mathtt{try}(e_1\,;x\,.\,e_2) \longmapsto k\,;\mathtt{try}(-\,;x\,.\,e_2) \triangleright e_1} \tag{29.5c}$$

$$\frac{}{k\,;\mathtt{try}(-\,;x\,.\,e_2\,)\triangleleft e \longmapsto k \triangleleft e} \tag{29.5d}$$

$$\frac{}{k\,;\mathtt{try}(-\,;x\,.\,e_2\,)\blacktriangleleft e \longmapsto k \triangleright \{e/x\}e_2} \tag{29.5e}$$

$$\frac{(f \neq \mathtt{try}(-\,;x\,.\,e_2\,))}{k\,;f\blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5f}$$

The main difference compared to Rules (29.2) is that an exception passes a values to the stack, whereas a failure does not.

The initial and final states of the **XPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \triangleright e\ \mathsf{initial}} \tag{29.6a}$$

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{29.6b}$$

$$\frac{e\ \mathsf{val}}{\epsilon \blacktriangleleft e\ \mathsf{final}} \tag{29.6c}$$

Theorem 29.2 (Safety for **XPCF**). *1. If s ok and $s \longmapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \longmapsto s'$.

29.3 Exception Values

The statics of **XPCF** is parameterized by the type `exn` of values associated to exceptions. The choice of `exn` is important because it determines how the source of an exception is identified in a program. If `exn` is the one-element type `unit`, then exceptions degenerate to failures, which are unable to identify their source. Thus `exn` must have more than one value to be useful.

This fact suggests that `exn` should be a finite sum. The classes of the sum identify the sources of exceptions, and the classified value carries information about the particular instance. For example, `exn` might be a sum type of the form

$$[\mathtt{div} \hookrightarrow \mathtt{unit}\,,\mathtt{fnf} \hookrightarrow \mathtt{string}\,,\ldots].$$

Here 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 that was not found.

Using a sum means that an exception handler can dispatch on the class of the exception value to identify its source and cause. For example, we might write

```
handle e1 ow x ↪
  match x {
    div ⟨⟩ ↪ e_div
  | fnf s ↪ e_fnf }
```

to handle the exceptions specified by the above sum type. Because the exception and its associated data are coupled in a sum type, there is no possibility of misinterpreting the data associated to one exception as being that of another.

The disadvantage of choosing a finite sum for `exn` is that it specifies a *closed world* of possible exception sources. All sources must be identified for the entire program, which impedes modular development and evolution. A more modular approach admits an *open world* of exception sources that can be introduced as the program evolves and even as it executes. A generalization of finite sums, called *dynamic classification*, defined in Chapter 33, is required for an open world. (See that Chapter for further discussion.)

When `exn` is a type of classified values, its classes are often called *exceptions*, so that one may speak of "the `fnf` exception" in the above example. This terminology is harmless, and all but unavoidable, but it invites confusion between two separate ideas:

1. Exceptions as a *control mechanism* that allows the course of evaluation to be altered by raising and handling exceptions.

2. Exceptions as a *data value* associated with such a deviation of control that allows the source of the deviation to be identified.

As a control mechanism exceptions can be eliminated using explicit *exception passing*. A computation of type τ that may raise an exception is interpreted as an exception-free computation of type $\tau + \texttt{exn}$.

29.4 Notes

Various forms of exceptions were considered in Lisp (Steele, 1990). The original formulation of ML (Gordon et al., 1979) as a metalanguage for mechanized logic used failures to implement backtracking proof search. Most modern languages now have exceptions, but differ in the forms of data that may be associated with them.

Exercises

29.1. Prove Theorem 29.2. Are any properties of `exn` required for the proof?

29.2. Give an evaluation dynamics for **XPCF** using the following judgment forms:

- Normal evaluation: $e \Downarrow v$, where $e : \tau$, $v : \tau$, and v val.
- Exceptional evaluation: $e \Uparrow v$, where $e : \tau$, and $v : \texttt{exn}$, and v val.

The first states that e evaluates normally to value v, the second that e raises an exception with value v.

29.3. Give a structural operational dynamics to **XPCF** by inductively defining the following judgment forms:

- $e \longmapsto e'$, stating that expression e transitions to expression e';
- e val, stating that expression e is a value.

Ensure that $e \Downarrow v$ iff $e \longmapsto^* v$, and $e \Uparrow v$ iff $e \longmapsto^* \texttt{raise}(v)$, where v val in both cases.

Part XIII

Symbolic Data

Part XIV

Mutable State

Chapter 34

Modernized Algol

Modernized Algol, or **MA**, is an imperative, block-structured programming language based on the classic language Algol. **MA** extends **PCF** 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 are combined by sequencing, and are iterated using recursion.

MA 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. The segregation of pure from impure ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, so that they can be manipulated just as in **PCF**. Commands, on the other hand, have a constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of **MA** 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. Stack allocation avoids the need for more complex forms of storage management, at the cost of reducing the expressive power of the language.

34.1 Basic Commands

The syntax of the language **MA** of modernized Algol distinguishes pure *expressions* from impure *commands*. The expressions include those of **PCF** (as described in Chapter 19), augmented with one construct, and the commands are those of a simple imperative programming language based on assignment. The language maintains a sharp distinction between *variables* and *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 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 34.3 for the general case.)

The syntax of **MA** is given by the following grammar, from which we have omitted repetition of the expression syntax of **PCF** for the sake of brevity.

Typ	τ	$::=$	$\mathtt{cmd}$	$\mathtt{cmd}$	command
Exp	e	$::=$	$\mathtt{cmd}(\,m\,)$	$\mathtt{cmd}\,m$	encapsulation
Cmd	m	$::=$	$\mathtt{ret}(\,e\,)$	$\mathtt{ret}\,e$	return
			$\mathtt{bnd}(\,e\,;x\,.\,m\,)$	$\mathtt{bnd}\,x \leftarrow e\,;m$	sequence
			$\mathtt{dcl}(\,e\,;a\,.\,m\,)$	$\mathtt{dcl}\,a := e\,\mathtt{in}\,m$	new assignable
			$\mathtt{get}[\,a\,]$	$@\,a$	fetch
			$\mathtt{set}[\,a\,](\,e\,)$	$a := e$	assign

The expression $\mathtt{cmd}(\,m\,)$ consists of the unevaluated command m thought of as a value of type $\mathtt{cmd}$. The command $\mathtt{ret}(\,e\,)$ returns the value of the expression e without having any effect on the assignables. The command $\mathtt{bnd}(\,e\,;x\,.\,m\,)$ evaluates e to an encapsulated command, then this command is executed for its effects on assignables, with its value substituted for x in m. The command $\mathtt{dcl}(\,e\,;a\,.\,m\,)$ introduces a new assignable, a, for use within the command m whose initial contents is given by the expression e. The command $\mathtt{get}[\,a\,]$ returns the current contents of the assignable a and the command $\mathtt{set}[\,a\,](\,e\,)$ changes the contents of the assignable a to the value of e, and returns that value.

34.1.1 Statics

The statics of **MA** consists of two forms of judgment:

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. As with other uses of symbols, the signature cannot be interpreted as a form of typing hypothesis (it enjoys no structural properties of entailment), but must be considered as an index of a family of judgments, one for each choice of Σ.

The statics of **MA** is inductively defined by the following rules:

$$\frac{\Gamma \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{cmd}(\,m\,) : \mathtt{cmd}} \tag{34.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat}}{\Gamma \vdash_\Sigma \mathtt{ret}(\,e\,)\ \mathsf{ok}} \tag{34.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd} \quad \Gamma, x : \mathtt{nat} \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{bnd}(\,e\,;x\,.\,m\,)\ \mathsf{ok}} \tag{34.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat} \quad \Gamma \vdash_{\Sigma,a} m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{dcl}(\,e\,;a\,.\,m\,)\ \mathsf{ok}} \tag{34.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \mathtt{get}[a]\ \mathsf{ok}} \tag{34.1e}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \mathtt{nat}}{\Gamma \vdash_{\Sigma,a} \mathtt{set}[a](e)\ \mathsf{ok}} \tag{34.1f}$$

Rule (34.1a) is the introduction rule for the type `cmd`, and rule (34.1c) is the corresponding elimination form. Rule (34.1d) introduces a new assignable for use within a specified command. The name a of the assignable is bound by the declaration, and so may be renamed to satisfy the implicit constraint that it not already occur in Σ. Rule (34.1e) leaves implicit that the command to retrieve the contents of an assignable a returns a natural number, as all do. Rule (34.1f) states that we may assign a natural number to an assignable.

34.1.2 Dynamics

The dynamics of **MA** 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 judgment forms:

1. $e\ \mathsf{val}_\Sigma$, stating that e is a value relative to Σ.
2. $e \underset{\Sigma}{\longmapsto} e'$, stating that the expression e steps to the expression e'.

These judgments are inductively defined by the following rules, together with the rules defining the dynamics of **PCF** (see Chapter 19). It is important, however, that the successor operation be given an *eager*, instead of *lazy*, dynamics so that a closed value of type `nat` is a numeral (for reasons that will be explained in Section 34.3).

$$\frac{}{\mathtt{cmd}(m)\ \mathsf{val}_\Sigma} \tag{34.2a}$$

Rule (34.2a) states that an encapsulated command is a value.

The dynamics of commands is defined in terms of states $\mu \parallel m$, where μ is a memory mapping assignables to values, and m is a command. There are two judgments governing such states:

1. $\mu \parallel m\ \mathsf{final}_\Sigma$. The state $\mu \parallel m$ is complete.
2. $\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$. The state $\mu \parallel m$ steps to the state $\mu' \parallel m'$; the set of active assignables is given by the signature Σ.

These judgments are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\mu \parallel \mathtt{ret}(e)\ \mathsf{final}_\Sigma} \tag{34.3a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mu \parallel \mathtt{ret}(e) \underset{\Sigma}{\longmapsto} \mu \parallel \mathtt{ret}(e')} \tag{34.3b}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mu \parallel \mathtt{bnd}(e\,;x\,.\,m) \underset{\Sigma}{\longmapsto} \mu \parallel \mathtt{bnd}(e'\,;x\,.\,m)} \tag{34.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mu \parallel \mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e))\,;x\,.\,m) \underset{\Sigma}{\longmapsto} \mu \parallel \{e/x\}m} \tag{34.3d}$$

$$\frac{\mu \parallel m_1 \underset{\Sigma}{\longmapsto} \mu' \parallel m_1'}{\mu \parallel \mathtt{bnd}(\mathtt{cmd}(m_1)\,;x\,.\,m_2) \underset{\Sigma}{\longmapsto} \mu' \parallel \mathtt{bnd}(\mathtt{cmd}(m_1')\,;x\,.\,m_2)} \tag{34.3e}$$

$$\frac{}{\mu \otimes a \hookrightarrow e \parallel \mathtt{get}[a] \underset{\Sigma,a}{\longmapsto} \mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(e)} \tag{34.3f}$$

$$\frac{e \underset{\Sigma,a}{\longmapsto} e'}{\mu \parallel \mathtt{set}[a](e) \underset{\Sigma,a}{\longmapsto} \mu \parallel \mathtt{set}[a](e')} \tag{34.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a}}{\mu \otimes a \hookrightarrow \_ \parallel \mathtt{set}[a](e) \underset{\Sigma,a}{\longmapsto} \mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(e)} \tag{34.3h}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mu \parallel \mathtt{dcl}(e\,;a\,.\,m) \underset{\Sigma}{\longmapsto} \mu \parallel \mathtt{dcl}(e'\,;a\,.\,m)} \tag{34.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad \mu \otimes a \hookrightarrow e \parallel m \underset{\Sigma,a}{\longmapsto} \mu' \otimes a \hookrightarrow e' \parallel m'}{\mu \parallel \mathtt{dcl}(e\,;a\,.\,m) \underset{\Sigma}{\longmapsto} \mu' \parallel \mathtt{dcl}(e'\,;a\,.\,m')} \tag{34.3j}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad e'\ \mathsf{val}_{\Sigma,a}}{\mu \parallel \mathtt{dcl}(e\,;a\,.\,\mathtt{ret}(e')) \underset{\Sigma}{\longmapsto} \mu \parallel \mathtt{ret}(e')} \tag{34.3k}$$

Rule (34.3a) specifies that a `ret` command is final if its argument is a value. Rules (34.3c) to (34.3e) specify the dynamics of sequential composition. The expression e must, by virtue of the type system, evaluate to an encapsulated command, which is executed to find its return value, which is then substituted into the command m before executing it.

Rules (34.3i) to (34.3k) define the concept of *block structure* in a programming language. Declarations adhere to the *stack discipline* in that an assignable is allocated during 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 way as their static scopes are nested inside one another. The stack-like behavior of assignables is a characteristic feature of what are known as *Algol-like languages*.

34.1.3 Safety

The judgment $\mu \parallel m \ \mathsf{ok}_\Sigma$ is defined by the rule

$$\frac{\vdash_\Sigma m \ \mathsf{ok} \quad \mu : \Sigma}{\mu \parallel m \ \mathsf{ok}_\Sigma} \tag{34.4}$$

where the auxiliary judgment $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e \ \mathsf{val}_\varnothing \text{ and } \vdash_\varnothing e : \mathtt{nat}}{\mu : \Sigma} \tag{34.5}$$

That is, the memory must bind a number to each assignable in Σ.

Theorem 34.1 (Preservation)**.**

1. *If* $e \xmapsto[\Sigma]{} e'$ *and* $\vdash_\Sigma e : \tau$, *then* $\vdash_\Sigma e' : \tau$.

2. *If* $\mu \parallel m \xmapsto[\Sigma]{} \mu' \parallel m'$, *with* $\vdash_\Sigma m \ \mathit{ok}$ *and* $\mu : \Sigma$, *then* $\vdash_\Sigma m' \ \mathit{ok}$ *and* $\mu' : \Sigma$.

Proof. Simultaneously, by induction on rules (34.2) and (34.3).

Consider rule (34.3j). Assume that $\vdash_\Sigma \mathtt{dcl}(e\,;a\,.\,m)\ \mathsf{ok}$ and $\mu : \Sigma$. By inversion of typing we have $\vdash_\Sigma e : \mathtt{nat}$ and $\vdash_{\Sigma,a} m \ \mathsf{ok}$. Because $e \ \mathsf{val}_\Sigma$ and $\mu : \Sigma$, we have $\mu \otimes a \hookrightarrow e : \Sigma, a$. By induction we have $\vdash_{\Sigma,a} m' \ \mathsf{ok}$ and $\mu' \otimes a \hookrightarrow e' : \Sigma, a$, from which the result follows immediately.

Consider rule (34.3k). Assume that $\vdash_\Sigma \mathtt{dcl}(e\,;a\,.\,\mathtt{ret}(e'))\ \mathsf{ok}$ and $\mu : \Sigma$. By inversion we have $\vdash_\Sigma e : \mathtt{nat}$, and $\vdash_{\Sigma,a} \mathtt{ret}(e')\ \mathsf{ok}$, and so $\vdash_{\Sigma,a} e' : \mathtt{nat}$. But because $e' \ \mathsf{val}_{\Sigma,a}$, and e' is a numeral, and we also have $\vdash_\Sigma e' : \mathtt{nat}$, as required. □

Theorem 34.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* $e \ \mathit{val}_\Sigma$, *or there exists* e' *such that* $e \xmapsto[\Sigma]{} e'$.

