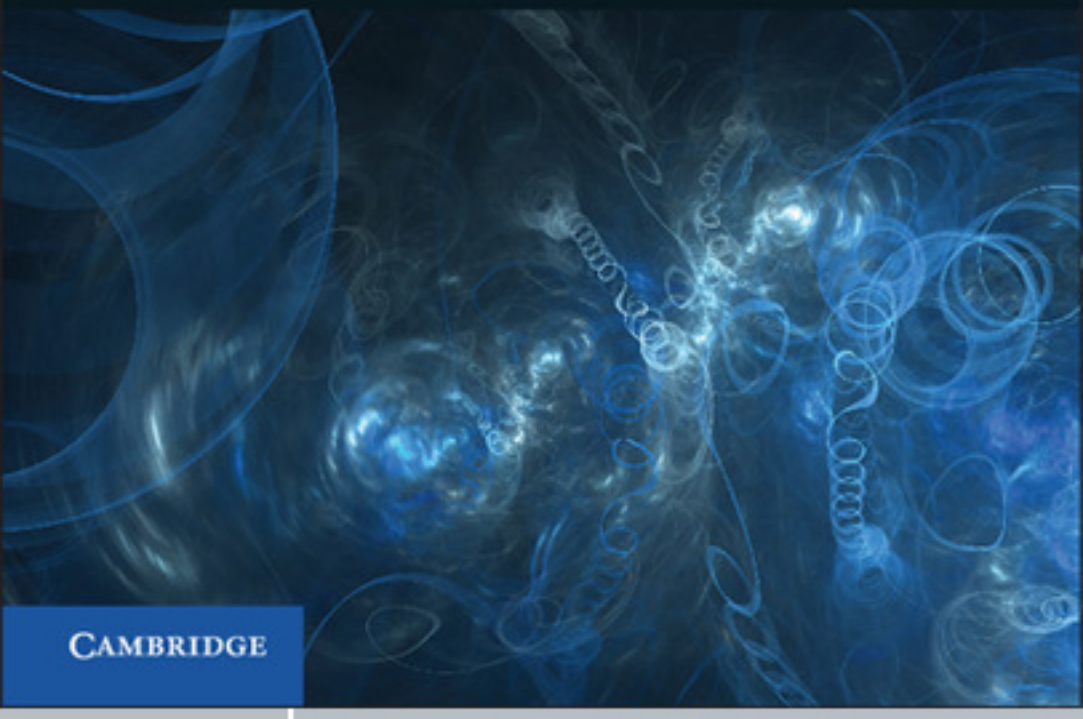
ROBERT HARPER

Practical Foundations for PROGRAMMING LANGUAGES



CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9781107029576

Practical Foundations for Programming Languages

Types are the central organizing principle of the theory of programming languages. In this innovative book, Professor Robert Harper offers a fresh perspective on the fundamentals of these languages through the use of type theory. Whereas most textbooks on the subject emphasize taxonomy, Harper instead emphasizes genetics, examining the building blocks from which all programming languages are constructed.

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.

Professor Harper's presentation is simultaneously rigorous and intuitive, relying on only elementary mathematics. The framework he outlines scales easily to a rich variety of language concepts and is directly applicable to their implementation. The result is a lucid introduction to programming theory that is both accessible and practical.

Robert Harper has been a member of the faculty of Computer Science at Carnegie Mellon University since 1988. His main research interest is in the application of type theory to the design and implementation of programming languages and to the development of systems for mechanization of mathematics. Professor Harper is a recipient of the Allen Newell Medal for Research Excellence and the Herbert A. Simon Award for Teaching Excellence at Carnegie Mellon, and he is a Fellow of the Association for Computing Machinery.

Practical Foundations for Programming Languages

Robert Harper

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Mexico City

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9781107029576

First published 2013

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data

Harper, Robert.
Practical foundations for programming languages / Robert Harper.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-107-02957-6 (hardback)
1. Programming languages (Electronic computers) I. Title.
QA76.7.H377 2012
005.13–dc23 2012018404

ISBN 978-1-107-02957-6 Hardback

Contents

Part IV Finite Data Types

Part V Infinite Data Types

Part VI Dynamic Types

Part VII Variable Types

Part XVIII Equational Reasoning

Part XIX Appendix

Preface

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, 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 Andrew Appel, Iliano Cervesato, Lin Chase, Derek Dreyer, Zhong Shao, and Todd Wilson for their extensive efforts in reading and criticizing the book. I also thank the following people for their suggestions: Arbob Ahmad, Zena Ariola, Eric Bergstrome, Guy Blelloch, William Byrd,

Luis Caires, Luca Cardelli, Manuel Chakravarty, Richard C. Cobbe, Karl Crary, Yi Dai, Daniel Dantas, Anupam Datta, Jake Donham, Favonia, Matthias Felleisen, Kathleen Fisher, Dan Friedman, Peter Gammie, Maia Ginsburg, Byron Hawkins, Kevin Hely, Justin Hsu, Cao Jing, Salil Joshi, Gabriele Keller, Scott Kilpatrick, Danielle Kramer, Akiva Leffert, Ruy Ley-Wild, Dan Licata, Karen Liu, Dave MacQueen, Chris Martens, Greg Morrisett, Tom Murphy, Aleksandar Nanevski, Georg Neis, David Neville, Doug Perkins, Frank Pfenning, Jean Pichon, Benjamin Pierce, Andrew M. Pitts, Gordon Plotkin, David Renshaw, John Reynolds, Carter Schonwald, Dale Schumacher, Dana Scott, Robert Simmons, Pawel Sobocinski, Daniel Spoonhower, Paulo Tanimoto, Peter Thiemann, Bernardo Toninho, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, Roger Wolff, Omer Zach, Luke Zarko, and Yu Zhang. I am 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 in Germany 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 grants 0702381 and 0716469. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Robert Harper
Pittsburgh
March 2012

PART I

Judgments and Rules

1 Syntactic Objects

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

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

We do not concern ourselves in this book with matters of concrete syntax, but instead work at the level of abstract syntax. To prepare the ground for the rest of the book, this chapter begins by defininining abstract syntax trees and abstract binding trees and some functions and relations associated with them. The definitions are a bit technical, but are absolutely fundamental to what follows. It is probably best to skim this chapter on first reading, returning to it only as the need arises.

