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 Stephanie Balzer, Stephen Brookes, Evan Cavallo, Karl Crary, Jon Sterling, 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

Preface to the First Edition

Types are the central organizing principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs. The soundness of a language design—the absence of ill-defined programs—follows naturally.

The purpose of this book is to explain this remark. A variety of programming language features are analyzed in the unifying framework of type theory. A language feature is defined by its *statics*, the rules governing the use of the feature in a program, and its *dynamics*, the rules defining how programs using this feature are to be executed. The concept of *safety* emerges as the coherence of the statics and the dynamics of a language.

In this way we establish a foundation for the study of programming languages. But why these particular methods? The main justification is provided by the book itself. The methods we use are both *precise* and *intuitive*, providing a uniform framework for explaining programming language concepts. Importantly, these methods *scale* to a wide range of programming language concepts, supporting rigorous analysis of their properties. Although it would require another book in itself to justify this assertion, these methods are also *practical* in that they are *directly applicable* to implementation and *uniquely effective* as a basis for mechanized reasoning. No other framework offers as much.

Being a consolidation and distillation of decades of research, this book does not provide an exhaustive account of the history of the ideas that inform it. Suffice it to say that much of the development is not original, but rather is largely a reformulation of what has gone before. The notes at the end of each chapter signpost the major developments, but are not intended as a complete guide to the literature. For further information and alternative perspectives, the reader is referred to such excellent sources as Constable (1986), Constable (1998), Girard (1989), Martin-Löf (1984), Mitchell (1996), Pierce (2002, 2004), and Reynolds (1998).

The book is divided into parts that are, in the main, independent of one another. Parts I and II, however, provide the foundation for the rest of the book, and must therefore be considered prior to all other parts. On first reading it may be best to skim Part I, and begin in earnest with Part II, returning to Part I for clarification of the logical framework in which the rest of the book is cast.

Numerous people have read and commented on earlier editions of this book, and have suggested corrections and improvements to it. I am particularly grateful to Umut Acar, Jesper Louis Andersen, Carlo Angiuli, Andrew Appel, Stephanie Balzer, Eric Bergstrom, Guy E. Blelloch, Iliano Cervesato, Lin Chase, Karl Crary, Rowan Davies, Derek Dreyer, Dan Licata, Zhong Shao,

Rob Simmons, and Todd Wilson for their extensive efforts in reading and criticizing the book. I also thank the following people for their suggestions: Joseph Abrahamson, Arbob Ahmad, Zena Ariola, Eric Bergstrome, William Byrd, Alejandro Cabrera, Luis Caires, Luca Cardelli, Manuel Chakravarty, Richard C. Cobbe, James Cooper, Yi Dai, Daniel Dantas, Anupam Datta, Jake Donham, Bill Duff, Matthias Felleisen, Kathleen Fisher, Dan Friedman, Peter Gammie, Maia Ginsburg, Byron Hawkins, Kevin Hely, Kuen-Bang Hou (Favonia), Justin Hsu, Wojciech Jedynak, Cao Jing, Salil Joshi, Gabriele Keller, Scott Kilpatrick, Danielle Kramer, Dan Kreysa, Akiva Leffert, Ruy Ley-Wild, Karen Liu, Dave MacQueen, Chris Martens, Greg Morrisett, Stefan Muller, Tom Murphy, Aleksandar Nanevski, Georg Neis, David Neville, Adrian Trejo Nuñez, Cyrus Omar, Doug Perkins, Frank Pfenning, Jean Pichon, Benjamin Pierce, Andrew M. Pitts, Gordon Plotkin, David Renshaw, John Reynolds, Andreas Rossberg, Carter Schonwald, Dale Schumacher, Dana Scott, Shayak Sen, Pawel Sobocinski, Kristina Sojakova, Daniel Spoonhower, Paulo Tanimoto, Joe Tassarotti, Peter Thiemann, Bernardo Toninho, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, Sergei Winitzki, Roger Wolff, Omer Zach, Luke Zarko, and Yu Zhang. I am very grateful to the students of 15–312 and 15–814 at Carnegie Mellon who have provided the impetus for the preparation of this book and who have endured the many revisions to it over the last ten years.

I thank the Max Planck Institute for Software Systems for its hospitality and support. I also thank Espresso a Mano in Pittsburgh, CB2 Cafe in Cambridge, and Thonet Cafe in Saarbrücken for providing a steady supply of coffee and a conducive atmosphere for writing.

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

Robert Harper
Pittsburgh
March, 2012

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 $\texttt{num}[\,n\,]$ 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

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

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

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

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:

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

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

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:

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. Define $\mathcal{P}(a)$ to mean that if $a = \texttt{succ}(\,b\,)$, then b nat. It suffices to show that $\mathcal{P}$ is closed under rules (2.2).

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

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

□

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 zero is zero.

Rule (2.2b) Assume that a is a. It follows that $\texttt{succ}(\,a\,)$ is $\texttt{succ}(\,a\,)$ 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

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

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

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

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 $\mathtt{succ}(\,\mathtt{zero}\,)\ \mathtt{even}$ from rules (2.8). Even so, the derivability

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

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

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

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:

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

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

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

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 `^` 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 $x \notin dom(\Gamma)$ 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 $x \notin dom(\Gamma)$ and any type τ.*

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

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

from which the result follows by rule (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, $e' = \mathtt{let}(\, e_1; z.e_2 \,)$, where z is chosen so that $z \neq x$ and $z \notin \mathit{dom}(\Gamma)$. 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

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

It follows by rule (4.1h) that $\Gamma \vdash [e/x]\mathtt{let}(\, e_1; z.e_2 \,) : \tau'$, 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 judgment $s \downarrow$ means that there is a complete transition sequence starting from s, which is to say that there exists s' final such that $s \longmapsto^* s'$.

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 $s \longmapsto^* ns'$, where $n \geq 0$, is inductively defined by the following rules.

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

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

Theorem 5.1. *For all states s and s', $s \longmapsto^* s'$ iff $s \longmapsto^* ks'$ 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:

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

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

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

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

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

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

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

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

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.

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

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:

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

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.

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

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

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

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.)

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

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

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

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

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

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

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

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

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

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

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

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

□

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:

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

But the expression $\texttt{div}(\texttt{num}[3]; \texttt{num}[0])$ 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:

$$\frac{e_1 \text{ val}}{\texttt{div}(e_1;\texttt{num}[0]) \text{ err}} \tag{6.1a}$$

$$\frac{e_1 \text{ err}}{\texttt{div}(e_1;e_2) \text{ err}} \tag{6.1b}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ err}}{\texttt{div}(e_1;e_2) \text{ err}} \tag{6.1c}$$

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:

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

$$\frac{}{\texttt{error err}} \tag{6.2b}$$

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.

Chapter 7

Evaluation Dynamics

In Chapter 5 we defined evaluation of expressions in **E** using a structural dynamics. Structural dynamics is very useful for proving safety, but for some purposes, such as writing a user manual, another formulation, called *evaluation dynamics* is preferable. An evaluation dynamics is a relation between a phrase and its value that is defined without detailing the step-by-step process of evaluation. A *cost dynamics* enriches an evaluation dynamics with a *cost measure* specifying the resource usage of evaluation. A prime example is time, measured as the number of transition steps required to evaluate an expression according to its structural dynamics.

7.1 Evaluation Dynamics

An *evaluation dynamics*, consists of an inductive definition of the evaluation judgment $e \Downarrow v$ stating that the closed expression e evaluates to the value v. The evaluation dynamics of **E** is defined by the following rules:

$$\frac{}{\mathtt{num}[\,n\,] \Downarrow \mathtt{num}[\,n\,]} \tag{7.1a}$$

$$\frac{}{\mathtt{str}[\,s\,] \Downarrow \mathtt{str}[\,s\,]} \tag{7.1b}$$

$$\frac{e_1 \Downarrow \mathtt{num}[\,n_1\,] \quad e_2 \Downarrow \mathtt{num}[\,n_2\,] \quad n_1 + n_2 = n}{\mathtt{plus}(\,e_1; e_2\,) \Downarrow \mathtt{num}[\,n\,]} \tag{7.1c}$$

$$\frac{e_1 \Downarrow \mathtt{str}[\,s_1\,] \quad e_2 \Downarrow \mathtt{str}[\,s_2\,] \quad s_1 \hat{} s_2 = s}{\mathtt{cat}(\,e_1; e_2\,) \Downarrow \mathtt{str}[\,s\,]} \tag{7.1d}$$

$$\frac{e \Downarrow \mathtt{str}[\,s\,] \quad |s| = n}{\mathtt{len}(\,e\,) \Downarrow \mathtt{num}[\,n\,]} \tag{7.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let}(\,e_1; x.e_2\,) \Downarrow v_2} \tag{7.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are not syntax-directed, because the premise of rule (7.1f) is not a sub-expression of the expression in the conclusion of that rule.

Rule (7.1f) specifies a by-name interpretation of definitions. For a by-value interpretation the following rule should be used instead:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let}(\, e_1; x.e_2 \,) \Downarrow v_2} \tag{7.2}$$

Because the evaluation judgment is inductively defined, we prove properties of it by rule induction. Specifically, to show that the property $\mathcal{P}(e \Downarrow v)$ holds, it is enough to show that $\mathcal{P}$ is closed under rules (7.1):

1. Show that $\mathcal{P}(\texttt{num}[\, n \,] \Downarrow \texttt{num}[\, n \,])$.
2. Show that $\mathcal{P}(\texttt{str}[\, s \,] \Downarrow \texttt{str}[\, s \,])$.
3. Show that $\mathcal{P}(\texttt{plus}(\, e_1; e_2 \,) \Downarrow \texttt{num}[\, n \,])$, if $\mathcal{P}(e_1 \Downarrow \texttt{num}[\, n_1 \,])$, $\mathcal{P}(e_2 \Downarrow \texttt{num}[\, n_2 \,])$, and $n_1 + n_2 = n$.
4. Show that $\mathcal{P}(\texttt{cat}(\, e_1; e_2 \,) \Downarrow \texttt{str}[\, s \,])$, if $\mathcal{P}(e_1 \Downarrow \texttt{str}[\, s_1 \,])$, $\mathcal{P}(e_2 \Downarrow \texttt{str}[\, s_2 \,])$, and $s_1 \hat{} s_2 = s$.
5. Show that $\mathcal{P}(\texttt{let}(\, e_1; x.e_2 \,) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

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

Lemma 7.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on rules (7.1). All cases except rule (7.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

7.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for **E**. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics suppresses the intermediate states, focusing attention on the initial and final states alone. This remark suggests that the right correspondence is between *complete* execution sequences in the structural dynamics and the evaluation judgment in the evaluation dynamics.

Theorem 7.2. *For all closed expressions e and values v, $e \longmapsto^* v$ iff $e \Downarrow v$.*

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

Lemma 7.3. *If $e \Downarrow v$, then $e \longmapsto^* v$.*

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

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

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

For the converse, recall from Chapter 5 the definitions of multi-step evaluation and complete evaluation. Because $v \Downarrow v$ when v val, it suffices to show that evaluation is closed under converse evaluation:[1]

Lemma 7.4. *If $e \longmapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

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

7.3 Type Safety, Revisited

Type safety is defined in Chapter 6 as preservation and progress (Theorem 6.1). These concepts are meaningful when applied to a dynamics given by a transition system, as we shall do throughout this book. But what if we had instead given the dynamics as an evaluation relation? How is type safety proved in that case?

The answer, unfortunately, is that we cannot. Although there is an analog of the preservation property for an evaluation dynamics, there is no clear analog of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. It can be readily proved by induction on the evaluation rules. But what is the analog of progress? We might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. Although this property is true for **E**, it demands much more than just progress — it requires that every expression evaluate to a value! If **E** were extended to admit operations that may result in an error (as discussed in Section 6.3), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude towards this situation is to conclude that type safety cannot be properly discussed in the context of an evaluation dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics with explicit checks for dynamic type errors, and to show that any expression with a dynamic type fault must be statically ill-typed. Re-stated in the contrapositive, this means that a statically well-typed program cannot incur a dynamic type error. A difficulty with this point of view is that we must explicitly account for a form of error

[1] Converse evaluation is also known as *head expansion*.

solely to prove that it cannot arise! Nevertheless, a semblance of type safety can be established using evaluation dynamics.

We define a judgment e ?? stating that the expression e *goes wrong* when executed. The exact definition of "going wrong" is given by a set of rules, but the intention is that it should cover all situations that correspond to type errors. The following rules are representative of the general case:

$$\frac{}{\texttt{plus}(\,\texttt{str}[\,s\,];e_2\,)\;??} \tag{7.3a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus}(\,e_1;\texttt{str}[\,s\,]\,)\;??} \tag{7.3b}$$

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

Theorem 7.5. *If e ??, then there is no τ such that $e : \tau$.*

Proof. By rule induction on rules (7.3). For example, for rule (7.3a), we note that $\texttt{str}[\,s\,] : \texttt{str}$, and hence $\texttt{plus}(\,\texttt{str}[\,s\,];e_2\,)$ is ill-typed. □

Corollary 7.6. *If $e : \tau$, then $\neg(e\;??)$.*

Apart from the inconvenience of having to define the judgment e ?? only to show that it is irrelevant for well-typed programs, this approach suffers a very significant methodological weakness. If we should omit one or more rules defining the judgment e ??, the proof of Theorem 7.5 remains valid; there is nothing to ensure that we have included sufficiently many checks for run-time type errors. We can prove that the ones we define cannot arise in a well-typed program, but we cannot prove that we have covered all possible cases. By contrast the structural dynamics does not specify any behavior for ill-typed expressions. Consequently, any ill-typed expression will "get stuck" without our explicit intervention, and the progress theorem rules out all such cases. Moreover, the transition system corresponds more closely to implementation—a compiler need not make any provisions for checking for run-time type errors. Instead, it relies on the statics to ensure that these cannot arise, and assigns no meaning to any ill-typed program. Therefore, execution is more efficient, and the language definition is simpler.

7.4 Cost Dynamics

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

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

$$\frac{}{\texttt{num}[\,n\,] \Downarrow^0 \texttt{num}[\,n\,]} \tag{7.4a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num}[n_1] \quad e_2 \Downarrow^{k_1} \mathtt{num}[n_2]}{\mathtt{plus}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{num}[n_1 + n_2]} \tag{7.4b}$$

$$\frac{}{\mathtt{str}[s] \Downarrow^{0} \mathtt{str}[s]} \tag{7.4c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\mathtt{cat}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{str}[s_1 \hat{} s_2]} \tag{7.4d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{7.4e}$$

For a by-value interpretation of `let`, rule (7.4e) is replaced by the following rule:

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad [v_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_1+k_2+1} v_2} \tag{7.5}$$

Theorem 7.7. *For any closed expression e and closed value v of the same type, $e \Downarrow^{k} v$ iff $e \longmapsto^{*} kv$.*

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

7.5 Notes

The structural similarity between evaluation dynamics and typing rules was first developed in *The Definition of Standard ML* (Milner et al., 1997). The advantage of evaluation dynamics is its directness; its disadvantage is that it is not well-suited to proving properties such as type safety. Robin Milner introduced the apt phrase "going wrong" as a description of a type error. Cost dynamics was introduced by Blelloch and Greiner (1996) in a study of parallel computation (see Chapter 37).

Exercises

7.1. Show that evaluation is deterministic: if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

7.2. Complete the proof of Lemma 7.3.

7.3. Complete the proof of Lemma 7.4. Then show that if $e \longmapsto^{*} e'$ with e' val, then $e \Downarrow e'$.

7.4. Augment the evaluation dynamics with checked errors, along the lines sketched in Chapter 5, using e ?? to say that e incurs a checked (or an unchecked) error. What remains unsatisfactory about the type safety proof? Can you think of a better alternative?

7.5. Consider generic hypothetical judgments of the form

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

where v_1 val, $\ldots$, v_n val, and v val. The hypotheses, written Δ, are called the *environment* of the evaluation; they provide the values of the free variables in e. The hypothetical judgment $\Delta \vdash e \Downarrow v$ is called an *environmental evaluation dynamics*.

Give a hypothetical inductive definition of the environmental evaluation dynamics *without making any use of substitution*. In particular, you should include the rule

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

defining the evaluation of a free variable.

Show that $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$ (using the by-value form of evaluation).

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
			$\rightarrow(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$	function
Exp	e	::=	x	x	variable
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(e)$	$\mathtt{s}(e)$	successor
			$\mathtt{rec}\{e_0;x.y.e_1\}(e)$	$\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow e_1\}$	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 $\mathtt{rec}\{e_0;x.y.e_1\}(e)$ 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*, $\mathtt{iter}\{e_0; y.e_1\}(\,e\,)$, 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:

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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:

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

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

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

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
			$\times(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	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	::=	$\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I})$	$\langle \tau_i \rangle_{i \in I}$	product
Exp	e	::=	$\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I})$	$\langle e_i \rangle_{i \in I}$	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 $\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\prod_{i \in I} \tau_i$ 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 $\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I})$, or $\langle e_i \rangle_{i \in I}$ 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:

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

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

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

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

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 $\mathtt{iter}\{e_0; x.e_1\}(\,e\,)$, we may define $\mathtt{rec}\{e_0; x.y.e_1\}(\,e\,)$ to be $e' \cdot \mathtt{r}$, where e' is the expression

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

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 $\prod_{i \in I} \tau$, 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
			$+(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	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
			$\mathtt{case}\{x_1.e_1;x_2.e_2\}(e)$	$\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}$	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 $e\ \mathsf{val}$ 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	$::=$	$\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$	$[\tau_i]_{i \in I}$	sum
Exp	e	$::=$	$\mathtt{in}[i]\{\vec{\tau}\}(e)$	$i \cdot e$	injection
			$\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$	$\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I}$	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 $\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\sum_{i \in I} \tau_i$ 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 $\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$.

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:

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

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

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

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

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:

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

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

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

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

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

$$\frac{}{\texttt{false val}} \tag{11.6b}$$

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

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

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

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

$$\texttt{bool} = \texttt{unit} + \texttt{unit} \tag{11.7a}$$
$$\texttt{true} = \texttt{l} \cdot \langle\rangle \tag{11.7b}$$
$$\texttt{false} = \texttt{r} \cdot \langle\rangle \tag{11.7c}$$
$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{case } e \, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.7d}$$

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

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

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = [\texttt{unit}]_{\_ \in I}$, 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 `chcode`(n) that yield the `char` associated to the code n, and `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
			$\texttt{ifnull}\{\tau\}\{e_1; x.e_2\}(e)$	$\texttt{ifnull}\,e\,\{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x) \hookrightarrow e_2\}$	null test

The type $\texttt{opt}(\tau)$ represents the type of "optional" values of type τ. The introduction forms are $\texttt{null}$, 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 $\texttt{true}$ if its argument is null, and $\texttt{false}$ otherwise. Such a test allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

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

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

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

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 $\tau\ \texttt{opt}$ to be the type $\texttt{bool} \times \tau$ by filling in the following chart:

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

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 $\texttt{bool} \times \texttt{bool} \times \texttt{bool} \times \texttt{bool}$. 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 $\texttt{nat} \to \texttt{bool}$ 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
			$\rightarrow(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$	function
			$\forall(t.\tau)$	$\forall(t.\tau)$	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*, $\forall(t.\tau)$, classifies polymorphic functions.

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

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

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:

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

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \rightarrow(\tau_1;\tau_2) \text{ type}} \tag{16.1b}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \forall(t.\tau) \text{ type}} \tag{16.1c}$$

The rules defining the typing judgment are as follows:

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

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \lambda\{\tau_1\}(x.e) : \rightarrow(\tau_1;\tau_2)} \tag{16.2b}$$

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

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

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

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

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 $\tau = \rightarrow(\,\tau_1;\tau_2\,)$, then $e = \lambda\{\tau_1\}(\,x.e_2\,)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \forall(\,t.\tau'\,)$, 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:

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

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:

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

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.

Chapter 17

Abstract Types

Data abstraction is perhaps the most important technique for structuring programs. The main idea is to introduce an *interface* that serves as a contract between the *client* and the *implementor* of an abstract type. The interface specifies what the client may rely on for its own work, and, simultaneously, what the implementor must provide to satisfy the contract. The interface serves to isolate the client from the implementor so that each may be developed in isolation from the other. In particular one implementation can be replaced by another without affecting the behavior of the client, provided that the two implementations meet the same interface and that each simulates the other with respect to the operations of the interface. This property is called *representation independence* for an abstract type.

Data abstraction is formalized by extending the language **F** with *existential types*. Interfaces are existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are packages, the introduction form for existential types, and clients are uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is captured so naturally and directly by the logical concept of existential type quantification. Existential types are closely connected with universal types, and hence are often treated together. The superficial reason is that both are forms of type quantification, and hence both require the machinery of type variables. The deeper reason is that existential types are *definable* from universals — surprisingly, data abstraction is actually just a form of polymorphism! Consequently, representation independence is an application of the parametricity properties of polymorphic functions discussed in Chapter 16.

17.1 Existential Types

The syntax of **FE** extends **F** with the following constructs:

Typ	τ	::=	$\exists(t.\tau)$	$\exists(t.\tau)$	interface
Exp	e	::=	$\texttt{pack}\{t.\tau\}\{\rho\}(e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$	implementation
			$\texttt{open}\{t.\tau\}\{\rho\}(e_1; t, x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$	client

The introduction form $\exists(\,t.\tau\,)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(\,t.\tau\,)$, where ρ is a type and e is an expression of type $[\rho/t]\tau$. The type ρ is the *representation type* of the package, and the expression e is the *implementation* of the package. The elimination form is the expression $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$, which *opens* the package e_1 for use within the *client* e_2 by binding its representation type to t and its implementation to x for use within e_2. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it can be varied without affecting the type correctness of the client.

The abstract syntax of the `open` construct specifies that the type variable t and the expression variable x are bound within the client. They may be renamed at will by α-equivalence without affecting the meaning of the construct, provided, of course, that the names do not conflict with any others in scope. In other words the type t is a "new" type, one that is distinct from all other types, when it is introduced. This principle is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior relies on the identification convention stated in Chapter 1.

17.1.1 Statics

The statics of **FE** is given by these rules:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \exists(\,t.\tau\,)\ \mathsf{type}} \tag{17.1a}$$

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}\{t.\tau\}\{\rho\}(\,e\,) : \exists(\,t.\tau\,)} \tag{17.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \exists(\,t.\tau\,) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}\{t.\tau\}\{\tau_2\}(\,e_1; t, x.e_2\,) : \tau_2} \tag{17.1c}$$

Rule (17.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, τ_2, must not involve the abstract type t. This restriction prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.
2. The body of the client, e_2, is type checked without knowledge of the representation type, t. The client is, in effect, polymorphic in the type variable t.

Lemma 17.1 (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$*. If* $\Delta \vdash \tau_i\ \mathsf{type}$ *for each* $x_i : \tau_i$ *in* Γ*, then* $\Delta \vdash \tau\ \mathsf{type}$*.*

Proof. By induction on rules (17.1), using substitution for expressions and types. □

17.1.2 Dynamics

The dynamics of **FE** is defined by the following rules (including the bracketed material for an eager interpretation, and omitting it for a lazy interpretation):

$$\frac{[e\ \mathsf{val}]}{\texttt{pack}\{t.\tau\}\{\rho\}(\,e\,)\ \mathsf{val}} \tag{17.2a}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{pack}\{t.\tau\}\{\rho\}(\,e\,) \longmapsto \mathtt{pack}\{t.\tau\}\{\rho\}(\,e'\,)}\right] \tag{17.2b}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{open}\{t.\tau\}\{\tau_2\}(\,e_1; t, x.e_2\,) \longmapsto \mathtt{open}\{t.\tau\}\{\tau_2\}(\,e_1'; t, x.e_2\,)} \tag{17.2c}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{open}\{t.\tau\}\{\tau_2\}(\,\mathtt{pack}\{t.\tau\}\{\rho\}(\,e\,); t, x.e_2\,) \longmapsto [\rho, e/t, x]e_2} \tag{17.2d}$$

It is important to see that, according to these rules, *there are no abstract types at run time*! The representation type is propagated to the client by substitution when the package is opened, thereby eliminating the abstraction boundary between the client and the implementor. Thus, data abstraction is a *compile-time discipline* that leaves no traces of its presence at execution time.

17.1.3 Safety

Safety of **FE** is stated and proved by decomposing it into progress and preservation.

Theorem 17.2 (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof. By rule induction on $e \longmapsto e'$, using substitution for both expression- and type variables. □

Lemma 17.3 (Canonical Forms)**.** *If $e : \exists(\,t.\tau\,)$ and e val, then $e = \mathtt{pack}\{t.\tau\}\{\rho\}(\,e'\,)$ for some type ρ and some e' such that $e' : [\rho/t]\tau$.*

Proof. By rule induction on the statics, using the definition of closed values. □

Theorem 17.4 (Progress)**.** *If $e : \tau$ then either e val or there exists e' such that $e \longmapsto e'$.*

Proof. By rule induction on $e : \tau$, using the canonical forms lemma. □

17.2 Data Abstraction

To illustrate the use of **FE**, we consider an abstract type of queues of natural numbers supporting three operations:

1. Forming the empty queue.
2. Inserting an element at the tail of the queue.
3. Removing the head of the queue, which is assumed non-empty.

This is clearly a bare-bones interface, but suffices to illustrate the main ideas of data abstraction. Queue elements are natural numbers, but nothing depends on this choice.

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. The behavior of a queue is expressed by the existential type $\exists(\,t.\tau\,)$ which serves as the interface of the queue abstraction:

$$\exists(\,t.\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times t \to t, \mathtt{rem} \hookrightarrow t \to (\,\mathtt{nat} \times t\,)\,\mathtt{opt}\rangle\,).$$

The representation type t of queues is *abstract* — all that is known about it is that it supports the operations `emp`, `ins`, and `rem`, with the given types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations. Here is a very simple implementation e_l in which queues are represented as lists:

$$\texttt{pack}\,\texttt{natlist}\,\texttt{with}\,\langle \texttt{emp} \hookrightarrow \texttt{nil}, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r \rangle\,\texttt{as}\,\exists(\,t.\tau\,),$$

where

$$e_i : \texttt{nat} \times \texttt{natlist} \to \texttt{natlist} = \lambda\,(\,x : \texttt{nat} \times \texttt{natlist}\,)\ldots,$$

and

$$e_r : \texttt{natlist} \to \texttt{nat} \times \texttt{natlist} = \lambda\,(\,x : \texttt{natlist}\,)\ldots.$$

The elided body of e_i conses the first component of x, the element, onto the second component of x, the queue, and the elided body of e_r reverses its argument, and returns the head element paired with the reversal of the tail. Both of these operations "know" that queues are represented as values of type `natlist`, and are programmed accordingly.

It is also possible to give another implementation e_p of the same interface $\exists(\,t.\tau\,)$, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half". This two-part representation avoids the need for reversals on each call, and, as a result, achieves amortized constant-time behavior:

$$\texttt{pack}\,\texttt{natlist} \times \texttt{natlist}\,\texttt{with}\,\langle \texttt{emp} \hookrightarrow \langle \texttt{nil}, \texttt{nil} \rangle, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r \rangle\,\texttt{as}\,\exists(\,t.\tau\,).$$

In this case e_i has type

$$\texttt{nat} \times (\,\texttt{natlist} \times \texttt{natlist}\,) \to (\,\texttt{natlist} \times \texttt{natlist}\,),$$

and e_r has type

$$(\,\texttt{natlist} \times \texttt{natlist}\,) \to \texttt{nat} \times (\,\texttt{natlist} \times \texttt{natlist}\,).$$

These operations "know" that queues are represented as values of type $\texttt{natlist} \times \texttt{natlist}$, and are implemented accordingly.

The important point is that the *same* client type checks regardless of which implementation of queues we choose, because the representation type is hidden, or *held abstract*, from the client during type checking. Consequently, it cannot rely on whether it is `natlist` or $\texttt{natlist} \times \texttt{natlist}$ or some other type. That is, the client is *independent* of the representation of the abstract type.

17.3 Definability of Existential Types

The language **FE** is not a proper extension of **F**, because existential types (under a lazy dynamics) are definable in terms of universal types. Why should this be possible? Note that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2 : \tau_2,$$

where $e_1 : \exists(\, t.\tau \,)$, specifies that $e_2 : \tau_2$ under the assumptions t type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(\, t.\tau \to \tau_2 \,),$$

where t may occur in τ (the type of the operations), but not in τ_2 (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned}
\exists(\, t.\tau \,) &\triangleq \forall(\, u.\forall(\, t.\tau \to u \,) \to u \,) \\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(\, t.\tau \,) &\triangleq \Lambda(\, u \,)\,\lambda\,(\, x : \forall(\, t.\tau \to u \,)\,)\,x[\,\rho\,](\, e \,) \\
\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2 &\triangleq e_1[\,\tau_2\,](\,\Lambda(\, t \,)\,\lambda\,(\, x : \tau \,)\,e_2 \,)
\end{aligned}$$

An existential is encoded as a polymorphic function taking the overall result type u as argument, followed by a polymorphic function representing the client with result type u, and yielding a value of type u as overall result. Consequently, the `open` construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, τ_2, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type τ_2 of the `open` construct.) Finally, a package consisting of a representation type ρ and an implementation e is a polymorphic function that, when given the result type u and the client x instantiates x with ρ and passes to it the implementation e.

17.4 Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types. More precisely, there is a criterion, *bisimilarity*, for relating two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation by another that is bisimilar to it. This principle leads to a simple method for proving the correctness of a *candidate* implementation of an abstract type, which is to show that it is bisimilar to an obviously correct *reference* implementation of it. Because the candidate and the reference implementations are bisimilar, no client may distinguish them from one another, and hence if the client behaves properly with the reference implementation, then it must also behave properly with the candidate.

To derive the definition of bisimilarity of implementations, it is helpful to examine the definition of existential types in terms of universals given in Section 17.3. It is immediately clear that the client of an abstract type is polymorphic in the representation of the abstract type. A client c of an abstract type $\exists(\, t.\tau \,)$ has type $\forall(\, t.\tau \to \tau_2 \,)$, where t does not occur free in τ_2 (but may, of course, occur in τ). Applying the parametricity property described informally in Chapter 16 (and developed rigorously in Chapter 48), this says that if R is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on them. The fact that t does not occur in the result type ensures that the behavior of the client is independent of the choice of relation between the implementations, provided that this relation is preserved by the operations that implement it.

Explaining what is a bisimulation is best done by example. Consider the existential type $\exists(\,t.\tau\,)$, where τ is the labeled tuple type

$$\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times t \to t, \mathtt{rem} \hookrightarrow t \to (\,\mathtt{nat} \times t\,)\,\mathtt{opt}\rangle.$$

This specifies an abstract type of queues. The operations `emp`, `ins`, and `rem` specify, respectively, the empty queue, an insert operation, and a remove operation. For the sake of simplicity the element type is the type of natural numbers. The result of removal is an optional pair, according to whether the queue is empty or not.

Theorem 48.12 ensures that if ρ and ρ' are any two closed types, and if R is a relation between expressions of these two types, then if the implementations $e : [\rho/x]\tau$ and $e' : [\rho'/x]\tau$ respect R, then $c[\,\rho\,]e$ behaves the same as $c[\,\rho'\,]e'$. It remains to define when two implementations respect the relation R. Let

$$e \triangleq \langle \mathtt{emp} \hookrightarrow e_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e_{\mathsf{r}}\rangle$$

and

$$e' \triangleq \langle \mathtt{emp} \hookrightarrow e'_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e'_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e'_{\mathsf{r}}\rangle.$$

For these implementations to respect R means that the following three conditions hold:

1. The empty queues are related: $R(e_{\mathsf{m}}, e'_{\mathsf{m}})$.
2. Inserting the same element on each of two related queues yields related queues: if $d : \tau$ and $R(q, q')$, then $R(e_{\mathsf{i}}(\,d\,)(\,q\,), e'_{\mathsf{i}}(\,d\,)(\,q'\,))$.
3. If two queues are related, then either they are both empty, or their front elements are the same and their back elements are related: if $R(q, q')$, then either
   (a) $e_{\mathsf{r}}(\,q\,) \cong \mathtt{null} \cong e'_{\mathsf{r}}(\,q'\,)$, or
   (b) $e_{\mathsf{r}}(\,q\,) \cong \mathtt{just}(\,\langle d, r\rangle\,)$ and $e'_{\mathsf{r}}(\,q'\,) \cong \mathtt{just}(\,\langle d', r'\rangle\,)$, with $d \cong d'$ and $R(r, r')$.

If such a relation R exists, then the implementations e and e' are *bisimilar*. The terminology stems from the requirement that the operations of the abstract type preserve the relation: if it holds before an operation is performed, then it must also hold afterwards, and the relation must hold for the initial state of the queue. Thus each implementation *simulates* the other up to the relationship specified by R.

To see how this works in practice, let us consider informally two implementations of the abstract type of queues defined earlier. For the reference implementation we choose ρ to be the type `natlist`, and define the empty queue to be the empty list, define insert to add the given element to the head of the list, and define remove to remove the last element of the list. The code is as follows:

$$\begin{aligned}
t &\triangleq \mathtt{natlist}\\
\mathtt{emp} &\triangleq \mathtt{nil}\\
\mathtt{ins} &\triangleq \lambda\,(\,x\!:\!\mathtt{nat}\,)\,\lambda\,(\,q\!:\!t\,)\,\mathtt{cons}(\,x; q\,)\\
\mathtt{rem} &\triangleq \lambda\,(\,q\!:\!t\,)\,\mathtt{case}\,\mathtt{rev}(\,q\,)\,\{\mathtt{nil} \hookrightarrow \mathtt{null} \mid \mathtt{cons}(\,f; qr\,) \hookrightarrow \mathtt{just}(\,\langle f, \mathtt{rev}(\,qr\,)\rangle\,)\}.
\end{aligned}$$

Removing an element takes time linear in the length of the list, because of the reversal.