2. *If* $\vdash_\Sigma m \ \mathit{ok}$ *and* $\mu : \Sigma$, *then either* $\mu \parallel m \ \mathit{final}_\Sigma$ *or* $\mu \parallel m \xmapsto[\Sigma]{} \mu' \parallel m'$ *for some* μ' *and* m'.

Proof. Simultaneously, by induction on rules (34.1). Consider rule (34.1d). By the first inductive hypothesis we have either $e \xmapsto[\Sigma]{} e'$ or $e \ \mathsf{val}_\Sigma$. In the former case rule (34.3i) applies. In the latter, we have by the second inductive hypothesis,

$$\mu \otimes a \hookrightarrow e \parallel m \ \mathsf{final}_{\Sigma,a} \quad \text{or} \quad \mu \otimes a \hookrightarrow e \parallel m \xmapsto[\Sigma,a]{} \mu' \otimes a \hookrightarrow e' \parallel m'.$$

In the former case we apply rule (34.3k), and in the latter, rule (34.3j). □

34.2 Some Programming Idioms

The language **MA** 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 **MA**.

We define the *sequential composition* of commands, written $\{x \leftarrow m_1 \,;\, m_2\}$, to stand for the command $\texttt{bnd}\, x \leftarrow \texttt{cmd}\,(\, m_1 \,)\,;\, m_2$. Binary composition readily 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 $\{\_ \leftarrow m_1 \,;\, m_2\}$ where the returned value from m_1 is ignored; this generalizes in the obvious way to an n-ary form.

A related idiom, the command $\texttt{do}\, e$, executes an encapsulated command and returns its result. By definition $\texttt{do}\, e$ stands for the command $\texttt{bnd}\, x \leftarrow e\,;\, \texttt{ret}\, x$.

The *conditional* command $\texttt{ifz}\,(\, m \,)\, m_1\, \texttt{else}\, m_2$ executes either m_1 or m_2 according to whether the result of executing m is zero or not:

$$\{x \leftarrow m \,;\, \texttt{do}\,(\ \texttt{ifz}\, x\, \{\texttt{z} \hookrightarrow \texttt{cmd}\, m_1 \mid \texttt{s}(\_) \hookrightarrow \texttt{cmd}\, m_2\}\ )\}.$$

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:

$$\texttt{do}\,(\ \texttt{fix}\, \mathit{loop} : \texttt{cmd}\, \texttt{is}\, \texttt{cmd}\,(\ \texttt{ifz}\,(\, m_1 \,)\, \{\texttt{ret}\, \texttt{z}\}\, \texttt{else}\, \{m_2 \,;\, \texttt{do}\, \mathit{loop}\}\ )\ ).$$

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 $\tau \rightharpoonup \texttt{cmd}$ that takes an argument of some type τ and yields an unexecuted command as result. Many procedures have the form $\lambda\,(\, x : \tau \,)\, \texttt{cmd}\, m$, which we abbreviate to $\texttt{proc}\,(\, x : \tau \,)\, m$. 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 $\texttt{call}\, e_1(\, e_2 \,)$ is defined to be the command $\texttt{do}\,(\, e_1(\, e_2 \,)\,)$, which immediately runs the result of applying e_1 to e_2.

As an example, here is a procedure of type $\texttt{nat} \rightharpoonup \texttt{cmd}$ that returns the factorial of its argument:

```
proc (x:nat) {
  dcl r := 1 in
  dcl a := x in
  { while ( @ a ) {
      y ← @ r
    ; z ← @ a
    ; r := (x-z+1)× y
    ; a := z-1
    }
    ; x ← @ r
    ; ret x
  }
}
```

The loop maintains the invariant that the contents of `r` is the factorial of the quantity 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.

34.3 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 returned and assignable values of other types may be uncovered by a close examination of the proof of Theorem 34.1. For the proof to go through it is crucial that values of type `nat`, the type of assignables and return values, cannot contain an assignable, for otherwise the embedded assignable would escape the scope of its declaration. This property is self-evidently true for eagerly evaluated natural numbers, but fails when they are evaluated lazily. Thus the safety of **MA** hinges on the evaluation order for the successor operation, in contrast to most other situations where either interpretation is also safe.

When extending **MA** to admit assignables and returned values of other types, it is necessary to pay close attention to whether assignables can be embedded in a value of a candidate type. For example, if return values of procedure type are allowed, then the following command violates safety:

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \{\texttt{ret}\, (\, \texttt{proc}\, (\, x : \texttt{nat}\,)\, \{a := x\}\,)\}.$$

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 procedure type is also unsound. For example, suppose that b is an assignable whose contents are of type $\texttt{nat} \rightharpoonup \texttt{cmd}$, and consider the command

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \{b := \texttt{proc}\, (\, x : \texttt{nat}\,)\, \{a := x\}\, ; \texttt{ret}\, \texttt{z}\}.$$

We assign to b 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 that return values other than `nat` and to admit assignables with contents of types other than `nat`, we must rework the statics of **MA** 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 mobile type to each active assignable so that Σ has the form of a finite set of assumptions of the form $a \sim \tau$, where a is an assignable. Second, we replace the judgment $\Gamma \vdash_\Sigma m$ ok by the more general form $\Gamma \vdash_\Sigma m \div \tau$, stating that m is a well-formed command returning a value of type τ. Third, the type `cmd` is 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 34.1.1 is generalized to admit typed commands and typed assignables as follows:

$$\frac{\Gamma \vdash_\Sigma m \div \tau}{\Gamma \vdash_\Sigma \texttt{cmd}(m) : \texttt{cmd}(\tau)} \tag{34.6a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{ret}(e) \div \tau} \tag{34.6b}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\tau) \quad \Gamma, x : \tau \vdash_\Sigma m \div \tau'}{\Gamma \vdash_\Sigma \texttt{bnd}(e; x.m) \div \tau'} \tag{34.6c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \text{ mobile} \quad \Gamma \vdash_{\Sigma, a \sim \tau} m \div \tau' \quad \tau' \text{ mobile}}{\Gamma \vdash_\Sigma \texttt{dcl}(e; a.m) \div \tau'} \tag{34.6d}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{get}[a] \div \tau} \tag{34.6e}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{set}[a](e) \div \tau} \tag{34.6f}$$

Apart from the generalization to track returned types and content types, the most important change is that in Rule (34.6d) both the type of a declared assignable and the return type of the declaration is required to be *mobile*. The definition of the judgment τ mobile is guided by the following *mobility condition*:

$$\textit{if } \tau \text{ mobile}, \vdash_\Sigma e : \tau \textit{ and } e \text{ val}_\Sigma, \textit{ then } \vdash_\varnothing e : \tau \textit{ and } e \text{ val}_\varnothing. \tag{34.7}$$

That is, a value of mobile type may not depend on any active assignables.

As long as the successor operation is evaluated eagerly, the type `nat` is mobile:

$$\frac{}{\texttt{nat mobile}} \tag{34.8}$$

Similarly, a product of mobile types may safely be deemed mobile, if pairs are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 \times \tau_2 \text{ mobile}} \tag{34.9}$$

And the same goes for sums, if the injections are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 + \tau_2 \text{ mobile}} \tag{34.10}$$

In each of these cases 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.

Because the body of a procedure may involve an assignable, no procedure type is mobile, nor is any command type. What about function types other than procedure types? We may think they are mobile, because a pure expression cannot depend on an assignable. Although this is the case, the mobility condition need not hold. For example, consider the following value of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$:

$$\lambda\,(\,x:\mathtt{nat}\,)\,(\;\lambda\,(\,\_:\tau\,\mathtt{cmd}\,)\,\mathtt{z}\,)(\;\mathtt{cmd}\,\{@\,a\}\,).$$

Although the assignable a is not actually needed to compute the result, it nevertheless occurs in the value, violating the mobility condition.

The mobility restriction on the statics of declarations ensures that the type associated to an assignable is always mobile. We may therefore assume, without loss of generality, that the types associated to the assignables in the signature Σ are mobile.

Theorem 34.3 (Preservation for Typed Commands)**.**

1. *If* $e \longmapsto_{\Sigma} e'$ *and* $\vdash_{\Sigma} e : \tau$*, then* $\vdash_{\Sigma} e' : \tau$*.*
2. *If* $\mu \parallel m \longmapsto_{\Sigma} \mu' \parallel m'$*, with* $\vdash_{\Sigma} m \div \tau$ *and* $\mu : \Sigma$*, then* $\vdash_{\Sigma} m' \div \tau$ *and* $\mu' : \Sigma$*.*

Theorem 34.4 (Progress for Typed Commands)**.**

1. *If* $\vdash_{\Sigma} e : \tau$*, then either* e val_{Σ}*, or there exists* e' *such that* $e \longmapsto_{\Sigma} e'$*.*
2. *If* $\vdash_{\Sigma} m \div \tau$ *and* $\mu : \Sigma$*, then either* $\mu \parallel m$ final_{Σ} *or* $\mu \parallel m \longmapsto_{\Sigma} \mu' \parallel m'$ *for some* μ' *and* m'*.*

The proofs of Theorems 34.3 and 34.4 follows very closely the proof of Theorems 34.1 and 34.2. The main difference is that we appeal to the mobility condition to ensure that returned values and stored values are independent of the active assignables.

34.4 Notes

Modernized Algol is a derivative of Reynolds's Idealized Algol (Reynolds, 1981). In contrast to Reynolds's formulation, Modernized Algol maintains a separation between computations that depend on the memory and those that do not, and does not rely on call-by-name for function application, but rather has a type of encapsulated commands that can be used where call-by-name

would otherwise be required. The modal distinction between expressions and commands was present in the original formulation of Algol 60, but is developed here in using the concept of monadic effects introduced by Moggi (1989). Its role in functional programming was emphasized by Wadler (1992). The modal separation in **MA** is adapted directly from Pfenning and Davies (2001), which stresses the connection to lax modal logic.

What are called *assignables* here are invariably called *variables* elsewhere. The distinction between variables and assignables is blurred in languages that allow assignables as forms of expression. (Indeed, Reynolds himself (personal communication, 2012) regards this as a defining feature of Algol, in opposition to the formulation given here.) In **MA** we choose to make the distinction between variables, which are given meaning by substitution, and assignables, which are given meaning by mutation. Drawing this distinction requires new terminology; the term *assignable* seems apt for the imperative programming concept.

The concept of mobility of a type was introduced in the ML5 language for distributed computing (Murphy et al., 2004), with the similar meaning that a value of a mobile type cannot depend on local resources. Here the mobility restriction is used to ensure that the language adheres to the stack discipline.

Exercises

34.1. Originally Algol had both *scalar* assignables, whose contents are atomic values, and *array* assignables, which is a finite sequence of scalar assignables. Like scalar assignables, array assignables are stack-allocated. Extend **MA** with array assignables, ensuring that the language remains type safe, but allowing that computation may abort if a non-existent array element is accessed.

34.2. Consider carefully the behavior of assignable declarations within recursive procedures, as in the following expression

$$\texttt{fix}\, p\, \texttt{is}\, \lambda\, (\, x : \tau\,)\, \texttt{dcl}\, a := e\, \texttt{in}\, \texttt{cmd}(\, m\,)$$

of type $\tau \rightharpoonup \rho\, \texttt{cmd}$ for some ρ. Because p is recursive, the body m of the procedure may call itself during its execution, causing the *same* declaration to be executed more than once. Explain the dynamics of getting and setting a in such a situation.

34.3. Originally Algol considered assignables as expressions that stand for their contents in memory. Thus, if a is an assignable containing a number, one could write expressions such as $a + a$ that would evaluate to twice the contents of a. Moreover, one could write commands such as $a := a + a$ to double the contents of a. These conventions encouraged programmers to think of assignables as variables, quite the opposite of their separation in **MA**. This convention, combined with an over-emphasis on concrete syntax, led to a conundrum about the different roles of a in the above assignment command: its meaning on the left of the assignment is different from its meaning on the right. These came to be called the *left-*, or *l-value*, and the *right-*, or *r-value* of the assignable a, corresponding to its position in the assignment statement. When viewed as abstract syntax, though, there is no ambiguity to be explained:

the assignment operator is indexed by its target assignable, instead of taking as argument an expression that happens to be an assignable, so that the command is $\texttt{set}[a](a+a)$, not $\texttt{set}(a;a+a)$.

This still leaves the puzzle of how to regard assignables as forms of expression. As a first cut, reformulate the dynamics of **MA** to account for this. Reformulate the dynamics of expressions in terms of the judgments $\mu \parallel e \underset{\Sigma}{\longmapsto} \mu' \parallel e'$ and $\mu \parallel e$ final that allow evaluation of e to depend on the contents of the memory. Each use of an assignable as an expression should require one access to the memory. Then prove *memory invariance*:: if $\mu \parallel e \underset{\Sigma}{\longmapsto} \mu' \parallel e'$, then $\mu' = \mu$.

A natural generalization is to allow any sequence of commands to be considered as an expression, if they are all *passive* in the sense that no assignments are allowed. Write $\texttt{do}\,\{m\}$, where m is a passive command, for a *passive block* whose evaluation consists of executing the command m on the current memory, using its return value as the value of the expression. Observe that memory invariance holds for passive blocks.

The use of an assignable a as an expression may now be rendered as the passive block $\texttt{do}\,\{@a\}$. More complex uses of assignables as expressions admit several different interpretations using passive blocks. For example, an expression such as $a + a$ might be rendered in one of two ways:

(a) $\texttt{do}\,\{@a\} + \texttt{do}\,\{@a\}$, or

(b) $\texttt{let}\,x\,\texttt{be}\,\texttt{do}\,\{@a\}\,\texttt{in}\,x + x$.

The latter formulation accesses a only once, but uses its value twice. Comment on there being two different interpretations of $a + a$.