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. Abstract syntax trees 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. Ast's may be combined by an *operator*, which has both a sort and an *arity*, a finite sequence of sorts specifying the number and sorts of its arguments. An operator of sort s and arity $s_1, \ldots, s_n$ combines $n \geq 0$ ast's of sort $s_1, \ldots, s_n$, into a compound ast of sort s. As a matter of terminology, a *nullary* operator is one that takes no arguments, a *unary* operator takes one, a *binary* operator takes two, and so forth.

The concept of a variable is central, and therefore deserves special emphasis. As in mathematics a variable is an *unknown* object drawn from some domain, its *range of signficance*. In school mathematics the (often implied) range of significance is the set of real numbers. Here variables range over ast's of a specified sort. Being an unknown, the meaning of a variable is given by *substitution*, the process of "plugging in" an object from the domain for the variable in a formula. So, in school, we might plug in π for x in a polynomial and calculate the result. Here we would plug in an ast of the appropriate sort for a variable in an ast to obtain another ast. The process of substitution is easily understood for ast's, because it amounts to a "physical" replacement of the variable by an ast within another ast. We subsequently consider a generalization of the concept of ast for which the substitution process is somewhat more complex, but the essential idea is the same and bears repeating: *A variable is given meaning by substitution*.

For example, consider a simple language of expressions built from numbers, addition, and multiplication. The abstract syntax of such a language would consist of a single sort Exp and an infinite collection of operators that generate the forms of expression: `num[`n`]` is a nullary operator of sort Exp whenever $n \in \mathbb{N}$; `plus` and `times` are binary operators of sort Exp whose arguments are both of sort Exp. The expression $2 + (3 \times x)$, which involves a variable x, would be represented by the ast

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

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

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

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

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

1. the property holds for any variable x of sort Exp; $\mathcal{P}(x)$;
2. the property holds for any number `num[`n`]`: For every $n \in \mathbb{N}$; $\mathcal{P}($`num[`n`]`$)$;
3. assuming the property holds for a_1 and a_2, it holds for `plus(`a_1`;`a_2`)` and `times(`a_1`;`a_2`)`: If $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}($`plus(`a_1`;`a_2`)`$)$ and $\mathcal{P}($`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.

For the sake of precision and to prepare the ground for further developments, precise definitions of the foregoing concepts are now given. Let $\mathcal{S}$ be a finite set of sorts. Let $\{ O_s \}_{s \in \mathcal{S}}$ be a sort-indexed family of *operators* o of sort s with arity $\mathsf{ar}(o) = (s_1, \ldots, s_n)$. Let $\{ \mathcal{X}_s \}_{s \in \mathcal{S}}$

be a sort-indexed family of *variables* x of each sort s. The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s\in\mathcal{S}}$ of ast's of sort s is defined as follows:

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

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

1. if $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$, and
2. if $o \in \mathcal{O}_s$ and $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, then if $\mathcal{P}_{s_1}(a_1)$ 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 if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

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

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

1. $[b/x]x = b$ 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 for ast's 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. □

In most cases it is possible to enumerate all of the operators that generate the ast's of a sort up front, as we have done in the foregoing examples. However, in some situations this is not possible—certain operators are available only within certain contexts. In such

cases we cannot fix the collection of operators $\mathcal{O}$ in advance, but rather must allow it to be *extensible*. This is achieved by considering families of operators that are indexed by symbolic *parameters* that serve as "names" for the instances. For example, in Chapter 34 we consider a family of nullary operators $\texttt{cls}[u]$, where u is a symbolic parameter drawn from the set of *active* parameters. It is essential that distinct parameters determine distinct operators: If u and v are active parameters and $u \neq v$, then $\texttt{cls}[u]$ and $\texttt{cls}[v]$ are different operators. Extensibility is achieved by introducing new active parameters. So, if u is not active, then $\texttt{cls}[u]$ makes no sense, but if u becomes active, then $\texttt{cls}[u]$ is a nullary operator.

Parameters are easily confused with variables, but they are fundamentally different concepts. As noted earlier, a variable stands for an unknown ast of its sort, but a parameter *does not stand for anything*. It is a purely symbolic identifier whose only significance is whether it is the same as or different from another parameter. Whereas variables are given meaning by substitution, it is not possible, or sensible, to substitute for a parameter. As a consequence, disequality of parameters is preserved by substitution, whereas disequality of variables is not (because the same ast may be substituted for two distinct variables).

To account for the set of active parameters, the set $\mathcal{A}[\mathcal{U};\mathcal{X}]$ is the set of ast's with variables drawn from $\mathcal{X}$ and with parameters drawn from $\mathcal{U}$. Certain operators, such as $\texttt{cls}[u]$, are *parameterized* by parameters u of a given sort. The parameters are distinguished from the arguments by the square brackets around them. Instances of such operators are permitted only for parameters drawn from the active set $\mathcal{U}$. So, for example, if $u \in \mathcal{U}$, then $\texttt{cls}[u]$ is a nullary operator, but if $u \notin \mathcal{U}$, then $\texttt{cls}[u]$ is not a valid operator. The next section introduces the means of extending $\mathcal{U}$ to make operators available within that context.

1.2 Abstract Binding Trees

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

`let` x `be` 7 `in` $x + x$, occurrences of x in the addition refer to the variable introduced by the `let`. However, in the expression `let` x `be` $x * x$ `in` $x + x$, occurrences of x within the multiplication refer to a variable different from those occurring within the addition. The latter occurrences refer to the binding introduced by the `let`, whereas the former refer to some outer binding not displayed here.