For the candidate implementation we choose ρ' to be the type $\texttt{natlist} \times \texttt{natlist}$ of pairs of lists $\langle b, f \rangle$ in which b is the "back half" of the queue, and f is the *reversal* of the "front half" of the queue. For this representation we define the empty queue to be a pair of empty lists, define insert to extend the back with that element at the head, and define remove based on whether the front is empty or not. If it is non-empty, the head element is removed from it, and returned along with the pair consisting of the back and the tail of the front. If it is empty, and the back is not, then we reverse the back, remove the head element, and return the pair consisting of the empty list and the tail of the now-reversed back. The code is as follows:

$$
\begin{aligned}
t &\triangleq \texttt{natlist} \times \texttt{natlist} \\
\texttt{emp} &\triangleq \langle \texttt{nil}, \texttt{nil} \rangle \\
\texttt{ins} &\triangleq \lambda\,(\,x:\texttt{nat}\,)\,\lambda\,(\,\langle bs, fs\rangle : t\,)\,\langle \texttt{cons}(\,x; bs\,), fs\rangle \\
\texttt{rem} &\triangleq \lambda\,(\,\langle bs, fs\rangle : t\,)\,\texttt{case}\,fs\,\{\texttt{nil} \hookrightarrow e \mid \texttt{cons}(\,f; fs'\,) \hookrightarrow \texttt{just}(\,\langle f, \langle bs, fs'\rangle\rangle\,)\}, \text{ where} \\
e &\triangleq \texttt{case}\,\texttt{rev}(\,bs\,)\,\{\texttt{nil} \hookrightarrow \texttt{null} \mid \texttt{cons}(\,b; bs'\,) \hookrightarrow \texttt{just}(\,\langle b, \langle \texttt{nil}, bs'\rangle\rangle\,)\}.
\end{aligned}
$$

The cost of the occasional reversal is amortized across the sequence of inserts and removes to show that each operation in a sequence costs unit time overall.

To show that the candidate implementation is correct, we show that it is bisimilar to the reference implementation. To do so, we specify a relation R between the types $\texttt{natlist}$ and $\texttt{natlist} \times \texttt{natlist}$ such that the two implementations satisfy the three simulation conditions given earlier. The required relation states that $R(l, \langle b, f\rangle)$ iff the list l is the list $\texttt{app}(\,b\,)(\,\texttt{rev}(\,f\,)\,)$, where app is the evident append function on lists. That is, thinking of l as the reference representation of the queue, the candidate must ensure that the elements of b followed by the elements of f in reverse order form precisely the list l. It is easy to check that the implementations just described preserve this relation. Having done so, we are assured that the client c behaves the same regardless of whether we use the reference or the candidate. Because the reference implementation is obviously correct (albeit inefficient), the candidate must also be correct in that the behavior of any client is not affected by using it instead of the reference.

17.5 Notes

The connection between abstract types in programming languages and existential types in logic was made by Mitchell and Plotkin (1988). Closely related ideas were already present in Reynolds (1974), but the connection with existential types was not explicitly drawn there. The present formulation of representation independence follows closely Mitchell (1986).

Exercises

17.1. Show that the statics and dynamics of existential types are correctly simulated using the interpretation given in Section 17.3.

17.2. Define in **FE** of the coinductive type of streams of natural numbers as defined in Chapter 15. *Hint*: Define the representation in terms of the generator (introduction form) for streams.

17.3. Define in **FE** an arbitrary coinductive type $\nu(t.\tau)$. *Hint*: generalize your answer to Exercise **17.2**.

17.4. Representation independence for abstract types is a corollary of the parametricity theorem for polymorphic types, using the interpretation of **FE** in **F** given in Section 17.3. Recast the proof of equivalence of the two implementations of queues given in Section 17.4 as an instance of parametricity as defined informally in Chapter 16.

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
			$\rightharpoonup(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	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:

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

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

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

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

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

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

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

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.

The 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 $\overline{\ulcorner e \urcorner}$ 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.

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

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 `nat` defined in 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 type `conat` defined in 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 a function of two arguments that returns $\mathtt{z}$ if either of its arguments is $\mathtt{z}$, and $\mathtt{s}(\mathtt{z})$ otherwise. That is, we wish to find an expression e satisfying the following properties:

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

Thus, e defines a total function of its two arguments, *even if* one of the arguments diverges. 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).

Chapter 20

System FPC of Recursive Types

In this chapter we study **FPC**, a language with products, sums, partial functions, and *recursive types*. Recursive types are solutions to type equations $t \cong \tau$ where there is no restriction on where t may occur in τ. Equivalently, a recursive type is a *fixed point* up to isomorphism of the associated unrestricted type operator $t.\tau$. By removing the restrictions on the type operator we may consider the solution of a type equation such as $t \cong t \rightharpoonup t$, which describes a type that is isomorphic to the type of partial functions defined on itself. If types were sets, such an equation could not be solved, because there are more partial functions on a set than there are elements of that set. But *types are not sets*: they classify *computable* functions, not *arbitrary* functions. With types we may solve such "dubious" type equations, even though we cannot expect to do so with sets. The penalty is that we must admit non-termination. For one thing, type equations involving functions have solutions only if the functions involved are partial.

A benefit of working in the setting of partial functions is that type equations have *unique* solutions (up to isomorphism). Therefore it makes sense, as we shall do in this chapter, to speak of *the* solution to a type equation. But what about the *distinct* solutions to a type equation given in Chapter 15? These turn out to coincide for any fixed dynamics, but give rise to different solutions according to whether the dynamics is eager or lazy (as illustrated in Section 19.4 for the special case of the natural numbers). Under a lazy dynamics (where all constructs are evaluated lazily), recursive types have a coinductive flavor, and the inductive analogs are inaccessible. Under an eager dynamics (where all constructs are evaluated eagerly), recursive types have an inductive flavor. But the coinductive analogs are accessible as well, using function types to selectively impose laziness. It follows that the eager dynamics is *more expressive* than the lazy dynamics, because it is impossible to go the other way around (one cannot define inductive types in a lazy language).

20.1 Solving Type Equations

The language **FPC** has products, sums, and partial functions inherited from the preceding development, extended with the new concept of recursive types. The syntax of recursive types is

defined as follows:

Typ	τ	$::=$	t	t	self-reference
			$\mathtt{rec}(\,t.\tau\,)$	$\mathtt{rec}\,t\,\mathtt{is}\,\tau$	recursive type
Exp	e	$::=$	$\mathtt{fold}\{t.\tau\}(\,e\,)$	$\mathtt{fold}(\,e\,)$	fold
			$\mathtt{unfold}(\,e\,)$	$\mathtt{unfold}(\,e\,)$	unfold

Recursive types have the same general form as the inductive and coinductive types discussed in Chapter 15, but without restriction on the type operator involved. Recursive type are formed according to the rule:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(\,t.\tau\,)\ \mathsf{type}} \tag{20.1}$$

The statics of folding and unfolding is given by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(\,t.\tau\,)/t]\tau}{\Gamma \vdash \mathtt{fold}\{t.\tau\}(\,e\,) : \mathtt{rec}(\,t.\tau\,)} \tag{20.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(\,t.\tau\,)}{\Gamma \vdash \mathtt{unfold}(\,e\,) : [\mathtt{rec}(\,t.\tau\,)/t]\tau} \tag{20.2b}$$

The dynamics of folding and unfolding is given by these rules:

$$\frac{[e\ \mathsf{val}]}{\mathtt{fold}\{t.\tau\}(\,e\,)\ \mathsf{val}} \tag{20.3a}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{fold}\{t.\tau\}(\,e\,) \longmapsto \mathtt{fold}\{t.\tau\}(\,e'\,)}\right] \tag{20.3b}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}(\,e\,) \longmapsto \mathtt{unfold}(\,e'\,)} \tag{20.3c}$$

$$\frac{\mathtt{fold}\{t.\tau\}(\,e\,)\ \mathsf{val}}{\mathtt{unfold}(\,\mathtt{fold}\{t.\tau\}(\,e\,)\,) \longmapsto e} \tag{20.3d}$$

The bracketed premise and rule are included for an *eager* interpretation of the introduction form, and omitted for a *lazy* interpretation. As mentioned in the introduction, the choice of eager or lazy dynamics affects the meaning of recursive types.

Theorem 20.1 (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 $e \longmapsto e'$.

20.2 Inductive and Coinductive Types

Recursive types may be used to represent inductive types such as the natural numbers. Using an *eager* dynamics for **FPC**, the recursive type

$$\rho = \mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t\,]$$

satisfies the type equation

$$\rho \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho\,],$$

and is isomorphic to the type of eager natural numbers. The introduction and elimination forms are defined on ρ by the following equations:[1]

$$\begin{aligned}
\mathtt{z} &\triangleq \mathtt{fold}(\,\mathtt{z} \cdot \langle\rangle\,) \\
\mathtt{s}(\,e\,) &\triangleq \mathtt{fold}(\,\mathtt{s} \cdot e\,) \\
\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\} &\triangleq \mathtt{case}\, \mathtt{unfold}(\,e\,)\, \{\mathtt{z} \cdot _ \hookrightarrow e_0 \mid \mathtt{s} \cdot x \hookrightarrow e_1\}.
\end{aligned}$$

It is a good exercise to check that the eager dynamics of natural numbers in **PCF** is correctly simulated by these definitions.

On the other hand, under a lazy dynamics for **FPC**, the same recursive type ρ',

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t\,],$$

satisfies the same type equation,

$$\rho' \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho'\,],$$

but is not the type of natural numbers! Rather, it is the type `lnat` of lazy natural numbers introduced in Section 19.4. As discussed there, the type ρ' contains the "infinite number" ω, which is of course not a natural number.

Similarly, using an eager dynamics for **FPC**, the type `natlist` of lists of natural numbers is defined by the recursive type

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{n} \hookrightarrow \mathtt{unit}, \mathtt{c} \hookrightarrow \mathtt{nat} \times t\,],$$

which satisfies the type equation

$$\mathtt{natlist} \cong [\mathtt{n} \hookrightarrow \mathtt{unit}, \mathtt{c} \hookrightarrow \mathtt{nat} \times \mathtt{natlist}\,].$$

The list introduction operations are given by the following equations:

$$\begin{aligned}
\mathtt{nil} &\triangleq \mathtt{fold}(\,\mathtt{n} \cdot \langle\rangle\,) \\
\mathtt{cons}(\,e_1; e_2\,) &\triangleq \mathtt{fold}(\,\mathtt{c} \cdot \langle e_1, e_2\rangle\,).
\end{aligned}$$

[1] The "underscore" stands for a variable that does not occur free in e_0.

A conditional list elimination form is given by the following equation:

$$\texttt{case}\, e\, \{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(\, x; y\,) \hookrightarrow e_1\} \triangleq \texttt{case}\, \texttt{unfold}(\, e\,)\, \{\texttt{n} \cdot \_ \hookrightarrow e_0 \mid \texttt{c} \cdot \langle x, y\rangle \hookrightarrow e_1\},$$

where we have used pattern-matching syntax to bind the components of a pair for the sake of clarity.

Now consider the *same* recursive type, but in the context of a lazy dynamics for **FPC**. What type is it? If all constructs are lazy, then a value of the recursive type

$$\texttt{rec}\, t\, \texttt{is}\, [\, \texttt{n} \hookrightarrow \texttt{unit}, \texttt{c} \hookrightarrow \texttt{nat} \times t\,],$$

has the form $\texttt{fold}(\, e\,)$, where e is an unevaluated computation of the sum type, whose values are injections of unevaluated computations of either the unit type or of the product type $\texttt{nat} \times t$. And the latter consists of pairs of an unevaluated computation of a (lazy!) natural number, and an unevaluated computation of another value of this type. In particular, this type contains infinite lists whose tails go on without end, as well as finite lists that eventually reach an end. The type is, in fact, a version of the type of infinite streams defined in Chapter 15, rather than a type of finite lists as is the case under an eager dynamics.

It is common in textbooks to depict data structures using "box-and-pointer" diagrams. These work well in the eager setting, provided that no functions are involved. For example, an eager list of eager natural numbers may be depicted using this notation. We may think of `fold` as an abstract pointer to a tagged cell consisting of either (a) the tag `n` with no associated data, or (b) the tag `c` attached to a pair consisting of an authentic natural number and another list, which is an abstract pointer of the same type. But this notation does not scale well to types involving functions, or to languages with a lazy dynamics. For example, the recursive type of "lists" in lazy **FPC** cannot be depicted using boxes and pointers, because of the unevaluated computations occurring in values of this type. It is a mistake to limit one's conception of data structures to those that can be drawn on the blackboard using boxes and pointers or similar informal notations. There is no substitute for a programming language to express data structures fully and accurately.

It is deceiving that the "same" recursive type can have two different meanings according to whether the underlying dynamics is eager or lazy. For example, it is common for lazy languages to use the name "list" for the recursive type of streams, or the name "nat" for the type of lazy natural numbers. This terminology is misleading, considering that such languages do not (and can not) have a proper type of finite lists or a type of natural numbers. *Caveat emptor!*

20.3 Self-Reference

In the general recursive expression $\texttt{fix}\{\tau\}(\, x.e\,)$ the variable x stands for the expression itself. Self-reference is effected by the unrolling transition

$$\texttt{fix}\{\tau\}(\, x.e\,) \longmapsto [\texttt{fix}\{\tau\}(\, x.e\,)/x]e,$$

which substitutes the expression itself for x in its body during execution. It is useful to think of x as an *implicit argument* to e that is instantiated to itself when the expression is used. In many well-known languages this implicit argument has a special name, such as `this` or `self`, to emphasize its self-referential interpretation.

Using this intuition as a guide, we may derive general recursion from recursive types. This derivation shows that general recursion may, like other language features, be seen as a manifestation of type structure, instead of as an *ad hoc* language feature. The derivation isolates a type of self-referential expressions given by the following grammar:

Typ	τ	$::=$	$\mathtt{self}(\tau)$	$\tau\ \mathtt{self}$	self-referential type
Exp	e	$::=$	$\mathtt{self}\{\tau\}(x.e)$	$\mathtt{self}\ x\ \mathtt{is}\ e$	self-referential expression
			$\mathtt{unroll}(e)$	$\mathtt{unroll}(e)$	unroll self-reference

The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \mathtt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \mathtt{self}\{\tau\}(x.e) : \mathtt{self}(\tau)} \tag{20.4a}$$

$$\frac{\Gamma \vdash e : \mathtt{self}(\tau)}{\Gamma \vdash \mathtt{unroll}(e) : \tau} \tag{20.4b}$$

The dynamics is given by the following rule for unrolling the self-reference:

$$\frac{}{\mathtt{self}\{\tau\}(x.e)\ \mathsf{val}} \tag{20.5a}$$

$$\frac{e \longmapsto e'}{\mathtt{unroll}(e) \longmapsto \mathtt{unroll}(e')} \tag{20.5b}$$

$$\frac{}{\mathtt{unroll}(\mathtt{self}\{\tau\}(x.e)) \longmapsto [\mathtt{self}\{\tau\}(x.e)/x]e} \tag{20.5c}$$

The main difference, compared to general recursion, is that we distinguish a type of self-referential expressions, instead of having self-reference at every type. However, as we shall see, the self-referential type suffices to implement general recursion, so the difference is a matter of taste.

The type $\mathtt{self}(\tau)$ is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type τ to depend on the expression itself. That is, we seek to define the type $\mathtt{self}(\tau)$ so that it satisfies the isomorphism

$$\mathtt{self}(\tau) \cong \mathtt{self}(\tau) \rightharpoonup \tau.$$

We seek a fixed point of the type operator $t.t \rightharpoonup \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\mathtt{rec}(t.t \rightharpoonup \tau),$$

which we take as the definition of $\mathtt{self}(\tau)$.

The self-referential expression $\mathtt{self}\{\tau\}(x.e)$ is the expression

$$\mathtt{fold}(\lambda(x : \mathtt{self}(\tau))\, e).$$

We may check that rule (20.4a) is derivable according to this definition. The expression $\texttt{unroll}(\,e\,)$ is correspondingly the expression

$$\texttt{unfold}(\,e\,)(\,e\,).$$

It is easy to check that rule (20.4b) is derivable from this definition. Moreover, we may check that

$$\texttt{unroll}(\,\texttt{self}\{\tau\}(\,y.e\,)\,) \longmapsto^* [\texttt{self}\{\tau\}(\,y.e\,)/y]e.$$

This completes the derivation of the type $\texttt{self}(\,\tau\,)$ of self-referential expressions of type τ.

The self-referential type $\texttt{self}(\,\tau\,)$ can be used to define general recursion for *any* type. We may define $\texttt{fix}\{\tau\}(\,x.e\,)$ to stand for the expression

$$\texttt{unroll}(\,\texttt{self}\{\tau\}(\,y.[\texttt{unroll}(\,y\,)/x]e\,)\,)$$

where the recursion at each occurrence of x is unrolled within e. It is easy to check that this verifies the statics of general recursion given in Chapter 19. Moreover, it also validates the dynamics, as shown by the following derivation:

$$\begin{aligned}\texttt{fix}\{\tau\}(\,x.e\,) &= \texttt{unroll}(\,\texttt{self}\{\tau\}(\,y.[\texttt{unroll}(\,y\,)/x]e\,)\,)\\ &\longmapsto^* [\texttt{unroll}(\,\texttt{self}\{\tau\}(\,y.[\texttt{unroll}(\,y\,)/x]e\,)\,)/x]e\\ &= [\texttt{fix}\{\tau\}(\,x.e\,)/x]e.\end{aligned}$$

It follows that recursive types can be used to define a non-terminating expression of every type, $\texttt{fix}\{\tau\}(\,x.x\,)$.

20.4 The Origin of State

The concept of *state* in a computation—which will be discussed in Part XIV—has its origins in the concept of recursion, or self-reference, which, as we have just seen, arises from the concept of recursive types. For example, the concept of a *flip-flop* or a *latch* is a circuit built from combinational logic elements (typically, nor or nand gates) that have the characteristic that they maintain an alterable state over time. An RS latch, for example, maintains its output at the logical level of zero or one in response to a signal on the R or S inputs, respectively, after a brief settling delay. This behavior is achieved using *feedback*, which is just a form of self-reference, or recursion: the output of the gate feeds back into its input so as to convey the current state of the gate to the logic that determines its next state.

We can implement an RS latch using recursive types. The idea is to use self-reference to model the passage of time, with the current output being computed from its input and its previous outputs. Specifically, an RS latch is a value of type τ_{rsl} given by

$$\texttt{rec}\,t\,\texttt{is}\,\langle \texttt{X} \hookrightarrow \texttt{bool}, \texttt{Q} \hookrightarrow \texttt{bool}, \texttt{N} \hookrightarrow t\rangle.$$

The `X` and `Q` components of the latch represent its current outputs (of which `Q` represents the current state of the latch), and the `N` component represents the next state of the latch. If e is of type τ_{rsl},

then we define e @ X to mean $\texttt{unfold}(\,e\,)\cdot\texttt{X}$, and define e @ Q and e @ N similarly. The expressions e @ X and e @ Q evaluate to the boolean outputs of the latch e, and e @ N evaluates to another latch representing its evolution over time based on these inputs.

For given values r and s, a new latch is computed from an old latch by the recursive function *rsl* defined as follows:[2]

$$\texttt{fix}\,\mathit{rsl}\,\texttt{is}\,\lambda\,(\,l:\tau_{rsl}\,)\,e_{rsl},$$

where e_{rsl} is the expression

$$\texttt{fix}\,\mathit{this}\,\texttt{is}\,\texttt{fold}(\,\langle\texttt{X}\hookrightarrow e_{nor}(\,\langle s, l\,@\,\texttt{Q}\rangle\,),\texttt{Q}\hookrightarrow e_{nor}(\,\langle r, l\,@\,\texttt{X}\rangle\,),\texttt{N}\hookrightarrow \mathit{rsl}(\,\mathit{this}\,)\rangle\,),$$

where e_{nor} is the obvious binary function on booleans. The outputs of the latch are computed in terms of the r and s inputs and the outputs of the previous state of the latch. To get the construction started, we define an initial state of the latch in which the outputs are arbitrarily set to `false`, and whose next state is determined by applying the recursive function *rsl* to that state:

$$\texttt{fix}\,\mathit{this}\,\texttt{is}\,\texttt{fold}(\,\langle\texttt{X}\hookrightarrow\texttt{false},\texttt{Q}\hookrightarrow\texttt{false},\texttt{N}\hookrightarrow \mathit{rsl}(\,\mathit{this}\,)\rangle\,).$$

Selection of the N component causes the outputs to be recalculated based on their current values. Notice the role of self-reference in maintaining the state of the latch.

20.5 Notes

The systematic study of recursive types in programming was initiated by Scott (1976, 1982) to give a mathematical model of the untyped λ-calculus. The derivation of recursion from recursive types is an application of Scott's theory. The category-theoretic view of recursive types was developed by Wand (1979) and Smyth and Plotkin (1982). Implementing state using self-reference is fundamental to digital logic (Ward and Halstead, 1990). The example given in Section 20.4 is inspired by Cook (2009) and Abadi and Cardelli (1996). The account of signals as streams (explored in the exercises) is inspired by the pioneering work of Kahn (MacQueen, 2009). The language name **FPC** is taken from Gunter (1992).

Exercises

20.1. Show that the recursive type $\texttt{D}\triangleq\texttt{rec}\,t\,\texttt{is}\,t\rightharpoonup t$ is non-trivial by interpreting the sk-combinators defined in Exercise 3.1 into it. Specifically, define elements `k : D` and `s : D` and a (left-associative) "application" function

$$x:\texttt{D}\,y:\texttt{D}\vdash x\cdot y:\texttt{D}$$

such that

[2] For convenience we assume that `fold` is evaluated lazily.

(a) $\mathtt{k} \cdot x \cdot y \longmapsto^* x$;

(b) $\mathtt{s} \cdot x \cdot y \cdot z \longmapsto^* (x \cdot z) \cdot (y \cdot z)$.

20.2. Recursive types admit the structure of both inductive and coinductive types. Consider the recursive type $\tau \triangleq \mathtt{rec}\, t\, \mathtt{is}\, \tau'$ and the associated inductive and coinductive types $\mu(t.\tau')$ and $\nu(t.\tau')$. Complete the following chart consistently with the statics of inductive and coinductive types on the left-hand side and with the statics of recursive types on the right:

$$\begin{aligned}
\mathtt{fold}\{t.t\ \mathtt{opt}\}(e) &\triangleq \mathtt{fold}(e) \\
\mathtt{rec}\{t.t\ \mathtt{opt}\}(x.e'; e) &\triangleq \mathtt{?} \\
\mathtt{unfold}\{t.t\ \mathtt{opt}\}(e) &\triangleq \mathtt{unfold}(e) \\
\mathtt{gen}\{t.t\ \mathtt{opt}\}(x.e'; e) &\triangleq \mathtt{?}
\end{aligned}$$

Check that the statics is derivable under these definitions. *Hint*: you will need to use general recursion on the right to fill in the missing cases. You may also find it useful to use generic programming.

Now consider the dynamics of these definitions, under both an eager and a lazy interpretation. What happens in each case?

20.3. Define the type `signal` of *signals* to be the coinductive type of infinite streams of booleans (bits). Define a *signal transducer* to be a function of type $\mathtt{signal} \rightharpoonup \mathtt{signal}$. Combinational logic gates, such as the NOR gate, can be defined as signal transducers. Give a coinductive definition of the type `signal`, and define NOR as a signal transducer. Be sure to take account of the underlying dynamics of **PCF**.

The passage from combinational to digital logic (circuit elements that maintain state) relies on self-reference. For example, an RS latch can be built from NOR two nor gates in this way. Define an RS latch using general recursion and two of the NOR gates just defined.

20.4. The type τ_{rsl} given in Section 20.4 above is the type of streams of pairs of booleans. Give another formulation of an RS latch as a value of type τ_{rsl}, but this time using the coinductive interpretation of the recursive type proposed in Exercise **20.2** (using the lazy dynamics for **FPC**). Expand and simplify this definition using your solution to Exercise **20.2**, and compare it with the formulation given in Section 20.4. *Hint*: the internal state of the stream is a pair of booleans corresponding to the `X` and `Q` outputs of the latch.

Part IX

Dynamic Types

Chapter 21

The Untyped λ-Calculus

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) λ-calculus*. This formalism was introduced by Church in the 1930's as a universal language of computable functions. It is distinctive for its austere elegance. The λ-calculus has but one "feature", the higher-order function. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a turn of phrase, in the λ-calculus it's functions all the way down.

21.1 The λ-Calculus

The abstract syntax of the untyped λ-calculus, called $\mathbf{\Lambda}$, is given by the following grammar:

Exp	u	$::=$	x	x	variable
			$\lambda(\,x.u\,)$	$\lambda\,(\,x\,)\,u$	λ-abstraction
			$\mathsf{ap}(\,u_1; u_2\,)$	$u_1(\,u_2\,)$	application

The statics of $\mathbf{\Lambda}$ is defined by general hypothetical judgments of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the variables when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{21.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash u_1(\,u_2\,)\ \mathsf{ok}} \tag{21.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda\,(\,x\,)\,u\ \mathsf{ok}} \tag{21.1c}$$

The dynamics of $\mathbf{\Lambda}$ is given equationally, rather than via a transition system. Definitional equality for $\mathbf{\Lambda}$ is a judgment of the form $\Gamma \vdash u \equiv u'$, where $\Gamma \vdash u \ \mathsf{ok}$ and $\Gamma \vdash u' \ \mathsf{ok}$. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{21.2a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{21.2b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{21.2c}$$

$$\frac{\Gamma \vdash u_1 \equiv u_1' \quad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash u_1(\,u_2\,) \equiv u_1'(\,u_2'\,)} \tag{21.2d}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda\,(\,x\,)\,u \equiv \lambda\,(\,x\,)\,u'} \tag{21.2e}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u_2 \ \mathsf{ok} \quad \Gamma \vdash u_1 \ \mathsf{ok}}{\Gamma \vdash (\,\lambda\,(\,x\,)\,u_2\,)(\,u_1\,) \equiv [u_1/x]u_2} \tag{21.2f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

21.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressiveness. It is a *Turing-complete* language in the sense that it has the same capability to express computations on the natural numbers as does any other known programming language. Church's Law states that any conceivable notion of computable function on the natural numbers is equivalent to the λ-calculus. This assertion is true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which predicts the outcome of all future measurements of the acceleration in a gravitational field.[1]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 19. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. In particular, we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

[1] It is debatable whether there are any scientific laws in Computer Science. In the opinion of the author, Church's Law, which is usually called *Church's Thesis*, is a strong candidate for being a scientific law.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} \triangleq \lambda(b)\lambda(s)\,b \tag{21.3a}$$
$$\overline{n+1} \triangleq \lambda(b)\lambda(s)\,s(\overline{n}(b)(s)) \tag{21.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

the n-fold application of u_2 to u_1. That is, $\overline{n}$ iterates its second argument (the induction step) n times, starting with its first argument (the basis).

Using this definition it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped λ-terms:

$$\mathtt{succ} \triangleq \lambda(x)\lambda(b)\lambda(s)\,s(x(b)(s)) \tag{21.4}$$
$$\mathtt{plus} \triangleq \lambda(x)\lambda(y)\,y(x)(\mathtt{succ}) \tag{21.5}$$
$$\mathtt{times} \triangleq \lambda(x)\lambda(y)\,y(\overline{0})(\mathtt{plus}(x)) \tag{21.6}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

To define $\mathtt{ifz}\{u_0; x.u_1\}(u)$ requires a bit of ingenuity. The key is to define the "cut-off predecessor", `pred`, such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{21.7}$$
$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{21.8}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ in terms of its value for $\overline{n}$. At first glance this seems simple—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This formulation invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the nth iteration the registers contain the predecessor of n and n itself, respectively. Given the result for n, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of n we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2\rangle \triangleq \lambda(f)\,f(u_1)(u_2) \tag{21.9}$$
$$u \cdot \mathtt{l} \triangleq u(\lambda(x)\lambda(y)\,x) \tag{21.10}$$
$$u \cdot \mathtt{r} \triangleq u(\lambda(x)\lambda(y)\,y) \tag{21.11}$$

It is easy to check that under this encoding $\langle u_1, u_2 \rangle \cdot \mathtt{l} \equiv u_1$, and that a similar equivalence holds for the second projection. We may now define the required representation, u_p, of the predecessor function:

$$u'_p \triangleq \lambda\,(\,x\,)\,x(\,\langle \overline{0}, \overline{0} \rangle\,)(\,\lambda\,(\,y\,)\,\langle y \cdot \mathtt{r}, \mathtt{succ}\,(\,y \cdot \mathtt{r}\,)\rangle\,) \tag{21.12}$$

$$u_p \triangleq \lambda\,(\,x\,)\,u'_p(\,x\,) \cdot \mathtt{l} \tag{21.13}$$

It is easy to check that this gives us the required behavior. Finally, define $\mathtt{ifz}\{u_0; x.u_1\}(\,u\,)$ to be the untyped term

$$u(\,u_0\,)(\,\lambda\,(\,\_\,)\,[u_p(\,u\,)/x]u_1\,).$$

This definition gives us all the apparatus of PCF, apart from general recursion. But general recursion is also definable in $\mathbf{\Lambda}$ using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the $\mathtt{Y}$ *combinator*:

$$\mathtt{Y} \triangleq \lambda\,(\,F\,)\,(\,\lambda\,(\,f\,)\,F(\,f(\,f\,)\,)\,)(\,\lambda\,(\,f\,)\,F(\,f(\,f\,)\,)\,).$$

It is easy to check that

$$\mathtt{Y}(\,F\,) \equiv F(\,\mathtt{Y}(\,F\,)\,).$$

Using the $\mathtt{Y}$ combinator, we may define general recursion by writing $\mathtt{Y}(\,\lambda\,(\,x\,)\,u\,)$, where x stands for the recursive expression itself.

Although it is clear that $\mathtt{Y}$ as just defined computes a fixed point of its argument, it is probably less clear why it works or how we might have invented it in the first place. The main idea is quite simple. If a function is recursive, it is given an extra first argument, which is arranged at call sites to be the function itself. Whenever we wish to call a self-referential function with an argument, we apply the function first to itself and then to its argument; this protocol is imposed on both the "external" calls to the function and on the "internal" calls that the function may make to itself. For this reason the first argument is often called `this` or `self`, to remind you that it will be, by convention, bound to the function itself.

With this in mind, it is easy to see how to derive the definition of $\mathtt{Y}$. If F is the function whose fixed point we seek, then the function $F' = \lambda\,(\,f\,)\,F(\,f(\,f\,)\,)$ is a variant of F in which the self-application convention has been imposed internally by substituting for each occurrence of f in $F(\,f\,)$ the self-application $f(\,f\,)$. Now check that $F'(\,F'\,) \equiv F(\,F'(\,F'\,)\,)$, so that $F'(\,F'\,)$ is the desired fixed point of F. Expanding the definition of F', we have derived that the desired fixed point of F is

$$\lambda\,(\,f\,)\,F(\,f(\,f\,)\,)(\,\lambda\,(\,f\,)\,F(\,f(\,f\,)\,)\,).$$

To finish the derivation, we need only note that nothing depends on the particular choice of F, which means that we can compute a fixed point for F uniformly in F. That is, we may define a *single* function, the term $\mathtt{Y}$ as defined above, that computes the fixed point of *any* F.

21.3 Scott's Theorem

Scott's Theorem states that definitional equality for the untyped λ-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equal. The proof

uses the concept of *inseparability*. Any two properties, $\mathcal{A}_0$ and $\mathcal{A}_1$, of λ-terms are *inseparable* if there is no decidable property, $\mathcal{B}$, such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ holds, and $\mathcal{A}_1\ u$ implies that $\mathcal{B}\ u$ does *not* hold. We say that a property, $\mathcal{A}$, of untyped terms is *behavioral* iff whenever $u \equiv u'$, then $\mathcal{A}\ u$ iff $\mathcal{A}\ u'$.

The proof of Scott's Theorem decomposes into two parts:

1. For any untyped λ-term u, we may find an untyped term v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of v, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 9 for a discussion of Gödel-numbering.)

2. Any two non-trivial[2] behavioral properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms are inseparable.

Lemma 21.1. *For any u there exists v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$.*

Proof Sketch. The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. $\mathtt{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.

2. $\mathtt{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms, and builds the representation of the application of one to the other. The second takes a numeral for n, and yields the representation of the Church numeral $\overline{n}$. Given these, we may find the required term v by defining $v \triangleq w(\overline{\ulcorner w \urcorner})$, where $w \triangleq \lambda(x)\, u(\mathtt{ap}(x)(\mathtt{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathtt{ap}(\overline{\ulcorner w \urcorner})(\mathtt{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathtt{Y}(u)$, except that u takes as input the representation of a term, and we find a v such that, when applied to the representation of v, the term u yields v itself. □

Lemma 21.2. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-trivial behavioral properties of untyped terms. Then there is no untyped term w such that*

1. *For every u either $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

2. *If $\mathcal{A}_0\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*

3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

[2] A property of untyped terms is *trivial* if it either holds for all untyped terms or never holds for any untyped term.

Proof. Suppose there is such an untyped term w. Let v be the untyped term

$$\lambda\,(\,x\,)\,\mathtt{ifz}\{u_1;\_.u_0\}(\,w(\,x\,)\,),$$

where u_0 and u_1 are chosen such that $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. (Such a choice must exist by non-triviality of the properties.) By Lemma 21.1 there is an untyped term t such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, because $\mathcal{A}_1$ is behavioral and $\mathcal{A}_1\ u_1$. But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of w, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

Corollary 21.3. *There is no algorithm to decide whether $u \equiv u'$.*

Proof. For fixed u, the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$ is a non-trivial behavioral property of untyped terms. So it is inseparable from its negation, and hence is undecidable. □

21.4 Untyped Means Uni-Typed

The untyped λ-calculus can be faithfully embedded in a typed language with recursive types. Thus every untyped λ-term has a representation as a typed expression in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the λ-calculus in **FPC**, but rather a direct representation of untyped λ-terms as typed expressions in a language with recursive types.