34.4. Recursive procedures in Algol are *declared* using a command of the form $\texttt{proc}\,p(x:\tau) : \rho\,\texttt{is}\,m\,\texttt{in}\,m'$, which is governed by the typing rule

$$\frac{\Gamma, p:\tau \rightharpoonup \rho\,\texttt{cmd}, x:\tau \vdash_\Sigma m \div \rho \quad \Gamma, p:\tau \rightharpoonup \rho\,\texttt{cmd} \vdash_\Sigma m' \div \tau'}{\Gamma \vdash_\Sigma \texttt{proc}\,p(x:\tau) : \rho\,\texttt{is}\,m\,\texttt{in}\,m' \div \tau'}\,. \tag{34.11}$$

From the present viewpoint it is peculiar to insist on declaring procedures at all, because they are simply values of procedure type, and even more peculiar to insist that they be confined for use within a command. One justification for this limitation, though, is that Algol included a peculiar feature, called an *own variable*[1] that was declared for use within the procedure, but whose state persisted across calls to the procedure. One application would be to a procedure that generated pseudo-random numbers based on a stored seed that influenced the behavior of successive calls to it. Give a formulation in **MA** of the extended declaration

$$\texttt{proc}\,p(x:\tau) : \rho\,\texttt{is}\,\{\texttt{own}\,a := e\,\texttt{in}\,m\}\,\texttt{in}\,m'$$

[1] That is to say, an *own assignable*.

where a is declared as an "own" of the procedure p. Contrast the meaning of the foregoing declaration with the following one:

$$\texttt{proc}\, p(\, x : \tau\,) : \rho\, \texttt{is}\, \{\texttt{dcl}\, a := e\, \texttt{in}\, m\}\, \texttt{in}\, m'.$$

34.5. A natural generalization of own assignables is to allow the creation of many such scenarios for a single procedure (or mutually recursive collection of procedures), with each instance creating its own persistent state. This ability motivated the concept of a *class* in Simula-67 as a collection of procedures, possibly mutually recursive, that shared common persistent state. Each instance of a class is called an *object* of that class; calls to its constituent procedures mutate the private persistent state. Formulate this 1967 precursor of imperative object-oriented programming in the context of **MA**.

34.6. There are several ways to formulate an abstract machine for **MA** that accounts for both the *control stack*, which sequences execution (as described in Chapter 28 for **PCF**), and the *data stack*, which records the contents of the assignables. A *consolidated stack* combines these two separate concepts into one, whereas *separated stacks* keeps the memory separate from the control stack, much as we have done in the structural dynamics given by Rules (34.3). In either case the storage required for an assignable is deallocated when exiting the scope of that assignable, a key benefit of the stack discipline for assignables in **MA**.

With a modal separation between expressions and commands it is natural to use a structural dynamics for expressions, and a stack machine dynamics for commands.

(a) Formulate a consolidated stack machine where both assignables and stack frames are recorded on the same stack. Consider states $k \triangleright_\Sigma m$, where $\vdash_\Sigma k \div \tau$ and $\vdash_\Sigma m \dot{\div} \tau$, and $k \triangleleft_\Sigma e$, where $\vdash_\Sigma k \div \tau$ and $\vdash_\Sigma e : \tau$. Comment on the implementation methods required for a consolidated stack.

(b) Formulate a separated stack machine where the memory is maintained separately from the control stack. Consider states of the form $k \parallel \mu \triangleright_\Sigma m$, where $\mu : \Sigma$, $\vdash_\Sigma k \div \tau$, and $\vdash_\Sigma m \dot{\div} \tau$, and of the form $k \parallel \mu \triangleleft_\Sigma e$, where $\vdash_\Sigma k \div \tau$, $\vdash_\Sigma e : \tau$, and $e\ \mathsf{val}_\Sigma$.

Chapter 35

Assignable References

A *reference* to an assignable a is a value, written $\&a$, of *reference type* that refers to the assignable a. A reference to an assignable provides the *capability* to get or set the contents of that assignable, even if the assignable itself is not in scope when it is used. Two references can be compared for equality to test whether they govern the same underlying assignable. If two references are equal, then setting one will affect the result of getting the other; if they are not equal, then setting one cannot influence the result of getting from the other. Two references that govern the same underlying assignable are *aliases*. Aliasing complicates reasoning about programs that use references, because any two references may refer to the same assignable.

Reference types are compatible with both a scoped and a scope-free allocation of assignables. When assignables are scoped, the range of significance of a reference type is limited to the scope of the assignable to which it refers. Reference types are therefore immobile, so that they cannot be returned from the body of a declaration, nor stored in an assignable. Although ensuring adherence to the stack discipline, this restriction precludes using references to create mutable data structures, those whose structure can be altered during execution. Mutable data structures have a number of applications in programming, including improving efficiency (often at the expense of expressiveness) and allowing cyclic (self-referential) structures to be created. Supporting mutability requires that assignables be given a scope-free dynamics, so that their lifetime persists beyond the scope of their declaration. Consequently, all types are mobile, so that a value of any type may be stored in an assignable or returned from a command.

35.1 Capabilities

The commands `get`$[a]$ and `set`$[a](e)$ in **MA** operate on statically specified assignable a. Even to write these commands requires that the assignable a be in scope where the command occurs. But suppose that we wish to define a procedure that, say, updates an assignable to double its previous value, and returns the previous value. We can write such a procedure for a specific assignable, a, but what if we wish to write a generic procedure that works uniformly for all assignables?

One way to do this is give the procedure the *capability* to get and set the contents of some caller-specified assignable. Such a capability is a pair consisting of a *getter* and a *setter* for that assignable. The getter for an assignable a is a command that, when executed, returns the contents of a. The setter for an assignable a is a procedure that, when applied to a value of suitable type, assigns that value to a. Thus, a capability for an assignable a containing a value of type τ is a value of type

$$\tau\,\mathtt{cap} \triangleq \tau\,\mathtt{cmd} \times (\,\tau \rightharpoonup \tau\,\mathtt{cmd}\,).$$

A capability for getting and setting an assignable a containing a value of type τ is given by the pair

$$\langle \mathtt{cmd}\,(\,@\,a\,), \mathtt{proc}\,(\,x:\tau\,)\,a := x\rangle$$

of type $\tau\,\mathtt{cap}$. Because a capability type is a product of a command type and a procedure type, no capability type is mobile. Thus, a capability cannot be returned from a command, nor stored into an assignable. This is as it should be, for otherwise we would violate the stack discipline for allocating assignables.

The proposed generic doubling procedure is programmed using capabilities as follows:

$$\mathtt{proc}\,(\,\langle \mathit{get}, \mathit{set}\rangle : \mathtt{nat\,cmd} \times (\,\mathtt{nat} \rightharpoonup \mathtt{nat\,cmd}\,)\,)\,\{x \leftarrow \mathtt{do}\,\mathit{get}\,;\, y \leftarrow \mathtt{do}\,(\,\mathit{set}(\,x + x\,)\,)\,;\, \mathtt{ret}\,x\}.$$

The procedure is called with the capability to access an assignable a. When executed, it invokes the getter to obtain the contents of a, and then invokes the setter to assign to a, returning the previous value. Observe that the assignable a need not be accessible by this procedure; the capability given by the caller comprises the commands required to get and set a.

35.2 Scoped Assignables

A weakness of using a capability to give indirect access to an assignable is that there is no guarantee that a given getter/setter pair are in fact the capability for a particular assignable. For example, we might pair the getter for a with the setter for b, leading to unexpected behavior. There is nothing in the type system that prevents creating such mismatched pairs.

To avoid this we introduce the concept of a *reference* to an assignable. A reference is a value from which we may obtain the capability to get and set a particular assignable. Moreover, two references can be tested for equality to see whether they act on the same assignable.[1] The *reference type* $\mathtt{ref}(\,\tau\,)$ has as values references to assignables of type τ. The introduction and elimination forms for this type are given by the following syntax chart:

Typ	τ	::=	$\mathtt{ref}(\,\tau\,)$	$\tau\,\mathtt{ref}$	assignable
Exp	e	::=	$\mathtt{ref}[\,a\,]$	$\&a$	reference
Cmd	m	::=	$\mathtt{getref}(\,e\,)$	$*\,e$	contents
			$\mathtt{setref}(\,e_1\,;e_2\,)$	$e_1 *\!= e_2$	update

[1] The getter and setter do not suffice to define equality, because not all types admit a test for equality. When they do, and when there are at least two distinct values of their type, we can determine whether they are aliases by assigning to one and checking whether the contents of the other is changed.

The statics of reference types is defined by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)} \tag{35.1a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{ref}(\tau)}{\Gamma \vdash_{\Sigma} \mathtt{getref}(e) \mathrel{\dot{\sim}} \tau} \tag{35.1b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \mathtt{ref}(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \tau}{\Gamma \vdash_{\Sigma} \mathtt{setref}(e_1; e_2) \mathrel{\dot{\sim}} \tau} \tag{35.1c}$$

Rule (35.1a) specifies that a reference to any active assignable is an expression of type $\mathtt{ref}(\tau)$.

The dynamics of reference types defers to the corresponding operations on assignables, and does not alter the underlying dynamics of assignables:

$$\frac{}{\mathtt{ref}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{35.2a}$$

$$\frac{e \xmapsto[\Sigma]{} e'}{\mu \parallel \mathtt{getref}(e) \xmapsto[\Sigma]{} \mu \parallel \mathtt{getref}(e')} \tag{35.2b}$$

$$\frac{}{\mu \parallel \mathtt{getref}(\mathtt{ref}[a]) \xmapsto[\Sigma, a \sim \tau]{} \mu \parallel \mathtt{get}[a]} \tag{35.2c}$$

$$\frac{e_1 \xmapsto[\Sigma]{} e_1'}{\mu \parallel \mathtt{setref}(e_1; e_2) \xmapsto[\Sigma]{} \mu \parallel \mathtt{setref}(e_1'; e_2)} \tag{35.2d}$$

$$\frac{}{\mu \parallel \mathtt{setref}(\mathtt{ref}[a]; e) \xmapsto[\Sigma, a \sim \tau]{} \mu \parallel \mathtt{set}[a](e)} \tag{35.2e}$$

A reference to an assignable is a value. The `getref` and `setref` operations on references defer to the corresponding operations on assignables once the referent has been resolved.

Because references give rise to capabilities, the reference type is immobile. As a result references cannot be stored in assignables or returned from commands. The immobility of references ensures safety, as can be seen by extending the safety proof given in Chapter 34.

As an example of using references, the generic doubling procedure discussed in the preceding section is programmed using references as follows:

$$\mathtt{proc}\,(r : \mathtt{nat\ ref})\,\{x \leftarrow *r; r *= x + x; \mathtt{ret}\, x\}.$$

Because the argument is a reference, rather than a capability, there is no possibility that the getter and setter refer to different assignables.

The ability to pass references to procedures comes at a price, because any two references might refer to the same assignable (if they have the same type). Consider 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:

$$\lambda\,(\,x : \texttt{nat ref}\,)\,\lambda\,(\,y : \texttt{nat ref}\,)\,\texttt{cmd}\,\{x' \leftarrow *\,x\,;\,y' \leftarrow *\,y\,;\,x *= x' + y' + y'\}.$$

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 the sum of its original contents and twice the contents of the assignable referenced by y.

But now consider the following seemingly equivalent implementation of this procedure:

$$\lambda\,(\,x : \texttt{nat ref}\,)\,\lambda\,(\,y : \texttt{nat ref}\,)\,\texttt{cmd}\,\{x \mathrel{+}= y\,;\,x \mathrel{+}= y\},$$

where $x \mathrel{+}= y$ is the command

$$\{x' \leftarrow *\,x\,;\,y' \leftarrow *\,y\,;\,x *= x' + y'\}$$

that adds the contents of y to the contents of x. The second implementation works right, as long as x and y do not refer to the same assignable. If they do refer to a common assignable a, with contents n, the result is that a is to set $4 \times n$, instead of the intended $3 \times n$. The second `get` of y is affected by the first `set` of x.

In this case it is clear 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 discovered, but in 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. The problem is that the number of possible aliasing relationships among n references grows combinatorially in n.

35.3 Free Assignables

Although it is interesting to note that references and capabilities are compatible with the stack discipline, for references to be useful requires that this restriction be relaxed. With immobile references it is impossible to build data structures containing references, or to return references from procedures. To allow this we must arrange that the lifetime of an assignable extend beyond its scope. In other words we must give up stack allocation for heap allocation. Assignables that persist beyond their scope of declaration are called *scope-free*, or just *free*, assignables. When all assignables are free, every type is mobile and so any value, including a reference, may be used in a data structure.

Supporting free assignables amounts to changing the dynamics so that allocation of assignables persists across transitions. We use transition judgments of the form

$$\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}.$$

Execution of a command may allocate new assignables, may alter the contents of existing assignables, and may give rise to a new command to be executed at the next step. The rules defining the dynamics of free assignables are as follows:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{ret}(\,e\,)\,\}\ \mathsf{final}} \tag{35.3a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{ret}(\,e\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{ret}(\,e'\,)\,\}} \tag{35.3b}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{bnd}(\,e\,;x\,.\,m\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{bnd}(\,e'\,;x\,.\,m\,)\,\}} \tag{35.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{ret}(\,e\,)\,)\,;x\,.\,m\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \{e/x\}m\,\}} \tag{35.3d}$$

$$\frac{\nu\,\Sigma\,\{\,\mu \parallel m_1\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m_1'\,\}}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{bnd}(\,\mathtt{cmd}(\,m_1\,)\,;x\,.\,m_2\,)\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel \mathtt{bnd}(\,\mathtt{cmd}(\,m_1'\,)\,;x\,.\,m_2\,)\,\}} \tag{35.3e}$$

$$\frac{}{\nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow e \parallel \mathtt{get}[\,a\,]\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(\,e\,)\,\}} \tag{35.3f}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{set}[\,a\,](\,e\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{set}[\,a\,](\,e'\,)\,\}} \tag{35.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a \sim \tau}}{\nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow \_ \parallel \mathtt{set}[\,a\,](\,e\,)\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(\,e\,)\,\}} \tag{35.3h}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(\,e\,;a\,.\,m\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(\,e'\,;a\,.\,m\,)\,\}} \tag{35.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(\,e\,;a\,.\,m\,)\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow e \parallel m\,\}} \tag{35.3j}$$

The language **RMA** extends **MA** with references to free assignables. Its dynamics is similar to that of references to scoped assignables given earlier.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,e\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,e'\,)\,\}} \tag{35.4a}$$

$$\nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,\mathtt{ref}[\,a\,]\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{get}[\,a\,]\,\} \tag{35.4b}$$

$$\frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,e_1\,;e_2\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,e_1'\,;e_2\,)\,\}} \tag{35.4c}$$

$$\nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,\mathtt{ref}[\,a\,]\,;e_2\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{set}[\,a\,](\,e_2\,)\,\} \tag{35.4d}$$

The expressions cannot alter or extend the memory, only commands may do so.

As an example of using **RMA**, consider the command $\mathtt{newref}[\,\tau\,](\,e\,)$ defined by

$$\mathtt{dcl}\,a := e\,\mathtt{in}\,\mathtt{ret}\,(\,\&a\,). \tag{35.5}$$

This command allocates a fresh assignable, and returns a reference to it. Its static and dynamics are derived from the foregoing rules as follows:

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{newref}[\,\tau\,](\,e\,) \mathrel{\dot{\sim}} \mathtt{ref}(\,\tau\,)} \tag{35.6}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{newref}[\,\tau\,](\,e\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{newref}[\,\tau\,](\,e'\,)\,\}} \tag{35.7a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{newref}[\,\tau\,](\,e\,)\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mu \otimes a \hookrightarrow e \parallel \mathtt{ret}(\,\mathtt{ref}[\,a\,]\,)\,\}} \tag{35.7b}$$

Oftentimes the command $\mathtt{newref}[\,\tau\,](\,e\,)$ is taken as primitive, and the declaration command is omitted. In that case all assignables are accessed by reference, and no direct access to assignables is provided.