The names of bound variables are immaterial insofar as they determine the same binding. So, for example, the expression `let` x `be` $x * x$ `in` $x + x$ could just as well have been written as `let` y `be` $x * x$ `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. The expression `let` x `be` $y * y$ `in` $x + x$ has a different meaning from 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 `let` x `be` 2 `in` `let` y `be` 3 `in` $x + x$ has a different meaning from the expression `let` y `be` 2 `in` `let` y `be` 3 `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 may be enriched to account for binding and scope of a variable. These enriched ast's are called abstract binding trees. Abt's generalize ast's by allowing an operator to bind any finite number (possibly zero) of variables in each argument position. An argument to an operator is called an *abstractor* and has the form $x_1, \ldots, x_k\,.\,a$. The sequence of variables $x_1, \ldots, x_k$ is 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 `let` x `be` a_1 `in` a_2 has the form `let`$(a_1; x\,.\,a_2)$, which more clearly specifies that the variable x is bound within a_2 and not within a_1. We often write $\vec{x}$ to stand for a finite sequence $x_1, \ldots, x_n$ of distinct variables and write $\vec{x}\,.\,a$ to mean $x_1, \ldots, x_n\,.\,a$.

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

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

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

The definition expression `let` x `be` $2 + 2$ `in` $x \times x$ is represented by the abt

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

Let $\mathcal{O}$ be a sort-indexed family of operators o with arities $\mathsf{ar}(o)$. For a given sort-indexed family $\mathcal{X}$ of variables, the sort-indexed family of abt's $\mathcal{B}[\mathcal{X}]$ is defined similarly to $\mathcal{A}[\mathcal{X}]$,

except that the set of active variables changes for each argument according to which variables are bound within it. A first cut at the definition is as follows:

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

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

This definition is *almost* correct, but fails to properly account for the behavior of bound variables. An abt of the form $\mathtt{let}(a_1; x\,.\,\mathtt{let}(a_2; x\,.\,a_3))$ is ill-formed according to this definition, because the first binding adjoins x to $\mathcal{X}$, which implies that the second cannot also adjoin x to $\mathcal{X}$, x without causing confusion. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names. This is achieved by altering the second clause of the definition using renaming as follows:[1]

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

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

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

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

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

As an example let us define the 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 ensure conditions that this is the case:

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

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, independent of the choice

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

of bound variable names in that argument, then it is free in the overall abt. This implies, in particular, that x is *not* free in $\texttt{let(zero;}x\,.\,x\texttt{)}$.

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

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

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

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

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

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

One way around this is to alter the definition of substitution so that the bound variables in the result are chosen fresh by substitution. By the principle of structural induction we know inductively that, for any renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$ with $\vec{x}'_i$ fresh, the substitution $[b/x](\pi_i \cdot a_i)$ is well-defined. Hence we may define

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

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

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

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

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

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

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

Parameters, as well as variables, may be bound within an argument of an operator. Such binders introduce a "new" or "fresh" parameter within the scope of the binder wherein it may be used to form further abt's. To allow for parameter declaration, the valence of an argument is generalized to indicate the sorts of the parameters bound within it, as well as the sorts of the variables, by writing $(\vec{s}_1;\vec{s}_2)s$, where $\vec{s}_1$ specifies the sorts of the parameters and $\vec{s}_2$ specifies the sorts of the variables. The sort-indexed family $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is the set of abt's determined by a fixed set of operators using the parameters $\mathcal{U}$ and the variables $\mathcal{X}$. We rely on naming conventions to distinguish parameters from variables, reserving u and v for parameters and x and y for variables.

1.3 Notes

The concept of abstract syntax has its orgins in the pioneering work of Church, Turing, and Gödel, who first considered the possibility of writing programs that act on representations of programs. Originally programs were represented by natural numbers, using encodings, now called *Gödel numberings*, based on the prime factorization theorem. Any standard text on mathematical logic, such as that by Kleene (1952), contains 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).

2 Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. This chapter develops the basic framework of inductive definitions and gives 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 syntactic objects of a specified 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 a syntactic object. We make use of many forms of judgment, including examples such as these:

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

A judgment states that one (or more) syntactic object has a property or stands in some relation to another. The property or relation itself is called a *judgment form*, and the judgment that an object (or objects) has that property or stands 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 for the judgment asserting that J holds for a. When it is not important to stress the subject of the judgment, we write J to stand for an unspecified judgment. For particular judgment forms, we freely use prefix, infix, or mixfix notation, as illustrated by the preceding 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 may be read as stating that the premises are *sufficient* for the conclusion: To show J, it is enough to show $J_1, \ldots, J_k$. When k is zero, a rule states that its conclusion holds unconditionally. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

For example, the following rules constitute an inductive definition of the judgment a `nat`:

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

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

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

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

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

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

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

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

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

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

In each of the preceding examples we have made use of a notational convention for specifying an infinite family of rules by a finite number of patterns, or *rule schemes*. For example, Rule (2.2b) is a rule scheme that determines one rule, called an *instance* of the rule scheme, for each choice of object a in the rule. We 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 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 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 may be thought of as a tree in which each node is a rule whose children are derivations of its premises. We sometimes say that a derivation of J is evidence for the validity of an inductively defined 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

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

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

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

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

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

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

Similarly, here is a derivation of a `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 toward the desired conclusion, whereas backward chaining starts with the desired conclusion and works backward toward the axioms.

More precisely, a 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, judgmnts 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 may 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 closed under a collection of rules, we may reason about them by *rule induction*. The principle of rule induction states that to show that a property $\mathcal{P}$ holds of a judgment J whenever J is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining J. Writing $\mathcal{P}(J)$ to mean that the property $\mathcal{P}$ holds of the judgment J, we say that $\mathcal{P}$ respects the rule

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

if $\mathcal{P}(J)$ holds whenever $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$. The assumptions $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$ are called the *inductive hypotheses*, and $\mathcal{P}(J)$ is called the *inductive conclusion* of the inference.

The principle of rule induction is simply the expression of the definition of an inductively defined judgment form as the *strongest* judgment form closed under the rules comprising the definition. This means that 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 $\mathcal{P}(a\ \mathsf{nat})$ whenever $a\ \mathsf{nat}$, it is enough to show that

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

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

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

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

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

We may also show by rule induction that the predecessor of a natural number is also a natural number. Although this may seem self-evident, the point of the example is to show how to derive this from first principles.

Lemma 2.1. *If* $\mathtt{succ}(a)\ \mathsf{nat}$*, then* $a\ \mathsf{nat}$.