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-calculus. It is not the *absence* of types that gives it its power, but rather that it has *only one* type, the recursive type

$$D \triangleq \mathtt{rec}\,t\,\mathtt{is}\,t \rightharpoonup t.$$

A value of type D is of the form $\mathtt{fold}(\,e\,)$ where e is a value of type $D \rightharpoonup D$ — a function whose domain and range are both D. Any such function can be regarded as a value of type D by "folding", and any value of type D can be turned into a function by "unfolding". As usual, a recursive type is a solution to a type equation, which in the present case is the equation

$$D \cong D \rightharpoonup D.$$

This isomorphism specifies that D is a type that is isomorphic to the space of partial functions on D itself, which is impossible if types are just sets.

This isomorphism leads to the following translation, of **Λ** into **FPC**:

$$x^\dagger \triangleq x \tag{21.14a}$$

$$\lambda\,(\,x\,)\,u^\dagger \triangleq \mathtt{fold}(\,\lambda\,(\,x : D\,)\,u^\dagger\,) \tag{21.14b}$$

$$u_1(\,u_2\,)^\dagger \triangleq \mathtt{unfold}(\,u_1^\dagger\,)(\,u_2^\dagger\,) \tag{21.14c}$$

Note that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unfolding the recursive type. And so we have

$$\begin{aligned}\lambda\,(\,x\,)\,u_1(\,u_2\,)^\dagger &= \mathtt{unfold}(\,\mathtt{fold}(\,\lambda\,(\,x:D\,)\,u_1^\dagger\,)\,)(\,u_2^\dagger\,)\\ &\equiv \lambda\,(\,x:D\,)\,u_1^\dagger(\,u_2^\dagger\,)\\ &\equiv [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is proved by induction on the structure of u_1. Thus β-reduction is implemented by evaluation of the embedded terms.

Thus we see that the canonical *untyped* language, $\mathbf{\Lambda}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all. Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the cost of incurring dynamic overhead to coerce values to and from the recursive type. In Chapter 22 we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This generalization shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus this traditional distinction cannot be considered an opposition, because dynamic languages are but particular forms of static languages in which undue emphasis is placed on a single recursive type.

21.5 Notes

The untyped λ-calculus was introduced by Church (1941) as a formalization of the informal concept of a computable function. Unlike the well-known machine models, such as the Turing machine or the random access machine, the λ-calculus codifies mathematical and programming practice. Barendregt (1984) is the definitive reference for all aspects of the untyped λ-calculus; the proof of Scott's theorem is adapted from Barendregt's account. Scott (1980a) gave the first model of the untyped λ-calculus in terms of an elegant theory of recursive types. This construction underlies Scott's apt description of the λ-calculus as "uni-typed", rather than "untyped." The idea to characterize Church's Law as such was communicated to the author, independently of each other, by Robert L. Constable and Mark Lillibridge.

Exercises

21.1. Define an encoding of finite products as defined in Chapter 10 in $\mathbf{\Lambda}$.

21.2. Define the factorial function in $\mathbf{\Lambda}$ two ways, one without using `Y`, and one using `Y`. In both cases show that your solution, u, has the property that $u(\,\overline{n}\,) \equiv \overline{n!}$.

21.3. Define the "Church booleans" in $\mathbf{\Lambda}$ by defining terms `true` and `false` such that

(a) $\texttt{true}(\,u_1\,)(\,u_2\,) \equiv u_1$.

(b) $\texttt{false}(\,u_1\,)(\,u_2\,) \equiv u_2$.

What is the encoding of $\texttt{if}\ u\ \texttt{then}\ u_1\ \texttt{else}\ u_2$?

21.4. Define an encoding of finite sums as defined in Chapter 11 in Λ.

21.5. Define an encoding of finite lists of natural numbers as defined in Chapter 15 in Λ.

21.6. Define an encoding of the infinite streams of natural numbers as defined in Chapter 15 in Λ.

21.7. Show that Λ can be "compiled" to sk-combinators using bracket abstraction (see Exercises 3.4 and 3.5. Define a translation u^* from Λ into sk combinators such that

$$\text{if } u_1 \equiv u_2, \text{ then } u_1^* \equiv u_2^*.$$

Hint: Define u^* by induction on the structure of u, using the compositional form of bracket abstraction considered in Exercise 3.5. Show that the translation is itself compositional in that it commutes with substitution:

$$([u_2/x]u_1)^* = [u_2^*/x]u^*.$$

Then proceed by rule induction on rules (21.2) to show the required correctness condition.

Chapter 22

Dynamic Typing

We saw in Chapter 21 that an untyped language is a uni-typed language in which "untyped" terms are just terms of single recursive type. Because all expressions of $\mathbf{\Lambda}$ are well-typed, type safety ensures that no misinterpretation of a value is possible. When spelled out for $\mathbf{\Lambda}$, type safety follows from there being exactly one class of values, that of functions on values. No application can get stuck, because every value is a function that may be applied to an argument.

This safety property breaks down once more than one class of value is admitted. For example, if the natural numbers are added as a primitive to $\mathbf{\Lambda}$, then it is possible to incur a run-time error by attempting to apply a number to an argument. One way to manage this is to embrace the possibility, treating class mismatches as checked errors, and weakening the progress theorem as outlined in Chapter 6. Such languages are called *dynamic languages* because an error such as the one described is postponed to run time, rather than precluded at compile time by type checking. Languages of the latter sort are called *static languages*.

Dynamic languages are often considered in opposition to static languages, but the opposition is illusory. Just as the untyped λ-calculus is uni-typed, so dynamic languages are but special cases of static languages in which there is only one recursive type (albeit with multiple classes of value).

22.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of **PCF**, called **DPCF**. The abstract syntax of **DPCF** is given by the following grammar:

Exp	d	::=	x	x	variable
			$\mathtt{num}[\,n\,]$	$\overline{n}$	numeral
			$\mathtt{zero}$	$\mathtt{zero}$	zero
			$\mathtt{succ}(\,d\,)$	$\mathtt{succ}(\,d\,)$	successor
			$\mathtt{ifz}\{d_0; x.d_1\}(\,d\,)$	$\mathtt{ifz}\,d\,\{\mathtt{zero} \hookrightarrow d_0 \mid \mathtt{succ}(\,x\,) \hookrightarrow d_1\}$	zero test
			$\mathtt{fun}(\,x.d\,)$	$\lambda\,(\,x\,)\,d$	abstraction
			$\mathtt{ap}(\,d_1; d_2\,)$	$d_1(\,d_2\,)$	application
			$\mathtt{fix}(\,x.d\,)$	$\mathtt{fix}\,x\,\mathtt{is}\,d$	recursion

There are two classes of values in **DPCF**, the *numbers*, which have the form $\mathtt{num}[\,n\,]$, and the *functions*, which have the form $\mathtt{fun}(\,x.d\,)$. The expressions $\mathtt{zero}$ and $\mathtt{succ}(\,d\,)$ are not themselves values, but rather are *constructors* that evaluate to values. General recursion is definable using a fixed point combinator, but is taken as primitive here to simplify the analysis of the dynamics in Section 22.3.

As usual, the abstract syntax of **DPCF** is what matters, but we use the concrete syntax to improve readability. However, notational conveniences can obscure important details, such as the tagging of values with their class and the checking of these tags at run-time. For example, the concrete syntax for a number, $\overline{n}$, suggests a "bare" representation, the abstract syntax reveals that the number is labeled with the class $\mathtt{num}$ to distinguish it from a function. Correspondingly, the concrete syntax for a function is $\lambda\,(\,x\,)\,d$, but its abstract syntax, $\mathtt{fun}(\,x.d\,)$, shows that it also sports a class label. The class labels are required to ensure safety by run-time checking, and must not be overlooked when comparing static with dynamic languages.

The statics of **DPCF** is like that of **Λ**; it merely checks that there are no free variables in the expression. The judgment

$$x_1\ \mathsf{ok}, \ldots x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

states that d is a well-formed expression with free variables among those in the hypotheses. If the assumptions are empty, then we write just $d\ \mathsf{ok}$ to mean that d is a closed expression of **DPCF**.

The dynamics of **DPCF** must check for errors that would never arise in a language such as **PCF**. For example, evaluation of a function application must ensure that the value being applied is indeed a function, signaling an error if it is not. Similarly the conditional branch must ensure that its principal argument is a number, signaling an error if it is not. To account for these possibilities,

the dynamics is given by several judgment forms, as summarized in the following chart:

d val	d is a (closed) value
$d \longmapsto d'$	d evaluates in one step to d'
d err	d incurs a run-time error
d is_num n	d is of class `num` with value n
d isnt_num	d is not of class `num`
d is_fun $x.d$	d is of class `fun` with body $x.d$
d isnt_fun	d is not of class `fun`

The last four judgment forms implement dynamic class checking. They are only relevant when d is already a value. The affirmative class-checking judgments have a second argument that represents the underlying structure of a value; this argument is *not* itself an expression of **DPCF**.

The value judgment d val states that d is a evaluated (closed) expression:

$$\frac{}{\texttt{num}[\,n\,]\ \mathsf{val}} \tag{22.1a}$$

$$\frac{}{\texttt{fun}(\,x.d\,)\ \mathsf{val}} \tag{22.1b}$$

The affirmative class-checking judgments are defined by the following rules:

$$\frac{}{\texttt{num}[\,n\,]\ \mathsf{is\_num}\ n} \tag{22.2a}$$

$$\frac{}{\texttt{fun}(\,x.d\,)\ \mathsf{is\_fun}\ x.d} \tag{22.2b}$$

The negative class-checking judgments are correspondingly defined by these rules:

$$\frac{}{\texttt{num}[\,n\,]\ \mathsf{isnt\_fun}} \tag{22.3a}$$

$$\frac{}{\texttt{fun}(\,x.d\,)\ \mathsf{isnt\_num}} \tag{22.3b}$$

The transition judgment $d \longmapsto d'$ and the error judgment d err are defined simultaneously by the following rules:

$$\frac{}{\texttt{zero} \longmapsto \texttt{num}[\,\texttt{z}\,]} \tag{22.4a}$$

$$\frac{d \longmapsto d'}{\texttt{succ}(\,d\,) \longmapsto \texttt{succ}(\,d'\,)} \tag{22.4b}$$

$$\frac{d\ \mathsf{err}}{\texttt{succ}(\,d\,)\ \mathsf{err}} \tag{22.4c}$$

$$\frac{d\ \mathsf{is\_num}\ n}{\texttt{succ}(\,d\,) \longmapsto \texttt{num}[\,\texttt{s}(\,n\,)\,]} \tag{22.4d}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{succ}(\,d\,)\ \mathsf{err}} \tag{22.4e}$$

$$\frac{d \longmapsto d'}{\mathtt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto \mathtt{ifz}\{d_0; x.d_1\}(\,d'\,)} \tag{22.4f}$$

$$\frac{d\ \mathsf{err}}{\mathtt{ifz}\{d_0; x.d_1\}(\,d\,)\ \mathsf{err}} \tag{22.4g}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{z}}{\mathtt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto d_0} \tag{22.4h}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{s}(\,n\,)}{\mathtt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto [\mathtt{num}[\,n\,]/x]d_1} \tag{22.4i}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{ifz}\{d_0; x.d_1\}(\,d\,)\ \mathsf{err}} \tag{22.4j}$$

$$\frac{d_1 \longmapsto d_1'}{\mathtt{ap}(\,d_1; d_2\,) \longmapsto \mathtt{ap}(\,d_1'; d_2\,)} \tag{22.4k}$$

$$\left[\frac{d_1\ \mathsf{val} \quad d_2 \longmapsto d_2'}{\mathtt{ap}(\,d_1; d_2\,) \longmapsto \mathtt{ap}(\,d_1; d_2'\,)}\right] \tag{22.4l}$$

$$\frac{d_1\ \mathsf{err}}{\mathtt{ap}(\,d_1; d_2\,)\ \mathsf{err}} \tag{22.4m}$$

$$\frac{d_1\ \mathsf{is\_fun}\ x.d \quad [d_2\ \mathsf{val}]}{\mathtt{ap}(\,d_1; d_2\,) \longmapsto [d_2/x]d} \tag{22.4n}$$

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{ap}(\,d_1; d_2\,)\ \mathsf{err}} \tag{22.4o}$$

$$\frac{}{\mathtt{fix}(\,x.d\,) \longmapsto [\mathtt{fix}(\,x.d\,)/x]d} \tag{22.4p}$$

Rule (22.4i) labels the predecessor with the class num to maintain the invariant that variables are bound to expressions of **DPCF**.

Lemma 22.1 (Class Checking)**.** *If d val, then*

1. *either d is_num n for some n, or d isnt_num;*
2. *either d is_fun $x.d'$ for some x and d', or d isnt_fun.*

Proof. By inspection of the rules defining the class-checking judgments. □

Theorem 22.2 (Progress)**.** *If d ok, then either d val, or d err, or there exists d' such that $d \longmapsto d'$.*

Proof. By induction on the structure of d. For example, if $d = \mathtt{succ}(\,d'\,)$, then we have by induction either d' val, or d' err, or $d' \longmapsto d''$ for some d''. In the last case we have by rule (22.4b) that $\mathtt{succ}(\,d'\,) \longmapsto \mathtt{succ}(\,d''\,)$, and in the second-to-last case we have by rule (22.4c) that $\mathtt{succ}(\,d'\,)$ err. If d' val, then by Lemma 22.1, either d' is_num n or d' isnt_num. In the former case $\mathtt{succ}(\,d'\,) \longmapsto \mathtt{num}[\,\mathtt{s}(\,n\,)\,]$, and in the latter $\mathtt{succ}(\,d'\,)$ err. The other cases are handled similarly. □

Lemma 22.3 (Exclusivity)**.** *For any d in* **DPCF**, *exactly one of the following holds: d val, or d err, or $d \longmapsto d'$ for some d'.*

Proof. By induction on the structure of d, making reference to rules (22.4). □

22.2 Variations and Extensions

The dynamic language **DPCF** defined in Section 22.1 parallels the static language **PCF** defined in Chapter 19. One discrepancy, however, is in the treatment of natural numbers. Whereas in **PCF** the zero and successor operations are introduction forms for the type `nat`, in **DPCF** they are elimination forms that act on specially defined numerals. The present formulation uses only a single class of numbers.

One could instead treat `zero` and `succ`$(\,d\,)$ as values of separate classes, and introduce the obvious class checking judgments for them. When written in this style, the dynamics of the conditional branch is given as follows:

$$\frac{d \longmapsto d'}{\texttt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto \texttt{ifz}\{d_0; x.d_1\}(\,d'\,)} \tag{22.5a}$$

$$\frac{d \text{ is\_zero}}{\texttt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto d_0} \tag{22.5b}$$

$$\frac{d \text{ is\_succ } d'}{\texttt{ifz}\{d_0; x.d_1\}(\,d\,) \longmapsto [d'/x]d_1} \tag{22.5c}$$

$$\frac{d \text{ isnt\_zero} \quad d \text{ isnt\_succ}}{\texttt{ifz}\{d_0; x.d_1\}(\,d\,) \text{ err}} \tag{22.5d}$$

Notice that the predecessor of a value of the successor class need not be a number, whereas in the previous formulation this possibility does not arise.

DPCF can be extended with structured data similarly. A classic example is to consider a class `nil`, consisting of a "null" value, and a class `cons`, consisting of pairs of values.

Exp	d	::=	`nil`	null
			`cons`$(\,d_1; d_2\,)$	pair
			`ifnil`$(\,d; d_0; x, y.d_1\,)$	conditional

The expression `ifnil`$(\,d; d_0; x, y.d_1\,)$ distinguishes the null value from a pair, and signals an error on any other class of value.

Lists (finite sequences) can be encoded using null and pairing. For example, the list consisting of three zeros can be represented by the value

$$\texttt{cons}(\,\texttt{zero}; \texttt{cons}(\,\texttt{zero}; \texttt{cons}(\,\texttt{zero}; \texttt{nil}\,)\,)\,).$$

But what to make of the following value?

$$\texttt{cons}(\,\texttt{zero}; \texttt{cons}(\,\texttt{zero}; \texttt{cons}(\,\texttt{zero}; \lambda\,(\,x\,)\,x\,)\,)\,)$$

It is not a list, because it does not end with `nil`, but it is a permissible value in the enriched language.

A difficulty with encoding lists using null and pair emerges when defining functions on them. For example, here is a definition of the function `append` that concatenates two lists:

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(\,x\,)\,\lambda\,(\,y\,)\,\texttt{ifnil}(\,x; y; x_1, x_2.\texttt{cons}(\,x_1; a(\,x_2\,)(\,y\,)\,)\,)$$

Nothing prevents us from applying this function to any two values, regardless of whether they are lists. If the first argument is not a list, then execution aborts with an error. But because the function does not traverse its second argument, it can be any value at all. For example, we may apply `append` with a list and a function to obtain the "list" that ends with a λ given above.

It might be argued that the conditional branch that distinguishes null from a pair is inappropriate in **DPCF**, because there are more than just these two classes in the language. One approach that avoids this criticism is to abandon pattern matching on the class of data, replacing it by a general conditional branch that distinguishes null from all other values, and adding to the language *predicates*[1] that test the class of a value and *destructors* that invert the constructors of each class.

We could instead reformulate null and and pairing as follows:

Exp	d	::=	$\texttt{cond}(\,d; d_0; d_1\,)$	$\texttt{cond}(\,d; d_0; d_1\,)$	conditional
			$\texttt{nil?}(\,d\,)$	$\texttt{nil?}(\,d\,)$	nil test
			$\texttt{cons?}(\,d\,)$	$\texttt{cons?}(\,d\,)$	pair test
			$\texttt{car}(\,d\,)$	$\texttt{car}(\,d\,)$	first projection
			$\texttt{cdr}(\,d\,)$	$\texttt{cdr}(\,d\,)$	second projection

The conditional $\texttt{cond}(\,d; d_0; d_1\,)$ distinguishes d between `nil` and *all other values*. If d is not `nil`, the conditional evaluates to d_0, and otherwise evaluates to d_1. In other words the value `nil` represents boolean falsehood, and all other values represent boolean truth. The predicates $\texttt{nil?}(\,d\,)$ and $\texttt{cons?}(\,d\,)$ test the class of their argument, yielding `nil` if the argument is not of the specified class, and yielding some non-`nil` if so. The destructors $\texttt{car}(\,d\,)$ and $\texttt{cdr}(\,d\,)$ decompose $\texttt{cons}(\,d_1; d_2\,)$ into d_1 and d_2, respectively.[2]

Written in this form, the function `append` is given by the expression

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(\,x\,)\,\lambda\,(\,y\,)\,\texttt{cond}(\,x; \texttt{cons}(\,\texttt{car}(\,x\,); a(\,\texttt{cdr}(\,x\,)\,)(\,y\,)\,); y\,).$$

The behavior of this formulation of `append` is no different from the earlier one; the only difference is that instead of dispatching on whether a value is either null or a pair, we instead allow discrimination on any predicate of the value, which includes such checks as special cases.

An alternative, which is not widely used, is to enhance, and not restrict, the conditional branch so that it includes cases for each possible class of value in the language. So in a language with numbers, functions, null, and pairing, the conditional would have four branches. The fourth branch, for pairing, would deconstruct the pair into its constituent parts. The difficulty with this approach is that in realistic languages there are many classes of data, and such a conditional would be rather unwieldy. Moreover, even once we have dispatched on the class of a value, it is nevertheless necessary for the primitive operations associated with that class to admit run-time checks. For example,

[1] Predicates evaluate to the null value to mean that a condition is false, and some non-null value to mean that it is true.

[2] The terminology for the projections is archaic, but well-established. It is said that `car` originally stood for "contents of the address register" and that `cdr` stood for "contents of the data register", referring to the details of the original implementation of Lisp.

we may determine that a value d is of the numeric class, but there is no way to propagate this information into the branch of the conditional that then adds d to some other number. The addition operation must still check the class of d, recover the underlying number, and create a new value of numeric class. It is an inherent limitation of dynamic languages that they do not allow values other than classified values.

22.3 Critique of Dynamic Typing

The safety theorem for **DPCF** is an advantage of dynamic over static typing. Unlike static languages, which rule out some candidate programs as ill-typed, every piece of abstract syntax in **DPCF** is well-formed, and hence, by Theorem 22.2, has a well-defined dynamics (albeit one with checked errors). But this convenience is also a disadvantage, because errors that could be ruled out at compile time by type checking are not signaled until run time.

Consider, for example, the addition function in **DPCF**, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[3]

$$\texttt{fun}(\,x.\texttt{fix}(\,p.\texttt{fun}(\,y.\texttt{ifz}\{x;y'.\texttt{succ}(\,p(\,y'\,)\,)\}(\,y\,)\,)\,)\,).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda\,(\,x\,)\,\texttt{fix}\,p\,\texttt{is}\,\lambda\,(\,y\,)\,\texttt{ifz}\,y\,\{\texttt{zero} \hookrightarrow x \mid \texttt{succ}(\,y'\,) \hookrightarrow \texttt{succ}(\,p(\,y'\,)\,)\}.$$

It is deceptive, because it obscures the class tags on values, and the operations that check the validity of those tags. Let us now examine the costs of these operations in a bit more detail.

First, note that the body of the fixed point expression is labeled with class `fun`. The dynamics of the fixed point construct binds p to this function. Consequently, the dynamic class check incurred by the application of p in the recursive call is guaranteed to succeed. But **DPCF** offers no means of suppressing the redundant check, because it cannot express the invariant that p is always bound to a value of class `fun`.

Second, note that the result of applying the inner λ-abstraction is either x, the argument of the outer λ-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class `num`, even though this condition is guaranteed to hold for all but the base case, which returns the given x, which can be of any class at all. In principle we can check that x is of class `num` once, and note that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, **DPCF** gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

Third, the argument y to the inner function is either the original argument to the addition function, or is the predecessor of some earlier recursive call. But as long as the original call is to a value of class `num`, then the dynamics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in **DPCF**, and hence there is no way to avoid the class check imposed by the conditional branch.

[3] This specification imposes no restrictions on the behavior of addition on arguments that are not classified as numbers, but we could make the further demand that the function abort when applied to arguments that are not classified by `num`.

Classification is not free—storage is required for the class label, and it takes time to detach the class from a value each time it is used and to attach a class to a value when it is created. Although the overhead of classification is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated when possible. But this is impossible within **DPCF**, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

22.4 Notes

The earliest dynamically typed language is Lisp (McCarthy, 1965), which continues to influence language design a half century after its invention. Dynamic PCF is the core of Lisp, but with a proper treatment of variable binding, correcting what McCarthy himself has described as an error in the original design. Informal discussions of dynamic languages are often complicated by the elision of the dynamic checks that are made explicit here. Although the surface syntax of dynamic PCF is almost the same as that for PCF, minus the type annotations, the underlying dynamics is different. It is for this reason that static PCF cannot be seen as a restriction of dynamic PCF by the imposition of a type system.

Exercises

22.1. Surface syntax can be deceiving; even simple arithmetic expressions do not have the same meaning in **DPCF** that they do in **PCF**. To see why, define the addition function, `plus`, in **DPCF**, and examine the dynamics of evaluating expressions such as $\texttt{plus}(\overline{5})(\overline{7})$. Even though this expression might be written as "$5 + 7$" in both static and dynamic languages, they have different meanings.

22.2. Give a precise dynamics to the data structuring primitives described informally in Section 22.2. What class restrictions should `cons` impose on its arguments? Check the dynamics of the `append` function when called with two lists as arguments.

22.3. To avoid the difficulties with the representation of lists using `cons` and `nil`, introduce a class of lists that are constructed using revised versions of `nil` and `cons` that operate on the class of lists. Revisit the dynamics of the `append` function when redefined using the class of lists.

22.4. Allowing multiple arguments to, and multiple results from, functions is a notorious source of trouble in dynamic languages. The restriction to a single type makes it impossible even to distinguish n things from m things, let alone express more subtle properties of a program. Numerous workarounds have been proposed. Explore the problem yourself by enriching **DPCF** with multi-argument and multi-result functions. Be sure to consider these questions:

(a) If a function is defined with n parameters, what should happen if it is called with more or fewer than n arguments?

(b) What happens if one were to admit functions with a *varying* number of arguments? How would you refer to these arguments within the body of such a function? How does this relate to pattern matching?

(c) What if one wished to admit keyword parameter passing by giving names to the arguments, and allowing them to be passed in any order by associating them with their names?

(d) What notation would you suggest for functions returning multiple results? For example, a division function might return the quotient and the remainder. How might one notate this in the function body? How would a caller access the results individually or collectively?

(e) How would one define the composition of two functions when either or both can take multiple arguments or return multiple results?

Part X

Subtyping

Chapter 24

Structural Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> *if τ' is a subtype of τ, then a value of type τ' may be provided when a value of type τ is required.*

The subsumption principle relaxes the strictures of a type system to allow values of one type to be treated as values of another.

Experience shows that the subsumption principle, although useful as a general guide, can be tricky to apply correctly in practice. The key to getting it right is the principle of introduction and elimination. To see whether a candidate subtyping relationship is sensible, it suffices to consider whether every *introduction* form of the subtype can be safely manipulated by every *elimination* form of the supertype. A subtyping principle makes sense only if it passes this test; the proof of the type safety theorem for a given subtyping relation ensures that this is the case.

A good way to get a subtyping principle wrong is to think of a type merely as a set of values (generated by introduction forms), and to consider whether every value of the subtype can also be considered to be a value of the supertype. The intuition behind this approach is to think of subtyping as akin to the subset relation in ordinary mathematics. But, as we shall see, this can lead to serious errors, because it fails to take account of the elimination forms that are applicable to the supertype. It is not enough to think only of the introduction forms; subtyping is a matter of *behavior*, and not *containment*.

24.1 Subsumption

A *subtyping judgment* has the form $\tau' <: \tau$, and states that τ' is a subtype of τ. At the least we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{24.1a}$$

$$\frac{\tau'' <: \tau' \quad \tau' <: \tau}{\tau'' <: \tau} \tag{24.1b}$$

In practice we either tacitly include these rules as primitive, or prove that they are admissible for a given set of subtyping rules.

The point of a subtyping relation is to enlarge the set of well-typed programs, which is accomplished by the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash e : \tau} \tag{24.2}$$

In contrast to most other typing rules, the rule of subsumption is *not* syntax-directed, because it does not constrain the form of e. That is, the subsumption rule can be applied to *any* form of expression. In particular, to show that $e : \tau$, we have two choices: either apply the rule appropriate to the particular form of e, or apply the subsumption rule, checking that $e : \tau'$ and $\tau' <: \tau$.

24.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping in the context of extensions of the language **FPC** introduced in Chapter 20.

Numeric Types

We begin with an informal discussion of numeric types such as are common in many programming languages. Our mathematical experience suggests subtyping relationships among numeric types. For example, in a language with types `int`, `rat`, and `real`, representing the integers, the rationals, and the reals, it is tempting to postulate the subtyping relationships

$$\texttt{int} <: \texttt{rat} <: \texttt{real}$$

by analogy with the set containments

$$\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}.$$

But are these subtyping relationships sensible? The answer depends on the representations and interpretations of these types. Even in mathematics, the containments just mentioned are usually not true—or are true only in a rough sense. For example, the set of rational numbers can be considered to consist of ordered pairs (m, n), with $n \neq 0$ and $\gcd(m, n) = 1$, representing the ratio m/n. The set $\mathbb{Z}$ of integers can be isomorphically embedded within $\mathbb{Q}$ by identifying $n \in \mathbb{Z}$ with the ratio $n/1$. Similarly, the real numbers are often represented as convergent sequences of rationals, so that strictly speaking the rationals are not a subset of the reals, but rather can be embedded in them by choosing a canonical representative (a particular convergent sequence) of each rational.

For mathematical purposes it is entirely reasonable to overlook fine distinctions such as that between $\mathbb{Z}$ and its embedding within $\mathbb{Q}$. Ignoring the difference is justified because the operations on rationals restrict to the embedding in the expected way: if we add two integers thought of as rationals in the canonical way, then the result is the rational associated with their sum. And similarly for the other operations, provided that we take some care in defining them to ensure that it all works out properly. For the purposes of computing, however, we must also take account of algorithmic efficiency and the finiteness of machine representations. For example, what are often

called "real numbers" in a programming language are, of course, floating point numbers, a finite subset of the rational numbers. Not every rational can be exactly represented as a floating point number, nor does floating point arithmetic restrict to rational arithmetic, even when its arguments are exactly represented as floating point numbers.

Product Types

Product types give rise to a form of subtyping based on the subsumption principle. The only elimination form applicable to a value of product type is a projection. Under mild assumptions about the dynamics of projections, we may consider one product type to be a subtype of another by considering whether the projections applicable to the supertype can be validly applied to values of the subtype.

Consider a context in which a value of type $\tau = \langle \tau_j \rangle_{j \in J}$ is required. The statics of finite products (rules (10.3)) ensures that the only operation we may perform on a value of type τ, other than to bind it to a variable, is to take the jth projection from it for some $j \in J$ to obtain a value of type τ_j. Now suppose that e is of type τ'. For the projection $e \cdot j$ to be well-formed, then τ' is a finite product type $\langle \tau'_i \rangle_{i \in I}$ such that $j \in I$. Moreover, for the projection to be of type τ_j, it is enough to require that $\tau'_j = \tau_j$. Because $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \ . \tag{24.3}$$

This rule sufices for the required subtyping, but not necessary; we will consider a more liberal form of this rule in Section 24.3. The justification for rule (24.3) is that we may evaluate $e \cdot i$ regardless of the actual form of e, provided only that it has a field indexed by $i \in I$.

Sum Types

By an argument dual to the one given for finite product types we may derive a related subtyping rule for finite sum types. If a value of type $\sum_{j \in J} \tau_j$ is required, the statics of sums (rules (11.3)) ensures that the only non-trivial operation that we may perform on that value is a J-indexed case analysis. If we provide a value of type $\sum_{i \in I} \tau'_i$ instead, no difficulty will arise so long as $I \subseteq J$ and each τ'_i is equal to τ_i. If the containment is strict, some cases cannot arise, but this does not disrupt safety.

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j} \ . \tag{24.4}$$

Note well the reversal of the containment as compared to rule (24.3).

Dynamic Types

A popular form of subtyping is associated with the type `dyn` introduced in Chapter 23. The type `dyn` provides no information about the class of a value of this type. One might argue that it is whole the point of dynamic typing to suppress this information statically, making it available only

dynamically. On the other hand, it is not much trouble to introduce subtypes of dyn that specify the class of a value, relying on subsumption to "forget" the class when it cannot be determined statically.

Working in the context of Chapter 23 this amounts to introduce two new types, dyn[num] and dyn[fun], governed by the following two subtyping axioms:

$$\frac{}{\texttt{dyn}[\texttt{num}] <: \texttt{dyn}} \tag{24.5a}$$

$$\frac{}{\texttt{dyn}[\texttt{fun}] <: \texttt{dyn}} \tag{24.5b}$$

Of course, in a richer language with more classes of dynamic values one would correspondingly introduce more such subtypes of dyn, one for each additional class. As a matter of notation, the type dyn is frequently spelled object, and its class-specific subtypes dyn[num] and dyn[fun], are often written as num and fun, respectively. But doing so invites confusion between the separate concepts of *class* and *type*, as discussed in detail in Chapters 22 and 23.

The class-specific subtypes of dyn come into play by reformulating the typing rules for introducing values of type dyn to note the class of the created value:

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new}[\texttt{num}](e) : \texttt{dyn}[\texttt{num}]} \tag{24.6a}$$

$$\frac{\Gamma \vdash e : \texttt{dyn} \rightharpoonup \texttt{dyn}}{\Gamma \vdash \texttt{new}[\texttt{fun}](e) : \texttt{dyn}[\texttt{fun}]} \tag{24.6b}$$

Thus, in this formulation, classified values "start life" with class-specific types, because in those cases it is statically apparent what is the class of the introduced value. Subsumption is used to weaken the type to dyn in those cases where no static prediction can be made—for example, when the branches of a conditional evaluate to dynamic values of different classes it is necessary to weaken the type of the branches to dyn.

The advantage of such a subtyping mechanism is that we can express more precise types, such as the type dyn[num] $\rightharpoonup$ dyn[num] of functions mapping a value of type dyn with class num to another such value. This typing is more precise than, say, dyn $\rightharpoonup$ dyn, which merely classifies functions that act on dynamically typed values. In this way weak invariants can be expressed and enforced, but only insofar as it is possible to track the classes of the values involved in a computation. Subtyping is not nearly a powerful enough mechanism for practical situations, rendering the additional specificity not worth the effort of including it. (A more powerful approach is developed in Chapter 25.)

24.3 Variance

In addition to basic subtyping principles such as those considered in Section 24.2, it is also important to consider the effect of subtyping on type constructors. A type constructor *covariant* in an

argument if subtyping in that argument is preserved by the constructor. It is *contravariant* if subtyping in that argument is reversed by the constructor. It is *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

Product and Sum Types

Finite product types are *covariant* in each field. For if e is of type $\prod_{i \in I} \tau_i'$, and the projection $e \cdot j$ is to be of type τ_j, then it suffices to require that $j \in I$ and $\tau_j' <: \tau_j$.

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\prod_{i \in I} \tau_i' <: \prod_{i \in I} \tau_i} \tag{24.7}$$

It is implicit in this rule that the dynamics of projection cannot be sensitive to the precise type of any of the fields of a value of finite product type.

Finite sum types are also covariant, because each branch of a case analysis on a value of the supertype expects a value of the corresponding summand, for which it suffices to provide a value of the corresponding subtype summand:

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\sum_{i \in I} \tau_i' <: \sum_{i \in I} \tau_i} \tag{24.8}$$

Partial Function Types

The variance of the function type constructors is a bit more subtle. Let us consider first the variance of the function type in its range. Suppose that $e : \tau_1 \rightharpoonup \tau_2'$. Then if $e_1 : \tau_1$, then $e(e_1) : \tau_2'$, and if $\tau_2' <: \tau_2$, then $e(e_1) : \tau_2$ as well.

$$\frac{\tau_2' <: \tau_2}{\tau_1 \rightharpoonup \tau_2' <: \tau_1 \rightharpoonup \tau_2} \tag{24.9}$$

Every function that delivers a value of type τ_2' also delivers a value of type τ_2, provided that $\tau_2' <: \tau_2$. Thus the function type constructor is covariant in its range.