35.4 Safety

Although the proof of safety for references to scoped assignables presents few difficulties, the safety for free assignables is tricky. The main difficulty is to account for cyclic dependencies within data structures (such as will arise in Section 35.5.) 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 $\mathtt{nat} \rightharpoonup \mathtt{nat}\ \mathtt{cmd}$:

$$\mathtt{proc}\,(\,x : \mathtt{nat}\,)\,\{\mathtt{ifz}\,(\,x\,)\,\mathtt{ret}\,(\,1\,)\,\mathtt{else}\,\{f \leftarrow @\,a\,; y \leftarrow f(\,x - 1\,)\,; \mathtt{ret}\,(\,x * y\,)\}\}.$$

Let μ be a memory of the form $\mu' \otimes a \hookrightarrow e$ 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.

Cyclic dependencies complicate the definition of the judgment $\nu\,\Sigma\,\{\,\mu \parallel m\,\}\ \mathsf{ok}$. It is defined by the following rule:

$$\frac{\vdash_\Sigma m \div \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{\,\mu \parallel m\,\}\ \mathsf{ok}} \tag{35.8}$$

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 all of* Σ so that cyclic dependencies are permitted. The judgment $\vdash_{\Sigma'} \mu : \Sigma$ is defined as follows:

$$\frac{\forall a \sim \tau \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{35.9}$$

In the safety proof to follow Σ' is chosen to be Σ to allow for cyclicity.

Theorem 35.1 (Preservation)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\longmapsto} e'$, *then* $\vdash_\Sigma e' : \tau$.
2. *If* $\nu\,\Sigma\,\{\,\mu \parallel m\,\}$ *ok and* $\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$, *then* $\nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$ *ok*.

Proof. Simultaneously, by induction on transition. We prove the following stronger form of the second statement:

> If $\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$, where $\vdash_\Sigma m \div \tau$, $\vdash_\Sigma \mu : \Sigma$, then Σ' extends Σ, and $\vdash_{\Sigma'} m' \div \tau$, and $\vdash_{\Sigma'} \mu' : \Sigma'$.

Consider the transition

$$\nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(\,e\,;a\,.\,m\,)\,\} \longmapsto \nu\,\Sigma, a \sim \rho\,\{\,\mu \otimes a \hookrightarrow e \parallel m\,\}$$

where $e\ \mathsf{val}_\Sigma$. By assumption and inversion of rule (34.6d) we have $\vdash_\Sigma e : \rho$, $\vdash_{\Sigma, a\sim\rho} m \div \tau$, and $\vdash_\Sigma \mu : \Sigma$. But because extension of Σ with a fresh assignable does not affect typing, we also have $\vdash_{\Sigma,a\sim\rho} \mu : \Sigma$ and $\vdash_{\Sigma,a\sim\rho} e : \rho$, from which it follows by rule (35.9) that $\vdash_{\Sigma,a\sim\rho} \mu \otimes a \hookrightarrow e : \Sigma, a \sim \rho$.

The other cases follow a similar pattern, and are left as an exercise for the reader.

□

Theorem 35.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* $e\ \mathsf{val}_\Sigma$ *or there exists* e' *such that* $e \underset{\Sigma}{\longmapsto} e'$.
2. *If* $\nu\,\Sigma\,\{\,\mu \parallel m\,\}$ *ok then either* $\nu\,\Sigma\,\{\,\mu \parallel m\,\}$ *final or* $\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$ *for some* Σ', μ', *and* m'.

Proof. Simultaneously, by induction on typing. For the second statement we prove

> If $\vdash_\Sigma m \div \tau$ and $\vdash_\Sigma \mu : \Sigma$, then either $\nu\,\Sigma\,\{\,\mu \parallel m\,\}$ final, or $\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$ for some Σ', μ', and m'.

Consider the typing rule

$$\frac{\Gamma \vdash_\Sigma e : \rho \quad \Gamma \vdash_{\Sigma, a \sim \rho} m \div \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}(e\,;a\,.\,m) \div \tau}$$

We have by the first inductive hypothesis that either e val_Σ or $e \xmapsto[\Sigma]{} e'$ for some e'. In the latter case we have by rule (35.3i)

$$\nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(e\,;a\,.\,m)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(e'\,;a\,.\,m)\,\}.$$

In the former case we have by rule (35.3j) that

$$\nu\,\Sigma\,\{\,\mu \parallel \mathtt{dcl}(e\,;a\,.\,m)\,\} \longmapsto \nu\,\Sigma, a \sim \rho\,\{\,\mu \otimes a \hookrightarrow e \parallel m\,\}.$$

Now consider the typing rule

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] \div \tau}$$

By assumption $\vdash_{\Sigma, a \sim \tau} \mu : \Sigma, a \sim \tau$, and hence there exists e $\mathsf{val}_{\Sigma, a \sim \tau}$ such that $\mu = \mu' \otimes a \hookrightarrow e$ and $\vdash_{\Sigma, a \sim \tau} e : \tau$. By rule (35.3f)

$$\nu\,\Sigma, a \sim \tau\,\{\,\mu' \otimes a \hookrightarrow e \parallel \mathtt{get}[a]\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mu' \otimes a \hookrightarrow e \parallel \mathtt{ret}(e)\,\},$$

as required. The other cases are handled similarly.

□

35.5 Benign Effects

The modal separation between commands and expressions ensures that the meaning of an expression does not depend on the (ever-changing) contents of assignables. Although this is helpful in many, perhaps most, situations, it also precludes programming techniques that use storage effects to implement purely functional behavior. A prime example is memoization. Externally, a suspended computation behaves exactly like the underlying computation; internally, an assignable is associated with the computation that stores the result of any evaluation of the computation for future use. Other examples are self-adjusting data structures, which use state to improve their efficiency without changing their functional behavior. For example, a splay tree is a binary search tree that uses mutation internally to re-balance the tree as elements are inserted, deleted, and retrieved, so that lookup takes time proportional to the logarithm of the number of elements.

These are examples of *benign storage effects*, uses of mutation in a data structure to improve efficiency without disrupting its functional behavior. One class of examples are self-adjusting data structures that reorganize themselves during one use to improve efficiency of later uses. Another class of examples are memoized, or lazy, data structures, which are discussed in Chapter 36. Benign effects such as these are impossible to implement if a strict separation between expressions and commands is maintained. For example, a self-adjusting tree involves mutation, but is a value

just like any other, and this cannot be achieved in **MA**. Although several special-case techniques are known, the most general solution is to do away with the modal distinction, coalescing expressions and commands into a single syntactic category. The penalty is that the type system no longer ensures that an expression of type τ denotes a value of that type; it might also have storage effects during its evaluation. The benefit is that one may freely use benign effects, but it is up to the programmer to ensure that they truly are benign.

The language **RPCF** extends **PCF** with references to free assignables. The following rules define the statics of the distinctive features of **RPCF**:

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a \sim \tau_1} e_2 : \tau_2}{\Gamma \vdash_\Sigma \mathtt{dcl}(e_1 \,;\, a \,.\, e_2) : \tau_2} \tag{35.10a}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] : \tau} \tag{35.10b}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{set}[a](e) : \tau} \tag{35.10c}$$

Correspondingly, the dynamics of **RPCF** is given by transitions of the form

$$\nu\, \Sigma \{\, \mu \parallel e \,\} \longmapsto \nu\, \Sigma' \{\, \mu' \parallel e' \,\},$$

where e is an expression, and not a command. The rules defining the dynamics are very similar to those for **RMA**, but with commands and expressions integrated into a single category.

To illustrate the concept of a benign effect, consider the technique of *back-patching* to implement recursion. Here is an implementation of the factorial function that uses an assignable to implement recursive calls:

```
dcl a := λn:nat.0 in
  { f ← a := λn:nat.ifz(n, 1, n′.n×(@a)(n′))
  ; ret(f)
  }
```

This declaration returns a function of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$ that is obtained by (a) allocating a free assignable initialized arbitrarily with a function of this type, (b) defining a λ-abstraction in which each "recursive call" consists of retrieving and applying the function stored in that assignable, (c) assigning this function to the assignable, and (d) returning that function. The result is a function on the natural numbers, even though it uses state in its implementation.

Backpatching is not expressible in **RMA**, because it relies on assignment. Let us attempt to recode the previous example in **RMA**:

```
dcl a := proc(n:nat){ret 0} in
  { f ← a := ...
  ; ret(f)
  },
```

where the elided procedure assigned to a is given by

```
proc(n:nat) {if (ret(n)) {ret(1)} else {f←@a; x←f(n-1); ret(n×x)}}.
```

The difficulty is that what we have is a command, not an expression. Moreover, the result of the command is of the procedure type `nat ⇀ ( nat cmd )`, and not of the function type `nat ⇀ nat`. Consequently, we cannot use the factorial procedure in an expression, but have to execute it as a command using code such as this:

```
{ f ← fact; x ← f(n); ret(x) }.
```

35.6 Notes

Reynolds (1981) uses capabilities to provide indirect access to assignables; references are just an abstract form of capability. References are often permitted only for free assignables, but with mobility restrictions one may also have references to scoped assignables. The proof of safety of free references outlined here follows those given by Wright and Felleisen (1994) and Harper (1994).

Benign effects are central to the distinction between Haskell, which provides an Algol-like separation between commands and expressions, and ML, which integrates evaluation with execution. The choice between them is classic trade-off, with neither superior to the other in all respects.

Exercises

35.1. Consider scoped array assignables as described in Exercise **34.1**. Extend the treatment of array assignables in Exercise **34.1**, to account for array assignable references.

35.2. References to scope-free assignables are often used to implement recursive data structures such as mutable lists and trees. Examine such data structures in the context of **RMA** enriched with sum, product, and recursive types.

Give six different types that could be considered a type of linked lists, according to the following characteristics:

(a) A mutable list may only be updated *in toto* by replacing it with another (immutable) list.

(b) A mutable list can be altered in one of two ways, to make it empty, or to change both its head and tail element simultaneously. The tail element is any other such mutable list, so circularities may arise.

(c) A mutable list is, permanently, either empty or non-empty. If not, both its head and tail can be modified simultaneously.

(d) A mutable list is, permanently, either empty or non-empty. If not, its tail, but not its head, can be set to another such list.

(e) A mutable list is, permanently, either empty or non-empty. If not, either its head or its tail elements can be modified independently.

(f) A mutable list can be altered to become either empty or non-empty. If it is non-empty, either it head, or its tail, can be modified independently of one another.

Discuss the merits and deficiencies of each representation.

Part XV

Parallelism

Chapter 37

Nested Parallelism

Parallel computation seeks to reduce the running times of programs by allowing many computations to be carried out simultaneously. For example, if we wish 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.

In this chapter we discuss the language **PPCF**, which is the extension of **PCF** with *nested parallelism*. Nested parallelism has a hierarchical structure arising from *forking* two (or more) parallel computations, then *joining* these computations to combine their results before proceeding. Nested parallelism is also known as *fork-join parallelism*. 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 7) that focuses attention on the sequential and parallel complexity (also known as the *work* and the *depth*, or *span*) of a parallel program by associating a *series-parallel graph* with each computation.

37.1 Binary Fork-Join

The syntax of **PPCF** extends that of **PCF** with the following construct:

$$\mathsf{Exp} \quad e \quad ::= \quad \mathtt{par}(e_1\,;e_2\,;x_1\,.\,x_2\,.\,e) \quad \mathtt{par}\,x_1 = e_1\,\mathtt{and}\,x_2 = e_2\,\mathtt{in}\,e \quad \text{parallel let}$$

The variables x_1 and x_2 are bound only within e, and not within e_1 or e_2, which ensures that they are not mutually dependent and hence can be evaluated simultaneously. The variable bindings represent a fork of two parallel computations e_1 and e_2, and the body e represents their join.

The statics of **PPCF** enriches that of **PCF** with the following rule for parallel let:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) : \tau} \tag{37.1}$$

The *sequential structural dynamics* of **PPCF** is defined by a transition judgment of the form $e \longmapsto_{\mathsf{seq}} e'$ defined by these rules:

$$\frac{e_1 \longmapsto_{\mathsf{seq}} e_1'}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{seq}} \mathtt{par}(\, e_1' \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,)} \tag{37.2a}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto_{\mathsf{seq}} e_2'}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{seq}} \mathtt{par}(\, e_1 \,;\, e_2' \,;\, x_1 \,.\, x_2 \,.\, e \,)} \tag{37.2b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{seq}} \{e_1, e_2 / x_1, x_2\} e} \tag{37.2c}$$

The *parallel structural dynamics* of **PPCF** is given by a transition judgment of the form $e \longmapsto_{\mathsf{par}} e'$, defined as follows:

$$\frac{e_1 \longmapsto_{\mathsf{par}} e_1' \quad e_2 \longmapsto_{\mathsf{par}} e_2'}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{par}} \mathtt{par}(\, e_1' \,;\, e_2' \,;\, x_1 \,.\, x_2 \,.\, e \,)} \tag{37.3a}$$

$$\frac{e_1 \longmapsto_{\mathsf{par}} e_1' \quad e_2 \ \mathsf{val}}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{par}} \mathtt{par}(\, e_1' \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,)} \tag{37.3b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto_{\mathsf{par}} e_2'}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{par}} \mathtt{par}(\, e_1 \,;\, e_2' \,;\, x_1 \,.\, x_2 \,.\, e \,)} \tag{37.3c}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \longmapsto_{\mathsf{par}} \{e_1, e_2 / x_1, x_2\} e} \tag{37.3d}$$

The parallel dynamics abstracts away from any limitations on processing capacity; such limitations are considered in Section 37.4.

The *implicit parallelism theorem* states that the sequential and the parallel dynamics coincide. Consequently, we need never be concerned with the *meaning* of a parallel program (its meaning is given by the sequential dynamics), but only with its *efficiency*. As a practical matter, this means that a program can be developed on a sequential platform, even if it is meant to run on a parallel platform, because the behavior is not affected by whether we execute it using a sequential or a parallel dynamics. Because the sequential dynamics is deterministic (every expression has at most

one value), the implicit parallelism theorem implies that the parallel dynamics is also deterministic. For this reason the implicit parallelism theorem is also known as the *deterministic parallelism theorem*. This terminology emphasizes the distinction between *deterministic parallelism*, the subject of this chapter, from *non-deterministic concurrency*, the subject of Chapters 39 and 40.

A proof of the implicit parallelism theorem can be given by giving an evaluation dynamics $e \Downarrow v$ in the style of Chapter 7, and showing that

$$e \xmapsto[\mathsf{par}]{}{}^* v \quad \text{iff} \quad e \Downarrow v \quad \text{iff} \quad e \xmapsto[\mathsf{seq}]{}{}^* v$$

(where v is a closed expression such that v val). The most important rule of the evaluation dynamics is for the evaluation of a parallel binding:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \{v_1, v_2/x_1, x_2\}e \Downarrow v}{\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e\,) \Downarrow v} \tag{37.4}$$

The other rules are easily derived from the structural dynamics of **PCF** as in Chapter 7.