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

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

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

This completes the proof. □

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

Lemma 2.2. *If* $a\ \mathsf{nat}$*, then* $a = a\ \mathsf{nat}$.

Proof By rule induction on Rules (2.2):

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

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

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

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

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 a list, stating 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 may 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:

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

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

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

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

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

As a simple example, we may use simultaneous rule induction to prove that (1) if a even, then a nat, and (2) if a odd, then a nat. That is, we define the property $\mathcal{P}$ by (1) $\mathcal{P}(a\ \mathsf{even})$ iff a nat, and (2) $\mathcal{P}(a\ \mathsf{odd})$ iff a nat. The principle of rule induction for Rules (2.8) states that it is sufficient to show the following facts:

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

2.6 Defining Functions by Rules

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

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

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

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 $c = c'$ nat.

For existence, let $\mathcal{P}(a\ \mathsf{nat})$ be the proposition *if* $b\ \mathsf{nat}$ *then there exists* $c\ \mathsf{nat}$ *such that* $\mathsf{sum}(a;b;c)$. We prove that if $a\ \mathsf{nat}$ then $\mathcal{P}(a\ \mathsf{nat})$ by rule induction on Rules (2.2). We have two cases to consider:

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

Rule (2.2b). Assuming $\mathcal{P}(a\ \mathsf{nat})$, we are to show that $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$. That is, we assume that if $b\ \mathsf{nat}$ then there exists c such that $\mathsf{sum}(a;b;c)$, and we are to show that if $b'\ \mathsf{nat}$, then there exists c' such that $\mathsf{sum}(\mathtt{succ}(a);b';c')$. To this end, suppose that $b'\ \mathsf{nat}$. Then by induction there exists c such that $\mathsf{sum}(a;b';c)$. Taking $c' = \mathtt{succ}(c)$ 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* $c_1 = c_2\ \mathsf{nat}$ by rule induction based on Rules (2.9).

Rule (2.9a). We have $a = \mathtt{zero}$ and $c_1 = b$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(\mathtt{zero};b;c_2)$, then c_2 is b. By Lemma 2.2 we obtain $b = b\ \mathsf{nat}$.

Rule (2.9b). We have that $a = \mathtt{succ}(a')$ and $c_1 = \mathtt{succ}(c_1')$, where $\mathsf{sum}(a';b;c_1')$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(a;b;c_2)$, then $c_2 = \mathtt{succ}(c_2')\ \mathsf{nat}$, where $\mathsf{sum}(a';b;c_2')$. By the outer inductive hypothesis $c_1' = c_2'\ \mathsf{nat}$ and so $c_1 = c_2\ \mathsf{nat}$. □

2.7 Modes

The statement that one (or more) argument of a judgment is (perhaps uniquely) determined by its other arguments is called a *mode specification* for that judgment. For example, we have shown that every two natural numbers have a sum according to Rules (2.9). This fact may be restated as a mode specification by saying that the judgment $\mathsf{sum}(a;b;c)$ has *mode* $(\forall, \forall, \exists)$. The notation arises from the form of the proposition it expresses: *For all* $a\ \mathsf{nat}$ *and for all* $b\ \mathsf{nat}$*, there exists* $c\ \mathsf{nat}$ *such that* $\mathsf{sum}(a;b;c)$. If we wish to further specify that c is *uniquely* determined by a and b, we would say that the judgment $\mathsf{sum}(a;b;c)$ has mode $(\forall, \forall, \exists!)$, corresponding to the proposition that *for all* $a\ \mathsf{nat}$ *and for all* $b\ \mathsf{nat}$*, there exists a unique* $c\ \mathsf{nat}$ *such that* $\mathsf{sum}(a;b;c)$. If we wish to specify only that the sum is unique, *if it exists*, then we would say that the addition judgment has mode $(\forall, \forall, \exists^{\leq 1})$, corresponding to the proposition *for all* $a\ \mathsf{nat}$ *and for all* $b\ \mathsf{nat}$ *there exists at most one* $c\ \mathsf{nat}$ *such that* $\mathsf{sum}(a;b;c)$.

As these examples illustrate, a given judgment may satisfy several different mode specifications. In general the universally quantified arguments are to be thought of as the *inputs* of the judgment, and the existentially quantified arguments are to be thought of as its *outputs*. We usually try to arrange things so that the outputs come after the inputs, but it is not essential that we do so. For example, addition also has the mode $(\forall, \exists^{\leq 1}, \forall)$, stating that the sum and the first addend uniquely determine the second addend, if there is any such addend

at all. Put in other terms, this says that addition of natural numbers has a (partial) inverse, namely subtraction. We could equally well show that addition has mode $(\exists^{\leq 1}, \forall, \forall)$, which is just another way of stating that addition of natural numbers has a partial inverse.

Often there is an intended, or *principal*, mode of a given judgment, which we often foreshadow by our choice of notation. For example, when giving an inductive definition of a function, we often use equations to indicate the intended input and output relationships. For example, we may restate the inductive definition of addition [given by Rules (2.9)] using the following equations:

$$\frac{a\ \mathsf{nat}}{a + \mathtt{zero} = a\ \mathsf{nat}} \tag{2.10a}$$

$$\frac{a + b = c\ \mathsf{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c)\ \mathsf{nat}}\,. \tag{2.10b}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgment is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse notation, writing $a + b$ for the unique c such that $a + b = c\ \mathsf{nat}$.

2.8 Notes

Aczel (1977) provides a thorough account of the theory of inductive definitions. The formulation given here is strongly influenced by Martin-Löf's development of the logic of judgments (Martin-Löf, 1983, 1987).

3 Hypothetical and General Judgments

A *hypothetical judgment* expresses an entailment between one or more hypotheses and a conclusion. We consider two notions of entailment, called *derivability* and *admissibility*. Both enjoy the same structural properties, but they differ in that derivability is stable under extension with new rules, admissibility is not. A *general judgment* expresses the universality, or generality, 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 hypothesis. 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 additional 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* $\mathcal{R}[J_1, \ldots, J_k]$ of the rules $\mathcal{R}$ with the additional axioms

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

We treat the *hypotheses*, or *antecedents*, of the judgment $J_1, \ldots, J_n$ 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, frequently Γ or Δ, to stand for a finite collection of basic judgments, and we write $\mathcal{R}[\Gamma]$ 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 $\mathcal{R}[\Gamma]$, 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

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

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

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

that comprises 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

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

is derivable from Rules (2.2).