Now let us consider the variance of the function type in its domain. Suppose again that $e : \tau_1 \rightharpoonup \tau_2$. Then e can be applied to any value of type τ_1 to obtain a value of type τ_2. Hence, by the subsumption principle, it can be applied to any value of a subtype τ_1' of τ_1, and it will still deliver a value of type τ_2. Consequently, we may just as well think of e as having type $\tau_1' \rightharpoonup \tau_2$.

$$\frac{\tau_1' <: \tau_1}{\tau_1 \rightharpoonup \tau_2 <: \tau_1' \rightharpoonup \tau_2} \tag{24.10}$$

The function type is contravariant in its domain position. Note well the reversal of the subtyping relation in the premise as compared to the conclusion of the rule!

Combining these rules we obtain the following general principle of contra- and covariance for function types:

$$\frac{\tau_1' <: \tau_1 \quad \tau_2' <: \tau_2}{\tau_1 \rightharpoonup \tau_2' <: \tau_1' \rightharpoonup \tau_2} \tag{24.11}$$

Beware of the reversal of the ordering in the domain!

Recursive Types

The language **FPC** has a partial function types, which behave the same under subtyping as total function types, sums and products, which behave as described above, and recursive types, which introduce some subtleties that have been the source of error in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\, \mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \langle \mathtt{data} \hookrightarrow \mathtt{nat}, \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle\,],$$

and the type of "bare" binary trees, without data attached to the nodes,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\, \mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle\,].$$

Is either a subtype of the other? Intuitively, we might expect the type of labeled binary trees to be a *subtype* of the type of bare binary trees, because any use of a bare binary tree can simply ignore the presence of the label.

Now consider the type of bare "two-three" trees with two sorts of nodes, those with two children, and those with three:

$$\mathtt{rec}\, t\, \mathtt{is}\, [\, \mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \tau_2, \mathtt{trinode} \hookrightarrow \tau_3\,],$$

where

$$\tau_2 \triangleq \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle, \text{and}$$
$$\tau_3 \triangleq \langle \mathtt{lft} \hookrightarrow t, \mathtt{mid} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle.$$

What subtype relationships should hold between this type and the preceding two tree types? Intuitively the type of bare two-three trees should be a *supertype* of the type of bare binary trees, because any use of a two-three tree proceeds by three-way case analysis, which covers both forms of binary tree.

To capture the pattern illustrated by these examples, we need a subtyping rule for recursive types. It is tempting to consider the following rule:

$$\frac{t\ \mathsf{type} \vdash \tau' <: \tau}{\mathtt{rec}\, t\, \mathtt{is}\, \tau' <: \mathtt{rec}\, t\, \mathtt{is}\, \tau}\ \textbf{??} \tag{24.12}$$

That is, to check whether one recursive type is a subtype of the other, we simply compare their bodies, with the bound variable treated as an argument. Notice that by reflexivity of subtyping, we have $t <: t$, and hence we may use this fact in the derivation of $\tau' <: \tau$.

Rule (24.12) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. Deriving this requires checking that the subtyping relationship

$$\langle \mathtt{data} \hookrightarrow \mathtt{nat}, \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle <: \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle,$$

holds generically in t, which is evidently the case.

Unfortunately, Rule (24.12) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\tau' = \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{a} \hookrightarrow t \rightharpoonup \mathtt{nat}, \mathtt{b} \hookrightarrow t \rightharpoonup \mathtt{int} \rangle$$

and

$$\tau = \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{a} \hookrightarrow t \rightharpoonup \mathtt{int}, \mathtt{b} \hookrightarrow t \rightharpoonup \mathtt{int} \rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using rule (24.12) we may derive $\tau' <: \tau$, which is incorrect. Let $e : \tau'$ be the expression

$$\mathtt{fold}(\, \langle \mathtt{a} \hookrightarrow \lambda\, (\, x : \tau'\,)\, 4, \mathtt{b} \hookrightarrow \lambda\, (\, x : \tau'\,)\, q(\, (\, \mathtt{unfold}(\, x\,) \cdot \mathtt{a}\,)(\, x\,)\,)\rangle\,),$$

where $q : \mathtt{nat} \rightharpoonup \mathtt{nat}$ is the discrete square root function. Because $\tau' <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(\, e\,) : \langle \mathtt{a} \hookrightarrow \tau \rightharpoonup \mathtt{int}, \mathtt{b} \hookrightarrow \tau \rightharpoonup \mathtt{int} \rangle.$$

Now let $e' : \tau$ be the expression

$$\mathtt{fold}(\, \langle \mathtt{a} \hookrightarrow \lambda\, (\, x : \tau\,)\, \mathtt{-4}, \mathtt{b} \hookrightarrow \lambda\, (\, x : \tau\,)\, \mathtt{0} \rangle\,).$$

(The important point about e' is that the `a` method returns a negative number; the `b` method is of no significance.) To finish the proof, observe that

$$(\, \mathtt{unfold}(\, e\,) \cdot \mathtt{b}\,)(\, e'\,) \longmapsto^* q(\, \mathtt{-4}\,),$$

which is a stuck state. We have derived a well-typed program that "gets stuck", refuting type safety!

Rule (24.12) is therefore incorrect. But what has gone wrong? The error lies in the choice of a single variable to stand for both recursive types, which does not correctly model self-reference. In effect we are treating two distinct recursive types as if they were equal while checking their bodies for a subtyping relationship. But this is clearly wrong! It fails to take account of the self-referential nature of recursive types. On the left side the bound variable stands for the subtype, whereas on the right the bound variable stands for the super-type. Confusing them leads to the unsoundness just illustrated.

As is often the case with self-reference, the solution is to *assume* what we are trying to prove, and check that this assumption can be maintained by examining the bodies of the recursive types. To do so we use hypothetical judgments of the form $\Delta \vdash \tau' <: \tau$, where Δ consists of hypotheses t type and $t <: \tau$ that declares a fresh type variable t that is not otherwise declared in Δ. Using such hypothetical judgments we may state the correct rule for subtyping recursive types as follows:

$$\frac{\Delta, t\ \mathsf{type}, t'\ \mathsf{type}, t' <: t \vdash \tau' <: \tau \quad \Delta, t'\ \mathsf{type} \vdash \tau'\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}\, t'\, \mathtt{is}\, \tau' <: \mathtt{rec}\, t\, \mathtt{is}\, \tau}. \tag{24.13}$$

That is, to check whether $\mathtt{rec}\, t'\, \mathtt{is}\, \tau' <: \mathtt{rec}\, t\, \mathtt{is}\, \tau$, we assume that $t' <: t$, because t' and t stand for the corresponding recursive types, and check that $\tau' <: \tau$ under this assumption. It is instructive to check that the unsound subtyping example given above is not derivable using this rule: the subtyping assumption is at odds with the contravariance of the function type in its domain.

Quantified Types

Consider extending **FPC** with the universal and existential quantified types discussed in Chapters 16 and 17. The variance principles for the quantifiers state that they are uniformly covariant in the quantified types:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau' <: \tau}{\Delta \vdash \forall(t.\tau') <: \forall(t.\tau)} \tag{24.14a}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau' <: \tau}{\Delta \vdash \exists(t.\tau') <: \exists(t.\tau)} \tag{24.14b}$$

Consequently, we may derive the principle of substitution:

Lemma 24.1. *If* $\Delta, t\ \mathit{type} \vdash \tau_1 <: \tau_2$, *and* $\Delta \vdash \tau\ \mathit{type}$, *then* $\Delta \vdash [\tau/t]\tau_1 <: [\tau/t]\tau_2$.

Proof. By induction on the subtyping derivation. □

It is easy to check that the above variance principles for the quantifiers are consistent with the principle of subsumption. For example, a package of the subtype $\exists(t.\tau')$ consists of a representation type ρ and an implementation e of type $[\rho/t]\tau'$. But if $t\ \mathsf{type} \vdash \tau' <: \tau$, we have by substitution that $[\rho/t]\tau' <: [\rho/t]\tau$, and hence e is also an implementation of type $[\rho/t]\tau$. So the package is also of the supertype.

It is natural to extend subtyping to the quantifiers by allowing quantification over all subtypes of a specified type; this is called *bounded quantification*.

$$\frac{}{\Delta, t\ \mathsf{type}, t <: \tau \vdash t <: \tau} \tag{24.15a}$$

$$\frac{\Delta \vdash \tau :: \mathtt{T}}{\Delta \vdash \tau <: \tau} \tag{24.15b}$$

$$\frac{\Delta \vdash \tau'' <: \tau' \quad \Delta \vdash \tau' <: \tau}{\Delta \vdash \tau'' <: \tau} \tag{24.15c}$$

$$\frac{\Delta \vdash \tau_1' <: \tau_1 \quad \Delta, t\ \mathsf{type}, t <: \tau_1' \vdash \tau_2 <: \tau_2'}{\Delta \vdash \forall\, t <: \tau_1.\tau_2 <: \forall\, t <: \tau_1'.\tau_2'} \tag{24.15d}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_1' \quad \Delta, t\ \mathsf{type}, t <: \tau_1 \vdash \tau_2 <: \tau_2'}{\Delta \vdash \exists\, t <: \tau_1.\tau_2 <: \exists\, t <: \tau_1'.\tau_2'} \tag{24.15e}$$

Rule (24.15d) states that the universal quantifier is contravariant in its bound, whereas rule (24.15e) states that the existential quantifier is covariant in its bound.

24.4 Dynamics and Safety

There is a subtle assumption in the definition of product subtyping in Section 24.2, namely that the *same* projection operation from an I-tuple applies also to a J-tuple, provided $J \supseteq I$. But this need not be the case. One could represent I-tuples differently from J-tuples at will, so that the meaning of the projection at position $i \in I \subseteq J$ is different in the two cases. Nothing rules out this possibility, yet product subtyping relies on it not being the case. From this point of view product subtyping is not well-justified, but one may instead consider that subtyping limits possible implementations to ensure that it makes sense.

Similar considerations apply to sum types. An J-way case analysis need not be applicable to an I-way value of sum type, even when $I \subseteq J$ and all the types in common agree. For example, one might represent values of a sum type with a "small" index set in a way that is not applicable for a "large" index set. In that case the "large" case analysis would not make sense on a value of "small" sum type. Here again we may consider either that subtyping is not justified, or that it imposes limitations on the implementation that are not otherwise forced.

These considerations merit careful consideration of the safety of languages with subtyping. As an illustrative case we consider the safety of **FPC** enriched with product subtyping. The main concern is that the subsumption rule obscures the "true" type of a value, complicating the canonical forms lemma. Moreover, we assume that the same projection makes sense for a wider tuple than a narrower one, provided that it is within range.

Lemma 24.2 (Structurality)**.**

1. *The tuple subtyping relation is reflexive and transitive.*
2. *The typing judgment $\Gamma \vdash e : \tau$ is closed under weakening and substitution.*

Proof.

1. Reflexivity is proved by induction on the structure of types. Transitivity is proved by induction on the derivations of the judgments $\tau'' <: \tau'$ and $\tau' <: \tau$ to obtain a derivation of $\tau'' <: \tau$.
2. By induction on rules (10.3), augmented by rule (24.2).

□

Lemma 24.3 (Inversion)**.**

1. *If $e \cdot j : \tau$, then $e : \prod_{i \in I} \tau_i$, $j \in I$, and $\tau_j <: \tau$.*
2. *If $\langle e_i \rangle_{i \in I} : \tau$, then $\prod_{i \in I} \tau_i' <: \tau$ where $e_i : \tau_i'$ for each $i \in I$.*
3. *If $\tau' <: \prod_{j \in J} \tau_j$, then $\tau' = \prod_{i \in I} \tau_i'$ for some I and some types τ_i' for $i \in I$.*
4. *If $\prod_{i \in I} \tau_i' <: \prod_{j \in J} \tau_j$, then $J \subseteq I$ and $\tau_j' <: \tau_j$ for each $j \in J$.*

Proof. By induction on the subtyping and typing rules, paying special attention to rule (24.2). □

Theorem 24.4 (Preservation). *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof. By induction on rules (10.4). For example, consider rule (10.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$ and $e' = e_k$. By Lemma 24.3 we have $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, with $k \in J$ and $\tau_k <: \tau$. By another application of Lemma 24.3 for each $i \in I$ there exists τ_i' such that $e_i : \tau_i'$ and $\prod_{i \in I} \tau_i' <: \prod_{j \in J} \tau_j$. By Lemma 24.3 again, we have $J \subseteq I$ and $\tau_j' <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaining cases are similar. □

Lemma 24.5 (Canonical Forms). *If e val and $e : \prod_{j \in J} \tau_j$, then e is of the form $\langle e_i \rangle_{i \in I}$, where $J \subseteq I$, and $e_j : \tau_j$ for each $j \in J$.*

Proof. By induction on rules (10.3) augmented by rule (24.2). □

Theorem 24.6 (Progress). *If $e : \tau$, then either e val or there exists e' such that $e \longmapsto e'$.*

Proof. By induction on rules (10.3) augmented by rule (24.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. rule (10.4d) follows from Lemma 24.5. □

24.5 Notes

Subtyping is perhaps the most widely misunderstood concept in programming languages. Subtyping is principally a convenience, akin to type inference, that makes some programs simpler to write. But the subsumption rule cuts both ways. Inasmuch as it allows the implicit passage from τ' to τ when τ' is a subtype of τ, it also weakens the meaning of a type assertion $e : \tau$ to mean that e has some type contained in the type τ. Subsumption precludes expressing the requirement that e has *exactly* the type τ, or that two expressions jointly have the *same* type. And it is just this weakness that creates so many of the difficulties with subtyping.

Much has been written about subtyping, often in relation to object-oriented programming. Standard ML (Milner et al., 1997) is one of the first languages to make use of subtyping, in two forms, called *enrichment* and *realization*. The former corresponds to product subtyping, and the latter to the "forgetful" subtyping associated with type definitions (see Chapter 43). The first systematic studies of subtyping include those by Mitchell (1984); Reynolds (1980), and Cardelli (1988). Pierce (2002) give a thorough account of subtyping, especially of recursive and polymorphic types, and proves that subtyping for bounded impredicative universal quantification is undecidable.

Exercises

24.1. Check the variance of the type

$$(\,\mathtt{unit} \rightharpoonup \tau\,) \times (\,\tau \rightharpoonup \mathtt{unit}\,).$$

When viewed as a constructor with argument τ, is it covariant or contravariant? Give a precise proof or counterexample in each case.

24.2. Consider the two recursive types,

$$\rho_1 \triangleq \texttt{rec}\, t\, \texttt{is}\, \langle \texttt{eq} \hookrightarrow (\, t \rightharpoonup \texttt{bool}\,) \rangle,$$

and

$$\rho_2 \triangleq \texttt{rec}\, t\, \texttt{is}\, \langle \texttt{eq} \hookrightarrow (\, t \rightharpoonup \texttt{bool}\,), \texttt{f} \hookrightarrow \texttt{bool} \rangle.$$

It is clear that ρ_1 could not be a subtype of ρ_2, because, viewed as a product after unrolling, a value of the former type lacks a component that a value of the latter has. But is ρ_2 a subtype of ρ_1? If so, prove it by exhibiting a derivation of this fact using the rules given in Section 24.3. If not, give a counterexample showing that the suggested subtyping would violate type safety.

24.3. Another approach to the dynamics of subtyping that ensures safety, but gives subsumption dynamic significance, associates a witness, called a *coercion*, to each subtyping relation, and inserts a coercion wherever subsumption is used. More precisely,

(a) Assign to each valid subtyping $\tau <: \tau'$ a coercion function $\chi : \tau \rightharpoonup \tau'$ that transforms a value of type τ into a value of type τ'.

(b) Interpret the subsumption rule as implicit coercion. Specifically, when $\tau <: \tau'$ is witnessed by $\chi : \tau \rightharpoonup \tau'$, applying subsumption to $e : \tau$ inserts an application of χ to obtain $\chi(\, e\,) : \tau'$.

Formulate this idea precisely for the case of a subtype relation generated by "width" subtyping for products, and the variance principles for product, sum and function types. Your solution should make clear that it evades the tacit projection assumption mentioned above.

But there may be more than one coercion $\chi : \tau \rightharpoonup \tau'$ corresponding to the subtyping $\tau <: \tau'$. The meaning of a program would then depend on which coercion is chosen when subsumption is used. If there is exactly one coercion for each subtyping relation, it is said to be *coherent*. Is your coercion interpretation of product subtyping coherent? (A proper treatment of coherence requires expression equivalence, which is discussed in Chapter 47.)

Part XI

Dynamic Dispatch

Chapter 27

Inheritance

In this chapter we build on Chapter 26 and consider the process of defining the dispatch matrix that determines the behavior of each method on each class. A common strategy is to build the dispatch matrix incrementally by adding new classes or methods to an existing dispatch matrix. To add a class requires that we define the behavior of each method on objects of that class, and to define a method requires that we define the behavior of that method on objects of the classes. The definition of these behaviors can be given by any means available in the language. However, it is often suggested that a useful means of defining a new class is to *inherit* the behavior of another class on some methods, and to *override* its behavior on others, resulting in an amalgam of the old and new behaviors. The new class is often called a *subclass* of the old class, which is then called the *superclass*. Similarly, a new method can be defined by inheriting the behavior of another method on some classes, and overriding the behavior on others. By analogy we may call the new method a *sub-method* of a given *super-method*. For the sake of clarity we restrict attention to the non-self-referential case in the following development.

27.1 Class and Method Extension

We begin by extending a given dispatch matrix, e_{dm}, of type

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d)$$

with a new class $c^* \notin C$ and a new method $d^* \notin D$ to obtain a new dispatch matrix e^*_{dm} of type

$$\prod_{c \in C^*} \prod_{d \in D^*} (\tau^c \to \rho_d),$$

where $C^* = C \cup \{ c^* \}$ and $D^* = D \cup \{ d^* \}$.

To add a new class c^* to the dispatch matrix, we must specify the following information:[1]

[1] The extension with a new method will be considered separately for the sake of clarity.

1. The instance type τ^{c^*} of the new class c^*.
2. The behavior $e_d^{c^*}$ of each method $d \in D$ on an object of the new class c^*, a function of type $\tau^{c^*} \to \rho_d$.

This data determines a new dispatch matrix e^*_{dm} such that the following conditions are satisfied:

1. For each $c \in C$ and $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c \cdot d$ is the same as the behavior $e_{\mathsf{dm}} \cdot c \cdot d$.
2. For each $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c^* \cdot d$ is given by $e_d^{c^*}$.

To define c^* as a subclass of some class $c \in C$ means to define the behavior $e_d^{c^*}$ to be e_d^c for some (perhaps many) $d \in D$. It is sensible to inherit a method d in this way only if the subtype relationship

$$\tau^c \to \rho_d <: \tau^{c^*} \to \rho_d$$

is valid, which will be the case if $\tau^{c^*} <: \tau^c$. This subtyping condition ensures that the inherited behavior can be invoked on the instance data of the new class.

Similarly, to add a new method d^* to the dispatch matrix, we must specify the following information:

1. The result type ρ_{d^*} of the new method d^*.
2. The behavior $e^c_{d^*}$ of the new method d^* on an object of each class $c \in C$, a function of type $\tau^c \to \rho_{d^*}$.

This data determines a new dispatch matrix e^*_{dm} such that the following conditions are satisfied:

1. For each $c \in C$ and $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c \cdot d$ is the same as $e_{\mathsf{dm}} \cdot c \cdot d$.
2. The behavior $e^*_{\mathsf{dm}} \cdot c \cdot d^*$ is given by $e^c_{d^*}$.

To define d^* as a sub-method of some $d \in D$ means to define the behavior $e^c_{d^*}$ to be e^c_d for some (perhaps many) classes $c \in C$. This definition is only sensible if the subtype relationship

$$\tau^c \to \rho_d <: \tau^c \to \rho_{d^*}$$

holds, which is the case if $\rho_d <: \rho_{d^*}$. This subtyping relationship ensures that the result of the old behavior suffices for the new behavior.

We will now consider how inheritance relates to the method- and class-based organizations of dynamic dispatch considered in Chapter 26.

27.2 Class-Based Inheritance

Recall that the class-based organization given in Chapter 26 consists of a class vector e_{cv} of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \to \rho),$$

where the object type ρ is the finite product type $\prod_{d \in D} \rho_d$. The class vector consists of a tuple of constructors that specialize the methods to a given object of each class.

Let us consider the effect of adding a new class c^* as described in Section 27.1. The new class vector e^*_{cv} has type

$$\tau^*_{\mathsf{cv}} \triangleq \prod_{c \in C^*} (\tau^c \to \rho).$$

There is an isomorphism, written $(\,)^\dagger$, between τ^*_{cv} and the type

$$\tau_{\mathsf{cv}} \times (\tau^{c^*} \to \rho),$$

which can be used to define the new class vector e^*_{cv} as follows:

$$\langle e_{\mathsf{cv}}, \lambda\,(\,u : \tau^{c^*}\,)\,\langle d \hookrightarrow e^{c^*}_d\,(\,u\,)\rangle_{d \in D}\rangle^\dagger.$$

This definition makes clear that the old class vector e_{cv} is reused intact in the new class vector, which extends the old class vector with a new constructor.

Although the object type ρ is the same both before and after the extension with the new class, the behavior of an object of class c^* may differ arbitrarily from that of any other object, even that of the superclass from which it inherits its behavior. So, knowing that c^* inherits from c tells us nothing about the behavior of its objects, but only about the means by which the class is defined. Inheritance carries no semantic significance, but is only a record of the history of how a class is defined.

Now let us consider the effect of adding a new method d^* as described in Section 27.1. The new class vector e^*_{cv} has type

$$\tau^*_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \to \rho^*),$$

where ρ^* is the product type $\prod_{d \in D^*} \rho_d$. There is an isomorphism, written $(\,)^\ddagger$, between ρ^* and the type $\rho \times \rho_{d^*}$, where ρ is the old object type. Using this the new class vector e^*_{cv} is defined by

$$\langle c \hookrightarrow \lambda\,(\,u : \tau^c\,)\,\langle\langle d \hookrightarrow (\,(\,e_{\mathsf{cv}} \cdot c\,)(\,u\,)\,) \cdot d\rangle_{d \in D}, e^c_{d^*}(\,u\,)\rangle^\ddagger\rangle_{c \in C}.$$

Observe that each constructor must be re-defined to account for the new method, but the definition makes use of the old class vector for the definitions of the old methods.

By this construction the new object type ρ^* is a subtype of the old object type ρ. Consequently, any objects with the new method can be used in situations expecting an object without the new method, as might be expected. To avoid redefining old classes when a new method is introduced, we may restrict inheritance so that new methods are only added to new subclasses. Subclasses may then have more methods than super-classes, and objects of the subclass can be provided when an object of the superclass is required.

27.3 Method-Based Inheritance

The method-based organization is dual to that of the class-based organization. Recall that the method-based organization given in Chapter 26 consists of a method vector e_{mv} of type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} \tau \to \rho_d,$$

where the instance type τ is the sum type $\sum_{c \in C} \tau^c$. The method vector consists of a tuple of functions that dispatch on the class of the object to determine their behavior.

Let us consider the effect of adding a new method d^* as described in Section 27.1. The new method vector e^*_{mv} has type

$$\tau^*_{\mathsf{mv}} \triangleq \prod_{d \in D^*} \tau \to \rho_d.$$

There is an isomorphism, written $(\,)^\ddagger$, between τ^*_{mv} and the type

$$\tau_{\mathsf{mv}} \times (\tau \to \rho_{d^*}).$$

Using this isomorphism, the new method vector e^*_{mv} is defined as

$$\langle e_{\mathsf{mv}}, \lambda\,(\,\mathit{this} : \tau\,)\ \mathtt{case}\ \mathit{this}\ \{c \cdot u \hookrightarrow e^c_{d^*}(\,u\,)\}_{c \in C} \rangle^\ddagger.$$

The old method vector is re-used intact, extended with a dispatch function for the new method.

The object type does not change under the extension with a new method, but because $\rho^* <: \rho$, there is no difficulty using a new object in a context expecting an old object—the added method is ignored.

Finally, let us consider the effect of adding a new class c^* as described in Section 27.1. The new method vector, e^*_{mv}, has the type

$$\tau^*_{\mathsf{mv}} \triangleq \prod_{d \in D} \tau^* \to \rho_d,$$

where τ^* is the new object type $\sum_{c \in C^*} \tau^c$, which is a super-type of the old object type τ. There is an isomorphism, written $(\,)^\dagger$, between τ^* and the sum type $\tau + \tau^{c^*}$, which we may use to define the new method vector e^*_{mv} as follows:

$$\langle d \hookrightarrow \lambda\,(\,\mathit{this} : \tau^*\,)\ \mathtt{case}\ \mathit{this}^\dagger\ \{\mathtt{l} \cdot u \hookrightarrow (\,e_{\mathsf{mv}} \cdot d\,)(\,u\,) \mid \mathtt{r} \cdot u \hookrightarrow e^{c^*}_d(\,u\,)\} \rangle_{d \in D}.$$

Every method must be redefined to account for the new class, but the old method vector is reused.

27.4 Notes

Abadi and Cardelli (1996) and Pierce (2002) provide thorough accounts of the interaction of inheritance and subtyping. Liskov and Wing (1994) discuss it from a behavioral perspective. They propose to require that subclasses respect the behavior of the superclass when inheritance is used.

Exercises

27.1. Consider the case of extending a dispatch matrix with self-reference by a new class c^* in which a method d is inherited from an existing class c. What requirements ensure that such an inheritance is properly defined? What happens if we extend a self-referential dispatch matrix with a new method, d^* that inherits its behavior on class c from another method d?

27.2. Consider the example of two mutually recursive methods given in Exercise 26.3. Suppose that `num*` is a new class with instance type $\tau^{\texttt{num*}} <: \tau^{\texttt{num}}$ that inherits the `ev` method from `num`, but defines its own version of the `od` method. What happens when message `ev` is sent to an instance of `num*`? Will the revised `od` method ever be invoked?

27.3. *Method specialization* consists of defining a new class by inheriting methods from another class or classes, while redefining some of the methods that the inherited methods might invoke. The behavior of the inherited methods on instances of the new class is altered to the extent that they invoke a method that is specialized to the new class. Reconsider Exercise 26.3 in light of Exercise 27.2, seeking to ensure that the specialization of `od` is invoked when the inherited method `ev` is invoked on instances of the new class.

(a) Redefine the class `num` along the following lines. The instance data of `num` is an object admitting methods `ev` and `od`. The class `num` admits these methods, and simply hands them off to the instance object.

(b) The classes `zero` or of `succ` admit both the `ev` and `od` methods, and are defined using message send to effect mutual recursion as necessary.

(c) Define a subclass `succ*` of `succ` that overrides the `od` method. Show that `ev` on an instance of `succ*` correctly invokes the overridden `od` method.

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.

$$\frac{}{k \rhd \lambda\{\tau\}(x.e) \longmapsto k \lhd \lambda\{\tau\}(x.e)} \tag{28.5a}$$

$$\frac{}{k \rhd \mathtt{ap}(e_1; e_2) \longmapsto k\,;\mathtt{ap}(-; e_2) \rhd e_1} \tag{28.5b}$$

$$\frac{}{k\,;\mathtt{ap}(-; e_2) \lhd \lambda\{\tau\}(x.e) \longmapsto k \rhd [e_2/x]e} \tag{28.5c}$$

$$\frac{}{k \rhd \mathtt{fix}\{\tau\}(x.e) \longmapsto k \rhd [\mathtt{fix}\{\tau\}(x.e)/x]e} \tag{28.5d}$$

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:

$$\frac{}{\epsilon \rhd e \text{ initial}} \tag{28.6a}$$

$$\frac{e \text{ val}}{\epsilon \lhd e \text{ final}} \tag{28.6b}$$

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'.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \rightsquigarrow \mathtt{nat}} \tag{28.8a}$$

$$\frac{e_0 : \tau \quad x : \mathtt{nat} \vdash e_1 : \tau}{\mathtt{ifz}\{e_0; x.e_1\}(-) : \mathtt{nat} \rightsquigarrow \tau} \tag{28.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-; e_2) : \rightarrow(\tau_2; \tau) \rightsquigarrow \tau} \tag{28.8c}$$

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 K 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 $s \leftarrow\!\!\shortmid e'$, 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:

$$\frac{k \bowtie e = e'}{k \triangleright e \leftarrow\!\!\shortmid e'} \tag{28.11a}$$

$$\frac{k \rhd\lhd e = e'}{k \lhd e \mapsto e'} \tag{28.11b}$$

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:

$$\frac{}{\epsilon \rhd\lhd e = e} \tag{28.12a}$$

$$\frac{k \rhd\lhd \mathtt{s}(e) = e'}{k\,;\mathtt{s}(-) \rhd\lhd e = e'} \tag{28.12b}$$

$$\frac{k \rhd\lhd \mathtt{ifz}\{e_0; x.e_1\}(e) = e'}{k\,;\mathtt{ifz}\{e_0; x.e_1\}(-) \rhd\lhd e = e'} \tag{28.12c}$$

$$\frac{k \rhd\lhd \mathtt{ap}(e_1; e_2) = e}{k\,;\mathtt{ap}(-; e_2) \rhd\lhd e_1 = e} \tag{28.12d}$$

These judgments both define total functions.

Lemma 28.5. *The judgment $s \mapsto e$ relates every state s to a unique expression e, and the judgment $k \rhd\lhd e = e'$ relates every stack k and expression e to a unique expression e'.*

We are therefore justified in writing $k \rhd\lhd e$ for the unique e' such that $k \rhd\lhd e = e'$.

The following lemma is crucial. It states that unraveling preserves the transition relation.

Lemma 28.6. *If $e \longmapsto e'$, $k \rhd\lhd e = d$, $k \rhd\lhd e' = d'$, 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 $k \rhd\lhd e = d$ and $k \rhd\lhd e' = d'$. It follows from rules (28.12) that $k\,;\mathtt{ap}(-; e_2) \rhd\lhd e_1 = d$ and $k\,;\mathtt{ap}(-; e_2) \rhd\lhd e_1' = d'$. 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 $k \rhd\lhd e = d$ and $k \rhd\lhd e' = d'$; 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 $k' \rhd\lhd \mathtt{ap}(e; c_2) = d$ and $k' \rhd\lhd \mathtt{ap}(e'; c_2) = d'$. 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 $s = k \rhd \mathtt{s}(e)$ and $s' = k\,;\mathtt{s}(-) \rhd e$. Note that $k \rhd\lhd \mathtt{s}(e) = e'$ iff $k\,;\mathtt{s}(-) \rhd\lhd e = e'$, from which the result follows immediately.

Suppose that $s = k\,;\mathtt{ap}(\,\lambda\{\tau\}(\,x.e_1\,);-\,) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k\,;\mathtt{ap}(\,\lambda\{\tau\}(\,x.e_1\,);-\,) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\,\lambda\{\tau\}(\,x.e_1\,);e_2\,) = e'$. 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:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \tag{29.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}(e_1; x.e_2) : \tau} \tag{29.4b}$$

The type τ_{exn} is some fixed, but as yet unspecified, type of exception values. (The choice of τ_{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. We use a by-value interpretation to avoid the problem of *imprecise exceptions* that arises under a by-name interpretation.

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:

$$\overline{k \rhd \mathtt{raise}(e) \longmapsto k ; \mathtt{raise}(-) \rhd e} \tag{29.5a}$$

$$\overline{k ; \mathtt{raise}(-) \lhd e \longmapsto k \blacktriangleleft e} \tag{29.5b}$$

$$\overline{k \rhd \mathtt{try}(e_1; x.e_2) \longmapsto k ; \mathtt{try}(-; x.e_2) \rhd e_1} \tag{29.5c}$$

$$\overline{k ; \mathtt{try}(-; x.e_2) \lhd e \longmapsto k \lhd e} \tag{29.5d}$$

$$\frac{}{k\,;\mathtt{try}(\,-;x.e_2\,)\blacktriangleleft e \longmapsto k \triangleright [e/x]e_2} \tag{29.5e}$$

$$\frac{(f \neq \mathtt{try}(\,-;x.e_2\,))}{k\,;f \blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5f}$$

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:

$$\frac{}{\epsilon \triangleright e\ \mathsf{initial}} \tag{29.6a}$$

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{29.6b}$$

$$\frac{}{\epsilon \blacktriangleleft e\ \mathsf{final}} \tag{29.6c}$$

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 τ_{exn} of values associated to exceptions. The choice of τ_{exn} is important because it determines how the source of an exception is identified in a program. If τ_{exn} is the one-element type `unit`, then exceptions degenerate to failures, which are unable to identify their source. Thus τ_{exn} must have more than one value to be useful.

This fact suggests that τ_{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, τ_{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 τ_{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 τ_{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 $\tau + \tau_{\mathsf{exn}}$.

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 τ_{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 $v : \tau_{\mathsf{exn}}$, 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) states that the command to retrieve the contents of an assignable a returns a natural number. 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* $e \underset{\Sigma}{\longmapsto} e'$ *and* $\vdash_\Sigma e : \tau$, *then* $\vdash_\Sigma e' : \tau$.

2. *If* $\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$, *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* $e \underset{\Sigma}{\longmapsto} e'$.