It is possible to show that the sequential dynamics of **PPCF** agrees with its evaluation dynamics by extending the proof of Theorem 7.2.

Lemma 37.1. *For all v val, $e \xmapsto[\mathit{seq}]{}{}^* v$ if, and only if, $e \Downarrow v$.*

Proof. It suffices to show that if $e \xmapsto[\mathsf{seq}]{} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \xmapsto[\mathsf{seq}]{}{}^* v_1$ and $e_2 \xmapsto[\mathsf{seq}]{}{}^* v_2$ and $\{v_1, v_2/x_1, x_2\}e \xmapsto[\mathsf{seq}]{}{}^* v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \xmapsto[\mathsf{seq}]{}{}^* v.$$

□

By a similar argument we may show that the parallel dynamics also agrees with the evaluation dynamics, and hence with the sequential dynamics.

Lemma 37.2. *For all v val, $e \xmapsto[\mathit{par}]{}{}^* v$ if, and only if, $e \Downarrow v$.*

Proof. It suffices to show that if $e \xmapsto[\mathsf{par}]{} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \xmapsto[\mathsf{par}]{}{}^* v_1$ and $e_2 \xmapsto[\mathsf{par}]{}{}^* v_2$ and $\{v_1, v_2/x_1, x_2\}e \xmapsto[\mathsf{par}]{}{}^* v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \xmapsto[\mathsf{par}]{}{}^* v.$$

The proof of the first is by induction on the parallel dynamics. The proof of the second proceeds by simultaneous induction on the derivations of $e_1 \xmapsto[\mathsf{par}]{}{}^* v_1$ and $e_2 \xmapsto[\mathsf{par}]{}{}^* v_2$. 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 $e_1 \xmapsto[\mathsf{par}]{} e_1' \xmapsto[\mathsf{par}]{}{}^* v_1$, then by induction we have that $\mathtt{par}\, x_1 = e_1' \,\mathtt{and}\, x_2 = v_2 \,\mathtt{in}\, e \xmapsto[\mathsf{par}]{}{}^* v$, and hence the result follows by an application of rule (37.3b). The symmetric case follows similarly by an application of rule (37.3c), and in the case that both e_1 and e_2 transition, the result follows by induction and rule (37.3a). □

Theorem 37.3 (Implicit Parallelism)**.** *The sequential and parallel dynamics coincide: for all v val, $e \longmapsto_{seq}^{*} v$ iff $e \longmapsto_{par}^{*} v$.*

Proof. By Lemmas 37.1 and 37.2. □

The implicit parallelism theorem states that parallelism does not affect the meaning of a program, only the efficiency of its execution. Correctness is not affected by parallelism, only efficiency.

37.2 Cost Dynamics

In this section we define a *parallel cost dynamics* that assigns a *cost graph* to the evaluation of a **PPCF** 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 *series-parallel* ordered 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 extra 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. The children of the source are ordered according to the sequential evaluation order. 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. At the structural level, we note that only the root has no ancestors, and only the final node of the cost graph has no descendents. Interior nodes may have one or two descendents, the former representing a sequential dependency, and the latter representing a *fork point*. Such nodes may have one or two ancestors, the former corresponding to a sequential dependency and the latter representing a *join point*.

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:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{37.5}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{37.6}$$

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 a lower bound on the number of steps required to complete the computation.

In Chapter 7 we introduced *cost dynamics* to assign time complexity to computations. The proof of Theorem 7.7 shows that $e \Downarrow^k v$ iff $e \longmapsto^k v$. That is, the step complexity of an evaluation of e to a value v is just the number of transitions required to derive $e \longmapsto^* v$. Here we use cost graphs as the measure of complexity, then relate these cost graphs to the structural dynamics given in Section 37.1.

The judgment $e \Downarrow^c v$, 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:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad \{v_1, v_2 / x_1, x_2\} e \Downarrow^{c} v}{\texttt{par}(e_1 ; e_2 ; x_1 \,.\, x_2 \,.\, e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{37.7}$$

The cost assignment specifies that, under ideal conditions, e_1 and e_2 are evaluated in parallel, and that their results are passed 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 by a constant-time mechanism for updating an environment. The cost dynamics of other language constructs is specified in a similar way, using only sequential combination to isolate the source of parallelism to the `par` construct.

Two simple facts about the cost dynamics are important to keep in mind. First, the cost assignment does not influence the outcome.

Lemma 37.4. *$e \Downarrow v$ iff $e \Downarrow^c v$ for some c.*

Proof. From right to left, erase the cost assignments to construct 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 37.5. *If $e \Downarrow^{c} v$ and $e \Downarrow^{c'} v$, then c is c'.*

Proof. By induction on the derivation of $e \Downarrow^{c} v$. □

The link between the cost dynamics and the structural dynamics is given 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 37.6. *If $e \Downarrow^{c} v$, then $e \xmapsto[\mathsf{seq}]{}{}^{w} v$ and $e \xmapsto[\mathsf{par}]{}{}^{d} v$, where $w = wk(c)$ and $d = dp(c)$. Conversely, if $e \xmapsto[\mathsf{seq}]{}{}^{w} v$, then there exists c such that $e \Downarrow^{c} v$ with $wk(c) = w$, and if $e \xmapsto[\mathsf{par}]{}{}^{d} v'$, then there exists c' such that $e \Downarrow^{c'} v'$ with $dp(c') = d$.*

Proof. The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being rule (37.7). By induction we have $e_1 \xmapsto[\mathsf{seq}]{}{}^{w_1} v_1$, $e_2 \xmapsto[\mathsf{seq}]{}{}^{w_2} v_2$, and $\{v_1, v_2/x_1, x_2\}e \xmapsto[\mathsf{seq}]{}{}^{w} v$, where $w_1 = wk(c_1)$, $w_2 = wk(c_2)$, and $w = wk(c)$. By pasting together derivations we get a derivation

$$\begin{aligned}
\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) &\xmapsto[\mathsf{seq}]{}{}^{w_1} \mathtt{par}(\, v_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \\
&\xmapsto[\mathsf{seq}]{}{}^{w_2} \mathtt{par}(\, v_1 \,;\, v_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \\
&\xmapsto[\mathsf{seq}]{} \{v_1, v_2/x_1, x_2\}e \\
&\xmapsto[\mathsf{seq}]{}{}^{w} v.
\end{aligned}$$

Noting that $wk((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \xmapsto[\mathsf{par}]{}{}^{d_1} v_1$, $e_2 \xmapsto[\mathsf{par}]{}{}^{d_2} v_2$, and $\{v_1, v_2/x_1, x_2\}e \xmapsto[\mathsf{par}]{}{}^{d} v$, where $d_1 = dp(c_1)$, $d_2 = dp(c_2)$, and $d = dp(c)$. Assume, without loss of generality, that $d_1 \leq d_2$ (otherwise simply swap the roles of d_1 and d_2 in what follows). We may paste together derivations as follows:

$$\begin{aligned}
\mathtt{par}(\, e_1 \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) &\xmapsto[\mathsf{par}]{}{}^{d_1} \mathtt{par}(\, v_1 \,;\, e_2' \,;\, x_1 \,.\, x_2 \,.\, e \,) \\
&\xmapsto[\mathsf{par}]{}{}^{d_2 - d_1} \mathtt{par}(\, v_1 \,;\, v_2 \,;\, x_1 \,.\, x_2 \,.\, e \,) \\
&\xmapsto[\mathsf{par}]{} \{v_1, v_2/x_1, x_2\}e \\
&\xmapsto[\mathsf{par}]{}{}^{d} v.
\end{aligned}$$

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 $e \xmapsto[\mathsf{seq}]{} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $wk(c) = wk(c') + 1$, and if $e \xmapsto[\mathsf{par}]{} e'$ 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 $e \xmapsto[\mathsf{seq}]{} e'$, 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, $e \xmapsto[\mathsf{par}]{} e'$ 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 $e \xmapsto[\mathsf{seq}]{} e'$, where $e' = \mathtt{par}(\, e_1' \,;\, e_2 \,;\, x_1 \,.\, x_2 \,.\, e_0 \,)$ and $e_1 \xmapsto[\mathsf{seq}]{} e_1'$. 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 $e \xmapsto[\mathsf{par}]{} e'$, where $e' = \mathtt{par}(\, e_1' \,;\, e_2' \,;\, x_1 \,.\, x_2 \,.\, e_0 \,)$ and $e_1 \xmapsto[\mathsf{par}]{} e_1'$, $e_2 \xmapsto[\mathsf{par}]{} e_2'$, 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 get

$$\begin{aligned} dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) \\ &= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) \\ &= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 \\ &= dp(c') + 1, \end{aligned}$$

which completes the proof. □

Corollary 37.7. *If $e \xmapsto[\mathsf{seq}]{}^{w} v$ and $e \xmapsto[\mathsf{par}]{}^{d} v'$, then v is v' and $e \Downarrow^{c} v$ for some c such that $wk(c) = w$ and $dp(c) = d$.*

37.3 Multiple Fork-Join

So far we have confined attention to binary fork/join parallelism induced by the parallel `par` construct. A generalizaton, called *data parallelism*, allows the simultaneous creation of any number of tasks that compute on the components of a data structure. The main example is a *sequence* of values of a specified type. The primitive operations on sequences are a natural source of unbounded parallelism. For example, we 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	::=	$\mathtt{seq}[\,n\,](\,e_0,\ldots,e_{n-1}\,)$	$\langle e_0,\ldots,e_{n-1}\rangle_n$	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 $\mathtt{seq}[\,n\,](\,e_0,\ldots,e_{n-1}\,)$ evaluates to a sequence whose length is n and 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:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \mathtt{seq}[\,n\,](\,e_0,\ldots,e_{n-1}\,) : \mathtt{seq}(\,\tau\,)} \tag{37.8a}$$

$$\frac{\Gamma \vdash e : \mathtt{seq}(\,\tau\,)}{\Gamma \vdash \mathtt{len}(\,e\,) : \mathtt{nat}} \tag{37.8b}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{seq}(\,\tau\,) \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{sub}(\,e_1\,;e_2\,) : \tau} \tag{37.8c}$$

$$\frac{\Gamma, x : \mathtt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{tab}(\,x\,.\,e_1\,;e_2\,) : \mathtt{seq}(\,\tau\,)} \tag{37.8d}$$

$$\frac{\Gamma \vdash e_2 : \mathtt{seq}(\,\tau\,) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \mathtt{map}(\,x\,.\,e_1\,;e_2\,) : \mathtt{seq}(\,\tau'\,)} \tag{37.8e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{seq}(\,\tau\,) \quad \Gamma \vdash e_2 : \mathtt{seq}(\,\tau\,)}{\Gamma \vdash \mathtt{cat}(\,e_1\,;e_2\,) : \mathtt{seq}(\,\tau\,)} \tag{37.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\mathtt{seq}[\,n\,](\,e_0,\ldots,e_{n-1}\,) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \mathtt{seq}[\,n\,](\,v_0,\ldots,v_{n-1}\,)} \tag{37.9a}$$

$$\frac{e \Downarrow^{c} \mathtt{seq}[\,n\,](\,v_0,\ldots,v_{n-1}\,)}{\mathtt{len}(\,e\,) \Downarrow^{c \oplus \mathbf{1}} \mathtt{num}[\,n\,]} \tag{37.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \mathtt{seq}[\,n\,](\,v_0,\ldots,v_{n-1}\,) \quad e_2 \Downarrow^{c_2} \mathtt{num}[\,i\,] \quad (0 \leq i < n)}{\mathtt{sub}(\,e_1\,;e_2\,) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{37.9c}$$

$$\frac{e_2 \Downarrow^{c} \texttt{num}[n] \quad \{\texttt{num}[0]/x\}e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad \{\texttt{num}[n-1]/x\}e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(x.e_1\,;e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}[n](v_0,\ldots,v_{n-1})} \tag{37.9d}$$

$$\frac{e_2 \Downarrow^{c} \texttt{seq}[n](v_0,\ldots,v_{n-1}) \quad \{v_0/x\}e_1 \Downarrow^{c_0} v'_0 \quad \ldots \quad \{v_{n-1}/x\}e_1 \Downarrow^{c_{n-1}} v'_{n-1}}{\texttt{map}(x.e_1\,;e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}[n](v'_0,\ldots,v'_{n-1})} \tag{37.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}[m](v_0,\ldots,v_{m-1}) \quad e_2 \Downarrow^{c_2} \texttt{seq}[n](v'_0,\ldots,v'_{n-1}) \quad p = m+n}{\texttt{cat}(e_1\,;e_2) \Downarrow^{c_1 \oplus c_2 \oplus \bigotimes_{i=0}^{m+n} \mathbf{1}} \texttt{seq}[p](v_0,\ldots,v_{m-1},v'_0,\ldots,v'_{n-1})} \tag{37.9f}$$

The cost dynamics for sequence operations is validated by introducing a sequential and parallel cost dynamics and extending the proof of Theorem 37.6 to cover this extension.

37.4 Bounded Implementations

Theorem 37.6 states that the cost dynamics accurately models the dynamics of the parallel `let` construct, whether executed sequentially or in parallel. The theorem 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. Besides limiting the available parallelism this also imposes some synchronization overhead that must be taken into account. A *bounded implementation* is one for which we may establish an asymptotic bound on the execution time once these overheads are taken into account.

A bounded implementation must take account of the limitations and capabilities of the hardware on which the program is run. Because 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 example of such a model is the *shared-memory multiprocessor*, or *SMP*, model. The basic assumption of the SMP model is that there are some fixed $p > 0$ processors coordinated by an interconnect that permits constant-time access to any object in memory shared by all p processors.[1] An SMP 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 are enough to provide a parallel fetch-and-add instruction that allows each processor to get 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 bounded implementation of parallelism involves two major tasks. First, we must show that the primitives of the language can be implemented efficiently on the abstract machine model. Second, we must show how to schedule the workload across the processors to minimize execution time by maximizing parallelism. When working with a low-level machine model such

[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.

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 *Brent's Theorem*.

Theorem 37.8. *If $e \Downarrow^c v$ with $wk(c) = w$ and $dp(c) = d$, then e can 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. Note 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 depth, then we are unable to exploit parallelism, and the overall time is proportional to the work alone.

Theorem 37.8 motivates consideration of a useful figure of merit, the *parallelizability ratio*, which is the ratio w/d of work to depth. If $w/d \gg p$, then the program is *parallelizable*, because then $w/p \gg d$, and we may therefore reduce running time by using p processors at each step. If 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 difficult problem in complexity theory to analyze which problems are parallelizable, and which are not.

Proving Brent's Theorem for an SMP would take us much too far afield for the present purposes. Instead we shall prove a Brent-type Theorem for an abstract machine, the **P** machine. The machine is unrealistic in that it is defined at a very high level of abstraction. But it is designed to match well the cost dynamics given earlier in this chapter. In particular, there are mechanisms that account for both sequential and parallel dependencies in a computation.