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

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

Proof Any derivation of J from $\mathcal{R}[\Gamma]$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}')[\Gamma]$, 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 unexercised options.

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

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

3.1.2 Admissibility

Admissibility, written as $\Gamma \models_{\mathcal{R}} J$, 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 $\mathcal{R}$ whenever the assumptions Γ are all derivable from rules $\mathcal{R}$. In particular if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the 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 $\mathcal{R}$. This means that given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may construct a derivation of J using the rules in $\mathcal{R}$.

For example, the admissibility judgment

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

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

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

is admissible for Rules (2.2).

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

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

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

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

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

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

To see that the converse fails, observe that there is no composition of rules such that

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

yet the admissibility judgment

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

holds vacuously.

Evidence for admissibility may be thought of as a mathematical function transforming derivations $\nabla_1, \ldots, \nabla_n$ of the hypotheses into a derivation ∇ of the consequent. Therefore the admissibility 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 also K are derivable from the original rules: If $\Gamma \models_{\mathcal{R}} J$, then $\Gamma, K \models_{\mathcal{R}} J$.

Transitivity. If $\Gamma, K \models_{\mathcal{R}} J$ and $\Gamma \models_{\mathcal{R}} K$, then $\Gamma \models_{\mathcal{R}} J$. If the 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 $\mathcal{R}, r$ may be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to show that $\mathcal{P}(J)$ whenever $\vdash_{\mathcal{R},r} J$, it is sufficient to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$ alone. That is, we need only consider the rules $\mathcal{R}$ in a proof by rule induction to derive $\mathcal{P}(J)$.

3.2 Hypothetical Inductive Definitions

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

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

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

The hypotheses Γ are the global hypotheses of the rule, and the hypotheses Γ_i are the local hypotheses of the ith premise of the rule. Informally, this rule states that J is a derivable consequence of Γ whenever each J_i is a derivable consequence of Γ, augmented with the additional hypotheses Γ_i. Thus one way to show that J is derivable from Γ is to show, in

turn, that each J_i is derivable from $\Gamma\,\Gamma_i$. The derivation of each premise involves a "context switch" in which we extend the global hypotheses with the local hypotheses of that premise, establishing a new set of global hypotheses for use within that derivation.

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

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

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

A hypothetical inductive definition is to be regarded as an ordinary inductive definition of a *formal derivability judgment* $\Gamma \vdash J$ consisting of a finite set of basic judgments Γ and a basic judgment J. A collection of hypothetical rules $\mathcal{R}$ defines the strongest formal derivability judgment that is *structural* and *closed* under 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. By a slight abuse of notation we write $\Gamma \vdash_{\mathcal{R}} J$ to indicate 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 $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under both the rules of $\mathcal{R}$ and under the structural rules. Thus, for each rule of the form (3.10), whether structural or in $\mathcal{R}$, we must show that

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

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

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

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 collection of

objects. A *generic* judgment expresses that a judgment holds for any choice of objects replacing designated variables in the judgment. Another form of general judgment codifies the handling of parameters. A *parametric* judgment expresses generality over any choice of fresh renamings of designated parameters of a judgment. To keep track of the active variables and parameters in a derivation, we write $\Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ to indicate that J is derivable from Γ according to rules $\mathcal{R}$, with objects consisting of abt's over parameters $\mathcal{U}$ and variables $\mathcal{X}$.

Generic derivability judgment is defined by

$$\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J,$$

where the quantification is restricted to variables $\vec{x}'$ not already active in $\mathcal{X}$. Evidence for generic derivability consists of a *generic derivation* ∇ involving the variables $\vec{x}$ such that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the derivation $\pi \cdot \nabla$ is evidence for $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$. The renaming ensures that the variables in the generic judgment are fresh (not already declared in $\mathcal{X}$) and that the meaning of the judgment is not dependent on the choice of variable names.

For example, the generic derivation ∇

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

is evidence for the judgment

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

This is because every fresh renaming of x to y in ∇ results in a valid derivation of the corresponding renaming of the indicated judgment.

The generic derivability judgment enjoys the following *structural properties* governing the behavior of variables:

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

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

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

(It is left implicit in the principle of substitution that sorts are to be respected in that the substituting object must be of the same sort as the variable that is being substituted.) 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. Substitution holds as long as the rules themselves are closed under substitution. This need not be the case, but in practice this requirement is usually met.

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

$$\vec{u}; \vec{x} \mid \Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \rho : \vec{u} \leftrightarrow \vec{u}'\ \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \rho \cdot \pi \cdot \Gamma \vdash^{\mathcal{U},\vec{u}';\mathcal{X},\vec{x}'}_{\mathcal{R}} \rho \cdot \pi \cdot J.$$

Evidence for parametric derivability consists of a derivation ∇ involving the parameters $\vec{u}$ and variables $\vec{x}$, each of whose fresh renamings is a derivation of the corresponding renaming of the underlying hypothetical judgment.

Recalling from Chapter 1 that parameters admit disequality, we cannot expect any substitution principle for parameters to hold of a parametric derivability. It does, however, validate the structural properties of proliferation and renaming because the presence of additional parameters does not affect the formation of an abt, and parametric derivability is defined to respect all fresh renamings of parameters.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The admissibility of Rule (3.14a) is, in practice, ensured by explicitly including it. The admissibility of Rules (3.14b) and (3.14c) is ensured if each of the generic rules is uniform, as we may assimilate the additional parameter x to the global parameters and the additional hypothesis J to the global hypotheses. The admissibility of Rule (3.14d) is ensured by the identification convention for the formal generic 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 $\vec{u}; \vec{x} \mid \Gamma \vdash J$, with parameters $\vec{u}$ as well as variables $\vec{x}$. Such formal judgments are identified up to the renaming of their variables and their parameters to ensure that the meaning is independent of the choice of variable and parameter names.