2. *If* $\vdash_\Sigma m \ \mathit{ok}$ *and* $\mu : \Sigma$, *then either* $\mu \parallel m \ \mathit{final}_\Sigma$ *or* $\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$ *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 $e \underset{\Sigma}{\longmapsto} e'$ or $e \ \mathsf{val}_\Sigma$. In the former case rule (34.3i) applies. In the latter, we have by the second inductive hypothesis,

$$\mu \otimes a \hookrightarrow e \parallel m \ \mathsf{final}_{\Sigma,a} \quad \text{or} \quad \mu \otimes a \hookrightarrow e \parallel m \underset{\Sigma,a}{\longmapsto} \mu' \otimes a \hookrightarrow e' \parallel m'.$$

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 $\texttt{if}\,(\, m \,)\, m_1\, \texttt{else}\, m_2$ executes either m_1 or m_2 according to whether the result of executing m is zero or not:

$$\{x \leftarrow m\,;\texttt{do}\,(\,\texttt{ifz}\, x\, \{\texttt{z} \hookrightarrow \texttt{cmd}\, m_1 \mid \texttt{s}(\_) \hookrightarrow \texttt{cmd}\, m_2\}\,)\}.$$

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:

$$\texttt{do}\,(\,\texttt{fix}\, \mathit{loop} : \texttt{cmd}\, \texttt{is}\, \texttt{cmd}\,(\,\texttt{if}\,(\, m_1 \,)\, \{\texttt{ret}\, \texttt{z}\}\, \texttt{else}\, \{m_2\,;\texttt{do}\, \mathit{loop}\}\,)\,).$$

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 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

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \{b := \texttt{proc}\, (\, x : \texttt{nat}\,)\, \texttt{cmd}(\, a := x\,)\, ; \texttt{ret}\, \texttt{z}\}.$$

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* $e \underset{\Sigma}{\longmapsto} e'$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$*.*
2. *If* $\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$*, with* $\vdash_\Sigma m \mathrel{\div\!\!\sim} \tau$ *and* $\mu : \Sigma$*, then* $\vdash_\Sigma m' \mathrel{\div\!\!\sim} \tau$ *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* $e \underset{\Sigma}{\longmapsto} e'$*.*
2. *If* $\vdash_\Sigma m \mathrel{\div\!\!\sim} \tau$ *and* $\mu : \Sigma$*, then either* $\mu \parallel m$ final_Σ *or* $\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$ *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 light of 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

$$\texttt{fix}\, p\, \texttt{is}\, \lambda\, (\, p : \tau\,)\, \texttt{dcl}\, a := e\, \texttt{in}\, \texttt{cmd}(\, m\,)$$

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 (given by the transition and value judgments, $e \longmapsto e'$ and $e\ \mathsf{val}$), 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 $k \rhd_\Sigma m$, where $\vdash_\Sigma k \div \tau$ and $\vdash_\Sigma m \dot{\div} \tau$, and $k \lhd_\Sigma e$, 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 $k \parallel \mu \rhd_\Sigma m$, where $\mu : \Sigma$, $\vdash_\Sigma k \div \tau$, and $\vdash_\Sigma m \dot{\div} \tau$, and of the form $k \parallel \mu \lhd_\Sigma e$, where $\vdash_\Sigma k \div \tau$, $\vdash_\Sigma e : \tau$, and $e\ \mathsf{val}$.

Chapter 35

Assignable References

A *reference* to an assignable a is a value, written $\&a$, of *reference type* that determines 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 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 any given assignable, a, but what if we wish to write a generic procedure that works for any given assignable?

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:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)} \tag{35.1a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{ref}(\tau)}{\Gamma \vdash_{\Sigma} \mathtt{getref}(e) \div \tau} \tag{35.1b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \mathtt{ref}(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \tau}{\Gamma \vdash_{\Sigma} \mathtt{setref}(e_1; e_2) \div \tau} \tag{35.1c}$$

Rule (35.1a) specifies that the name of 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 refere 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.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,e\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,e'\,)\,\}} \tag{35.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{getref}(\,\mathtt{ref}[\,a\,]\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{get}[\,a\,]\,\}} \tag{35.4b}$$

$$\frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,e_1; e_2\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,e_1'; e_2\,)\,\}} \tag{35.4c}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mu \parallel \mathtt{setref}(\,\mathtt{ref}[\,a\,];e_2\,)\,\} \longmapsto \nu\,\Sigma\,\{\,\mu \parallel \mathtt{set}[\,a\,](\,e_2\,)\,\}} \tag{35.4d}$$

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. 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 $\mathtt{nat} \rightharpoonup \mathtt{nat}\ \mathtt{cmd}$:

$$\mathtt{proc}\,(\,x : \mathtt{nat}\,)\,\{\mathtt{if}\,(\,x\,)\,\mathtt{ret}\,(\,1\,)\,\mathtt{else}\,\{f \leftarrow @a\,;y \leftarrow f(\,x-1\,)\,;\mathtt{ret}\,(\,x * y\,)\}\}.$$

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 $\mu : \Sigma$. It is defined by the following rule:

$$\frac{\vdash_\Sigma m \mathrel{\dot{\sim}} \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{\,\mu \parallel m\,\}\ \mathsf{ok}} \tag{35.8}$$

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:

$$\frac{\forall a \sim \tau \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{35.9}$$

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 val_Σ. 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 val_Σ *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 the stronger form

> 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 $e \underset{\Sigma}{\longmapsto} e'$ 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

$$\overline{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{get}[a] \div \tau}$$

By assumption $\vdash_{\Sigma,a\sim\tau} \mu : \Sigma, a \sim \tau$, and hence there exists e $\mathrm{val}_{\Sigma,a\sim\tau}$ 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 `nat ⇀ 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 static 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 $e \xmapsto[\mathsf{seq}]{} e'$ defined by these rules:

$$\frac{e_1 \xmapsto[\mathsf{seq}]{} e_1'}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{seq}]{} \mathtt{par}(\, e_1'; e_2; x_1.x_2.e\,)} \tag{37.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \xmapsto[\mathsf{seq}]{} e_2'}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{seq}]{} \mathtt{par}(\, e_1; e_2'; x_1.x_2.e\,)} \tag{37.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{seq}]{} [e_1, e_2/x_1, x_2]e} \tag{37.2c}$$

The *parallel structural dynamics* of **PPCF** is given by a transition judgment of the form $e \xmapsto[\mathsf{par}]{} e'$, defined as follows:

$$\frac{e_1 \xmapsto[\mathsf{par}]{} e_1' \quad e_2 \xmapsto[\mathsf{par}]{} e_2'}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{par}]{} \mathtt{par}(\, e_1'; e_2'; x_1.x_2.e\,)} \tag{37.3a}$$

$$\frac{e_1 \xmapsto[\mathsf{par}]{} e_1' \quad e_2\ \mathsf{val}}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{par}]{} \mathtt{par}(\, e_1'; e_2; x_1.x_2.e\,)} \tag{37.3b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \xmapsto[\mathsf{par}]{} e_2'}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{par}]{} \mathtt{par}(\, e_1; e_2'; x_1.x_2.e\,)} \tag{37.3c}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \xmapsto[\mathsf{par}]{} [e_1, e_2/x_1, x_2]e} \tag{37.3d}$$

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

$$e \longmapsto_{\mathsf{par}}^{*} v \quad \text{iff} \quad e \Downarrow v \quad \text{iff} \quad e \longmapsto_{\mathsf{seq}}^{*} v$$

(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 `let`:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow v}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \Downarrow v} \tag{37.4}$$

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, $e \longmapsto_{\mathit{seq}}^{*} v$ if, and only if, $e \Downarrow v$.*

Proof. It suffices to show that if $e \longmapsto_{\mathsf{seq}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \longmapsto_{\mathsf{seq}}^{*} v_1$ and $e_2 \longmapsto_{\mathsf{seq}}^{*} v_2$ and $[v_1, v_2/x_1, x_2]e \longmapsto_{\mathsf{seq}}^{*} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \longmapsto_{\mathsf{seq}}^{*} v.$$

□

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, $e \longmapsto_{\mathit{par}}^{*} v$ if, and only if, $e \Downarrow v$.*

Proof. It suffices to show that if $e \longmapsto_{\mathsf{par}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \longmapsto_{\mathsf{par}}^{*} v_1$ and $e_2 \longmapsto_{\mathsf{par}}^{*} v_2$ and $[v_1, v_2/x_1, x_2]e \longmapsto_{\mathsf{par}}^{*} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \longmapsto_{\mathsf{par}}^{*} v.$$

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 $e_1 \longmapsto_{\mathsf{par}}^{*} v_1$ and $e_2 \longmapsto_{\mathsf{par}}^{*} v_2$. 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 $e_1 \longmapsto_{\mathsf{par}} e_1' \longmapsto_{\mathsf{par}}^{*} v_1$, then by induction we have that $\mathtt{par}\, x_1 = e_1' \,\mathtt{and}\, x_2 = v_2 \,\mathtt{in}\, e \longmapsto_{\mathsf{par}}^{*} v$, 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:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{37.7}$$

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 `let` construct.

Two simple facts about the cost dynamics are important to keep in mind. First, the cost assignment does not influence the outcome.

Lemma 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 $w = \mathit{wk}(c)$ and $d = \mathit{dp}(c)$. Conversely, if $e \xmapsto[\mathsf{seq}]{}{}^{w} v$, then there exists c such that $e \Downarrow^{c} v$ with $\mathit{wk}(c) = w$, and if $e \xmapsto[\mathsf{par}]{}{}^{d} v'$, then there exists c' such that $e \Downarrow^{c'} v'$ with $\mathit{dp}(c') = d$.*

Proof. The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being rule (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 $w_1 = \mathit{wk}(c_1)$, $w_2 = \mathit{wk}(c_2)$, and $w = \mathit{wk}(c)$. 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 $\mathit{wk}((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \xmapsto[\mathsf{par}]{}{}^{d_1} v_1$, $e_2 \xmapsto[\mathsf{par}]{}{}^{d_2} v_2$, and $[v_1, v_2/x_1, x_2]e \xmapsto[\mathsf{par}]{}{}^{d} v$, where $d_1 = \mathit{dp}(c_1)$, $d_2 = \mathit{dp}(c_2)$, and $d = \mathit{dp}(c)$. Assume, without loss of generality, that $d_1 \leq d_2$ (otherwise simply swap the roles of d_1 and d_2 in what follows). We may paste together derivations as follows:

$$\begin{aligned}
\mathtt{par}(\, e_1; e_2; x_1.x_2.e\,) &\xmapsto[\mathsf{par}]{}{}^{d_1} \mathtt{par}(\, v_1; e_2'; x_1.x_2.e\,) \\
&\xmapsto[\mathsf{par}]{}{}^{d_2 - d_1} \mathtt{par}(\, v_1; v_2; x_1.x_2.e\,) \\
&\xmapsto[\mathsf{par}]{} [v_1, v_2/x_1, x_2]e \\
&\xmapsto[\mathsf{par}]{}{}^{d} v.
\end{aligned}$$

Calculating $\mathit{dp}((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1, d_2) + 1 + d$ 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 $\mathit{wk}(c) = \mathit{wk}(c') + 1$, and if $e \xmapsto[\mathsf{par}]{} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $\mathit{dp}(c) = \mathit{dp}(c') + 1$.

Suppose that $e = \mathtt{par}(\, e_1; e_2; x_1.x_2.e_0\,)$ with e_1 val and e_2 val. Then $e \underset{\mathsf{seq}}{\longmapsto} e'$, 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, $e \underset{\mathsf{par}}{\longmapsto} e'$ 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 $e \underset{\mathsf{seq}}{\longmapsto} e'$, where $e' = \mathtt{par}(\, e_1'; e_2; x_1.x_2.e_0\,)$ and $e_1 \underset{\mathsf{seq}}{\longmapsto} e_1'$. 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 $e \underset{\mathsf{par}}{\longmapsto} e'$, where $e' = \mathtt{par}(\, e_1'; e_2'; x_1.x_2.e_0\,)$ and $e_1 \underset{\mathsf{par}}{\longmapsto} e_1'$, $e_2 \underset{\mathsf{par}}{\longmapsto} e_2'$, 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 $e \underset{seq}{\longmapsto}^{w} v$ and $e \underset{par}{\longmapsto}^{d} v'$, 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 `let` 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 $\texttt{seq}\{n\}(\,e_0,\ldots,e_{n-1}\,)$ evaluates to a sequence whose length is n and whose elements are given by the expressions $e_0,\ldots,e_{n-1}$. The operation $\texttt{len}(\,e\,)$ returns the number of elements in the sequence given by e. The operation $\texttt{sub}(\,e_1;e_2\,)$ retrieves the element of the sequence given by e_1 at the index given by e_2. The tabulate operation, $\texttt{tab}(\,x.e_1;e_2\,)$, yields the sequence of length given by e_2 whose ith element is given by $[i/x]e_1$. The operation $\texttt{map}(\,x.e_1;e_2\,)$ 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 $\texttt{cat}(\,e_1;e_2\,)$ concatenates two sequences of the same type.

The statics of these operations is given by the following typing rules:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \texttt{seq}\{n\}(\,e_0,\ldots,e_{n-1}\,) : \texttt{seq}(\,\tau\,)} \tag{37.8a}$$

$$\frac{\Gamma \vdash e : \texttt{seq}(\,\tau\,)}{\Gamma \vdash \texttt{len}(\,e\,) : \texttt{nat}} \tag{37.8b}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\,\tau\,) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{sub}(\,e_1;e_2\,) : \tau} \tag{37.8c}$$

$$\frac{\Gamma, x : \texttt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{tab}(\,x.e_1;e_2\,) : \texttt{seq}(\,\tau\,)} \tag{37.8d}$$

$$\frac{\Gamma \vdash e_2 : \texttt{seq}(\,\tau\,) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{map}(\,x.e_1;e_2\,) : \texttt{seq}(\,\tau'\,)} \tag{37.8e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\,\tau\,) \quad \Gamma \vdash e_2 : \texttt{seq}(\,\tau\,)}{\Gamma \vdash \texttt{cat}(\,e_1;e_2\,) : \texttt{seq}(\,\tau\,)} \tag{37.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{seq}\{n\}(\,e_0,\ldots,e_{n-1}\,) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \texttt{seq}\{n\}(\,v_0,\ldots,v_{n-1}\,)} \tag{37.9a}$$

$$\frac{e \Downarrow^{c} \texttt{seq}\{n\}(\,v_0,\ldots,v_{n-1}\,)}{\texttt{len}(\,e\,) \Downarrow^{c \oplus \mathbf{1}} \texttt{num}[\,n\,]} \tag{37.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}\{n\}(\,v_0,\ldots,v_{n-1}\,) \quad e_2 \Downarrow^{c_2} \texttt{num}[\,i\,] \quad (0 \leq i < n)}{\texttt{sub}(\,e_1;e_2\,) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{37.9c}$$

$$\frac{e_2 \Downarrow^{c} \texttt{num}[\,n\,] \quad [\texttt{num}[\,0\,]/x]e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad [\texttt{num}[\,n-1\,]/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(\,x.e_1;e_2\,) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}\{n\}(\,v_0,\ldots,v_{n-1}\,)} \tag{37.9d}$$

$$\frac{\begin{array}{c} e_2 \Downarrow^{c} \texttt{seq}\{n\}(\,v_0,\ldots,v_{n-1}\,) \\ [v_0/x]e_1 \Downarrow^{c_0} v'_0 \qquad \ldots \qquad [v_{n-1}/x]e_1 \Downarrow^{c_{n-1}} v'_{n-1} \end{array}}{\texttt{map}(\,x.e_1;e_2\,) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}\{n\}(\,v'_0,\ldots,v'_{n-1}\,)} \tag{37.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}\{m\}(v_0,\ldots,v_{m-1}) \quad e_2 \Downarrow^{c_2} \texttt{seq}\{n\}(v'_0,\ldots,v'_{n-1}) \quad p = m+n}{\texttt{cat}(e_1;e_2) \Downarrow^{c_1 \oplus c_2 \oplus \bigotimes_{i=0}^{m+n} \mathbf{1}} \texttt{seq}\{p\}(v_0,\ldots,v_{m-1},v'_0,\ldots,v'_{n-1})} \tag{37.9f}$$

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 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 $O(\max(w/p, d))$.*

[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.

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 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

[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 Chapter 28.

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 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.

The local transitions of the **P** machine corresponding to binary fork and join are as follows:

$$\frac{}{\left\{\begin{array}{c} \nu\, a\, \{\, a \hookrightarrow \mathtt{par}(\,e_1; e_2; x_1.x_2.e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a_1, a_2, a\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1; a_2\,](\,x_1; x_2.e\,)\,\} \end{array}\right\}} \tag{37.11a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1, a_2, a\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[\,a_1; a_2\,](\,x_1; x_2.e\,)\,\} \\ \underset{\mathsf{loc}}{\longmapsto} \\ \nu\, a\, \{\, a \hookrightarrow [e_1, e_2/x_1, x_2]e\,\} \end{array}\right\}} \tag{37.11b}$$

[3] Here and elsewhere typing information is omitted from Σ, because it is not relevant to the dynamics.

Rule (37.11a) creates two parallel tasks on which the executing task depends. The expression $\texttt{join}[\,a_1;a_2\,](\,x_1;x_2.e\,)$ 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.

$$\frac{\begin{array}{c}\nu\,\Sigma_1\,a_1\,\{\,\mu_1 \otimes a_1 \hookrightarrow s_1\,\} \mathrel{\underset{\text{loc}}{\longmapsto}} \nu\,\Sigma_1'\,a_1\,\{\,\mu_1' \otimes a_1 \hookrightarrow s_1'\,\} \\ \cdots \\ \nu\,\Sigma_n\,a_n\,\{\,\mu_n \otimes a_n \hookrightarrow s_n\,\} \mathrel{\underset{\text{loc}}{\longmapsto}} \nu\,\Sigma_n'\,a_n\,\{\,\mu_n' \otimes a_n \hookrightarrow s_n'\,\}\end{array}}{\left\{\begin{array}{c}\nu\,\Sigma_0\,\Sigma_1\,a_1\,\ldots\,\Sigma_n\,a_n\,\{\,\mu_0 \otimes \mu_1 \otimes a_1 \hookrightarrow s_1 \otimes \ldots \otimes \mu_n \otimes a_n \hookrightarrow s_n\,\} \\ \mathrel{\underset{\text{glo}}{\longmapsto}} \\ \nu\,\Sigma_0\,\Sigma_1'\,a_1\,\ldots\,\Sigma_n'\,a_n\,\{\,\mu_0 \otimes \mu_1' \otimes a_1 \hookrightarrow s_1' \otimes \ldots \otimes \mu_n' \otimes a_n \hookrightarrow s_n'\,\}\end{array}\right\}} \tag{37.12}$$

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 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 τ_{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:

[4] Thus the local transition given by Rule (37.11b) is never applicable; the dynamics of joins will be described shortly.

(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.

Chapter 38

Futures and Speculations

A *future* is a computation that is performed before it is value is needed. Like a suspension, a future represents a value that is to be determined later. Unlike a suspension, a future is always evaluated, regardless of whether its value is required. In a sequential setting futures are of little interest; a future of type τ is just an expression of type τ. In a parallel setting, however, futures are of interest because they provide a means of initiating a parallel computation whose result is not needed until later, by which time it will have been completed.

The prototypical example of the use of futures is to implementing *pipelining*, a method for overlapping the stages of a multistage computation to the fullest extent possible. Pipelining minimizes the latency caused by one stage waiting for a previous stage to complete by allowing the two stages to execute in parallel until an explicit dependency arises. Ideally, the computation of the result of an earlier stage is finished by the time a later stage needs it. At worst the later stage is delayed until the earlier stage completes, incurring what is known as a *pipeline stall*.

A *speculation* is a delayed computation whose result might be needed for the overall computation to finish. The dynamics for speculations executes suspended computations in parallel with the main thread of computation, without regard to whether the value of the speculation is needed by the main thread. If the value of the speculation is needed, then such a dynamics pays off, but if not, the effort to compute it is wasted.

Futures are *work efficient* in that the overall work done by a computation involving futures is no more than the work done by a sequential execution. Speculations, in contrast, are *work inefficient* in that speculative execution might be in vain—the overall computation may involve more steps than the work needed to compute the result. For this reason speculation is a risky strategy for exploiting parallelism. It can make use available resources, but perhaps only at the expense of doing more work than necessary!

38.1 Futures

The syntax of futures is given by the following grammar:

Typ	τ	$::=$	$\texttt{fut}(\tau)$	$\tau\,\texttt{fut}$	future
Exp	e	$::=$	$\texttt{fut}(e)$	$\texttt{fut}(e)$	future
			$\texttt{fsyn}(e)$	$\texttt{fsyn}(e)$	synchronize
			$\texttt{fcell}[a]$	$\texttt{fcell}[a]$	indirection

The type $\tau\,\texttt{fut}$ is the type of futures of type τ. Futures are introduced by the expression $\texttt{fut}(e)$, which schedules e for evaluation and returns a reference to it. Futures are eliminated by the expression $\texttt{fsyn}(e)$, which synchronizes with the future referred to by e, returning its value. Indirect references to future values are represented by $\texttt{fcell}[a]$, indicating a future value to be stored at a.

38.1.1 Statics

The statics of futures is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{fut}(e) : \texttt{fut}(\tau)} \tag{38.1a}$$

$$\frac{\Gamma \vdash e : \texttt{fut}(\tau)}{\Gamma \vdash \texttt{fsyn}(e) : \tau} \tag{38.1b}$$

These rules are unsurprising, because futures add no new capabilities to the language beyond providing an opportunity for parallel evaluation.

38.1.2 Sequential Dynamics

The sequential dynamics of futures is easily defined. Futures are evaluated eagerly; synchronization returns the value of the future.

$$\frac{e\ \mathsf{val}}{\texttt{fut}(e)\ \mathsf{val}} \tag{38.2a}$$

$$\frac{e \longmapsto e'}{\texttt{fut}(e) \longmapsto \texttt{fut}(e')} \tag{38.2b}$$

$$\frac{e \longmapsto e'}{\texttt{fsyn}(e) \longmapsto \texttt{fsyn}(e')} \tag{38.2c}$$

$$\frac{e\ \mathsf{val}}{\texttt{fsyn}(\texttt{fut}(e)) \longmapsto e} \tag{38.2d}$$

Under a sequential dynamics futures have little purpose: they introduce a pointless level of indirection.

38.2 Speculations

The syntax of (non-recursive) speculations is given by the following grammar:[1]

Typ	τ	$::=$	$\mathtt{spec}(\tau)$	$\tau\,\mathtt{spec}$	speculation
Exp	e	$::=$	$\mathtt{spec}(e)$	$\mathtt{spec}(e)$	speculate
			$\mathtt{ssyn}(e)$	$\mathtt{ssyn}(e)$	synchronize
			$\mathtt{scell}[a]$	$\mathtt{scell}[a]$	indirection

The type $\tau\,\mathtt{spec}$ is the type of speculations of type τ. The introduction form $\mathtt{spec}(e)$ creates a computation that can be speculatively evaluated, and the elimination form $\mathtt{ssyn}(e)$ synchronizes with a speculation. A reference to the result of a speculative computation stored at a is written $\mathtt{scell}[a]$.

38.2.1 Statics

The statics of speculations is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{spec}(e) : \mathtt{spec}(\tau)} \tag{38.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{spec}(\tau)}{\Gamma \vdash \mathtt{ssyn}(e) : \tau} \tag{38.3b}$$

Thus, the statics for speculations as given by rules (38.3) is equivalent to the statics for futures given by rules (38.1).

38.2.2 Sequential Dynamics

The definition of the sequential dynamics of speculations is like that of futures, except that speculations are values.

$$\frac{}{\mathtt{spec}(e)\ \mathsf{val}} \tag{38.4a}$$

$$\frac{e \longmapsto e'}{\mathtt{ssyn}(e) \longmapsto \mathtt{ssyn}(e')} \tag{38.4b}$$

$$\frac{}{\mathtt{ssyn}(\mathtt{spec}(e)) \longmapsto e} \tag{38.4c}$$

Under a sequential dynamics speculations are simply a re-formulation of suspensions.

[1] We confine ourselves to the non-recursive case to ease the comparison with futures.

38.3 Parallel Dynamics

Futures are only interesting insofar as they admit a parallel dynamics that allows the computation of the future to go ahead concurrently with some other computation. In this section we give a parallel dynamics of futures and speculation in which the creation, execution, and synchronization of tasks is made explicit. The parallel dynamics of futures and speculations is *identical*, except for the termination condition. Whereas futures require that all tasks are completed before termination, speculations may be abandoned before they are completed. For the sake of concision we will give the parallel dynamics of futures, remarking only where alterations are made for the parallel dynamics of speculations.

The parallel dynamics of futures relies on a modest extension to the language given in Section 38.1 to introduce *names* for tasks. Let Σ be a finite mapping assigning types to names. As mentioned earlier, the expression $\mathtt{fcell}[a]$ is a value referring to the outcome of task a. The statics of this expression is given by the following rule:[2]

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{fcell}[a] : \mathtt{fut}(\tau)} \tag{38.5}$$

Rules (38.1) carry over in the obvious way with Σ recording the types of the task names.

States of the parallel dynamics have the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is the *focus* of evaluation, and μ records the active parallel futures (or speculations). Formally, μ is a finite mapping assigning expressions to the task names declared in Σ. A state is well-formed according to the following rule:

$$\frac{\vdash_\Sigma e : \tau \quad (\forall a \in dom(\Sigma))\ \vdash_\Sigma \mu(a) : \Sigma(a)}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{ok}} \tag{38.6}$$

As discussed in Chapter 35 this rule admits self-referential and mutually referential futures. A more refined condition could as well be given that avoids circularities; we leave this as an exercise for the reader.

The parallel dynamics is divided into two phases, the *local* phase, which defines the basic steps of evaluation of an expression, and the *global* phase, which executes all possible local steps in parallel. The local dynamics of futures is defined by the following rules:[3]

$$\frac{}{\mathtt{fcell}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{38.7a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{fut}(e) \parallel \mu\,\} \longmapsto[\mathsf{loc}] \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{fcell}[a] \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{38.7b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto[\mathsf{loc}] \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{fsyn}(e) \parallel \mu\,\} \longmapsto[\mathsf{loc}] \nu\,\Sigma'\,\{\,\mathtt{fsyn}(e') \parallel \mu'\,\}} \tag{38.7c}$$

[2] A similar rule applies to $\mathtt{scell}[a]$ in the case of speculations.

[3] These rules are augmented by a reformulation of the dynamics of the other constructs of the language phrased in terms of the present notion of state.

$$\frac{e' \ \mathsf{val}_{\Sigma,a\sim\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a\sim\tau\,\{\,\mathtt{fsyn}(\,\mathtt{fcell}[\,a\,]\,) \parallel \mu \otimes a \hookrightarrow e'\,\} \\ \longmapsto_{\mathsf{loc}} \\ \nu\,\Sigma, a\sim\tau\,\{\,e' \parallel \mu \otimes a \hookrightarrow e'\,\} \end{array}\right\}} \tag{38.7d}$$

Rule (38.7b) activates a future named a executing the expression e and returns a reference to it. Rule (38.7d) synchronizes with a future whose value has been determined. Note that a local transition always has the form

$$\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto_{\mathsf{loc}} \nu\,\Sigma\,\Sigma'\,\{\,e' \parallel \mu \otimes \mu'\,\}$$

where Σ' is either empty or declares the type of a single symbol, and μ' is either empty or of the form $a \hookrightarrow e'$ for some expression e'.

A global step of the parallel dynamics consists of at most one local step for the focal expression and one local step for each of up to p futures, where $p > 0$ is a fixed parameter representing the number of processors.

$$\frac{\begin{array}{c} \mu = \mu_0 \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n \\ \mu'' = \mu_0 \otimes a_1 \hookrightarrow e_1' \otimes \ldots \otimes a_n \hookrightarrow e_n' \\ \nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto_{\mathsf{loc}}^{?} \nu\,\Sigma\,\Sigma'\,\{\,e' \parallel \mu \otimes \mu'\,\} \\ (\forall 1 \leq i \leq n \leq p) \quad \nu\,\Sigma\,\{\,e_i \parallel \mu\,\} \longmapsto_{\mathsf{loc}} \nu\,\Sigma\,\Sigma_i'\,\{\,e_i' \parallel \mu \otimes \mu_i'\,\} \end{array}}{\left\{\begin{array}{c} \nu\,\Sigma\,\{\,e \parallel \mu\,\} \\ \longmapsto_{\mathsf{glo}} \\ \nu\,\Sigma\,\Sigma'\,\Sigma_1'\,\ldots\,\Sigma_n'\,\{\,e' \parallel \mu'' \otimes \mu' \otimes \mu_1' \otimes \ldots \otimes \mu_n'\,\} \end{array}\right\}} \tag{38.8a}$$

Rule (38.8a) allows the focus expression to take either zero or one steps because it might be blocked awaiting the completion of evaluation of a parallel future (or synchronizing with a speculation). The futures allocated by the local steps of execution are consolidated in the result of the global step. We assume without loss of generality that the names of the new futures in each local step are pairwise disjoint so that the combination makes sense. In implementation terms satisfying this disjointness assumption means that the processors must synchronize their access to memory.

The initial state of a computation, for futures or speculations, is defined by the rule

$$\frac{}{\nu\,\emptyset\,\{\,e \parallel \emptyset\,\}\ \mathsf{initial}} \tag{38.9}$$

For futures a state is final only if the focus and all parallel futures have completed evaluation:

$$\frac{e\ \mathsf{val}_\Sigma \quad \mu\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{38.10a}$$

$$\frac{(\forall a \in dom(\Sigma))\ \mu(a)\ \mathsf{val}_\Sigma}{\mu\ \mathsf{val}_\Sigma} \tag{38.10b}$$

For speculations a state is final only if the focus is a value, regardless of whether any other speculations have completed:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{38.11}$$

All futures must terminate to ensure that the work performed in parallel matches that performed sequentially; no future is created whose value is not needed according to the sequential dynamics. In contrast, speculations can be abandoned when their values are not needed.

38.4 Pipelining With Futures

Pipelining is an interesting example of the use of parallel futures. Consider a situation in which a *producer* builds a list whose elements represent units of work, and a *consumer* traverses the work list and acts on each element of that list. The elements of the work list can be thought of as "instructions" to the consumer, which maps a function over that list to carry out those instructions. An obvious sequential implementation first builds the work list, then traverses it to perform the work indicated by the list. This strategy works well provided that the elements of the list can be produced quickly, but if each element needs a lot of computation, it would be preferable to overlap production of the next list element with execution of the previous unit of work, which can be programmed using futures.

Let `flist` be the recursive type $\texttt{rec}\,t\,\texttt{is}\,\texttt{unit} + (\,\texttt{nat} \times t\,\texttt{fut}\,)$, whose elements are `nil`, defined to be $\texttt{fold}(\,\texttt{l} \cdot \langle\rangle\,)$, and $\texttt{cons}(\,e_1,e_2\,)$, defined to be $\texttt{fold}(\,\texttt{r} \cdot \langle e_1, \texttt{fut}(\,e_2\,)\rangle\,)$. The producer is a recursive function that generates a value of type `flist`:

```
fix produce : (nat → nat opt) → nat → flist is
  λ f. λ i.
    case f(i) {
      null ↪ nil
    | just x ↪ cons(x, fut (produce f (i+1)))
    }
```

On each iteration the producer generates a parallel future to produce the tail. The future continues to execute after the producer returns so that its evaluation overlaps with subsequent computation.

The consumer folds an operation over the work list as follows:

```
fix consume : ((nat×nat)→nat) → nat → flist → nat is
  λ g. λ a. λ xs.
    case xs {
      nil ↪ a
    | cons (x, xs) ↪ consume g (g (x, a)) (fsyn xs)
    }
```

The consumer synchronizes with the tail of the work list just at the point where it makes a recursive call and hence needs the head element of the tail to continue processing. At this point the consumer will block, if necessary, to await computation of the tail before continuing the recursion.

Speculations arise naturally in lazy languages. But although they provide opportunities for parallelism, they are not, in general, *work efficient*: a speculation might be evaluated even though its value is never needed. An alternative is to combine suspensions (see Chapter 36) with futures so that the programmer may specify which suspensions ought to be evaluated in parallel. The notion of a *spark* is designed to achieve this. A spark evaluates a computation in parallel only for its effect on suspensions that are likely to be needed later. Specifically, we may define

$$\texttt{spark}(\,e_1; e_2\,) \triangleq \texttt{letfut}\ \_\ \texttt{be}\ \texttt{force}(\,e_1\,)\ \texttt{in}\ e_2,$$

where $e_1 : \tau_1\ \texttt{susp}$ and $e_2 : \tau_2$.[4] The expression $\texttt{force}(\,e_1\,)$ is evaluated in parallel, forcing the evaluation of e_1, in hopes that it will have completed evaluation before its value is needed by e_2.

As an example, consider the type `strm` of streams of numbers defined by the recursive type $\texttt{rec}\,t\ \texttt{is}\ (\,\texttt{unit} + (\,\texttt{nat} \times t\,)\,)\ \texttt{susp}$. Elements of this type are suspended computations that, when forced, either signals the end of stream, or produces a number and another such stream. Suppose that s is such a stream, and assume that we know, for reasons of its construction, that it is finite. We wish to compute $\texttt{map}(\,f\,)(\,s\,)$ for some function f, and to overlap this computation with the production of the stream elements. We will make use of a function `mapforce` that forces successive elements of the input stream, but yields no useful output. The computation

$$\texttt{letfut}\ \_\ \texttt{be}\ \texttt{map}(\,\texttt{force}\,)(\,s\,)\ \texttt{in}\ \texttt{map}(\,f\,)(\,s\,)$$

forces the elements of the stream in parallel with the computation of $\texttt{map}(\,f\,)(\,s\,)$, with the intention that all suspensions in s are forced before their values are needed by the main computation.

38.5 Notes

Futures were introduced by Friedman and Wise (1976), and featured in the MultiLisp language (Halstead, 1985) for parallel programming. A similar concept is proposed by Arvind et al. (1986) under the name "I-structures." The formulation given here is derived from Greiner and Blelloch (1999). Sparks were introduced by Trinder et al. (1998).