At the highest level, the state of the **P** machine consists of a global task graph whose structure corresponds to a "diagonal cut" through the cost graph of the overall computation. Nodes immediately above the cut are eligible to be executed, higher ancestors having already been completed, and whose immediate descendents are waiting for their ancestors to complete. Further descendents in the full task graph are tasks yet to be created, once the immediate descendents are finished. The **P** machine discards completed tasks, and future tasks beyond the immediate dependents are only created as execution proceeds. Thus it is only those nodes next to the cut line through the cost graph that are represented in the **P** machine state.

The *global state* of the **P** machine is a configuration of the form $\nu\,\Sigma\,\{\,\mu\,\}$, where Σ is degenerated to just a finite set of (pairwise distinct) *task names* and μ is a finite mapping of the task names in Σ to *local states*, representing the state of an individual task. A *local state* is either a closed **PCF** expression, or one of two special *join points* that implement the sequential and parallel dependencies of a task on one or two ancestors, respectively.[2] Thus, when expanded out, a global state has the

[2]The use of join points for each sequential dependency is profligate, but aligns the machine with the cost dynamics. Realistically, individual tasks manage sequential dependencies without synchronization, by using local control stacks as in

form

$$\nu\, a_1, \ldots, a_n \, \{\, a_1 \hookrightarrow s_1 \otimes \ldots \otimes a_n \hookrightarrow s_n \,\},$$

where $n \geq 1$, and each s_i is a local state. The ordering of the tasks in a state, like the order of declarations in the signature, is not significant.

A **P** machine state transition has the form $\nu\, \Sigma \, \{\, \mu \,\} \longmapsto \nu\, \Sigma' \, \{\, \mu' \,\}$. There are two forms of such transitions, the *global* and the *local*. A global step selects as many tasks as are available, up to a pre-specified parameter $p > 0$, which represents the number of processors available at each round. (Such a scheduler is *greedy* in the sense that it never fails to execute an available task, up to the specified limit for each round.) A task is *finished* if it consists of a closed **PCF** value, or is a join point whose dependents are not yet finished; otherwise a task is *available*, or *ready*. A ready task is always capable of taking a local step consisting of either a step of **PCF**, expressed in the setting of the **P** machine, or a *synchronization* step that manages the join-point logic. Because the **P** machine employs a greedy scheduler, it must complete execution in time proportional to $\max(w/p, d)$ steps by doing up to p steps of work at a time, insofar as it is possible within the limits of the depth of the computation. We thus get a *Brent-type Theorem* for the abstract machine that illustrates more sophisticated Brent-type Theorems for other models, such as the PRAM, that are used in the analysis of parallel algorithms.

The local transitions of the **P** machine corresponding to the steps of **PCF** itself are illustrated by the following example rules for application; the others follow a similar pattern.[3]

$$\frac{\neg(e_1 \ \mathsf{val})}{\nu\, a \, \{\, a \hookrightarrow e_1(\, e_2 \,) \,\} \underset{\mathsf{loc}}{\longmapsto} \nu\, a\, a_1 \, \{\, a \hookrightarrow \mathtt{join}[\, a_1 \,](\, x_1 \,.\, x_1(\, e_2 \,)\,) \otimes a_1 \hookrightarrow e_1 \,\}} \tag{37.10a}$$

$$\frac{e_1 \ \mathsf{val} \quad \neg(e_2 \ \mathsf{val})}{\nu\, a \, \{\, a \hookrightarrow e_1(\, e_2 \,) \,\} \underset{\mathsf{loc}}{\longmapsto} \nu\, a\, a_2 \, \{\, a \hookrightarrow \mathtt{join}[\, a_2 \,](\, x_2 \,.\, e_1(\, x_2 \,)\,) \otimes a_2 \hookrightarrow e_2 \,\}} \tag{37.10b}$$

$$\frac{e_1 \ \mathsf{val}}{\nu\, a\, a_1 \, \{\, a \hookrightarrow \mathtt{join}[\, a_1 \,](\, x_1 \,.\, x_1(\, e_2 \,)\,) \otimes a_1 \hookrightarrow e_1 \,\} \underset{\mathsf{loc}}{\longmapsto} \nu\, a \, \{\, a \hookrightarrow e_1(\, e_2 \,) \,\}} \tag{37.10c}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\nu\, a\, a_2 \, \{\, a_1 \hookrightarrow \mathtt{join}[\, a_2 \,](\, x_2 \,.\, e_1(\, x_2 \,)\,) \otimes a_2 \hookrightarrow e_2 \,\} \underset{\mathsf{loc}}{\longmapsto} \nu\, a \, \{\, a \hookrightarrow e_1(\, e_2 \,) \,\}} \tag{37.10d}$$

$$\frac{e_2 \ \mathsf{val}}{\nu\, a \, \{\, a \hookrightarrow (\ \lambda\, (\, x : \tau_2 \,)\, e \,)(\, e_2 \,) \,\} \underset{\mathsf{loc}}{\longmapsto} \nu\, a \, \{\, a \hookrightarrow \{e_2/x\}e \,\}} \tag{37.10e}$$

Rules (37.10a) and (37.10b) create tasks for the evaluation of the function and argument of an expression. Rules (37.10c) and (37.10d) propagate the result of evaluation of the function or argument of an application to the appropriate application expression. This rule mediates between the first two rules and Rule (37.10e), which effects a β-reduction in-place.

Chapter 28.

[3] Types are omitted from Σ for brevity.

The local transitions of the **P** machine corresponding to binary fork and join are as follows:

$$\left\{\begin{array}{c} \nu\, a\, \{\, a \hookrightarrow \mathtt{par}(\, e_1\,;e_2\,;x_1\,.\,x_2\,.\,e\,)\,\} \\ \longmapsto_{\mathsf{loc}} \\ \nu\, a_1, a_2, a\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \end{array}\right\} \tag{37.11a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1, a_2, a\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)\,\} \\ \longmapsto_{\mathsf{loc}} \\ \nu\, a\, \{\, a \hookrightarrow \{e_1, e_2/x_1, x_2\}e\,\} \end{array}\right\}} \tag{37.11b}$$

Rule (37.11a) creates two parallel tasks on which the executing task depends. The expression $\mathtt{join}[\,a_1\,;a_2\,](\,x_1\,;x_2\,.\,e\,)$ is blocked on tasks a_1 and a_2, so that no local step applies to it. Rule (37.11b) synchronizes a task with the tasks on which it depends once their execution has completed; those tasks are no longer required, and are eliminated from the state.

Each global transition is the simultaneous execution of one step of computation on as many as $p \geq 1$ processors.

$$\frac{\begin{array}{c} \nu\, \Sigma_1\, a_1\, \{\, \mu_1 \otimes a_1 \hookrightarrow s_1\,\} \longmapsto_{\mathsf{loc}} \nu\, \Sigma_1'\, a_1\, \{\, \mu_1' \otimes a_1 \hookrightarrow s_1'\,\} \\ \ldots \\ \nu\, \Sigma_n\, a_n\, \{\, \mu_n \otimes a_n \hookrightarrow s_n\,\} \longmapsto_{\mathsf{loc}} \nu\, \Sigma_n'\, a_n\, \{\, \mu_n' \otimes a_n \hookrightarrow s_n'\,\} \end{array}}{\left\{\begin{array}{c} \nu\, \Sigma_0\, \Sigma_1\, a_1\, \ldots\, \Sigma_n\, a_n\, \{\, \mu_0 \otimes \mu_1 \otimes a_1 \hookrightarrow s_1 \otimes \ldots \otimes \mu_n \otimes a_n \hookrightarrow s_n\,\} \\ \longmapsto_{\mathsf{glo}} \\ \nu\, \Sigma_0\, \Sigma_1'\, a_1\, \ldots\, \Sigma_n'\, a_n\, \{\, \mu_0 \otimes \mu_1' \otimes a_1 \hookrightarrow s_1' \otimes \ldots \otimes \mu_n' \otimes a_n \hookrightarrow s_n'\,\} \end{array}\right\}} \tag{37.12}$$

At each global step some number $1 \leq n \leq p$ of ready tasks are scheduled for execution, where n is maximal among the number of ready tasks. Because no two distinct tasks may depend on the same task, 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 adds two fresh tasks to the state resulting from 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 *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 accidentally reused among the parallel tasks.

The proof of a Brent-type Theorem for the **P** machine is now obvious. We need only ensure that the parameter n of Rule (37.12) is chosen as large as possible at each step, limited only by the parameter p and the number of ready tasks. A scheduler with this property is *greedy*; it never allows a processor to go idle if work remains to be done. Consequently, if there are always p available tasks at each global step, then the evaluation will complete in w/p steps, where w is

the work complexity of the program. If, at some stage, fewer than p tasks are available, then performance degrades according to the sequential dependencies among the sub-computations. In the limiting case the **P** machine must take at least d steps, where d is the depth of the computation.

37.5 Scheduling

The global transition relation of the **P** machine defined in Section 37.4 affords wide latitude in the choice of tasks that are advanced by taking a local transition. Doing so abstracts from implementation details that are irrelevant to the proof of the Brent-type Theorem given later in that section, the only requirement being that the number of tasks chosen be as large as possible up to the specified bound p, representing the number of available processors. When taking into account factors not considered here, it is necessary to specify the scheduling policy more precisely—for example, different scheduling policies may have asymptotically different space requirements. The overall idea is to consider scheduling a computation on p processors as a *p-way parallel traversal* of its cost graph, visiting up to p nodes at a time in an order consistent with the dependency ordering. In this section we will consider one such traversal, *p-way parallel depth-first-search*, or *p-DFS*, which specializes to the familiar depth-first traversal in the case that $p = 1$.

Recall that the depth first-search of a directed graph maintain a stack of unvisited nodes, which is initialized with the start node. At each round, a node is popped from the stack and visited, and then its unvisited children are pushed on the stack (in reverse order in the case of ordered graphs), completing that round. The traversal terminates when the stack is empty. When viewed as a scheduling strategy, visiting a node of a cost graph consists of scheduling the work associated with that node on a processor. The job of such as scheduler is to do the work of the computation in depth-first order, visiting the children of a node from left to right, consistently with the sequential dynamics (which would, in particular, treat a parallel binding as two sequential bindings). Notice that because a cost graph is directed acyclic, there are no "back edges" arising from the traversal, and because it is series-parallel in structure, there are no "cross edges". Thus, all children of a node are unvisited, and no task is considered more than once.

Although evocative, viewing scheduling as graph traversal invites one to imagine that the cost graph is given explicitly as a data structure, which is not at all the case. Instead the graph is created dynamically as the sub-computations are executed. At each round the computation associated with a node may *complete* (when it has achieved its value), *continue* (when more work is yet to be done), or *fork* (when it generates parallel sub-computations with a specified join point). Once a computation has completed and its value has been passed to the associated join point, its node in the cost graph is discarded. Furthermore, the children of a node only come into existence as a result of its execution, according to whether it completes (no children), continues (one child), or forks (two children). Thus one may envision that the cost graph "exists" as a cut through the abstract cost graph representing pending tasks that have not yet been activated by the traversal.

A parallel depth-first search works much the same way, except that as many as p nodes are visited at each round, constrained only by the presence of unvisited (yet-to-be-scheduled) nodes. One might naively think that this simply means popping up to p nodes from the stack on each round, visiting them all simultaneously, and pushing their dependents on the stack in reverse order, just as for conventional depth-first search. But a moment's thought reveals that this is not

correct. Because the cost graphs are ordered, the visited nodes form a sequence determined by the left-to-right ordering of the children of a node. If a node completes, it has no children and is removed from its position in the sequence in the next round. If a node continues, it has one child that occupies the same relative position as its parent in the next round. And if a node forks two children, they are inserted into the sequence after the predecessor, and immediately prior to that node, related to each other by the left-to-right ordering of the children. The task associated to the visited node itself becomes the join point of the immediately preceding pair of tasks, with which it will synchronize when they complete. Thus the visited sequence of $k \leq p$ nodes becomes, on the next round, anywhere from 0 (if all nodes completes) to $3 \times k$ nodes (if each node forks). These are placed into consideration, in the specified order, for the next round to ensure that they are processed in depth-first order. Importantly, the data structure maintaining the unvisited nodes of the graph is not a simple pushdown stack, because of the "in-place" replacement of each visited node by zero, one, or two nodes in between its predecessor and successor in the sequential ordering of the visited nodes.

Consider a variant of the **P** machine in which the order of the tasks is significant. A task is *finished* if it is a value, *blocked* if it is a join, and *ready* otherwise. Local transitions remain the same as in Section 37.4, bearing in mind that the ordering is significant. A global transition, however, consists of making a local transition on each of the first $k \leq p$ ready tasks.[4] After this selection the global state is depicted as follows:

$$\nu\, \Sigma_0\, a_1\, \Sigma_1\, \ldots\, a_k\, \Sigma_k\, \Sigma\, \{\, \mu_0 \otimes a_1 \hookrightarrow e_1 \otimes \mu_1 \otimes \ldots a_k \hookrightarrow e_k \otimes \mu \,\}$$

where each μ_i consists of finished or blocked tasks, and each e_i is ready. A schedule is greedy If $k < p$ only when no task in μ is ready.

After a local transition is made on each of the k selected tasks, the resulting global state has the form

$$\nu\, \Sigma_0 \Sigma_1'\, a_1\, \Sigma_1\, \ldots\, \Sigma_k'\, a_k\, \Sigma_k\, \Sigma\, \{\, \mu_0 \otimes \mu_1' \otimes a_1 \hookrightarrow e_1' \otimes \mu_1 \otimes \ldots \mu_k' \otimes a_k \hookrightarrow e_k' \otimes \mu \,\}$$

where each μ_i' represents the newly created task(s) of the local transition on task $a_i \hookrightarrow e_i$, and each e_i' is the expression resulting from the transition on that task. Next, all possible synchronizations are made by replacing each occurrence of an adjacent triple of the form

$$a_{i,1} \hookrightarrow e_1 \otimes a_{i,2} \hookrightarrow e_2 \otimes a_i \hookrightarrow \mathtt{join}[\, a_{i,1}\, ;\, a_{i,2}\,](\, x_1\, ;\, x_2\, .\, e\,)$$

(with e_1 and e_2 finished) by the task $a_i \hookrightarrow \{e_1, e_2 / x_1, x_2\} e$. Doing so propagates the values of tasks $a_{i,1}$ and $a_{i,2}$ to the join point, enabling the computation to continue. The two finished tasks are removed from the state, and the join point is no longer blocked.

37.6 Notes

Parallelism is a high-level programming concept that increases efficiency by carrying out multiple computations simultaneously when they are mutually independent. Parallelism does not change

[4] Thus the local transition given by Rule (37.11b) is never applicable; the dynamics of joins will be described shortly.

the meaning of a program, but only how fast it is executed. The cost dynamics specifies the number of steps required to execute a program sequentially and with maximal parallelism. A bounded implementation provides a bound on the number of steps when the number of processors is limited, limiting the degree of parallelism that can be realized. This formulation of parallelism was introduced by Blelloch (1990). The concept of a cost dynamics and the idea of a bounded implementation studied here are derived from Blelloch and Greiner (1995, 1996).