3.5 Notes

The concepts of entailment and generality are fundamental to logic and programming languages. The formulation given here builds on the work of Martin-Löf (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 permit arbitrarily nested combinations of hypothetical and general judgments, whereas the present account considers only general hypothetical judgments over basic judgment forms.

The failure to distinguish parameters from variables is the source of many errors in language design. The crucial distinction is that whereas it makes sense to distinguish cases based on whether two parameters are the same or distinct, it makes no sense to do so for variables, because disequality is not preserved by substitution. Adhering carefully to this distinction avoids much confusion and complication in language design (see, for example, Chapter 41).

PART II

Statics and Dynamics

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 collection 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 accurate; if not, the statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

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

4.1 Syntax

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

We accomplish both of these purposes with a *syntax chart*, whose meaning is best illustrated by example. The following chart summarizes the abstract and concrete syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$:

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

This chart defines two sorts, Typ, ranged over by τ, and Exp, ranged over by e. This chart also defines a collection of operators and their arities. For example, the operator `let` has arity (Exp, (Exp)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 $\mathcal{L}\{\texttt{num str}\}$ consists of an inductive definition of generic hypothetical judgments of the form

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

where $\vec{x}$ is a finite set of variables and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of variables, using the letters x and y for variables that serve as parameters of the typing judgment. We write $\mathrm{dom}(\Gamma)$ to indicate that there is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

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

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

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

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

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

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

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

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

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

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 inverts 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 parameters we may assume z is chosen such that $z \notin dom(\Gamma)$ and $z \neq x$. By induction we have

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

from which the result follows by Rule (4.1h). □

Lemma 4.4 (Substitution)**.** *If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$.*

Proof By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We again consider only Rule (4.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where z may be chosen so that $z \neq x$ and $z \notin dom(\Gamma)$. 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

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

It follows by Rule (4.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 4.3 allows us to use an expression in any context that binds its free variables: If e is well-typed in a context Γ, then we may "import" it into any context that includes the assumptions Γ. In other words the introduction of new variables beyond those required by an expression e does not invalidate e itself; it remains well-formed, with the same type.[1] More significantly, Lemma 4.4 expresses the concepts of *modularity* and *linking*. We may think of expressions e and e' as two *components* of a larger system in which component e' is to be thought of as a *client* of the *implementation* e. The client declares a variable specifying the type of the implementation and is type checked knowing only this information. The implementation must be of the specified type in order to satisfy the assumptions of the client. If so, then we may link them to form the composite system, $[e/x]e'$. This may itself be the client of another component, represented by a variable y, that is replaced by that component during linking. When all such variables have been implemented, the result is a *closed expression* that is ready for execution (evaluation).

The converse of Lemma 4.4 is called *decomposition*. It states that any (large) expression may be decomposed into a client and implementor by introducing a variable to mediate their interaction.

Lemma 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. □

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

The statics of $\mathcal{L}\{\texttt{num str}\}$ given by Rules (4.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introductory* and the *eliminatory*. The introductory forms for a type determine the *values*, or *canonical forms*, of that type. The eliminatory forms determine how to manipulate the values of a type to

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

form a computation of another (possibly the same) type. In $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the introductory forms for the type `num` are the numerals and those for the type `str` are the literals. The eliminatory forms for the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification becomes apparent once we have defined the dynamics of the language in Chapter 5. Then we see that the eliminatory forms are *inverse* to the introductory forms in that they "take apart" what the introductory forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 6.

4.4 Notes

The concept of the static semantics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. The concept of static semantics in the sense considered here was introduced in the definition of the Standard ML programming language (Milner et al., 1997), building on much earlier work by Church and others on the typed λ-calculus (Barendregt, 1992). The concept of introduction and elimination and the associated inversion principle were 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 (1980, 1984).

5 Dynamics

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

In practice we always arrange things so that no transition is possible from a final state: If s final, then there is no s' state such that $s \mapsto s'$. A state from which no transition is possible is sometimes said to be *stuck*. Whereas all final states are, by convention, stuck, there may be stuck states in a transition system that are not final. A transition system is *deterministic* iff for every state s there exists at most one state s' such that $s \mapsto s'$; otherwise it is *nondeterministic*.

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Structural Dynamics

A *structural dynamics* for $\mathcal{L}\{\texttt{num str}\}$ is given by a transition system whose states are closed expressions. All states are initial. The final states are the *(closed) values*, which represent the completed computations. The judgment e val, which states that e is a value, is inductively defined by the following rules:

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

$$\frac{}{\texttt{str[}s\texttt{]} \text{ val}}. \tag{5.3b}$$

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

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

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

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

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

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

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

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

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

We have omitted rules for multiplication and computing the length of a string, which follows 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 in which instructions are executed.

The bracketed rule, Rule (5.4g), and bracketed premise on Rule (5.4h), are to be included for a *by-value* interpretation of `let` and omitted for a *by-name* interpretation. The by-value intepretation 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:

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

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

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

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

The principle of rule induction for the structural dynamics of $\mathcal{L}\{\texttt{num str}\}$ states that to show $\mathcal{P}(e \mapsto e')$ whenever $e \mapsto e'$, it is sufficient to show that $\mathcal{P}$ is closed under Rules (5.4).

For example, we may show by rule induction that structural dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is *determinate*, which means that an expression may make a 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 $e \mapsto e'$ for some e'.*

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

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

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

Rules (5.4) exemplify the *inversion principle* of language design, which states that the eliminatory forms are *inverse* to the introductory forms of a language. The search rules determine the *principal arguments* of each eliminatory form, and the instruction rules specify how to evaluate an eliminatory form when all of its principal arguments are in introductory form. For example, Rules (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, it is 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, $e_1 \to e_2$, for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string:

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

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

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

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

$$\frac{}{\circ \text{ ectxt}} \tag{5.6a}$$

$$\frac{\mathcal{E}_1 \text{ ectxt}}{\texttt{plus}(\mathcal{E}_1; e_2) \text{ ectxt}} \tag{5.6b}$$

$$\frac{e_1 \text{ val} \quad \mathcal{E}_2 \text{ ectxt}}{\texttt{plus}(e_1; \mathcal{E}_2) \text{ ectxt}}. \tag{5.6c}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here," at the point of the occurrence of the hole. The remaining rules correspond one-to-one to the search rules of the structural dynamics. For example, Rule (5.6c) states that in an expression $\texttt{plus}(e_1; e_2)$, if the first argument e_1 is a value, then the next instruction step, if any, lies at or within the second argument e_2.