Exercises

38.1. Use futures to define $\texttt{letfut}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$, a parallel `let` in which e_2 is evaluated in parallel with e_1 up to the point that e_2 needs the value of x.

38.2. Use futures to encode binary nested parallelism by giving a definition of $\texttt{par}(\,e_1; e_2; x_1.x_2.e\,)$. *Hint*: Only one future is needed if you are careful.

[4] The expression evaluates e_1 simultaneously with e_2, up to the point that the value of x is needed. Its definition in terms of futures is the subject of Exercise 38.1.

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 $p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p'$ states that the process p evolves in one step to the process p' while undertaking action α.

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\,\Sigma'\,\{\,m' \otimes p\,\}}{\texttt{run}(\,m\,) \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\,\Sigma'\{\texttt{run}(\,m'\,) \otimes p\}} \tag{40.3a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{run}(\,\texttt{ret}\,e\,) \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}} \mathbf{1}} \tag{40.3b}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1'}{p_1 \otimes p_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \otimes p_2} \tag{40.3c}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \quad p_2 \overset{\bar{\alpha}}{\underset{\Sigma}{\longmapsto}} p_2'}{p_1 \otimes p_2 \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}} p_1' \otimes p_2'} \tag{40.3d}$$

$$\frac{p \overset{\alpha}{\underset{\Sigma, a \sim \tau}{\longrightarrow}} p' \quad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\,a{\sim}\tau.p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\,a{\sim}\tau.p'} \tag{40.3e}$$

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]

$$m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\,\Sigma'\,\{\,m' \otimes p'\,\}$$

[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**:

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\, e'} \tag{40.4a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\, \Sigma' \{\, m_1' \otimes p' \,\}}{\mathtt{bnd}\, x \leftarrow \mathtt{cmd}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\, \Sigma' \{ \mathtt{bnd}\, x \leftarrow \mathtt{cmd}\, m_1' \,;\, m_2 \otimes p' \}} \tag{40.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd}\, x \leftarrow \mathtt{cmd}\,(\,\mathtt{ret}\, e\,)\,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} [e/x]m_2} \tag{40.4c}$$

$$\frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\mathtt{bnd}\, x \leftarrow e_1 \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{bnd}\, x \leftarrow e_1' \,;\, m_2} \tag{40.4d}$$

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 $\mathtt{newch}\{\tau\}$ returns a reference to a fresh class carrying values of type τ.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\,\mathtt{unit}\,)}{\Gamma \vdash_\Sigma \mathtt{spawn}(\,e\,) \div \mathtt{unit}} \tag{40.5a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma \mathtt{emit}(\,e\,) \div \mathtt{unit}} \tag{40.5b}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{acc} \div \mathtt{clsfd}} \tag{40.5c}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{newch}\{\tau\} \div \mathtt{cls}(\,\tau\,)} \tag{40.5d}$$

Execution of these commands is defined as follows:

$$\frac{}{\mathtt{spawn}(\,\mathtt{cmd}(\,m\,)\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \mathtt{ret}\,\langle\rangle \otimes \mathtt{run}(\,m\,)} \tag{40.6a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{spawn}(\,e\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \mathtt{spawn}(\,e'\,)} \tag{40.6b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{emit}(\,e\,) \underset{\Sigma}{\overset{e!}{\Longrightarrow}} \mathtt{ret}\,\langle\rangle} \tag{40.6c}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{emit}(\,e\,) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \mathtt{emit}(\,e'\,)} \tag{40.6d}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{acc} \underset{\Sigma}{\overset{e?}{\Longrightarrow}} \mathtt{ret}\,e} \tag{40.6e}$$

$$\frac{}{\mathtt{newch}\{\tau\} \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \nu\, a \sim \tau\,\{\,\mathtt{ret}\,(\,\&\,a\,)\,\}} \tag{40.6f}$$

Rule (40.6c) specifies that $\mathtt{emit}(\,e\,)$ has the effect of emitting the message e. Correspondingly, rule (40.6e) specifies that $\mathtt{acc}$ 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 ithe nability 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:

$$\texttt{fix}\,\mathit{loop} : \tau\,\texttt{cmd}\,\texttt{is}\,\texttt{cmd}\,\{x \leftarrow \texttt{acc}\,;\texttt{match}\,x\,\texttt{as}\,a \cdot y \hookrightarrow \texttt{ret}\,y\,\texttt{ow} \hookrightarrow \texttt{emit}(x)\,;\texttt{do}\,\mathit{loop}\}$$

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 $\texttt{event}(\tau)$ of *events* consists of a finite choice of accepts, all of whose payloads are of type τ.

Typ	τ	::=	$\texttt{event}(\tau)$	$\tau\,\texttt{event}$	events
Exp	e	::=	$\texttt{rcv}[a]$	$?\,a$	selective read
			$\texttt{never}\{\tau\}$	$\texttt{never}$	null
			$\texttt{or}(e_1; e_2)$	$e_1\,\texttt{or}\,e_2$	choice
			$\texttt{wrap}(e_1; x.e_2)$	$e_1\,\texttt{as}\,x\,\texttt{in}\,e_2$	post-composition
Cmd	m	::=	$\texttt{sync}(e)$	$\texttt{sync}(e)$	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:

$$\frac{\Sigma \vdash a \sim \tau}{\mathtt{rcv}[a]\ \mathsf{val}_\Sigma} \tag{40.8a}$$

$$\frac{}{\mathtt{never}\{\tau\}\ \mathsf{val}_\Sigma} \tag{40.8b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{val}_\Sigma}{\mathtt{or}(e_1; e_2)\ \mathsf{val}_\Sigma} \tag{40.8c}$$

$$\frac{e_1\ \mathsf{val}_\Sigma}{\mathtt{wrap}(e_1; x.e_2)\ \mathsf{val}_\Sigma} \tag{40.8d}$$

$$\frac{e_1 \xmapsto[\Sigma]{} e_1'}{\mathtt{or}(e_1; e_2) \xmapsto[\Sigma]{} \mathtt{or}(e_1'; e_2)} \tag{40.8e}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2 \xmapsto[\Sigma]{} e_2'}{\mathtt{or}(e_1; e_2) \xmapsto[\Sigma]{} \mathtt{or}(e_1; e_2')} \tag{40.8f}$$

$$\frac{e_1 \xmapsto[\Sigma]{} e_1'}{\mathtt{wrap}(e_1; x.e_2) \xmapsto[\Sigma]{} \mathtt{wrap}(e_1'; x.e_2')} \tag{40.8g}$$

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.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{sync}(\,e\,) \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \texttt{sync}(\,e'\,)} \tag{40.10a}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \tau \quad \Sigma \vdash a \sim \tau}{\texttt{sync}(\,\texttt{rcv}[\,a\,]\,) \overset{a \cdot e\,?}{\underset{\Sigma}{\Longrightarrow}} \texttt{ret}(\,e\,)} \tag{40.10b}$$

$$\frac{\texttt{sync}(\,e_1\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m_1}{\texttt{sync}(\,\texttt{or}(\,e_1; e_2\,)\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m_1} \tag{40.10c}$$

$$\frac{\texttt{sync}(\,e_2\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m_2}{\texttt{sync}(\,\texttt{or}(\,e_1; e_2\,)\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m_2} \tag{40.10d}$$

$$\frac{\texttt{sync}(\,e_1\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} m_1}{\texttt{sync}(\,\texttt{wrap}(\,e_1; x.e_2\,)\,) \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \texttt{bnd}(\,\texttt{cmd}(\,m_1\,); x.\texttt{ret}(\,e_2\,)\,)} \tag{40.10e}$$

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:

$$\lambda\ (\ u : \tau_{srvr}\ \texttt{cls}\ )\ \texttt{fix}\, srvr : \tau \rightharpoonup \texttt{void}\ \texttt{cmd}\ \texttt{is}\ \lambda\ (\ x : \tau\ )\ \texttt{cmd}\ \{y \leftarrow \texttt{sync}(\ ??\, u\ )\,;\, e_{(40.12)}\}. \tag{40.11}$$

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

Chapter 42

Modularity and Linking

Modularity is the most important technique for controlling the complexity of programs. Programs are decomposed into separate *components* with precisely specified, and tightly controlled, interactions. The pathways for interaction among components determine dependencies that constrain the process by which the components are integrated, or *linked*, to form a complete system. Different systems may use the same components, and a single system may use multiple instances of a single component. Sharing of components amortizes the cost of their development across systems, and helps limit errors by limiting coding effort.

Modularity is not limited to programming languages. In mathematics the proof of a theorem is decomposed into a collection of definitions and lemmas. References among the lemmas determine a dependency relation that constrains their integration to form a complete proof of the main theorem. Of course, one person's theorem is another person's lemma; there is no intrinsic limit on the depth and complexity of the hierarchies of results in mathematics. Mathematical structures are themselves composed of separable parts, for example, a ring comprises a group and a monoid structure on the same underlying set.

Modularity arises from the structural properties of the hypothetical and general judgments. Dependencies among components are expressed by free variables whose typing assumptions state the presumed properties of the component. Linking amounts to substitution to discharge the hypothesis.

42.1 Simple Units and Linking

Decomposing a program into units amounts to exploiting the transitivity of the hypothetical judgment (see Chapter 3). The decomposition may be described as an interaction between two parties, the *client* and the *implementor*, mediated by an agreed-upon contract, an *interface*. The client *assumes* that the implementor upholds the contract, and the implementor *guarantees* that the contract will be upheld. The assumption made by the client amounts to a declaration of its dependence on the implementor discharged by *linking* the two parties accordng to their agreed-upon contract.

The interface that mediates the interaction between a client and an implementor is a *type*. Linking is the implementation of the composite structural rules of substitution and transitivity:

$$\frac{\Gamma \vdash e_{impl} : \tau_{intf} \quad \Gamma, x : \tau_{intf} \vdash e_{client} : \tau_{client}}{\Gamma \vdash [e_{impl}/x]e_{client} : \tau_{client}} \tag{42.1}$$

The type τ_{intf} is the interface type. It defines the operations provided by the implementor e_{impl} and relied upon by the client e_{client}. The free variable x expresses the dependency of e_{client} on e_{impl}. That is, the client accesses the implementation by using the variable x.

The interface type τ_{intf} is the contract between the client and the implementor. It determines the properties of the implementation on which the client may depend and, at the same time, determines the obligations that the implementor must fulfill. The simplest form of interface type is a finite product type of the form $\langle f_1 \hookrightarrow \tau_1, \ldots, f_n \hookrightarrow \tau_n \rangle$, specifying a component with components f_i of type τ_i. Such a type is an *application program interface*, or *API*, because it determines the operations that the client (application) may expect from the implementor. A more advanced form of interface is one that defines an abstract type of the form $\exists(\, t.\langle f_1 \hookrightarrow \tau_1, \ldots, f_n \hookrightarrow \tau_n \rangle\,)$, which defines an abstract type t representing the internal state of an "abstract machine" whose "instruction set" consists of the operations $f_1, \ldots, f_n$ whose types may involve t. Being abstract, the type t is not revealed to the client, but is known only to the implementor.[1]

Conceptually, linking is just substitution, but practically this can be implemented in many ways. One method is *separate compilation*. The expressions e_{client} and e_{impl}, the *source modules*, are translated (compiled) into another, lower-level, language, resulting in *object modules*. Linking consists of performing the required substitution at the level of the object language in such a way that the result corresponds to translating $[e_{impl}/x]e_{client}$. Another method, *separate checking*, shifts the requirement for translation to the linker. The client and implementor units are checked for type correctness with respect to the interface, but are not translated into lower-level form. Linking then consists of translating the composite program as a whole, often resulting in a more efficient outcome than would be possible when compiling separately.

The foregoing are all forms of *static linking* because the program is composed before it is executed. Another method, *dynamic linking*, defers program composition until run-time, so that a component is loaded only if it is actually required during execution. This might seem to involve executing programs with free variables, but it does not. Each client implemented by a *stub* that forwards accesses to a stored implementation (typically, in an ambient file system). The difficulty with dynamic linking is that it refers to components by name (say, a path in a file system), and the binding of that name may change at any time, wreaking havoc on program behavior.

42.2 Initialization and Effects

Linking resolves the dependencies among the components of a program by substitution. This view is valid so long as the components are given by pure expressions, those that evaluate to a value without inducing any effects. For in such cases there is no problem with the replication, or

[1] See Chapters 17 and 48 for a discussion of type abstraction.

complete omission, of a component arising from repeated, or absent, uses of a variable representing it. But what if the expression defining the implementation of a component has an effect when evaluated? At a minimum replication of the component implies replication of its effects. Worse, effects introduce *implicit dependencies* among components that are not apparent from their types. For example, if each of two components mutates a shared assignable, the order in which they are linked with a client program affects the behavior of the whole.

This may raise doubts about the treatment of linking as substitution, but on closer inspection it becomes clear that implicit dependencies are naturally expressed by the modal distinction between expressions and commands introduced in Chapter 34. Specifically, a component that may have an effect when executed does not have type τ_{intf} of implementations of the interface type, but rather the type τ_{intf} `cmd` of encapsulated commands that, when executed, have effects and yield implementations. Being encapsulated, a value of this type is itself free of effects, but it may have effects when evaluated.

The distinction between the types τ_{intf} and τ_{intf} `cmd` is mediated by the sequencing command introduced in Chapter 34. For the sake of generality, let us assume that the client is itself an encapsulated command of type τ_{client} `cmd`, so that it may itself have effects when executed, and may serve as a component of a yet larger system. Assuming that the client refers to the encapsulated implementation by the variable x, the command

$$\texttt{bnd}\, x \leftarrow x\, \texttt{;}\, \texttt{do}\, e_{client}$$

first determines the implementation of the interface by running the encapsulated command x then running the client code with the result bound to x. The implicit dependencies of the client on the implementor are made explicit by the sequencing command, which ensures that the implementor's effects occur prior to those of the client, precisely because the client depends on the implementor for its execution.

More generally, to manage such interactions in a large program it is common to isolate an *initialization procedure* whose role is to stage the effects engendered by the various components according to some policy or convention. Rather than attempt to survey all possible policies, let us just note that the upshot of such conventions is that the initialization procedure is a command of the form

$$\{x_1 \leftarrow x_1\, \texttt{;} \ldots x_n \leftarrow x_n\, \texttt{;}\, m_{main}\},$$

where $x_1, \ldots, x_n$ represent the components of the system and m_{main} is the main (startup) routine. After linking the initialization procedure has the form

$$\{x_1 \leftarrow e_1\, \texttt{;} \ldots x_n \leftarrow e_n\, \texttt{;}\, m_{main}\},$$

where $e_1, \ldots, e_n$ are the encapsulated implementations of the linked components. When the initialization procedure is executed, it results in the substitution

$$[v_1, \ldots, v_n / x_1, \ldots, x_n] m_{main},$$

where the expressions $v_1, \ldots, v_n$ represent the values resulting from executing $e_1, \ldots, e_n$, respectively, and the implicit effects have occurred in the order specified by the initializer.

42.3 Notes

The relationship between the structural properties of entailment and the practical problem of separate development was implicit in much early work on programming languages, but became explicit once the correspondence between propositions and types was developed. There are many indications of this correspondence in sources such as *Proofs and Types* (Girard, 1989) and *Intuitionistic Type Theory* (Martin-Löf, 1984), but it was first made explicit by Cardelli (1997).

Chapter 44

Type Abstractions and Type Classes

An interface is a contract that specifies the rights of a client and the responsibilities of an implementor. Being a specification of behavior, an interface is a type. In principle any type may serve as an interface, but in practice it is usual to structure code into *modules* consisting of separable and reusable components. An interface specifies the behavior of a module expected by a client and imposed on the implementor. It is the fulcrum balancing the tension between separation and integration. As a rule, a module ought to have a well-defined behavior that can be understood separately, but it is equally important that it be easy to combine modules to form an integrated whole.

A fundamental question is, what is the type of a module? That is, what form should an interface take? One long-standing idea is that an interface is a labeled tuple of functions and procedures with specified types. The types of the fields of the tuple are often called *function headers*, because they summarize the call and return types of each function. Using interfaces of this form is called *procedural abstraction*, because it limits the dependencies between modules to a specified set of procedures. We may think of the fields of the tuple as being the instruction set of a virtual machine. The client makes use of these instructions in its code, and the implementor agrees to provide their implementations.

The problem with procedural abstraction is that it does not provide as much insulation as one might like. For example, a module that implements a dictionary must expose in the types of its operations the exact representation of the tree as, say, a recursive type (or, in more rudimentary languages, a pointer to a structure that itself may contain such pointers). Yet the client ought not depend on this representation: the purpose of abstraction is to get rid of pointers. The solution, as discussed in Chapter 17, is to extend the abstract machine metaphor to allow the internal state of the machine to be hidden from the client. In the case of a dictionary the representation of the dictionary as a binary search tree is hidden by existential quantification. This concept is called *type abstraction*, because the type of the underlying data (state of the abstract machine) is hidden.

Type abstraction is a powerful method for limiting the dependencies among the modules that constitute a program. It is very useful in many circumstances, but is not universally applicable. It is often useful to expose, rather than obscure, type information across a module boundary. A typical example is the implementation of a dictionary, which is a mapping from keys to values. To

use, say, a binary search tree to implement a dictionary, we require that the key type admit a total ordering with which keys can be compared. The dictionary abstraction does not depend on the exact type of the keys, but only requires that the key type be constrained to provide a comparison operation. A *type class* is a specification of such a requirement. The class of comparable types, for example, specifies a type t together with an operation `leq` of type $(t \times t) \to \texttt{bool}$ with which to compare them. Superficially, such a specification looks like a type abstraction, because it specifies a type and one or more operations on it, but with the important difference that the type t is not hidden from the client. For if it were, the client would only be able to compare keys using `leq`, but would have no means of obtaining keys to compare. A type class, in contrast to a type abstraction, is not intended to be an exhaustive specification of the operations on a type, but as a constraint on its behavior expressed by demanding that certain operations, such as comparison, be available, without limiting the other operations that might be defined on it.

Type abstractions and type classes are the extremal cases of a general concept of module type that we shall discuss in detail in this chapter. The crucial idea is the *controlled revelation* of type information across module boundaries. Type abstractions are opaque; type classes are transparent. These are both instances of *translucency*, which arises from combining existential types (Chapter 17), subtyping (Chapter 24), and singleton kinds and subkinding (Chapter 43). Unlike in Chapter 17, however, we will distinguish the types of modules, which are called *signatures*, from the types of ordinary values. The distinction is not essential, but it will be helpful to keep the two concepts separate at the outset, deferring discussion of how to ease the segregation once the basic concepts are in place.

44.1 Type Abstraction

Type abstraction is captured by a form of existential type quantification similar to that described in Chapter 17. For example, a dictionary with keys of type τ_{key} and values of type τ_{val} implements the signature σ_{dict} defined by $[\![t :: \mathtt{T} \,;\, \tau_{\mathsf{dict}}]\!]$, where τ_{dict} is the labeled tuple type

$$\langle \mathtt{emp} \hookrightarrow t \,,\, \mathtt{ins} \hookrightarrow \tau_{\mathsf{key}} \times \tau_{\mathsf{val}} \times t \to t \,,\, \mathtt{fnd} \hookrightarrow \tau_{\mathsf{key}} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt} \rangle.$$

The type variable t occurring in τ_{dict} and bound by σ_{dict} is the abstract type of dictionaries on which are defined three operations `emp`, `ins`, and `fnd` with the specified types. The type τ_{val} is immaterial to the discussion, because the dictionary operations impose no restrictions on the values that are associated to keys. However, it is important that the type τ_{key} be some fixed type, such as `str`, equipped with a suite of operations such as comparison. Observe that the signature σ_{dict} merely specifies that a dictionary is a value of some type that admits the operations `emp`, `ins`, and `fnd` with the types given by τ_{dict}.

An implementation of the signature σ_{dict} is a *structure* M_{dict} of the form $[\![\rho_{\mathsf{dict}} \,;\, e_{\mathsf{dict}}]\!]$, where ρ_{dict} is some concrete representation of dictionaries, and e_{dict} is a labeled tuple of type $[\rho_{\mathsf{dict}} / t]\tau_{\mathsf{dict}}$ of the general form

$$\langle \mathtt{emp} \hookrightarrow \ldots \,,\, \mathtt{ins} \hookrightarrow \ldots \,,\, \mathtt{fnd} \hookrightarrow \ldots \rangle.$$

The elided parts implement the dictionary operations in terms of the chosen representation type ρ_{dict} making use of the comparison operation that we assume is available of values of type τ_{key}. For

example, the type ρ_{dict} might be a recursive type defining a balanced binary search tree, such as a red-black tree. The dictionary operations work on the underlying representation of the dictionary as such a tree, just as would a package of existential type (see Chapter 17). The supposition about τ_{key} is temporary, and is lifted in Section 44.2.

To ensure that the representation of the dictionary is hidden from a client, the structure M_{dict} is *sealed* with the signature σ_{dict} to obtain the module

$$M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}}.$$

The effect of sealing is to ensure that the *only* information about M_{dict} that propagates to the client is given by σ_{dict}. In particular, because σ_{dict} only specifies that the type t have kind $\mathtt{T}$, no information about the choice of t as ρ_{dict} in M_{dict} is made available to the client.

A module is a *two-phase* object consisting of a *static part* and a *dynamic part*. The static part is a constructor of a specified kind; the dynamic part is a value of a specified type. There are two elimination forms that extract the static and dynamic parts of a module. These are, respectively, a form of constructor and a form of expression. More precisely, the constructor $M \cdot \mathtt{s}$ stands for the static part of M, and the expression $M \cdot \mathtt{d}$ stands for its dynamic part. According to the inversion principle, if a module M has introduction form, then $M \cdot \mathtt{s}$ should be equivalent to the static part of M. So, for example, $M_{\mathsf{dict}} \cdot \mathtt{s}$ should be equivalent to ρ_{dict}.

But consider the static part of a sealed module, which has the form $(\, M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}} \,) \cdot \mathtt{s}$. Because sealing hides the representation of an abstract type, this constructor should not be equivalent to ρ_{dict}. If M'_{dict} is another implementation of σ_{dict}, should $(\, M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}} \,) \cdot \mathtt{s}$ be equivalent to $(\, M'_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}} \,) \cdot \mathtt{s}$? To ensure reflexivity of type equivalence this equation should hold when M and M' are equivalent modules. But this violates representation independence for abstract types by making equivalence of abstract types sensitive to their implementation.

It would seem, then, that there is a contradiction between two very fundamental concepts, type equivalence and representation independence. The way out of this conundrum is to *disallow* reference to the static part of a sealed module: the type expression $M \rceil \sigma \cdot \mathtt{s}$ is deemed ill-formed. More generally, the formation of $M \cdot \mathtt{s}$ is disallowed unless M is a *module value*, whose static part is always manifest. An explicit structure is a module value, as is any module variable (provided that module variables are bound by-value).

One effect of this restriction is that sealed modules must be bound to a variable before they are used. Because module variables are bound by-value, doing so has the effect of imposing abstraction at the binding site. In fact, we may think of sealing as a kind of computational effect that "occurs" at the binding site, much as the bind operation in Algol discussed in Chapter 34 engenders the effects induced by an encapsulated command. As a consequence two bindings of the same sealed module result in two abstract types. The type system willfully ignores the identity of the two occurrences of the same module in order to ensure that their representations can be changed independently of one another without disrupting the behavior of any client code (because the client cannot rely on their identity, it must regard them as different).

44.2 Type Classes

Type abstraction is an essential tool for limiting dependencies among modules in a program. The signature of a type abstraction determines all that is known about a module by a client; no other uses of the values of an abstract type are permissible. A complementary tool is to use a signature to partially specify the capabilities of a module. Such a signature is a *type class*, or a *view*; an *instance* of the type class is an implementation of it. Because the signature of a type class only constrains the minimum capabilities of an unknown module, there must be some other means of working with values of that type. The way to achieve this is to expose, rather than hide, the identity of the static part of a module. In this sense type classes are the "opposite" of type abstractions, but we shall see below that there is a smooth progression between them, mediated by a subsignature judgment.

Let us consider the implementation of dictionaries as a client of the implementation of its keys. To implement a dictionary using a binary search tree the only requirement is that keys come equipped with a total ordering given by a comparison operation. This requirement can be expressed by a signature σ_{ord} given by

$$[\![t :: \mathtt{T} \,;\, \langle \mathtt{leq} \hookrightarrow (\, t \times t \,) \to \mathtt{bool} \rangle]\!] \,.$$

Because a given type can be ordered in many ways, it is essential that the ordering be packaged with the type to determine a type of keys.

The implementation of dictionaries as binary search trees takes the form

$$X : \sigma_{\mathsf{ord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}}.$$

Here σ^X_{dict} is the signature $[\![t :: \mathtt{T} \,;\, \tau^X_{\mathsf{dict}}]\!]$, whose body, τ^X_{dict}, is the tuple type

$$\langle \mathtt{emp} \hookrightarrow t \,,\, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t \,,\, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt} \rangle,$$

and M^X_{bstdict} is a structure (not given explicitly here) that implements the dictionary operations using binary search trees.[1] Within M^X_{bstdict}, the static and dynamic parts of the module X are accessed by writing $X \cdot \mathtt{s}$ and $X \cdot \mathtt{d}$, respectively. In particular, the comparison operation on keys is accessed by the projection $X \cdot \mathtt{d} \cdot \mathtt{leq}$.

The declared signature of the module variable X expresses a constraint on the capabilities of a key type by specifying an upper bound on its signature in the subsignature ordering. So any module bound to X must provide a type of keys and a comparison operation on that type, but nothing else is assumed of it. Because this is all we know about the unknown module X the dictionary implementation is constrained to rely only on these specified capabilities, and no others. When linking with a module defining X, the implementation need not be sealed with this signature, but must instead have a signature that is no larger than it in the subsignature relation. Indeed, the signature σ_{ord} is useless for sealing, as is easily seen by example. Suppose that $M_{\mathsf{natord}} : \sigma_{\mathsf{ord}}$ is an instance of the class of ordered types under the usual ordering. If we seal M_{natord} with σ_{ord} by writing

$$M_{\mathsf{natord}} \upharpoonright \sigma_{\mathsf{ord}},$$

[1] Here and elsewhere in this chapter and the next, the superscript X serves as a reminder that the module variable X may occur free in the annotated module or signature.

the resulting module is *useless*, because we would then have no way to create values of the key type.

We see, then, that a type class is a description (or view) of a pre-existing type, and is not a means of introducing a new type. Rather than obscure the identity of the static part of M_{natord}, we wish to propagate its identity as `nat` while specifying a comparison with which to order them. Type identity propagation is achieved using singleton kinds (as described in Chapter 43). Specifically, the most precise, or *principal*, signature of a structure is the one that exposes its static part using a singleton kind. In the case of the module M_{natord}, the principal signature is the signature σ_{natord} given by

$$[\![t :: \mathtt{S}(\,\mathtt{nat}\,)\,;\mathtt{leq} \hookrightarrow (\,t \times t\,) \to \mathtt{bool}]\!]\,,$$

which, by the rules of equivalence (defined formally in Section 44.3), is equivalent to the signature

$$[\![\_ :: \mathtt{S}(\,\mathtt{nat}\,)\,;\mathtt{leq} \hookrightarrow (\,\mathtt{nat} \times \mathtt{nat}\,) \to \mathtt{bool}]\!]\,.$$

The derivation of such an equivalence is called *equivalence propagation*, because it propagates the identity of the type t into its scope.

The dictionary implementation M^X_{bstdict} expects a module X with signature σ_{ord}, but the module M_{natord} provides the signature σ_{natord}. Applying the rules of subkinding given in Chapter 43, together with the covariance principle for signatures, we obtain the subsignature relationship

$$\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}.$$

By the subsumption principle, a module of signature σ_{natord} may be provided when a module of signature σ_{ord} is required. Therefore M_{natord} may be linked to X in M^X_{bstdict}.

Combining subtyping with sealing provides a smooth gradation between type classes and type abstractions. The principal signature for M^X_{bstdict} is the signature ρ^X_{dict} given by

$$\left[\!\!\left[t :: \mathtt{S}(\,\tau^X_{\mathsf{bst}}\,)\,;\langle \mathtt{emp} \hookrightarrow t\,,\,\mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t\,,\,\mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\,\mathtt{opt} \rangle \right]\!\!\right],$$

where τ^X_{bst} is the type of binary search trees with keys given by the module X of signature σ_{ord}. This signature is a subsignature of σ^X_{dict} given earlier, so that the sealed module

$$M^X_{\mathsf{bstdict}} \upharpoonright \sigma^X_{\mathsf{dict}}$$

is well-formed, and has type σ^X_{dict}, which hides the representation type of the dictionary abstraction.

After linking X to M_{natord}, the signature of the dictionary is specialized by propagating the identity of the static part of M_{natord} using the subsignature judgment. As remarked earlier, the dictionary implementation satisfies the typing

$$X : \sigma_{\mathsf{ord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}}.$$

But because $\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}$, we have, by contravariance, that

$$X : \sigma_{\mathsf{natord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}}.$$

is also a valid typing judgment. If $X : \sigma_{\mathsf{natord}}$, then $X \cdot \mathtt{s}$ is equivalent to `nat`, because it has kind `S( nat )`, so that the typing

$$X : \sigma_{\mathsf{natord}} \vdash M^{X}_{\mathsf{bstdict}} : \sigma_{\mathsf{natdict}}$$

is also valid. The closed signature $\sigma_{\mathsf{natdict}}$ is given explicitly by

$$[\![t :: \mathtt{T} \,;\, \langle \mathtt{emp} \hookrightarrow t \,,\, \mathtt{ins} \hookrightarrow \mathtt{nat} \times \tau_{\mathsf{val}} \times t \to t \,,\, \mathtt{fnd} \hookrightarrow \mathtt{nat} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt} \rangle]\!] .$$

The representation of dictionaries is hidden, but the representation of keys as natural numbers is not. The dependency on X has been eliminated by replacing all occurrences of $X \cdot \mathtt{s}$ within $\sigma^{X}_{\mathsf{dict}}$ by the type `nat`. Having derived this typing we may link X with M_{natord} as described in Chapter 42 to obtain a composite module, M_{natdict}, of signature $\sigma_{\mathsf{natdict}}$, in which keys are natural numbers ordered as specified by M_{natord}.

It is convenient to exploit subtyping for labeled tuple types to avoid creating an *ad hoc* module specifying the standard ordering on the natural numbers. Instead we can extract the required module directly from the implementation of the abstract type of numbers using subsumption. As an illustration, let X_{nat} be a module variable of signature σ_{nat}, which has the form

$$[\![t :: \mathtt{T} \,;\, \langle \mathtt{zero} \hookrightarrow t \,,\, \mathtt{succ} \hookrightarrow t \to t \,,\, \mathtt{leq} \hookrightarrow (\, t \times t \,) \to \mathtt{bool} \,, \ldots \rangle]\!]$$

The fields of the tuple provide all and only the operations that are available on the abstract type of natural numbers. Among them is the comparison operation `leq`, which is required by the dictionary module. Applying the subtyping rules for labeled tuples given in Chapter 24, together with the covariance of signatures, we obtain the subsignature relationship

$$\sigma_{\mathsf{nat}} <: \sigma_{\mathsf{ord}},$$

so that by subsumption the variable X_{nat} may be linked to the variable X postulated by the dictionary implementation. Subtyping takes care of extracting the required `leq` field from the abstract type of natural numbers, demonstrating that the natural numbers are an instance of the class of ordered types. Of course, this approach only works if we wish to order the natural numbers in the natural way provided by the abstract type. If, instead, we wish to use another ordering, then we must construct instances of σ_{ord} "by hand" to define the appropriate ordering.

44.3 A Module Language

The module language **Mod** formalizes theideas outlined in the preceding section. The syntax is divided into five levels: expressions classified by types, constructors classified by kinds, and modules classified by signatures. The expression and type level consists of various language mechanisms described earlier in this book, including at least product, sum, and partial function types. The constructor and kind level is as described in Chapters 18 and 43, with singleton and dependent

kinds. The following grammar summarizes the syntax of modules.

Sig	σ	::=	$\mathtt{sig}\{\kappa\}(\,t.\tau\,)$	$[\![t :: \kappa\,;\tau]\!]$	signature
Mod	M	::=	X	X	variable
			$\mathtt{str}(\,c;e\,)$	$[\![c\,;e]\!]$	structure
			$\mathtt{seal}\{\sigma\}(\,M\,)$	$M \mathbin{\upharpoonright} \sigma$	seal
			$\mathtt{let}\{\sigma\}(\,M_1; X.M_2\,)$	$(\,\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2\,) : \sigma$	definition
Con	c	::=	$\mathtt{s}(\,M\,)$	$M \cdot \mathtt{s}$	static part
Exp	e	::=	$\mathtt{d}(\,M\,)$	$M \cdot \mathtt{d}$	dynamic part

The statics of **Mod** consists of the following forms of judgment:

$\Gamma \vdash \sigma\ \mathsf{sig}$	well-formed signature
$\Gamma \vdash \sigma_1 \equiv \sigma_2$	equivalent signatures
$\Gamma \vdash \sigma_1 <: \sigma_2$	subsignature
$\Gamma \vdash M : \sigma$	well-formed module
$\Gamma \vdash M\ \mathsf{val}$	module value
$\Gamma \vdash e\ \mathsf{val}$	expression value

Rather than segregate hypotheses into zones, we instead admit the following three forms of hypothesis groups:

$X : \sigma, X\ \mathsf{val}$	module value variable
$u :: \kappa$	constructor variable
$x : \tau, x\ \mathsf{val}$	expression value variable

It is important that module and expression variables are always regarded as values to ensure that type abstraction is properly enforced. Correspondingly, each module and expression variable appears in Γ paired with the hypothesis that it is a value. As a notational convenience we will not explicitly state the value hypotheses associated with module and expression variables, under the convention that all such variables implicitly come paired with such an assumption.