Exercises

37.1. Consider extending **PPCF** with exceptions, as described in Chapter 29, under the assumption that `exn` has at least two exception values. Give a sequential and a parallel structural dynamics to parallel let in such a way that determinacy continues to hold.

37.2. Give a matching cost dynamics to **PPCF** extended with exceptions (descibed in Exercise **37.1**) by inductively defining the following two judgments:

(a) $e \Downarrow^c v$, stating that e evaluates to value v with cost c;

(b) $e \Uparrow^c v$, stating that e raises the value v with cost c.

The analog of Theorem 37.6 remains valid for the dynamics. In particular, if $e \Uparrow^c v$, then both $e \xmapsto[\mathsf{seq}]{}{}^w \mathtt{raise}(v)$, where $w = wk(c)$, and $e \xmapsto[\mathsf{par}]{}{}^d \mathtt{raise}(v)$, where $d = dp(c)$, and conversely.

37.3. Extend the **P** machine to admit exceptions to match your solution to Exercise **37.2**. Argue that the revised machine supports a Brent-type validation of the cost dynamics.

37.4. Another way to express the dynamics of **PPCF** enriched with exceptions is by rewriting $\mathtt{par}(e_1 ; e_2 ; x_1 . x_2 . e)$ into another such parallel binding, $\mathtt{par}(e_1' ; e_2' ; x_1' . x_2' . e')$, which implements the correct dynamics to ensure determinacy. *Hint*: Extend **XPCF** with sums (Chapter 11), using them to record the outcome of each parallel sub-computation (e_1' derived from e_1, and e_2' derived from e_2), then check the outcomes (e' derived from e) in such a way to ensure determinacy.

Part XVI

Concurrency and Distribution

Chapter 40

Concurrent Algol

In this chapter we integrate concurrency into the framework of Modernized Algol described in Chapter 34. The resulting language, called Concurrent Algol, or **CA**, illustrates the integration of the mechanisms of the process calculus described in Chapter 39 into a practical programming language. To avoid distracting complications, we drop assignables from Modernized Algol entirely. (There is no loss of generality, however, because free assignables are definable in Concurrent Algol using processes as cells.)

The process calculus described in Chapter 39 is intended as a self-standing model of concurrent computation. When viewed in the context of a programming language, however, it is possible to streamline the machinery to take full advantage of types that are in any case required for other purposes. In particular the concept of a *channel*, which features prominently in Chapter 39, is identified with the concept of a *dynamic class* as described in Chapter 33. More precisely, we take *broadcast communication* of dynamically classified values as the basic synchronization mechanism of the language. Being dynamically classified, messages consist of a *payload* tagged with a *class*, or *channel*. The type of the channel determines the type of the payload. Importantly, only those processes that have access to the channel may decode the message; all others must treat it as inscrutable data that can be passed around but not examined. In this way we can model not only the mechanisms described in Chapter 39, but also formulate an abstract account of encryption and decryption in a network using the methods described in Chapter 39.

Concurrent Algol features a modal separation between commands and expressions like in Modernized Algol. It is also possible to combine these two levels (so as to allow benign concurrency effects), but we do not develop this approach in detail here.

40.1 Concurrent Algol

The syntax of **CA** is obtained by removing assignables from **MA**, and adding a syntactic level of *processes* to represent the global state of a program:

Typ	τ	$::=$	$\mathtt{cmd}(\tau)$	$\tau\,\mathtt{cmd}$	commands
Exp	e	$::=$	$\mathtt{cmd}(m)$	$\mathtt{cmd}\,m$	command
Cmd	m	$::=$	$\mathtt{ret}\,e$	$\mathtt{ret}\,e$	return
			$\mathtt{bnd}(e\,;x\,.\,m)$	$\mathtt{bnd}\,x \leftarrow e\,;m$	sequence
Proc	p	$::=$	$\mathtt{stop}$	$\mathbf{1}$	idle
			$\mathtt{run}(m)$	$\mathtt{run}(m)$	atomic
			$\mathtt{conc}(p_1\,;p_2)$	$p_1 \otimes p_2$	concurrent
			$\mathtt{newch}[\tau](a\,.\,p)$	$\nu\,a \sim \tau\,.\,p$	new channel

The process $\mathtt{run}(m)$ is an atomic process executing the command m. The other forms of process are adapted from Chapter 39. If Σ has the form $a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$, then we sometimes write $\nu\,\Sigma\{p\}$ for the iterated form $\nu\,a_1 \sim \tau_1 \ldots \nu\,a_n \sim \tau_n\,.\,p$.

The statics of **CA** is given by these judgments:

$\Gamma \vdash_\Sigma e : \tau$	expression typing
$\Gamma \vdash_\Sigma m \div \tau$	command typing
$\Gamma \vdash_\Sigma p\ \mathsf{proc}$	process formation
$\Gamma \vdash_\Sigma \alpha\ \mathsf{action}$	action formation

The expression and command typing judgments are essentially those of **MA**, augmented with the constructs described below.

Process formation is defined by the following rules:

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{40.1a}$$

$$\frac{\vdash_\Sigma m \div \tau}{\vdash_\Sigma \mathtt{run}(m)\ \mathsf{proc}} \tag{40.1b}$$

$$\frac{\vdash_\Sigma p_1\ \mathsf{proc} \quad \vdash_\Sigma p_2\ \mathsf{proc}}{\vdash_\Sigma p_1 \otimes p_2\ \mathsf{proc}} \tag{40.1c}$$

$$\frac{\vdash_{\Sigma, a \sim \tau} p\ \mathsf{proc}}{\vdash_\Sigma \nu\,a \sim \tau\,.\,p\ \mathsf{proc}} \tag{40.1d}$$

Processes are identified up to structural congruence, as described in Chapter 39.

Action formation is defined by the following rules:

$$\frac{}{\vdash_\Sigma \varepsilon\ \mathsf{action}} \tag{40.2a}$$

$$\frac{\vdash_\Sigma e : \texttt{clsfd} \quad e\ \mathsf{val}_\Sigma}{\vdash_\Sigma e\,!\ \mathsf{action}} \tag{40.2b}$$

$$\frac{\vdash_\Sigma e : \texttt{clsfd} \quad e\ \mathsf{val}_\Sigma}{\vdash_\Sigma e\,?\ \mathsf{action}} \tag{40.2c}$$

Messages are values of the type `clsfd` defined in Chapter 33.

The dynamics of **CA** is defined by transitions between processes, which represent the state of the computation. More precisely, the judgment $p \xmapsto[\Sigma]{\alpha} p'$ states that the process p evolves in one step to the process p' while undertaking action α.

$$\frac{m \xRightarrow[\Sigma]{\alpha} \nu\,\Sigma'\,\{\,m' \otimes p\,\}}{\texttt{run}(\,m\,) \xmapsto[\Sigma]{\alpha} \nu\,\Sigma'\{\texttt{run}(\,m'\,) \otimes p\}} \tag{40.3a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{run}(\,\texttt{ret}\,e\,) \xmapsto[\Sigma]{\varepsilon} \mathbf{1}} \tag{40.3b}$$

$$\frac{p_1 \xmapsto[\Sigma]{\alpha} p_1'}{p_1 \otimes p_2 \xmapsto[\Sigma]{\alpha} p_1' \otimes p_2} \tag{40.3c}$$

$$\frac{p_1 \xmapsto[\Sigma]{\alpha} p_1' \quad p_2 \xmapsto[\Sigma]{\overline{\alpha}} p_2'}{p_1 \otimes p_2 \xmapsto[\Sigma]{\varepsilon} p_1' \otimes p_2'} \tag{40.3d}$$

$$\frac{p \xmapsto[\Sigma, a \sim \tau]{\alpha} p' \quad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\,a \sim \tau\,.\,p \xmapsto[\Sigma]{\alpha} \nu\,a \sim \tau\,.\,p'} \tag{40.3e}$$

Rule (40.3a) states that a step of execution of the atomic process $\texttt{run}(\,m\,)$ consists of a step of execution of the command m, which may allocate some set Σ' of symbols or create a concurrent process p. This rule implements scope extrusion for classes (channels) by expanding the scope of the channel declaration to the context in which the command m occurs. Rule (40.3b) states that a completed command evolves to the inert (stopped) process; processes are executed solely for their effect, and not for their value.

Executing a command in **CA** may, in addition to evolving to another command, allocate a new channel or may spawn a new process. More precisely, the judgment[1]

$$m \xRightarrow[\Sigma]{\alpha} \nu\,\Sigma'\,\{\,m' \otimes p'\,\}$$

[1]The right-hand side of this judgment is a triple consisting of Σ', m', and p', not a process expression comprising these parts.

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.

The following rules define the execution of the basic forms of command inherited from **MA**:

$$\frac{e \xmapsto[\Sigma]{} e'}{\mathtt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\, e'} \tag{40.4a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\, \Sigma'\, \{\, m_1' \otimes p' \,\}}{\mathtt{bnd}\, x \leftarrow \mathtt{cmd}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\, \Sigma' \{\mathtt{bnd}\, x \leftarrow \mathtt{cmd}\, m_1' \,;\, m_2 \otimes p'\}} \tag{40.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd}\, x \leftarrow \mathtt{cmd}\,(\ \mathtt{ret}\, e\ )\,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \{e/x\}m_2} \tag{40.4c}$$

$$\frac{e_1 \xmapsto[\Sigma]{} e_1'}{\mathtt{bnd}\, x \leftarrow e_1 \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{bnd}\, x \leftarrow e_1' \,;\, m_2} \tag{40.4d}$$

These rules are supplemented by rules governing communication and synchronization among processes in the next two sections.

40.2 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 33. 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 33.4.1 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	::=	$\mathtt{spawn}(e)$	$\mathtt{spawn}(e)$	spawn
			$\mathtt{emit}(e)$	$\mathtt{emit}(e)$	emit message
			$\mathtt{acc}$	$\mathtt{acc}$	accept message
			$\mathtt{newch}[\tau]$	$\mathtt{newch}$	new channel

The command $\mathtt{spawn}(e)$ spawns a process that executes the encapsulated command given by e. The commands $\mathtt{emit}(e)$ and $\mathtt{acc}$ emit and accept messages, which are classified values whose

class is the channel on which the message is sent. The command $\texttt{newch}[\,\tau\,]$ returns a reference to a fresh class carrying values of type τ.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\,\texttt{unit}\,)}{\Gamma \vdash_\Sigma \texttt{spawn}(\,e\,) \div \texttt{unit}} \tag{40.5a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma \texttt{emit}(\,e\,) \div \texttt{unit}} \tag{40.5b}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{acc} \div \texttt{clsfd}} \tag{40.5c}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{newch}[\,\tau\,] \div \texttt{cls}(\,\tau\,)} \tag{40.5d}$$

Execution of these commands is defined as follows:

$$\frac{}{\texttt{spawn}(\,\texttt{cmd}(\,m\,)\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \texttt{ret}\,\langle\rangle \otimes \texttt{run}(\,m\,)} \tag{40.6a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{spawn}(\,e\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \texttt{spawn}(\,e'\,)} \tag{40.6b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{emit}(\,e\,) \underset{\Sigma}{\overset{e!}{\Longrightarrow}} \texttt{ret}\,\langle\rangle} \tag{40.6c}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{emit}(\,e\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \texttt{emit}(\,e'\,)} \tag{40.6d}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{acc} \underset{\Sigma}{\overset{e?}{\Longrightarrow}} \texttt{ret}\,e} \tag{40.6e}$$

$$\frac{}{\texttt{newch}[\,\tau\,] \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \nu\, a \sim \tau\,\{\,\texttt{ret}\,(\,\&\,a\,)\,\}} \tag{40.6f}$$

Rule (40.6c) specifies that $\texttt{emit}(\,e\,)$ has the effect of emitting the message e. Correspondingly, rule (40.6e) specifies that acc may accept (any) message that is being sent.

As usual, the preservation theorem for **CA** ensures that well-typed programs remain well-typed during execution. The proof of preservation requires a lemma about command execution.

Lemma 40.1. *If* $m \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} \nu\,\Sigma'\,\{\,m' \otimes p'\,\}$, $\vdash_\Sigma m \div \tau$, *then* $\vdash_\Sigma \alpha$ *action*, $\vdash_{\Sigma\,\Sigma'} m' \div \tau$, *and* $\vdash_{\Sigma\,\Sigma'} p'$ *proc.*

Proof. By induction on rules (40.4). □

With this in hand the proof of preservation goes along familiar lines.

Theorem 40.2 (Preservation). *If $\vdash_\Sigma p$ proc and $p \xmapsto[\Sigma]{} p'$, then $\vdash_\Sigma p'$ proc.*

Proof. By induction on transition, appealing to Lemma 40.1 for the crucial steps. □

Typing does not, however, guarantee progress with respect to unlabeled transition, for the simple reason that there may be no other process with which to communicate. By extending progress to labeled transitions we may state that this is the *only* way for process exceution to get stuck. But some care must be taken to account for allocating new channels.

Theorem 40.3 (Progress). *If $\vdash_\Sigma p$ proc, then either $p \equiv \mathbf{1}$, or $p \equiv \nu\,\Sigma'\{p'\}$ such that $p' \xmapsto[\Sigma\,\Sigma']{\alpha} p''$ for some $\vdash_{\Sigma\,\Sigma'} p''$ and some $\vdash_{\Sigma\,\Sigma'} \alpha$ action.*

Proof. By induction on rules (40.1) and (40.5). □

The progress theorem says that no process can get stuck for any reason other than the inability to communicate with another process. For example, a process that receives on a channel for which there is no sender is "stuck", but this does not violate Theorem 40.3.

40.3 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:

$$\mathtt{fix}\,\mathit{loop} : \tau\,\mathtt{cmd}\,\mathtt{is}\,\mathtt{cmd}\,\{x \leftarrow \mathtt{acc}\,;\mathtt{match}\,x\,\mathtt{as}\,a \cdot y \hookrightarrow \mathtt{ret}\,y\,\mathtt{ow} \hookrightarrow \mathtt{emit}(x)\,;\mathtt{do}\,\mathit{loop}\}$$

This command is always capable of receiving a broadcast message. When one arrives, it is examined to see whether it is classified by a. If so, the underlying classified value is returned; otherwise the message is re-broadcast so that another process may consider it. *Polling* consists of repeatedly executing the above command until a message of channel a is successfully accepted, if ever it is.

Polling is evidently impractical in most situations. An alternative is to change the language to allow for *selective communication*. Rather than accept any broadcast message, we may confine attention to messages sent only on certain channels. The type $\mathtt{event}(\tau)$ of *events* consists of a finite choice of accepts, all of whose payloads are of type τ.