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

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

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

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

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

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

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \to e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'}. \tag{5.8}$$

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 dynamics and the contextual dynamics define the same transition relation. For the sake of the proof, let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural dynamics [Rules (5.4)], and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual dynamics [Rule (5.8)].

Theorem 5.4. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof From left to right, proceed by rule induction on Rules (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 $e_0 \to e_0'$.

For example, for Rule (5.4a), take $\mathcal{E} = \circ$, and observe that $e \to e'$. 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 $e_0 \to e_0'$. Take $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, and observe that $e = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0\}$ and $e' = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0'\}$ with $e_0 \to e_0'$.

From right to left, observe that if $e \mapsto_{\mathtt{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \to e_0'$. We prove by induction on Rules (5.7) that $e \mapsto_{\mathtt{s}} e'$. For example, for Rule (5.7a), e_0 is e, e_0' is e', and $e \to e'$. Hence $e \mapsto_{\mathtt{s}} e'$. 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 $e_1 \mapsto_{\mathtt{s}} e_1'$. Therefore e is $\mathtt{plus}(e_1; e_2)$, e' is $\mathtt{plus}(e_1'; e_2)$, and therefore by Rule (5.4b), $e \mapsto_{\mathtt{s}} e'$. □

Because the two transition judgments coincide, contextual dynamics may be seen as an alternative way of presenting 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:

$$\frac{e_0 \to e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}. \tag{5.9}$$

This formulation is superficially simpler in that it does not make explicit how an expression is to be decomposed into an evaluation context and a reducible expression. The deeper advantage of contextual dynamics is that all transitions are between complete programs. We need never consider a transition between expressions of any type other than the ultimate observable type. This simplifies certain arguments, notably the proof of Lemma 48.16.

5.4 Equational Dynamics

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

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where Γ consists of one assumption of the form $x : \tau$ for each $x \in \mathcal{X}$. We only consider definitional equality of well-typed expressions, so that when considering the 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 parameters $\mathcal{X}$ when they can be determined from the forms of the assumptions Γ.

Definitional equality of expressions in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ under the by-name interpretation of `let` is inductively defined by the following rules:

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

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

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash \texttt{let}(e_1; x\,.\,e_2) \equiv [e_1/x]e_2 : \tau}\,. \tag{5.10i}$$

Rules (5.10a)–(5.10c) state that definitional equality is an *equivalence relation*. Rules (5.10d)–(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)–(5.10i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. For the sake of concision, Rules (5.10) may be characterized as defining the *strongest congruence* closed under Rules (5.10g), (5.10h), and (5.10i).

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

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

by applying Rules (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

$$x : \texttt{num}, y : \texttt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \texttt{num}, \tag{5.11}$$

which, intuitively, expresses the commutativity of addition. Although we do 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 47.)

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

Theorem 5.5. *$e \equiv e' : \tau$ iff there exists e_0 val such that $e \mapsto^* e_0$ and $e' \mapsto^* e_0$.*

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

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

then there exists e_0 val such that

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

and

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

This is proved by rule induction on Rules (5.10). □

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 is

used 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.

6 Type Safety

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

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

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

Theorem 6.1 (Type Safety)**.**

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

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

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

6.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num str}\}$ 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* $e \mapsto e'$*, then* $e' : \tau$*.*

Proof Two cases are considered, leaving the rest to the reader. Consider Rule (5.4b),

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

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

Now consider Rule (5.4h),

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

Assume that $\texttt{let}(e_1;x\,.\,e_2) : \tau_2$. By inversion lemma 4.2, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By 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 $a + b = c$ nat, then c nat. □

The proof of preservation is naturally structured as an induction on the transition judgment, as 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, in some cases, 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 $e \mapsto e'$.*

Proof The proof proceeds by induction on the typing derivation. We 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 $e_1 \mapsto e_1'$. In the latter case it follows that $\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1'; e_2)$, as required. In the former we also have by induction that either e_2 val or there exists e_2' such that $e_2 \mapsto e_2'$. In the latter case we have that $\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1; e_2')$, as required. In the former, we have, by the canonical forms Lemma 6.3, $e_1 = \texttt{num}[n_1]$ and $e_2 = \texttt{num}[n_2]$, and hence

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

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 may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

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

6.3 Run-Time Errors

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

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

But the expression $\texttt{div}(\texttt{num}[3]; \texttt{num}[0])$ 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, feasible, but the most common approach is the second. The first requires that the type checker prove that an expression is nonzero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs as ill-formed. This is because we cannot reliably predict statically whether an expression will turn out to be nonzero when executed (because this is an undecidable property). We therefore consider the second approach, which is typical of current practice.

The general idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure

that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamics need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. However, 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 appear in a full inductive definition of this judgment:

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

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

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

Rule (6.1a) signals an error condition for division by zero. The other rules propagate this error upward: If an evaluated subexpression 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 static and dynamic semantics:

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

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

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

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

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

6.4 Notes

The concept of type safety as it is understood today 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. This analysis is replaced by the progress theorem given here, which relies on the concept of canonical forms introduced by Martin-Löf (1980).

7 Evaluation Dynamics

In Chapter 5 we defined the evaluation of expressions in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using the method of structural dynamics. This approach is useful as a foundation for proving properties of a language, but other methods are often more appropriate for other purposes, such as writing user manuals. Another method, called evaluation dynamics, presents the dynamics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Evaluation dynamics suppresses the step-by-step details of determining the value of an expression, and hence does not provide any useful notion of the time complexity of a program. Cost dynamics rectifies this by augmenting evaluation dynamics with a *cost measure*. Various cost measures may be assigned to an expression. One example is the number of steps in the structural dynamics required for an expression to reach a value.