The following rules define the formation, equivalence, and subsignature judgments.

$$\frac{\Gamma \vdash \kappa\ \mathsf{kind} \quad \Gamma, u :: \kappa \vdash \tau\ \mathsf{type}}{\Gamma \vdash [\![u :: \kappa\,;\tau]\!]\ \mathsf{sig}} \tag{44.1a}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![u :: \kappa_1\,;\tau_1]\!] \equiv [\![u :: \kappa_2\,;\tau_2]\!]} \tag{44.1b}$$

$$\frac{\Gamma \vdash \kappa_1 <:: \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Gamma \vdash [\![u :: \kappa_1\,;\tau_1]\!] <: [\![u :: \kappa_2\,;\tau_2]\!]} \tag{44.1c}$$

Most importantly, signatures are covariant in both the kind and type positions: subkinding and subtyping are preserved by the formation of a signature. It follows from rule (44.1b) that

$$[\![u :: \mathtt{S}(\,c\,)\,;\tau]\!] \equiv [\![\_ :: \mathtt{S}(\,c\,)\,;[c/u]\tau]\!]$$

and, further, it follows from rule (44.1c) that

$$[\![\_ :: \mathtt{S}(\,c\,)\,;[c/u]\tau]\!] <: [\![\_ :: \mathtt{T}\,;[c/u]\tau]\!]$$

and so

$$[\![u :: \mathtt{S}(\,c\,)\,;\tau]\!] <: [\![\_ :: \mathtt{T}\,;[c/u]\tau]\!].$$

It is also the case that

$$[\![u :: \mathtt{S}(\,c\,)\,;\tau]\!] <: [\![u :: \mathtt{T}\,;\tau]\!].$$

But the two supersignatures of $[\![u :: \mathtt{S}(\,c\,)\,;\tau]\!]$ are *incomparable* with respect to the subsignature judgment.

The statics of expressions of **Mod** is given by the following rules:

$$\frac{}{\Gamma, X : \sigma \vdash X : \sigma} \tag{44.2a}$$

$$\frac{\Gamma \vdash c :: \kappa \quad \Gamma \vdash e : [c/u]\tau}{\Gamma \vdash [\![c\,;e]\!] : [\![u :: \kappa\,;\tau]\!]} \tag{44.2b}$$

$$\frac{\Gamma \vdash \sigma\ \mathsf{sig} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash M \upharpoonright \sigma : \sigma} \tag{44.2c}$$

$$\frac{\Gamma \vdash \sigma\ \mathsf{sig} \quad \Gamma \vdash M_1 : \sigma_1 \quad \Gamma, X : \sigma_1 \vdash M_2 : \sigma}{\Gamma \vdash (\,\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2\,) : \sigma : \sigma} \tag{44.2d}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash M : \sigma'} \tag{44.2e}$$

In rule (44.2b) it is always possible to choose κ to be the most specific kind of c in the subkind ordering, which uniquely determines c up to constructor equivalence. For such a choice, the signature $[\![u :: \kappa\,;\tau]\!]$ is equivalent to $[\![\_ :: \kappa\,;[c/u]\tau]\!]$, which propagates the identity of the static part of the module expression into the type of its dynamic part. Rule (44.2c) is used together with the subsumption (rule (44.2e)) to ensure that M has the specified signature.

The need for a signature annotation on a module definition is a manifestation of the *avoidance problem*. Rule (44.2d) would be perfectly sensible were the signature σ omitted from the syntax of the definition. However, omitting this information greatly complicates type checking. If σ were omitted from the syntax of the definition, the type checker would be required to find a signature σ for the body of the definition that *avoids* the module variable X. Inductively, we may suppose that we have found a signature σ_1 for the module M_1, and a signature σ_2 for the module M_2, under the assumption that X has signature σ_1. To find a signature for an unadorned definition, we must find a supersignature σ of σ_2 that avoids X. To ensure that all possible choices of σ are accounted for, we seek the least (most precise) such signature with respect to the subsignature relation; this is called the *principal signature* of a module. The problem is that there may not be a least supersignature of a given signature that avoids a specified variable. (Consider the example above of a signature with two incomparable supersignatures. The example can be chosen so that the supersignatures avoid a variable X that occurs in the subsignature.) Consequently, modules do not have principal signatures, a significant complication for type checking. To avoid this problem, we insist that the

avoiding supersignature σ be given by the programmer so that the type checker is not required to find one.

Modules give rise to a new form of constructor expression, $M \cdot \mathtt{s}$, and a new form of value expression, $M \cdot \mathtt{d}$. These operations, respectively, extract the static and dynamic parts of the module M. Their formation rules are as follows:

$$\frac{\Gamma \vdash M \ \mathsf{val} \quad \Gamma \vdash M : [\![u :: \kappa \,;\, \tau]\!]}{\Gamma \vdash M \cdot \mathtt{s} :: \kappa} \tag{44.3a}$$

$$\frac{\Gamma \vdash M : [\![\_ :: \kappa \,;\, \tau]\!]}{\Gamma \vdash M \cdot \mathtt{d} : \tau} \tag{44.3b}$$

Rule (44.3a) requires that the module expression M be a value according to the following rules:

$$\frac{}{\Gamma, X : \sigma, X \ \mathsf{val} \vdash X \ \mathsf{val}} \tag{44.4a}$$

$$\frac{\Gamma \vdash e \ \mathsf{val}}{\Gamma \vdash [\![c \,;\, e]\!] \ \mathsf{val}} \tag{44.4b}$$

(It is not strictly necessary to insist that the dynamic part of a structure be a value for the structure to itself be a value.)

Rule (44.3a) specifies that only structure values have well-defined static parts, and hence precludes reference to the static part of a sealed structure, which is not a value. This property ensures representation independence for abstract types, as discussed in Section 44.1. For if $M \cdot \mathtt{s}$ were admissible when M is a sealed module, it would be a type whose identity depends on the underlying implementation, in violation of the abstraction principle. Module variables are, on the other hand, values, so that if $X : [\![t :: \mathtt{T} \,;\, \tau]\!]$ is a module variable, then $X \cdot \mathtt{s}$ is a well-formed type. What this means in practice is that sealed modules must be bound to variables before they can be used. It is for this reason that we include definitions among module expressions.

Rule (44.3b) requires that the signature of the module, M, be non-dependent, so that the result type, τ, does not depend on the static part of the module. This independence may not always be the case. For example, if M is a sealed module, say $N \upharpoonright [\![t :: \mathtt{T} \,;\, t]\!]$ for some module N, then projection $M \cdot \mathtt{d}$ is ill-formed. For if it were well-formed, its type would be $M \cdot \mathtt{s}$, which would violate representation independence for abstract types. But if M is a module value, then it is always possible to derive a non-dependent signature for it, provided that we include the following rule of *self-recognition*:

$$\frac{\Gamma \vdash M : [\![u :: \kappa \,;\, \tau]\!] \quad \Gamma \vdash M \ \mathsf{val}}{\Gamma \vdash M : [\![u :: \mathtt{S}(\, M \cdot \mathtt{s} :: \kappa \,) \,;\, \tau]\!]} \tag{44.5}$$

This rule propagates the identity of the static part of a module value into its signature. The dependency of the type of the dynamic part on the static part is then eliminable by sharing propagation.

The following rule of constructor equivalence states that a type projection from a module value is eliminable:

$$\frac{\Gamma \vdash [\![c \,;\, e]\!] : [\![t :: \kappa \,;\, \tau]\!] \quad \Gamma \vdash [\![c \,;\, e]\!] \ \mathsf{val}}{\Gamma \vdash [\![c \,;\, e]\!] \cdot \mathtt{s} \equiv c :: \kappa} \tag{44.6}$$

The requirement that the expression e be a value, which is implicit in the second premise of the rule, is not strictly necessary, but does no harm. A consequence is that apparent dependencies of closed constructors (or kinds) on modules may always be eliminated. In particular the identity of the constructor $[\![c\,;e]\!]\cdot \mathtt{s}$ is independent of e, as would be expected if representation independence is to be assured.

The dynamics of modules is given as follows:

$$\frac{e \longmapsto e'}{[\![c\,;e]\!] \longmapsto [\![c\,;e']\!]} \tag{44.7a}$$

$$\frac{e\ \mathsf{val}}{[\![c\,;e]\!]\cdot \mathtt{d} \longmapsto e} \tag{44.7b}$$

There is no need to evaluate constructors at run-time, because the dynamics of expressions does not depend on their types. It is not difficult to prove type safety for this dynamics relative to the foregoing statics.

44.4 First- and Second-Class

It is common to draw a distinction between *first-class* and *second-class* modules based on whether signatures are types, and hence whether modules are just a form of expression like any other. When modules are first-class their values can depend on the state of the world at run-time. When modules are second-class signatures are a separate form of classifier from types, and module expressions may not be used in the same way as ordinary expressions. For example, it may not be possible to compute a module based on the phase of the moon.

Superficially, it seems as though first-class modules are uniformly superior to second-class modules, because you can do more with them. But on closer examination we see that the "less is more" principle applies here as well, much as in the distinction between dynamic and static languages discussed in Chapters 22 and 23. In particular if modules are first-class, then one must adopt a "pessimistic" attitude towards expressions that compute them, precisely because they represent fully general, even state-dependent, computations. One consequence is that it is difficult, or even impossible, to track the identity of the static part of a module during type checking. A general module expression need not have a well-defined static component, precluding its use in type expressions. Second-class modules, on the other hand, can be permissive with the use of the static components of modules in types, precisely because the range of possible computations is reduced. In this respect second-class modules are more powerful than first-class, despite initial impressions. More importantly, a second-class module system can always be enriched to *allow* first-class modules, without *requiring* that they be first-class. Thus we have the best of both worlds: the flexibility of first-class modules and the precision of second-class modules. In short you pay for only what you use: if you use first-class capabilities, you should expect to pay a cost, but if you do not, you should not be taxed on the unrealized gain.

First-class modules are added to **Mod** in the following way. First, enrich the type system with existential types, as described in Chapter 17, so that "first-class modules" are just packages of existential type. A second-class module M of signature $[\![t :: \kappa\,;\tau]\!]$ is made first-class by forming

the package $\mathtt{pack}\, M \cdot \mathtt{s}\, \mathtt{with}\, M \cdot \mathtt{d}\, \mathtt{as}\, \exists(\, t.\tau\,)$ of type $\exists\, t :: \kappa.\tau$ consisting of the static and dynamic parts of M. Second, to allow packages to act like modules, we introduce the module expression $\mathtt{open}\, e$ that opens the contents of a package as a module:

$$\frac{\Gamma \vdash e : \exists\, t :: \kappa.\tau}{\Gamma \vdash \mathtt{open}\, e : [\![t :: \kappa\,;\tau]\!]} \tag{44.8}$$

Because the package e is an arbitrary expression of existential type, the module expression $\mathtt{open}\, e$ may not be regarded as a value, and hence does not have a well-defined static part. Instead we must generally bind it to a variable before it is used, mimicking the composite behavior of the existential elimination form given in Chapter 17.

44.5 Notes

The use of dependent types to express modularity was first proposed by MacQueen (1986). Later studies extended this proposal to model the *phase distinction* between compile- and run-time (Harper et al., 1990), and to account for type abstraction as well as type classes (Harper and Lillibridge, 1994; Leroy, 1994). The avoidance problem was first isolated by Castagna and Pierce (1994) and by Harper and Lillibridge (1994). It has come to play a central role in subsequent work on modules, such as Lillibridge (1997) and Dreyer (2005). The self-recognition rule was introduced by Harper and Lillibridge (1994) and by Leroy (1994). That rule was later identified as a manifestation of higher-order singletons (Stone and Harper, 2006). A consolidation of these ideas is used as the foundation for a mechanization of the meta-theory of modules (Lee et al., 2007). A thorough summary of the main issues in module system design is given in Dreyer (2005).

The presentation given here focuses attention on the type structure required to support modularity. An alternative formulation uses *elaboration*, a translation of modularity constructs into more primitive notions such as polymorphism and higher-order functions. *The Definition of Standard ML* (Milner et al., 1997) pioneered the elaboration approach. Building on earlier work of Russo, a more rigorous type-theoretic formulation was given by Rossberg et al. (2010). The advantage of the elaboration-based approach is that it can make do with a simpler type theory as the target language, but at the expense of making the explanation of modularity more complex.

Exercises

44.1. Consider the type abstraction σ_{set} of *finite sets* of elements of type τ_{elt} given by the following equations:

$$\sigma_{\mathsf{set}} \triangleq [\![t :: \mathtt{T}\,;\tau_{\mathsf{set}}]\!]$$
$$\tau_{\mathsf{set}} \triangleq \langle \mathtt{emp} \hookrightarrow t\,, \mathtt{ins} \hookrightarrow \tau_{\mathsf{elt}} \times t \to t\,, \mathtt{mem} \hookrightarrow \tau_{\mathsf{elt}} \times t \to \mathtt{bool} \rangle.$$

Define an implementation

$$\Gamma, D : \sigma_{\mathsf{dict}} \vdash M_{\mathsf{set}} : \sigma_{\mathsf{set}}$$

of finite sets of elements in terms of a dictionary whose key and value types are chosen appropriately.

44.2. Fix an ordered type τ_{nod} of *nodes*, and consider the type abstraction σ_{grph} of *finite graphs* given by the following equations:

$$\begin{aligned}
\sigma_{\mathsf{grph}} &\triangleq [\![t_{\mathsf{grph}} :: \mathtt{T} \,;\, [\![t_{\mathsf{edg}} :: \mathtt{S}(\,\tau_{\mathsf{edg}}\,) \,;\, \tau_{\mathsf{grph}}]\!]]\!] \\
\tau_{\mathsf{edg}} &\triangleq \tau_{\mathsf{nod}} \times \tau_{\mathsf{nod}} \\
\tau_{\mathsf{grph}} &\triangleq \langle \mathtt{emp} \hookrightarrow t_{\mathsf{grph}} \,,\, \mathtt{ins} \hookrightarrow \tau_{\mathsf{edg}} \times t_{\mathsf{grph}} \to t_{\mathsf{grph}} \,,\, \mathtt{mem} \hookrightarrow \tau_{\mathsf{edg}} \times t_{\mathsf{grph}} \to \mathtt{bool} \rangle.
\end{aligned}$$

The signature σ_{grph} is translucent, with both opaque and transparent type components: graphs themselves are abstract, but edges are pairs of nodes.

Define an implementation

$$N : \sigma_{\mathsf{ord}}, S : \sigma_{\mathsf{nodset}}, D : \sigma_{\mathsf{nodsetdict}} \vdash M_{\mathsf{grph}} : \sigma_{\mathsf{grph}}$$

in terms of an implementation of nodes, sets of nodes, and a dictionary mapping nodes to sets of nodes. Represent the graph by a dictionary assigning to each node the set of nodes incident upon it. Define the node type τ_{nod} to be the type $N \cdot \mathtt{s}$, and choose the signatures of the set and dictionary abstractions appropriately in terms of this choice of node type.

44.3. Define *signature modification*, a variant of kind modification defined in Exercise 43.3, in which a definition of a constructor component can be imposed on a signature. Let P stand for a composition of static and dynamic projections of the form $\cdot\,\mathtt{d} \ldots \cdot\,\mathtt{d} \cdot \mathtt{s}$, so that $X \cdot P$ stands for $X \cdot \mathtt{d} \ldots \cdot \mathtt{d} \cdot \mathtt{s}$. Assume that $\Gamma \vdash \sigma$ sig, $\Gamma, X : \sigma \vdash X \cdot P :: \kappa$, and $\Gamma \vdash c :: \kappa$. Define signature $\sigma\{P := c\}$ such that $\Gamma \vdash \sigma\{P := c\} <: \sigma$ and $\Gamma, X : \sigma\{P := c\} \vdash X \cdot P \equiv c :: \kappa$.

44.4. The signature σ_{grph} is a subsignature (instance) of the type class

$$\sigma_{\mathsf{grphcls}} \triangleq [\![t_{\mathsf{grph}} :: \mathtt{T} \,;\, [\![t_{\mathsf{edg}} :: \mathtt{T} \,;\, \tau_{\mathsf{grph}}]\!]]\!]$$

in which the definition of t_{edg} has been made explicit as the product of two nodes.

Check that $\Gamma \vdash \sigma_{\mathsf{grph}} \equiv \sigma_{\mathsf{grphcls}}\{\cdot\,\mathtt{d} \cdot \mathtt{s} := \tau_{\mathsf{nod}} \times \tau_{\mathsf{nod}}\}$, so that the former can be defined as the latter.

Part XVIII

Equational Reasoning

Chapter 46

Equality for System T

The beauty of functional programming is that equality of expressions in a functional language follows familiar patterns of mathematical reasoning. For example, in the language **T** of Chapter 9 in which we can express addition as the function `plus`, the expressions

$$\lambda\,(\,x\!:\!\texttt{nat}\,)\,\lambda\,(\,y\!:\!\texttt{nat}\,)\,\texttt{plus}(\,x\,)(\,y\,)$$

and

$$\lambda\,(\,x\!:\!\texttt{nat}\,)\,\lambda\,(\,y\!:\!\texttt{nat}\,)\,\texttt{plus}(\,y\,)(\,x\,)$$

are equal. In other words, the addition function *as programmed in* **T** is commutative.

Commutativity of addition may seem self-evident, but *why* is it true? What does it mean for two expressions to be equal? These two expressions are not *definitionally* equal; their equality requires proof, and is not merely a matter of calculation. Yet the two expressions are interchangeable because they give the same result when applied to the same number. In general two functions are *logically equivalent* if they give equal results for equal arguments. Because this is all that matters about a function, we may expect that logically equivalent functions are interchangeable in any program. Thinking of the programs in which these functions occur as *observations* of their behavior, these functions are said to be *observationally equivalent*. The main result of this chapter is that observational and logical equivalence coincide for a variant of **T** in which the successor is evaluated eagerly, so that a value of type `nat` is a numeral.

46.1 Observational Equivalence

When are two expressions equal? Whenever we cannot tell them apart! It may seem tautological to say so, but it is not, because it all depends on what we consider to be a means of telling expressions apart. What "experiment" are we permitted to perform on expressions in order to distinguish them? What counts as an observation that, if different for two expressions, is a sure sign that they are different?

If we permit ourselves to consider the syntactic details of the expressions, then very few expressions could be considered equal. For example, if it is significant that an expression contains, say, more than one function application, or that it has an occurrence of λ-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation, and not to the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamics of **T** is defined only for expressions without free variables, so a complete program should clearly be a *closed* expression. Second, the outcome of a computation should be *observable*, so that it is evident whether the outcome of two computations differs or not. We define a *complete program* to be a closed expression of type `nat`, and define the *observable behavior* of the program to be the numeral to which it evaluates.

An *experiment* on, or *observation* about, an expression is any means of using that expression within a complete program. We define an *expression context* to be an expression with a "hole" in it serving as a place-holder for another expression. The hole is permitted to occur anywhere, including within the scope of a binder. The bound variables within whose scope the hole lies are *exposed to capture* by the expression context. A *program context* is a closed expression context of type `nat`—that is, it is a complete program with a hole in it. The meta-variable $\mathcal{C}$ stands for any expression context.

Replacement is the process of filling a hole in an expression context $\mathcal{C}$ with an expression e which is written $\mathcal{C}\{e\}$. Importantly, the free variables of e that are exposed by $\mathcal{C}$ are *captured* by replacement (which is why replacement is not a form of substitution, which is defined so as to avoid capture). If $\mathcal{C}$ is a program context, then $\mathcal{C}\{e\}$ is a complete program iff all free variables of e are captured by the replacement. For example, if $\mathcal{C} = \lambda\,(\,x\!:\!\mathtt{nat}\,)\,\circ$, and $e = x + x$, then

$$\mathcal{C}\{e\} = \lambda\,(\,x\!:\!\mathtt{nat}\,)\,x + x.$$

The free occurrences of x in e are captured by the λ-abstraction as a result of the replacement of the hole in $\mathcal{C}$ by e.

We sometimes write $\mathcal{C}\{\circ\}$ to emphasize the occurrence of the hole in $\mathcal{C}$. Expression contexts are closed under *composition* in that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are expression contexts, then so is

$$\mathcal{C}\{\circ\} \triangleq \mathcal{C}_1\{\mathcal{C}_2\{\circ\}\},$$

and we have $\mathcal{C}\{e\} = \mathcal{C}_1\{\mathcal{C}_2\{e\}\}$. The *trivial*, or *identity*, expression context is the "bare hole", written $\circ$, for which $\circ\{e\} = e$.

The statics of expressions of **T** is extended to expression contexts by defining the typing judgment

$$\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$$

so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$. This judgment is inductively defined by a collection of rules derived from the statics of **T** (see rules (9.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \tag{46.1a}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})}{\mathtt{s}(\,\mathcal{C}\,) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})} \tag{46.1b}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,\mathcal{C}\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,)\,\mathtt{with}\,y \hookrightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{46.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau') \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow \mathcal{C}_0 \mid \mathtt{s}(\,x\,)\,\mathtt{with}\,y \hookrightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{46.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \rhd \tau')}{\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,)\,\mathtt{with}\,y \hookrightarrow \mathcal{C}_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{46.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \tau_1 \rhd \tau_2)}{\lambda\,(\,x : \tau_1\,)\,\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2)} \tag{46.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(\,e_2\,) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{46.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}{e_1(\,\mathcal{C}_2\,) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{46.1h}$$

Lemma 46.1. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, *then* $\Gamma' \subseteq \Gamma$, *and if* $\Gamma \vdash e : \tau$, *then* $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$.

Contexts are closed under composition, with the trivial context acting as an identity for it.

Lemma 46.2. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, *and* $\mathcal{C}' : (\Gamma' \rhd \tau') \rightsquigarrow (\Gamma'' \rhd \tau'')$, *then* $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma'' \rhd \tau'')$.

Lemma 46.3. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ *and* $x \notin \mathit{dom}(\Gamma)$, *then* $\mathcal{C} : (\Gamma, x : \tau'' \rhd \tau) \rightsquigarrow (\Gamma', x : \tau'' \rhd \tau')$.

Proof. By induction on rules (46.1). □

A *complete program* is a closed expression of type `nat`.

Definition 46.4. *Two complete programs,* e *and* e', *are* Kleene equal, *written* $e \simeq e'$, *iff there exists* $n \geq 0$ *such that* $e \longmapsto^* \overline{n}$ *and* $e' \longmapsto^* \overline{n}$.

Kleene equality is obviously symmetric; its transitivity follows from determinacy of evaluation. Closure under converse evaluation follows similarly. It is immediate from the definition that $\overline{0} \not\simeq \overline{1}$.

Definition 46.5. *Suppose that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$ *are two expressions of the same type. Two such expressions are* observationally equivalent, *written* $\Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *for every program context* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\emptyset \rhd \mathtt{nat})$.

In other words, for all possible experiments, the outcome of an experiment on e is the same as the outcome on e', which is an equivalence relation. For the sake of brevity, we often write $e \cong_\tau e'$ for $\emptyset \vdash e \cong e' : \tau$.

A family of relations $\Gamma \vdash e_1 \;\mathcal{E}\; e_2 : \tau$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } \Gamma \vdash e \;\mathcal{E}\; e' : \tau, \text{ then } \Gamma' \vdash \mathcal{C}\{e\} \;\mathcal{E}\; \mathcal{C}\{e'\} : \tau'$$

for every expression context $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$. Such a family of relations is *consistent* iff $\emptyset \vdash e \;\mathcal{E}\; e' : \mathtt{nat}$ implies $e \simeq e'$.

Theorem 46.6. *Observational equivalence is the coarsest consistent congruence on expressions.*

Proof. Consistency follows from the definition by noting that the trivial context is a program context. Observational equivalence is clearly an equivalence relation. To show that it is a congruence, we need only observe that type-correct composition of a program context with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $\Gamma \vdash e \mathrel{\mathcal{E}} e' : \tau$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$, then by congruence $\varnothing \vdash \mathcal{C}\{e\} \mathrel{\mathcal{E}} \mathcal{C}\{e'\} : \mathtt{nat}$, and hence by consistency $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$. □

A *closing substitution* γ for the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning closed expressions $e_1 : \tau_1, \ldots, e_n : \tau_n$ to $x_1, \ldots, x_n$, respectively. We write $\hat{\gamma}(e)$ for the substitution $[e_1, \ldots, e_n / x_1, \ldots, x_n]e$, and write $\gamma : \Gamma$ to mean that if $x : \tau$ occurs in Γ, then there exists a closed expression e such that $\gamma(x) = e$ and $e : \tau$. We write $\gamma \cong_\Gamma \gamma'$, where $\gamma : \Gamma$ and $\gamma' : \Gamma$, to express that $\gamma(x) \cong_{\Gamma(x)} \gamma'(x)$ for each x declared in Γ.

Lemma 46.7. *If $\Gamma \vdash e \cong e' : \tau$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. Moreover, if $\gamma \cong_\Gamma \gamma'$, then $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e)$ and $\hat{\gamma}(e') \cong_\tau \widehat{\gamma'}(e')$.*

Proof. Let $\mathcal{C} : (\varnothing \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$ be a program context; we are to show that $\mathcal{C}\{\hat{\gamma}(e)\} \simeq \mathcal{C}\{\hat{\gamma}(e')\}$. Because $\mathcal{C}$ has no free variables, this is equivalent to showing that $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Let $\mathcal{D}$ be the context

$$\lambda\,(\,x_1 : \tau_1\,) \ldots \lambda\,(\,x_n : \tau_n\,)\,\mathcal{C}\{\circ\}(\,e_1\,)\ldots(\,e_n\,),$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $\gamma(x_1) = e_1, \ldots, \gamma(x_n) = e_n$. By Lemma 46.3 we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$. Because $\Gamma \vdash e \cong e' : \tau$, we have $\mathcal{D}\{e\} \simeq \mathcal{D}\{e'\}$. But by construction $\mathcal{D}\{e\} \simeq \hat{\gamma}(\mathcal{C}\{e\})$, and $\mathcal{D}\{e'\} \simeq \hat{\gamma}(\mathcal{C}\{e'\})$, so $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Because $\mathcal{C}$ is arbitrary, it follows that $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$.

Defining $\mathcal{D}'$ like $\mathcal{D}$, but based on γ', rather than γ, we may also show that $\mathcal{D}'\{e\} \simeq \mathcal{D}'\{e'\}$, and hence $\widehat{\gamma'}(e) \cong_\tau \widehat{\gamma'}(e')$. Now if $\gamma \cong_\Gamma \gamma'$, then by congruence we have $\mathcal{D}\{e\} \cong_{\mathtt{nat}} \mathcal{D}'\{e\}$, and $\mathcal{D}\{e'\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$. It follows that $\mathcal{D}\{e\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$, and so, by consistency of observational equivalence, we have $\mathcal{D}\{e\} \simeq \mathcal{D}'\{e'\}$, which is to say that $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. □

Theorem 46.6 licenses the principle of *proof by coinduction*: to show that $\Gamma \vdash e \cong e' : \tau$, it is enough to exhibit a consistent congruence $\mathcal{E}$ such that $\Gamma \vdash e \mathrel{\mathcal{E}} e' : \tau$. It can be difficult to construct such a relation. In the next section we will provide a general method for doing so that exploits types.

46.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of expressions of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that manipulate expressions without inspecting them. For example, we may pass an expression of type τ to a function that simply returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only

the active uses that matter; the passive uses manipulate expressions at arm's length, affording no opportunities to distinguish one from another.

Logical equivalence is therefore defined as follows.

Definition 46.8. Logical equivalence *is a family of relations $e \sim_\tau e'$ between closed expressions of type τ. It is defined by induction on τ as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad \textit{if } e_1 \sim_{\tau_1} e_1', \textit{ then } e(e_1) \sim_{\tau_2} e'(e_1')$$

The definition of logical equivalence at type `nat` licenses the following principle of *proof by* `nat`*-induction*. To show that $\mathcal{E}(e, e')$ whenever $e \sim_{\mathtt{nat}} e'$, it is enough to show that

1. $\mathcal{E}(\overline{0}, \overline{0})$, and
2. if $\mathcal{E}(\overline{n}, \overline{n})$, then $\mathcal{E}(\overline{n+1}, \overline{n+1})$.

This assertion is justified by mathematical induction on $n \geq 0$, where $e \longmapsto^* \overline{n}$ and $e' \longmapsto^* \overline{n}$ by the definition of Kleene equivalence.

Lemma 46.9. *Logical equivalence is symmetric and transitive: if $e \sim_\tau e'$, then $e' \sim_\tau e$, and if $e \sim_\tau e'$ and $e' \sim_\tau e''$, then $e \sim_\tau e''$.*

Proof. Simultaneously, by induction on the structure of τ. If $\tau = \mathtt{nat}$, the result is immediate. If $\tau = \tau_1 \to \tau_2$, then we may assume that logical equivalence is symmetric and transitive at types τ_1 and τ_2. For symmetry, assume that $e \sim_\tau e'$; we wish to show $e' \sim_\tau e$. Assume that $e_1' \sim_{\tau_1} e_1$; it suffices to show that $e'(e_1') \sim_{\tau_2} e(e_1)$. By induction we have that $e_1 \sim_{\tau_1} e_1'$. Therefore by assumption $e(e_1) \sim_{\tau_2} e'(e_1')$, and hence by induction $e'(e_1') \sim_{\tau_2} e(e_1)$. For transitivity, assume that $e \sim_\tau e'$ and $e' \sim_\tau e''$; we are to show $e \sim_\tau e''$. Suppose that $e_1 \sim_{\tau_1} e_1''$; it is enough to show that $e(e_1) \sim_\tau e''(e_1'')$. By symmetry and transitivity we have $e_1 \sim_{\tau_1} e_1$, so by assumption $e(e_1) \sim_{\tau_2} e'(e_1)$. We also have by assumption $e'(e_1) \sim_{\tau_2} e''(e_1'')$. By transitivity we have $e'(e_1) \sim_{\tau_2} e''(e_1'')$, which suffices for the result. □

Logical equivalence is extended to open terms by substitution of related closed terms to obtain related results. If γ and γ' are two substitutions for Γ, we define $\gamma \sim_\Gamma \gamma'$ to hold iff $\gamma(x) \sim_{\Gamma(x)} \gamma'(x)$ for every variable, x, such that $\Gamma \vdash x : \tau$. *Open logical equivalence*, written $\Gamma \vdash e \sim e' : \tau$, is defined to mean that $\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

Lemma 46.10. *Open logical equivalence is symmetric and transitive.*

Proof. Follows from Lemma 46.9 and the definition of open logical equivalence. □

At this point we are "two thirds of the way" to justifying the use of the name "open logical equivalence." The remaining third, reflexivity, is established in the next section.

46.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

Lemma 46.11 (Converse Evaluation). *Suppose that $e \sim_\tau e'$. If $d \longmapsto e$, then $d \sim_\tau e'$, and if $d' \longmapsto e'$, then $e \sim_\tau d'$.*

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows from the closure of Kleene equivalence under converse evaluation. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \sim_\tau e'$, and $d \longmapsto e$. To show that $d \sim_\tau e'$, we assume $e_1 \sim_{\tau_1} e_1'$ and show $d(e_1) \sim_{\tau_2} e'(e_1')$. It follows from the assumption that $e(e_1) \sim_{\tau_2} e'(e_1')$. Noting that $d(e_1) \longmapsto e(e_1)$, the result follows by induction. □

Lemma 46.12 (Consistency). *If $e \sim_{\mathtt{nat}} e'$, then $e \simeq e'$.*

Proof. Immediate, from Definition 46.8. □

Theorem 46.13 (Reflexivity). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash e \sim e : \tau$.*

Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e)$. The proof proceeds by induction on typing derivations; we consider two representative cases.

Consider the case of rule (8.4a), in which $\tau = \tau_1 \to \tau_2$ and $e = \lambda(x:\tau_1)\, e_2$. We are to show that

$$\lambda(x:\tau_1)\,\hat{\gamma}(e_2) \sim_{\tau_1\to\tau_2} \lambda(x:\tau_1)\,\widehat{\gamma'}(e_2).$$

Assume that $e_1 \sim_{\tau_1} e_1'$; by Lemma 46.11, it is enough to show that $[e_1/x]\hat{\gamma}(e_2) \sim_{\tau_2} [e_1'/x]\widehat{\gamma'}(e_2)$. Let $\gamma_2 = \gamma \otimes x \hookrightarrow e_1$ and $\gamma_2' = \gamma' \otimes x \hookrightarrow e_1'$, and observe that $\gamma_2 \sim_{\Gamma,x:\tau_1} \gamma_2'$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim_{\tau_2} \hat{\gamma}_2'(e_2)$, from which the result follows easily.

Now consider the case of rule (9.1d), for which we are to show that

$$\mathtt{rec}\{\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)\}(\hat{\gamma}(e)) \sim_\tau \mathtt{rec}\{\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1)\}(\widehat{\gamma'}(e)).$$

By the induction hypothesis applied to the first premise of rule (9.1d), we have

$$\hat{\gamma}(e) \sim_{\mathtt{nat}} \widehat{\gamma'}(e).$$

We proceed by $\mathtt{nat}$-induction. It suffices to show that

$$\mathtt{rec}\{\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)\}(\mathtt{z}) \sim_\tau \mathtt{rec}\{\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1)\}(\mathtt{z}), \tag{46.2}$$

and that

$$\mathtt{rec}\{\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)\}(\mathtt{s}(\overline{n})) \sim_\tau \mathtt{rec}\{\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1)\}(\mathtt{s}(\overline{n})), \tag{46.3}$$

assuming

$$\mathtt{rec}\{\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)\}(\overline{n}) \sim_\tau \mathtt{rec}\{\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1)\}(\overline{n}). \tag{46.4}$$

To show (46.2), by Lemma 46.11 it is enough to show that $\hat{\gamma}(e_0) \sim_\tau \widehat{\gamma'}(e_0)$. This condition is assured by the outer inductive hypothesis applied to the second premise of rule (9.1d).