Typ	τ	::=	$\mathtt{event}(\tau)$	$\tau\,\mathtt{event}$	events
Exp	e	::=	$\mathtt{rcv}[a]$	$?\,a$	selective read
			$\mathtt{never}[\tau]$	$\mathtt{never}$	null
			$\mathtt{or}(e_1\,;e_2)$	$e_1\,\mathtt{or}\,e_2$	choice
			$\mathtt{wrap}(e_1\,;x\,.\,e_2)$	$e_1\,\mathtt{as}\,x\,\mathtt{in}\,e_2$	post-composition
Cmd	m	::=	$\mathtt{sync}(e)$	$\mathtt{sync}(e)$	synchronize

Events in **CA** are similar to those of the asynchronous process calculus described in Chapter 39. The chief difference is that post-composition is considered as a general operation on events, instead of one tied to the receive event itself.

The statics of event expressions is given by the following rules:

$$\frac{\Sigma \vdash a \sim \tau}{\Gamma \vdash_\Sigma \mathtt{rcv}[a] : \mathtt{event}(\tau)} \tag{40.7a}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{never}[\tau] : \mathtt{event}(\tau)} \tag{40.7b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{event}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{or}(e_1 ; e_2) : \mathtt{event}(\tau)} \tag{40.7c}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{event}(\tau_1) \quad \Gamma, x : \tau_1 \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \mathtt{wrap}(e_1 ; x \,.\, e_2) : \mathtt{event}(\tau_2)} \tag{40.7d}$$

The corresponding dynamics is defined by these rules:

$$\frac{\Sigma \vdash a \sim \tau}{\mathtt{rcv}[a] \ \mathsf{val}_\Sigma} \tag{40.8a}$$

$$\frac{}{\mathtt{never}[\tau] \ \mathsf{val}_\Sigma} \tag{40.8b}$$

$$\frac{e_1 \ \mathsf{val}_\Sigma \quad e_2 \ \mathsf{val}_\Sigma}{\mathtt{or}(e_1 ; e_2) \ \mathsf{val}_\Sigma} \tag{40.8c}$$

$$\frac{e_1 \ \mathsf{val}_\Sigma}{\mathtt{wrap}(e_1 ; x \,.\, e_2) \ \mathsf{val}_\Sigma} \tag{40.8d}$$

$$\frac{e_1 \mathrel{\underset{\Sigma}{\longmapsto}} e_1'}{\mathtt{or}(e_1 ; e_2) \mathrel{\underset{\Sigma}{\longmapsto}} \mathtt{or}(e_1' ; e_2)} \tag{40.8e}$$

$$\frac{e_1 \ \mathsf{val}_\Sigma \quad e_2 \mathrel{\underset{\Sigma}{\longmapsto}} e_2'}{\mathtt{or}(e_1 ; e_2) \mathrel{\underset{\Sigma}{\longmapsto}} \mathtt{or}(e_1 ; e_2')} \tag{40.8f}$$

$$\frac{e_1 \mathrel{\underset{\Sigma}{\longmapsto}} e_1'}{\mathtt{wrap}(e_1 ; x \,.\, e_2) \mathrel{\underset{\Sigma}{\longmapsto}} \mathtt{wrap}(e_1' ; x \,.\, e_2')} \tag{40.8g}$$

Event values are identified up to structural congruence as described in Chapter 39.

The statics of the synchronization command is given by the following rule:

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \div \tau} \tag{40.9a}$$

The type of the event determines the type of value returned by the synchronization command.

Execution of a synchronization command depends on the event.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{sync}(\,e\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \texttt{sync}(\,e'\,)} \tag{40.10a}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \tau \quad \Sigma \vdash a \sim \tau}{\texttt{sync}(\,\texttt{rcv}[a]\,) \underset{\Sigma}{\overset{a \cdot e\,?}{\Longrightarrow}} \texttt{ret}(\,e\,)} \tag{40.10b}$$

$$\frac{\texttt{sync}(\,e_1\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_1}{\texttt{sync}(\,\texttt{or}(\,e_1\,;e_2\,)\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_1} \tag{40.10c}$$

$$\frac{\texttt{sync}(\,e_2\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_2}{\texttt{sync}(\,\texttt{or}(\,e_1\,;e_2\,)\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_2} \tag{40.10d}$$

$$\frac{\texttt{sync}(\,e_1\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_1}{\texttt{sync}(\,\texttt{wrap}(\,e_1\,;x\,.\,e_2\,)\,) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} \texttt{bnd}(\,\texttt{cmd}(\,m_1\,)\,;x\,.\,\texttt{ret}(\,e_2\,)\,)} \tag{40.10e}$$

Rule (40.10b) states that an acceptance on a channel a may synchronize only with messages classified by a. When combined with structural congruence, Rules (40.10c) and (40.10d) state that either event between two choices may engender an action. Rule (40.10e) yields the command that performs the command m_1 resulting from the action α taken by the event e_1, then returns e_2 with x bound to the return value of m_1.

Selective communication and dynamic events can 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 $\texttt{cls}(\,\tau\,)$, and let b be a channel carrying values of type τ, so that $\&\,b$ can 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 is accepted, the event $??\,x$ specifies a selective receipt on the channel referenced by x. So, if $\&\,b$ is sent along a, then the event $??\,\&\,b$ evaluates to $?\,b$, which accepts selectively on channel b, even though the receiving process may have no direct access to the channel b itself.

40.4 Free Assignables as Processes

Scope-free assignables are definable in **CA** by associating to each assignable a server process that sets and gets the contents of the assignable. To each assignable a of type τ is associated a server that selectively accepts a message on channel a with one of two forms:

1. $\texttt{get} \cdot (\ \&\, b\ )$, where b is a channel of type τ. This message requests that the contents of a be sent on channel b.
2. $\texttt{set} \cdot (\ \langle e, \&\, b\rangle\ )$, where e is a value of type τ, and b is a channel of type τ. 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

$$[\texttt{get} \hookrightarrow \tau\ \texttt{cls}\,, \texttt{set} \hookrightarrow \tau \times \tau\ \texttt{cls}].$$

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 τ 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:

$$\lambda\ (\ u : \tau_{srvr}\ \texttt{cls}\ )\ \texttt{fix}\, srvr : \tau \rightharpoonup \texttt{unit}\ \texttt{cmd}\ \texttt{is}\ \lambda\ (\ x : \tau\ )\ \texttt{cmd}\ \{y \leftarrow \texttt{sync}(\ ??\, u\ )\,; e_{(40.12)}\}. \tag{40.11}$$

The server is a procedure that takes an argument of type τ, 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:

$$\texttt{case}\, y\, \{\texttt{get} \cdot z \hookrightarrow e_{(40.13)} \mid \texttt{set} \cdot \langle x', z\rangle \hookrightarrow e_{(40.14)}\}. \tag{40.12}$$

A request to get the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\ \texttt{inref}(\ z\,; x\ )\ )\,; \texttt{do}\, srvr(\ x\ )\} \tag{40.13}$$

A request to set the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\ \texttt{inref}(\ z\,; x'\ )\ )\,; \texttt{do}\, srvr(\ x'\ )\} \tag{40.14}$$

The type $\tau\ \texttt{ref}$ is defined to be $\tau_{srvr}\ \texttt{cls}$, the type of channels (classes) to servers providing a cell containing a value of type τ. A new free assignable is created by the command $\texttt{ref}\, e_0$, which is defined to be

$$\{x \leftarrow \texttt{newch}\,; \_ \leftarrow \texttt{spawn}(\ e_{(40.11)}(\ x\ )(\ e_0\ )\ )\,; \texttt{ret}\, x\}. \tag{40.15}$$

A channel carrying a value of type τ_{srvr} is allocated to serve as the name of the assignable, and a new server is spawned that accepts requests on that channel, with initial value e_0 of type τ_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:

$$\{x \leftarrow \texttt{newch}\,; \_ \leftarrow \texttt{emit}(\ \texttt{inref}(\ e_0\,; \texttt{get} \cdot x\ )\ )\,; \texttt{sync}(\ ??\,(\ x\ )\ )\} \tag{40.16}$$

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:

$$\{x \leftarrow \texttt{newch}\,; \_ \leftarrow \texttt{emit}(\ \texttt{inref}(\ e_0\,; \texttt{set} \cdot \langle e_1, x\rangle\ )\ )\,; \texttt{sync}(\ ??\,(\ x\ )\ )\} \tag{40.17}$$

40.5 Notes

Concurrent Algol is a synthesis of process calculus and Modernized Algol; is essentially an "Algol-like" formulation of Concurrent ML (Reppy, 1999). The design is influenced by Parallel Algol (Brookes, 2002). Much work on concurrent interaction takes communication channels as a basic concept, but see Linda (Gelernter, 1985) for an account similar to the one suggested here.

Exercises

40.1. In Section 40.2 channels are allocated using the command `newch`, which returns a channel reference. Alternatively one may extend **CA** with a means of declaring channels just as assignables are declared in **MA**. Formulate the syntax, statics, and dynamics of such a construct, and derive `newch` using this extension.

40.2. Extend selective communication (Section 40.3) to account for channel references, which give rise to a new form of event. Give the syntax, statics, and dynamics of this extension.

40.3. Adapt the implementation of an RS latch given in Exercise **39.3** to **CA**.

Part XVII

Modularity

Part XVIII

Equational Reasoning

Part XIX

Appendices

Bibliography

Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 783–818. North-Holland, 1977. 20

John Allen. *Anatomy of LISP*. Computer Science Series. McGraw-Hill, 1978. 10

S.F. Allen, M. Bickford, R.L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006. ISSN 1570-8683. doi: 10.1016/j.jal.2005.10.005. 87

Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *LICS*, pages 215–221, 1987.

Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3): 265–301, 1997.

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. ISBN 3-540-18420-1.

Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991. 30

Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

Henk 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. 39

Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors. *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*. Cambridge University Press, 2009. 546

Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X. 361

Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995. 361

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996. 361

Manuel Blum. On the size of machines. *Information and Control*, 11(3):257–265, September 1967.

Stephen D. Brookes. The essence of parallel algol. *Inf. Comput.*, 179(1):118–149, 2002. 398

Samuel R. Buss, editor. *Handbook of Proof Theory*. Elsevier, Amsterdam, 1998. 544

Luca Cardelli. Structural subtyping and the notion of power type. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 70–79, 1988.

Luca Cardelli. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 266–277, 1997.

Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 151–162, 1994.

Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. 10

R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986. 10

Robert L. Constable. Types in logic, mathematics, and programming. In Buss (1998), chapter X.

Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987.

William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, pages 557–572, 2009.

Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University School of Computer Science, May 2005. Available as Technical Report CMU–CS–05–110.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In Martin Odersky and Philip Wadler, editors, *ICFP*, pages 198–208. ACM, 2000. ISBN 1-58113-202-6.

Ewen Denney. Refinement types for specification. In David Gries and Willem P. de Roever, editors, *PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 148–166. Chapman & Hall, 1998. ISBN 0-412-83760-9.

Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2003. ISBN 3-540-00897-7.

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.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS: Theoretical Computer Science*, 103, 1992. 49

Tim Freeman and Frank Pfenning. Refinement types for ml. In David S. Wise, editor, *PLDI*, pages 268–277. ACM, 1991. ISBN 0-89791-428-7.

Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, 1976.

David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 398

Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213. North-Holland, Amsterdam, 1969. 39

J.-Y. Girard. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*. These d'etat, Universite Paris VII, 1972. 148

Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated by Paul Taylor and Yves Lafont.

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. 79

Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979. 10, 273

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.

Timothy Griffin. A formulae-as-types notion of control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990.

Carl Gunter. *Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1992.

Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

Robert Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1): 71–84, 1992.

Robert Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 51(4):201–206, 1994. 332

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 123–137, 1994.

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 341–354, 1990.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1993. 10, 30

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. ISBN 3-540-20194-7.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

Tony Hoare. Null references: The billion dollar mistake. Presentation at QCon 2009, August 2009. 95

S. C. Kleene. *Introduction to Metamathematics*. van Nostrand, 1952. 10

Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976.

P. J. Landin. A correspondence between Algol 60 and Church's lambda notation. *CACM*, 8:89–101; 158–165, 1965. 48, 267

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 173–184, 2007.

Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 142–153, 1995.

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, May 1997.

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, second edition, 1998.

David B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Symposium on Principles of Programming Languges*, pages 277–286, 1986.

David B. MacQueen. Kahn networks at the dawn of functional programming. In Bertot et al. (2009), chapter 5.

Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In Runciman and Shivers (2003), pages 213–225. ISBN 1-58113-756-7.

Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science IV*, pages 153–175. North-Holland, 1980. 39, 55

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Unpublished Lecture Notes, 1983. 20, 30

Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Naples, Italy, 1984. 39

Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73 (3):407–420, 1987. 20, 30

John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1965. 10

N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.

Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978. 55

Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 39

John C. Mitchell. Coercion and type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

John C. Mitchell. Representation independence and data abstraction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 263–276, 1986.

John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989. ISBN 0-8186-1954-6. 320

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295, 2004. 320

Chetan R. Murthy. An evaluation semantics for classical proofs. In *LICS*, pages 96–107. IEEE Computer Society, 1991.

Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP*, pages 207–218. ACM, 2003. ISBN 1-58113-705-2.

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. 10, 30

B. Nordstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990. URL `http://www.cs.chalmers.se/Cs/Research/Logic/book`. 10

OCaml. Ocaml, 2012. URL http://caml.inria.fr/ocaml/.

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. ISBN 3-540-10576-X.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. 320

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. 87

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

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. ISBN 3-540-64781-3.

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. ISBN 3-540-44044-5.

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. ISBN 3-540-57182-5. 10

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, 1981. 48, 267

Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977. 175

Gordon D. Plotkin. The origins of structural operational semantics. *J. of Logic and Algebraic Programming*, 60:3–15, 2004. 49

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. 398

J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983. 148

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. ISBN 3-540-06859-7. 148

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. ISBN 3-540-10250-7.

John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981. 319, 332

John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In Andrew Kennedy and Nick Benton, editors, *TLDI*, pages 89–102. ACM, 2010. ISBN 978-1-60558-891-9.

Colin Runciman and Olin Shivers, editors. *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, 2003. ACM. ISBN 1-58113-756-7. 546, 550

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, 1980a.

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

Dana S Scott. Relating theories of the lambda calculus. *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 403–450, 1980b.

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. ISBN 3-540-11576-5.

Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65 (2/3):85–97, 1985.

Guy L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition edition, 1990. 273

Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722, 2006.

Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999.

P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + strategy = parallelism. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 8:23–60, 1998.

Jaap van Oosten. Realizability: A historical essay. *Mathematical Structures in Computer Science*, 12 (3):239–263, 2002.

Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989. 148

Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. 320

Philip Wadler. Call-by-value is dual to call-by-name. In Runciman and Shivers (2003), pages 189–201. ISBN 1-58113-756-7.

Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theor. Comput. Sci.*, 8:13–30, 1979.

Stephen A. Ward and Robert H. Halstead. *Computation structures*. MIT electrical engineering and computer science series. MIT Press, 1990. ISBN 978-0-262-23139-8.

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.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. 55, 332

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 249–257. ACM, 1998. ISBN 0-89791-987-4.

Index