To show (46.3), define

$$\delta = \gamma \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}\{\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)\}(\overline{n})$$

and

$$\delta' = \gamma' \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}\{\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1)\}(\overline{n}).$$

By (46.4) we have $\delta \sim_{\Gamma, x:\mathtt{nat}, y:\tau} \delta'$. Consequently, by the outer inductive hypothesis applied to the third premise of rule (9.1d), and Lemma 46.11, the required follows. □

Corollary 46.14 (Equivalence)**.** *Open logical equivalence is an equivalence relation.*

Corollary 46.15 (Termination)**.** *If $e : \mathtt{nat}$, then $e \longmapsto^* e'$ for some e' val.*

Lemma 46.16 (Congruence)**.** *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $\Gamma \vdash e \sim e' : \tau$, then $\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0$.*

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$. We consider a representative case in which $\mathcal{C}_0 = \lambda(x : \tau_1)\,\mathcal{C}_2$ so that $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_1 \to \tau_2)$ and $\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $\Gamma \vdash e \sim e' : \tau$, we are to show that

$$\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_1 \to \tau_2,$$

which is to say

$$\Gamma_0 \vdash \lambda(x : \tau_1)\,\mathcal{C}_2\{e\} \sim \lambda(x : \tau_1)\,\mathcal{C}_2\{e'\} : \tau_1 \to \tau_2.$$

We know, by induction, that

$$\Gamma_0, x : \tau_1 \vdash \mathcal{C}_2\{e\} \sim \mathcal{C}_2\{e'\} : \tau_2.$$

Suppose that $\gamma_0 \sim_{\Gamma_0} \gamma_0'$, and that $e_1 \sim_{\tau_1} e_1'$. Let $\gamma_1 = \gamma_0 \otimes x \hookrightarrow e_1$, $\gamma_1' = \gamma_0' \otimes x \hookrightarrow e_1'$, and observe that $\gamma_1 \sim_{\Gamma_0, x:\tau_1} \gamma_1'$. By Definition 46.8 and Lemma 46.11 it is enough to show that

$$\hat{\gamma}_1(\mathcal{C}_2\{e\}) \sim_{\tau_2} \hat{\gamma}_1'(\mathcal{C}_2\{e'\}),$$

which follows from the inductive hypothesis.

□

Theorem 46.17. *If $\Gamma \vdash e \sim e' : \tau$, then $\Gamma \vdash e \cong e' : \tau$.*

Proof. By Lemmas 46.12 and 46.16, and Theorem 46.6. □

Corollary 46.18. *If $e : \mathtt{nat}$, then $e \cong_{\mathtt{nat}} \overline{n}$, for some $n \geq 0$.*

Proof. By Theorem 46.13 we have $e \sim_{\mathtt{nat}} e$. Hence for some $n \geq 0$, we have $e \sim_{\mathtt{nat}} \overline{n}$, and so by Theorem 46.17, $e \cong_{\mathtt{nat}} \overline{n}$. □

Lemma 46.19. *For closed expressions $e : \tau$ and $e' : \tau$, if $e \cong_\tau e'$, then $e \sim_\tau e'$.*

Proof. We proceed by induction on the structure of τ. If $\tau = \mathtt{nat}$, consider the empty context to obtain $e \simeq e'$, and hence $e \sim_{\mathtt{nat}} e'$. If $\tau = \tau_1 \to \tau_2$, then we are to show that whenever $e_1 \sim_{\tau_1} e_1'$, we have $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 46.17 we have $e_1 \cong_{\tau_1} e_1'$, and hence by congruence of observational equivalence it follows that $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

Theorem 46.20. *If* $\Gamma \vdash e \cong e' : \tau$, *then* $\Gamma \vdash e \sim e' : \tau$.

Proof. Assume that $\Gamma \vdash e \cong e' : \tau$, and that $\gamma \sim_\Gamma \gamma'$. By Theorem 46.17 we have $\gamma \cong_\Gamma \gamma'$, so by Lemma 46.7 $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. Therefore, by Lemma 46.19, $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. □

Corollary 46.21. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \sim e' : \tau$.

Definitional equality is sufficient for observational equivalence:

Theorem 46.22. *If* $\Gamma \vdash e \equiv e' : \tau$, *then* $\Gamma \vdash e \sim e' : \tau$, *and hence* $\Gamma \vdash e \cong e' : \tau$.

Proof. By an argument similar to that used in the proof of Theorem 46.13 and Lemma 46.16, then appealing to Theorem 46.17. □

Corollary 46.23. *If* $e \equiv e' : \mathtt{nat}$, *then there exists* $n \geq 0$ *such that* $e \longmapsto^* \overline{n}$ *and* $e' \longmapsto^* \overline{n}$.

Proof. By Theorem 46.22 we have $e \sim_{\mathtt{nat}} e'$ and hence $e \simeq e'$. □

46.4 Some Laws of Equality

In this section we summarize some useful principles of observational equivalence for **T**. For the most part these are laws of logical equivalence, and then transferred to observational equivalence by appeal to Corollary 46.21. The laws are presented as inference rules with the meaning that if all of the premises are true judgments about observational equivalence, then so are the conclusions. In other words each rule is admissible as a principle of observational equivalence.

46.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau} \tag{46.5a}$$

$$\frac{\Gamma \vdash e' \cong e : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{46.5b}$$

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau} \tag{46.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equalities are observational equivalences.

$$\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{46.6a}$$

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\Gamma' \vdash \mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'} \tag{46.7a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e/x]e_2 \cong [e/x]e_2' : \tau'} \tag{46.8a}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'} \tag{46.8b}$$

46.4.2 Equality Laws

Two functions are equal if they are equal on all arguments.

$$\frac{\Gamma, x : \tau_1 \vdash e(\,x\,) \cong e'(\,x\,) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2} \tag{46.9}$$

Consequently, every expression of function type is equal to a λ-abstraction:

$$\frac{}{\Gamma \vdash e \cong \lambda\,(\,x : \tau_1\,)\, e(\,x\,) : \tau_1 \to \tau_2} \tag{46.10}$$

46.4.3 Induction Law

An equation involving a free variable x of type `nat` can be proved by induction on x.

$$\frac{\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \text{ (for every } n \in \mathbb{N})}{\Gamma, x : \mathtt{nat} \vdash e \cong e' : \tau} \tag{46.11a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $\Gamma \vdash [\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau$, and
2. $\Gamma \vdash [\mathtt{s}(\,\overline{n}\,)/x]e \cong [\mathtt{s}(\,\overline{n}\,)/x]e' : \tau$, if $\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau$.

46.5 Notes

The method of *logical relations* interprets types as relations (here, equivalence relations) by associating with each type constructor a relational action that transforms the relation interpreting its arguments to the relation interpreting the constructed type. Logical relations (Statman, 1985) are a fundamental tool in proof theory and provide the foundation for the semantics of the NuPRL type theory (Constable, 1986; Allen, 1987; Harper, 1992). The use of logical relations to characterize observational equivalence is an adaptation of the NuPRL semantics to the simpler setting of Gödel's System **T**.

Chapter 47

Equality for System PCF

In this Chapter we develop the theory of observational equivalence for **PCF**, with an eager interpretation of the type of natural numbers. The development proceeds along lines similar to those in Chapter 46, but is complicated by the presence of general recursion. The proof depends on the concept of an *admissible relation*, one that admits the principle of *proof by fixed point induction*.

47.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 46, but modified to account for the possibility of non-termination.

The collection of well-formed **PCF** contexts is inductively defined in a manner directly analogous to that in Chapter 46. Specifically, we define the judgment $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ by rules similar to rules (46.1), modified for **PCF**. (We leave the precise definition as an exercise for the reader.) When Γ and Γ' are empty, we write just $\mathcal{C} : \tau \rightsquigarrow \tau'$.

A *complete program* is a closed expression of type `nat`.

Definition 47.1. *We say that two complete programs, e and e', are* Kleene equal*, written $e \simeq e'$, iff for every $n \geq 0$, $e \longmapsto^* \overline{n}$ iff $e' \longmapsto^* \overline{n}$.*

Kleene equality is clearly an equivalence relation and is closed under converse evaluation. Moreover, $\overline{0} \not\simeq \overline{1}$ and, if e and e' are both divergent, then $e \simeq e'$.

Observational equivalence is defined just as it is in Chapter 46.

Definition 47.2. *We say that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are* observationally*, or* contextually*,* equivalent *iff for every program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$, $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.*

Theorem 47.3. *Observational equivalence is the coarsest consistent congruence.*

Proof. See the proof of Theorem 46.6. □

Lemma 47.4 (Substitution and Functionality)**.** *If $\Gamma \vdash e \cong e' : \tau$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. Moreover, if $\gamma \cong_\Gamma \gamma'$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}'(e)$ and $\hat{\gamma}(e') \cong_\tau \hat{\gamma}'(e')$.*

Proof. See Lemma 46.7. □

47.2 Logical Equivalence

Definition 47.5. Logical equivalence, $e \sim_\tau e'$, *between closed expressions of type* τ *is defined by induction on* τ *as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad e_1 \sim_{\tau_1} e_1' \textit{ implies } e(e_1) \sim_{\tau_2} e'(e_1')$$

Formally, logical equivalence is defined as in Chapter 46, except that the definition of Kleene equivalence is altered to account for non-termination. Logical equivalence is extended to open terms by substitution. Specifically, we define $\Gamma \vdash e \sim e' : \tau$ to mean that $\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

By the same argument as given in the proof of Lemma 46.9 logical equivalence is symmetric and transitive, as is its open extension.

Lemma 47.6 (Strictness)**.** *If* $e : \tau$ *and* $e' : \tau$ *are both divergent, then* $e \sim_\tau e'$.

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows immediately from the definition of Kleene equivalence. If $\tau = \tau_1 \to \tau_2$, then $e(e_1)$ and $e'(e_1')$ diverge, so by induction $e(e_1) \sim_{\tau_2} e'(e_1')$, as required. □

Lemma 47.7 (Converse Evaluation)**.** *Suppose that* $e \sim_\tau e'$. *If* $d \longmapsto e$, *then* $d \sim_\tau e'$, *and if* $d' \longmapsto e'$, *then* $e \sim_\tau d'$.

47.3 Logical and Observational Equivalence Coincide

The proof of coincidence of logical and observational equivalence relies on the concept of *bounded recursion*, which we define by induction on $m \geq 0$ as follows:

$$\overset{0}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, e \triangleq \mathtt{fix}\, x : \tau \,\mathtt{is}\, x$$

$$\overset{m+1}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, e \triangleq [\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, e / x]e$$

When $m = 0$, bounded recursion is defined to be a divergent expression of type τ. When $m > 0$, bounded recursion is defined by unrolling the recursion m times by iterated substitution. Intuitively, the bounded recursive expression $\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e$ is as good as $\mathtt{fix}\, x : \tau \,\mathtt{is}\, e$ for up to m unrollings, after which it is divergent.

It is easy to check that the following rule is derivable for each $m \geq 0$:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}^m\{\tau\}(x.e) : \tau}\,. \tag{47.1a}$$

The proof is by induction on $m \geq 0$, and amounts to an iteration of the substitution lemma for the statics of **PCF**.

The key property of bounded recursion is the principle of fixed point induction, which permits reasoning about a recursive computation by induction on the number of unrollings required to reach a value. The proof relies on *compactness*, which will be stated and proved in Section 47.4 below.

Theorem 47.8 (Fixed Point Induction)**.** *Suppose that* $x : \tau \vdash e : \tau$. *If*

$$(\forall m \geq 0)\ \overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, e \sim_\tau \overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, e',$$

then $\mathtt{fix}\, x : \tau \,\mathtt{is}\, e \sim_\tau \mathtt{fix}\, x : \tau \,\mathtt{is}\, e'$.

Proof. Define an *applicative context* $\mathcal{A}$ to be either a hole, $\circ$, or an application of the form $\mathcal{A}(e)$, where $\mathcal{A}$ is an applicative context. The typing judgment for applicative contexts, $\mathcal{A} : \tau_0 \rightsquigarrow \tau$, is a special case of the general typing judgment for contexts. Define logical equivalence of applicative contexts, $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau$, by induction on the structure of $\mathcal{A}$ as follows:

1. $\circ \sim \circ : \tau_0 \rightsquigarrow \tau_0$;
2. if $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau_2 \rightharpoonup \tau$ and $e_2 \sim_{\tau_2} e_2'$, then $\mathcal{A}(e_2) \sim \mathcal{A}'(e_2') : \tau_0 \rightsquigarrow \tau$.

We prove by induction on the structure of τ, if $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau$ and

$$\text{for every } m \geq 0,\ \mathcal{A}\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_\tau \mathcal{A}'\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e'\}, \tag{47.2}$$

then

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}. \tag{47.3}$$

Choosing $\mathcal{A} = \mathcal{A}' = \circ$ with $\tau_0 = \tau$ completes the proof.

If $\tau = \mathtt{nat}$, then assume that $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \mathtt{nat}$ and (47.2). By Definition 47.5, we are to show

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

By Corollary 47.17 there exists $m \geq 0$ such that

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e\}.$$

By (47.2) we have

$$\mathcal{A}\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}'\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

By Corollary 47.17

$$\mathcal{A}'\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e'\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 47.5, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\}(e_1) \sim_{\tau_2} \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}(e_1')$$

whenever $e_1 \sim_{\tau_1} e_1'$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (47.2) that for every $m \geq 0$

$$\mathcal{A}_2\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\overset{m}{\mathtt{fix}}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

Noting that $\mathcal{A}_2 \sim \mathcal{A}_2' : \tau_0 \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\},$$

as required. □

Lemma 47.9 (Reflexivity)**.** *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e \sim e : \tau$.

Proof. The proof proceeds along the same lines as the proof of Theorem 46.13. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider rule (19.1g). Assuming $\gamma \sim_\Gamma \gamma'$, we are to show that

$$\mathtt{fix}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e).$$

By Theorem 47.8 it is enough to show that, for every $m \geq 0$,

$$\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e).$$

We proceed by an inner induction on m. When $m = 0$ the result is immediate, because both sides of the desired equivalence diverge. Assuming the result for m, and applying Lemma 47.7, it is enough to show that $\hat{\gamma}(e_1) \sim_\tau \widehat{\gamma'}(e_1')$, where

$$e_1 = [\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{47.4}$$

$$e_1' = [\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e). \tag{47.5}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e),$$

then

$$[\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e) \sim_\tau [\overset{m}{\mathtt{fix}}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e).$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows.

To handle the conditional $\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$, we proceed by cases on whether e diverges, in which case the conditional is divergent and therefore self-related by Lemma 47.6, or e converges, in which case we can proceed by an inner mathematical induction on its value, appealing to the inductive hypotheses governing the branches of the conditional to complete the argument. □

Symmetry and transitivity of eager logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager logical equivalence is therefore an equivalence relation.

Lemma 47.10 (Congruence). *If* $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, *and* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0$.

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$, following along similar lines to the proof of Lemma 47.9. □

Logical equivalence is consistent, by definition. Consequently, it is contained in observational equivalence.

Theorem 47.11. *If* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma \vdash e \cong e' : \tau$.

Proof. By consistency and congruence of logical equivalence. □

Lemma 47.12. *If* $e \cong_\tau e'$, *then* $e \sim_\tau e'$.

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result is immediate, because the empty expression context is a program context. If $\tau = \tau_1 \rightharpoonup \tau_2$, then suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 47.11 $e_1 \cong_{\tau_1} e_1'$, and hence by Lemma 47.4 $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

Theorem 47.13. *If* $\Gamma \vdash e \cong e' : \tau$, *then* $\Gamma \vdash e \sim e' : \tau$.

Proof. Assume that $\Gamma \vdash e \cong e' : \tau$. Suppose that $\gamma \sim_\Gamma \gamma'$. By Theorem 47.11 we have $\gamma \cong_\Gamma \gamma'$, and so by Lemma 47.4 we have

$$\hat{\gamma}(e) \cong_\tau \hat{\gamma'}(e').$$

Therefore by Lemma 47.12 we have

$$\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e').$$

□

Corollary 47.14. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \sim e' : \tau$.

47.4 Compactness

The principle of fixed point induction is derived from a critical property of **PCF**, called *compactness*. This property states that only finitely many unwindings of a fixed point expression are needed in a complete evaluation of a program. Although intuitively obvious (one cannot complete infinitely many recursive calls in a finite computation), it is rather tricky to state and prove rigorously.

The proof of compactness (Theorem 47.16) makes use of the stack machine for **PCF** defined in Chapter 28, augmented with the following transitions for bounded recursive expressions:

$$\frac{}{k \triangleright \mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e \longmapsto k \triangleright \mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e} \tag{47.6a}$$

$$\frac{}{k \triangleright \mathtt{fix}^{m+1}\, x : \tau \,\mathtt{is}\, e \longmapsto k \triangleright [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e / x]e} \tag{47.6b}$$

It is not difficult to extend the proof of Corollary 28.4 to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function f in **PCF**:

$$\mathtt{fix}\, f : \mathtt{nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda\,(\,x : \mathtt{nat}\,)\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow \mathtt{s}(\,\mathtt{z}\,) \mid \mathtt{s}(\,x'\,) \hookrightarrow x * f(\,x'\,)\}.$$

Obviously evaluation of $f(\,\overline{n}\,)$ requires n recursive calls to the function itself. That is, for a given input n we may place a *bound* m on the recursion that is sufficient to ensure termination of the computation. This property can be expressed formally using the m-bounded form of general recursion,

$$\overset{m}{\mathtt{fix}}\, f : \mathtt{nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda\,(\,x : \mathtt{nat}\,)\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow \mathtt{s}(\,\mathtt{z}\,) \mid \mathtt{s}(\,x'\,) \hookrightarrow x * f(\,x'\,)\}.$$

Call this expression $f^{(m)}$. It follows from the definition of f that if $f(\,\overline{n}\,) \longmapsto^* \overline{p}$, then $f^{(m)}(\,\overline{n}\,) \longmapsto^* \overline{p}$ for some $m \geq 0$ (in fact, $m = n$ suffices).

When considering expressions of higher type, we cannot expect to get the *same* result from the bounded recursion as from the unbounded. For example, consider the addition function a of type $\tau = \mathtt{nat} \rightharpoonup (\,\mathtt{nat} \rightharpoonup \mathtt{nat}\,)$, given by the expression

$$\mathtt{fix}\, p : \tau\, \mathtt{is}\, \lambda\,(\,x : \mathtt{nat}\,)\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow \mathit{id} \mid \mathtt{s}(\,x'\,) \hookrightarrow s \circ (\,p(\,x'\,)\,)\},$$

where $\mathit{id} = \lambda\,(\,y : \mathtt{nat}\,)\, y$ is the identity, $e' \circ e = \lambda\,(\,x : \tau\,)\, e'(\,e(\,x\,)\,)$ is composition, and $s = \lambda\,(\,x : \mathtt{nat}\,)\, \mathtt{s}(\,x\,)$ is the successor function. The application $a(\,\overline{n}\,)$ terminates after three transitions, regardless of the value of n, resulting in a λ-abstraction. When n is positive, the result contains a *residual* copy of a itself, which is applied to $n - 1$ as a recursive call. The m-bounded version of a, written $a^{(m)}$, is also such that $a^{(m)}(\,\overline{n}\,)$ terminates in three steps, provided that $m > 0$. But the result is not the same, because the residuals of a appear as $a^{(m-1)}$, rather than as a itself.

Turning now to the proof of compactness, it is helpful to introduce some notation. Suppose that $x : \tau \vdash e_x : \tau$ for some arbitrary abstractor $x.e_x$. Let $f^{(\omega)} = \mathtt{fix}\, x : \tau\, \mathtt{is}\, e_x$, and let $f^{(m)} = \mathtt{fix}^m\, x : \tau\, \mathtt{is}\, e_x$. Observe that $f^{(\omega)} : \tau$ and $f^{(m)} : \tau$ for any $m \geq 0$.

The following technical lemma governing the stack machine allows the bound on occurrences of a recursive expression to be raised without affecting the outcome of evaluation.

Lemma 47.15. *For every $m \geq 0$, if $[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}$, then $[f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}$.*

Proof. By induction on $m \geq 0$, and then induction on transition. □

Theorem 47.16 (Compactness)**.** *Suppose that $y : \tau \vdash e : \mathtt{nat}$ where $y \notin f^{(\omega)}$. If $[f^{(\omega)}/y]e \longmapsto^* \overline{n}$, then there exists $m \geq 0$ such that $[f^{(m)}/y]e \longmapsto^* \overline{n}$.*

Proof. We prove simultaneously the stronger statements that if

$$[f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n},$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n},$$

and if

$$[f^{(\omega)}/y]k \lhd [f^{(\omega)}/y]e \longmapsto^* \epsilon \lhd \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \lhd [f^{(m)}/y]e \longmapsto^* \epsilon \lhd \overline{n}.$$

(Note that if $[f^{(\omega)}/y]e$ val, then $[f^{(m)}/y]e$ val for all $m \geq 0$.) The result then follows by the correctness of the stack machine (Corollary 28.4).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \rhd f^{(\omega)},$$

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \rhd f^{(\omega)} \longmapsto [f^{(\omega)}/y]k \rhd [f^{(\omega)}/x]e_x \longmapsto^* \epsilon \lhd \overline{n}.$$

Noting that $[f^{(\omega)}/x]e_x = [f^{(\omega)}/y][y/x]e_x$, we have by induction that there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \rhd [f^{(m)}/x]e_x \longmapsto^* \epsilon \lhd \overline{n}.$$

By Lemma 47.15

$$[f^{(m+1)}/y]k \rhd [f^{(m)}/x]e_x \longmapsto^* \epsilon \lhd \overline{n}$$

and we need only recall that

$$[f^{(m+1)}/y]k \rhd f^{(m+1)} = [f^{(m+1)}/y]k \rhd [f^{(m)}/x]e_x$$

to complete the proof. If, on the other hand, the initial step is an unrolling, but $e \neq y$, then we have for some $z \notin f^{(\omega)}$ and $z \neq y$

$$[f^{(\omega)}/y]k \rhd \mathtt{fix}\, z : \tau \,\mathtt{is}\, d_\omega \longmapsto [f^{(\omega)}/y]k \rhd [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_\omega / z]d_\omega \longmapsto^* \epsilon \lhd \overline{n}.$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \rhd [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_m / z]d_m \longmapsto^* \epsilon \lhd \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 47.15 we have

$$[f^{(m+1)}/y]k \rhd [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_{m+1} / z]d_{m+1} \longmapsto^* \epsilon \lhd \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

Corollary 47.17. *There exists $m \geq 0$ such that $[f^{(\omega)}/y]e \simeq [f^{(m)}/y]e$.*

Proof. If $[f^{(\omega)}/y]e$ diverges, then it suffices to take m to be zero. Otherwise, apply Theorem 47.16 to obtain m, and note that the required Kleene equivalence follows. □

47.5 Lazy Natural Numbers

Recall from Chapter 19 that if the successor is evaluated lazily, then the type `nat` changes its meaning to that of the lazy natural numbers, which we shall write `lnat` for emphasis. This type contains an "infinite number" ω, which is essentially an endless stack of successors.

To account for the lazy successor the definition of logical equivalence must be reformulated. Rather than being defined *inductively* as the strongest relation closed under specified conditions, it is now defined *coinductively* as the weakest relation consistent with two analogous conditions. We may then show that two expressions are related using the principle of *proof by coinduction*.

The definition of Kleene equivalence must be altered to account for the lazily evaluated successor operation. To account for ω, two computations are compared based solely on the outermost form of their values, if any. We define $e \simeq e'$ to hold iff (a) if $e \longmapsto^* \mathtt{z}$, then $e' \longmapsto^* \mathtt{z}$, and *vice versa*; and (b) if $e \longmapsto^* \mathtt{s}(e_1)$, then $e' \longmapsto^* \mathtt{s}(e_1')$, and *vice versa*.

Corollary 47.17 can be proved for the co-natural numbers by essentially the same argument as before.

The definition of logical equivalence at type `lnat` is defined to be the *weakest* equivalence relation $\mathcal{E}$ between closed terms of type `lnat` satisfying the following *consistency conditions*: if $e \mathrel{\mathcal{E}} e' : \mathtt{lnat}$, then

1. If $e \longmapsto^* \mathtt{z}$, then $e' \longmapsto^* \mathtt{z}$, and *vice versa*.
2. If $e \longmapsto^* \mathtt{s}(e_1)$, then $e' \longmapsto^* \mathtt{s}(e_1')$ with $e_1 \mathrel{\mathcal{E}} e_1' : \mathtt{lnat}$, and *vice versa*.

It is immediate that if $e \sim_{\mathtt{lnat}} e'$, then $e \simeq e'$, and so logical equivalence is consistent. It is also strict in that if e and e' are both divergent expressions of type `lnat`, then $e \sim_{\mathtt{lnat}} e'$.

The principle of *proof by coinduction* states that to show $e \sim_{\mathtt{lnat}} e'$, it suffices to exhibit a relation, $\mathcal{E}$, such that

1. $e \mathrel{\mathcal{E}} e' : \mathtt{lnat}$, and
2. $\mathcal{E}$ satisfies the above consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in logical equivalence at type `lnat`, and hence $e \sim_{\mathtt{lnat}} e'$, as required.

As an application of coinduction, let us consider the proof of Theorem 47.8. The overall argument remains as before, but the proof for the type `lnat` must be altered as follows. Suppose that $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \mathtt{lnat}$, and let $a = \mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\}$ and $a' = \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}$. Writing $a^{(m)} = \mathcal{A}\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\}$ and $a'^{(m)} = \mathcal{A}'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\}$, assume that

$$\text{for every } m \geq 0,\ a^{(m)} \sim_{\mathtt{lnat}} a'^{(m)}.$$

We are to show that

$$a \sim_{\mathtt{lnat}} a'.$$

Define the functions p_n for $n \geq 0$ on closed terms of type `lnat` by the following equations:

$$p_0(d) = d$$
$$p_{(n+1)}(d) = \begin{cases} d' & \text{if } p_n(d) \longmapsto^* \mathtt{s}(\,d'\,) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

For $n \geq 0$, let $a_n = p_n(a)$ and $a'_n = p_n(a')$. Correspondingly, let $a_n^{(m)} = p_n(a^{(m)})$ and ${a'_n}^{(m)} = p_n(a'^{(m)})$. Define $\mathcal{E}$ to be the strongest relation such that $a_n \;\mathcal{E}\; a'_n : \mathtt{lnat}$ for all $n \geq 0$. We will show that the relation $\mathcal{E}$ satisfies the consistency conditions, and so it is contained in logical equivalence. Because $a \;\mathcal{E}\; a' : \mathtt{lnat}$ (by construction), the result follows immediately.

To show that $\mathcal{E}$ is consistent, suppose that $a_n \;\mathcal{E}\; a'_n : \mathtt{lnat}$ for some $n \geq 0$. We have by Corollary 47.17 $a_n \simeq a_n^{(m)}$, for some $m \geq 0$, and hence, by the assumption, $a_n \simeq {a'_n}^{(m)}$, and so by Corollary 47.17 again, ${a'_n}^{(m)} \simeq a'_n$. Now if $a_n \longmapsto^* \mathtt{s}(\,b_n\,)$, then $a_n^{(m)} \longmapsto^* \mathtt{s}(\,b_n^{(m)}\,)$ for some $b_n^{(m)}$, and hence there exists ${b'_n}^{(m)}$ such that ${a'_n}^{(m)} \longmapsto^* {b'_n}^{(m)}$, and so there exists b'_n such that $a'_n \longmapsto^* \mathtt{s}(\,b'_n\,)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n \;\mathcal{E}\; b'_n : \mathtt{lnat}$ by construction, as required.

47.6 Notes

The use of logical relations to characterize observational equivalence for **PCF** is inspired by the treatment of partiality in type theory by Constable and Smith (1987) and by the studies of observational equivalence by Pitts (2000). Although the technical details differ, the proof of compactness here is inspired by Pitts's structurally inductive characterization of termination using an abstract machine. It is critical to restrict attention to transition systems whose states are complete programs (closed expressions of observable type). Structural operational semantics usually does not fulfill this requirement, thereby requiring a considerably more complex argument than given here.

Part XIX

Appendices

Appendix B

Background on Finite Sets

We make frequent use of the concepts of a *finite set* of *discrete objects* and of *finite functions* between them. A set X is *discrete* iff equality of its elements is decidable: for every $x, y \in X$, either $x = y \in X$ or $x \neq y \in X$. This condition is to be understood constructively as stating that we may effectively determine whether any two elements of the set X are equal or not. Perhaps the most basic example of a discrete set is the set $\mathbb{N}$ of natural numbers. A set X is *countable* iff there is a bijection $f : X \cong \mathbb{N}$ between X and the set of natural numbers, and it is *finite* iff there is a bijection, $f : X \cong \{ 0, \ldots, n-1 \}$, where $n \in \mathbb{N}$, between it and some inital segment of the natural numbers. This condition is again to be understood constructively in terms of computable mappings, so that countable and finite sets are computably enumerable and, in the finite case, have a computable size.

Given countable sets, U and V, a *finite function* is a computable partial function $\phi : U \to V$ between them. The *domain* $dom(\phi)$ of ϕ is the set $\{ u \in U \mid \phi(u) \downarrow \}$, of objects $u \in U$ such that $\phi(u) = v$ for some $v \in V$. Two finite functions, ϕ and ψ, between U and V are *disjoint* iff $dom(\phi) \cap dom(\psi) = \emptyset$. The *empty* finite function, $\emptyset$, between U and V is the totally undefined partial function between them. If $u \in U$ and $v \in V$, the finite function, $u \hookrightarrow v$, between U and V sends u to v, and is undefined otherwise; its domain is therefore the singleton set $\{ u \}$. In some situations we write $u \rightsquigarrow v$ for the finite function $u \hookrightarrow v$.

If ϕ and ψ are two disjoint finite functions from U to V, then $\phi \otimes \psi$ is the finite function from U to V defined by the equation

$$(\phi \otimes \psi)(u) = \begin{cases} \phi(u) & \text{if } u \in dom(\phi) \\ \psi(v) & \text{if } v \in dom(\psi) \\ \text{undefined} & \text{otherwise} \end{cases}$$

If $u_1, \ldots, u_n \in U$ are pairwise distinct, and $v_1, \ldots, v_n \in V$, then we sometimes write $u_1 \hookrightarrow v_1, \ldots, u_n \hookrightarrow v_n$, or $u_1 \rightsquigarrow v_1, \ldots, u_n \rightsquigarrow v_n$, for the finite function $u_1 \hookrightarrow v_1 \otimes \ldots \otimes u_n \hookrightarrow v_n$.

Bibliography

Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996. 183, 248, 254

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, 297

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. 452

Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3): 265–301, 1997. 339

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. 365

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. 193

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. 545

Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X. 356

Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995. 356

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996. 61, 356

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. 394

Samuel R. Buss, editor. *Handbook of Proof Theory*. Elsevier, Amsterdam, 1998. 542

Luca Cardelli. Structural subtyping and the notion of power type. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 70–79, 1988. 224

Luca Cardelli. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 266–277, 1997. 408

Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 151–162, 1994. 429

Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. 10, 71, 193

R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986. v, 10, 237, 452

Robert L. Constable. Types in logic, mathematics, and programming. In Buss (1998), chapter X. v

Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987. 461

William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, pages 557–572, 2009. 183

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. 237

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. 237

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. 237

Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005. 429

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. 237

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. 382

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS: Theoretical Computer Science*, 103, 1992. 48, 279

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. 237

Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, 1976. 365

David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 394

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, 474

Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated by Paul Taylor and Yves Lafont. v, 408

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, 271

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. 365

Timothy Griffin. A formulae-as-types notion of control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990. 119

Carl Gunter. *Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1992. 183

Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. 365

Robert Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1): 71–84, 1992. 452

Robert Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 51(4):201–206, 1994. 330

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. 417, 429, 439

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. 429

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. 127

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 382

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. 119

P. J. Landin. A correspondence between Algol 60 and Church's lambda notation. *CACM*, 8:89–101; 158–165, 1965. 48, 265

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. 429

Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994. 417, 429, 440

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 142–153, 1995. 440

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, May 1997. 429

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. 254

Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, second edition, 1998. 127, 136

David B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Symposium on Principles of Programming Languges*, pages 277–286, 1986. 429

David B. MacQueen. Kahn networks at the dawn of functional programming. In Bertot et al. (2009), chapter 5. 183

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. 237

Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science IV*, pages 153–175. North-Holland, 1980. 39, 54

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. v, 39, 408

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, 202, 288

N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987. 136

Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978. 54

Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0. 288, 304, 382, 481

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 39, 61, 224, 304, 417, 429, 439

John C. Mitchell. Coercion and type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984. 224

John C. Mitchell. Representation independence and data abstraction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 263–276, 1986. 157, 474

John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996. v

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988. 157, 440

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989. ISBN 0-8186-1954-6. 318

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295, 2004. 318, 400

Chetan R. Murthy. An evaluation semantics for classical proofs. In *LICS*, pages 96–107. IEEE Computer Society, 1991. 119

Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP*, pages 207–218. ACM, 2003. ISBN 1-58113-705-2. 297

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/`. 440

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. 481

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. 318

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. v, 87, 224, 248, 254

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. v

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. 474

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. 461

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, 288

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, 1981. 48, 265

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. 48

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. 382, 394

J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983. 148, 474

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, 157

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. 224

John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981. 317, 329

John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993. 279

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998. v

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. 429

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. 545, 548

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. 193

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976. 183

Dana S Scott. Relating theories of the lambda calculus. *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 403–450, 1980b. 211

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. 183

Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982. 183

Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65 (2/3):85–97, 1985. 452

Guy L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition edition, 1990. 271

Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722, 2006. 417, 429

Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999. 136

P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + strategy = parallelism. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 8:23–60, 1998. 365

Jaap van Oosten. Realizability: A historical essay. *Mathematical Structures in Computer Science*, 12 (3):239–263, 2002. 237

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. 318

Philip Wadler. Call-by-value is dual to call-by-name. In Runciman and Shivers (2003), pages 189–201. ISBN 1-58113-756-7. 119

Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theor. Comput. Sci.*, 8:13–30, 1979. 183

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. 183

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. 163

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. 54, 330

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. 237

Index