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

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

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

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

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

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

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

Rule (7.1e) 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 $\mathcal{P}(\texttt{num[}n\texttt{]} \Downarrow \texttt{num[}n\texttt{]})$.
2. Show that $\mathcal{P}(\texttt{str[}s\texttt{]} \Downarrow \texttt{str[}s\texttt{]})$.
3. Show that $\mathcal{P}(\texttt{plus}(e_1; e_2) \Downarrow \texttt{num[}n\texttt{]})$, if $\mathcal{P}(e_1 \Downarrow \texttt{num[}n_1\texttt{]})$, $\mathcal{P}(e_2 \Downarrow \texttt{num[}n_2\texttt{]})$, and $n_1 + n_2 = n$ nat.
4. Show that $\mathcal{P}(\texttt{cat}(e_1; e_2) \Downarrow \texttt{str[}s\texttt{]})$, if $\mathcal{P}(e_1 \Downarrow \texttt{str[}s_1\texttt{]})$, $\mathcal{P}(e_2 \Downarrow \texttt{str[}s_2\texttt{]})$, and $s_1 \hat{} s_2 = s$ str.
5. Show that $\mathcal{P}(\texttt{let}(e_1; x\,.\,e_2) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e 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 (7.1e) 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 $\mathcal{L}\{\texttt{num str}\}$. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The 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 suggests that the appropriate correspondence is between *complete* execution sequences in the structural dynamics and the evaluation judgment in the evaluation dynamics. (We consider only numeric expressions, but analogous results hold also for string-valued expressions.)

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

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

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

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 $e_1 \mapsto^* \texttt{num}[n_1]$ and $e_2 \mapsto^* \texttt{num}[n_2]$. We reason as follows:

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

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

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

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

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

7.3 Type Safety, Revisited

Theorem 6.1 states that a language is safe iff it satisfies both preservation and progress. This formulation depends critically on the use of a transition system to specify the dynamics. But what if we had instead specified the dynamics as an evaluation relation, instead of using a transition system? Can we state and prove safety in such a setting?

The answer, unfortunately, is that we cannot. Although there is an analogue 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 $v : \tau$. This 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 $\mathcal{L}\{\texttt{num str}\}$, it demands much more than just progress—it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num str}\}$ were extended to admit operations that may result in an error (as discussed in Section 6.3) or to admit nonterminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude toward this situation is to simply conclude that type safety cannot be properly discussed in the context of an evaluation dynamics, but only by reference to

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

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. Restated 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 solely to prove that it cannot arise! Nevertheless, we press on to show how a semblance of type safety can be established by using evaluation dynamics.

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

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

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

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

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

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

Apart from the inconvenience of having to define the judgment $e\Uparrow$ only to show that it is irrelevant for well-typed programs, this approach suffers a very significant methodological weakness. If we should omit one or more rules defining the judgment $e\Uparrow$, 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. Execution is therefore 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 complexity. 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:

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

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

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

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

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

For a by-value interpretation of `let`, Rule (7.4e) should be 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 $e \mapsto^k v$.*

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

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 semantics is that it directly defines the relation of interest, that between a program and its outcome. The disadvantage is that it is not as well-suited to metatheory as structural semantics, precisely because it glosses over the fine structure of computation. The concept of a cost dynamics was introduced by Blelloch and Greiner (1996) in their study of parallelism (discussed further in Chapter 39).

PART III

Function Types

8 Function Definitions and Values

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

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

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

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

8.1 First-Order Functions

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

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

The expression $\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e)$ binds the function name f within e to the pattern $x_1.e_2$, which has parameter x_1 and definition e_2. The domain and range of the function are the types τ_1 and τ_2, respectively. The expression $\texttt{call}[f](e)$ instantiates the binding of f with the argument e.

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

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

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

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

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e) : \tau} \tag{8.1a}$$

$$\frac{\Gamma \vdash f(\tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{call}[f](e) : \tau_2}. \tag{8.1b}$$

Function substitution, written as $[\![x.e/f]\!]e'$, is defined by induction on the structure of e', much like the definition of ordinary substitution. However, a function name f is not a form of expression, but rather can occur only in a call of the form $\texttt{call}[f](e)$. Function substitution for such expressions is defined by the following rule:

$$\frac{}{[\![x.e/f]\!]\texttt{call}[f](e') = \texttt{let}([\![x.e/f]\!]e';x.e)}. \tag{8.2}$$

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

Lemma 8.1. *If* $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1.e_2/f]\!]e : \tau$.

Proof By induction on the structure of e. □

The dynamics of $\mathcal{L}\{\texttt{num str fun}\}$ is defined using function substitution:

$$\frac{}{\texttt{fun}[\tau_1;\tau_2](x_1.e_2;f.e) \mapsto [\![x_1.e_2/f]\!]e}. \tag{8.3}$$

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

The safety of $\mathcal{L}\{\texttt{num str fun}\}$ may, with some effort, be derived from the safety theorem for higher-order functions, which we discuss next.

8.2 Higher-Order Functions

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

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

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

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

The statics of $\mathcal{L}\{\texttt{num str} \to\}$ is given by extending Rules (4.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam[}\tau_1\texttt{]}\,(x\,.\,e) : \texttt{arr}(\tau_1; \tau_2)} \tag{8.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1; e_2) : \tau}. \tag{8.4b}$$

Lemma 8.2 (Inversion)**.** *Suppose that* $\Gamma \vdash e : \tau$.

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

Proof The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies and that the premises of the rule in question provide the required result. □

Lemma 8.3 (Substitution)**.** *If* $\Gamma, x : \tau \vdash e' : \tau'$*, and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$.

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

The dynamics of $\mathcal{L}\{\texttt{num str} \rightarrow\}$ extends that of $\mathcal{L}\{\texttt{num str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam[}\tau\texttt{]}(x\,.\,e)\ \mathsf{val}} \tag{8.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1'; e_2)} \tag{8.5b}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1; e_2')}\right] \tag{8.5c}$$

$$\frac{[e_2\ \mathsf{val}]}{\texttt{ap}(\texttt{lam[}\tau_2\texttt{]}(x\,.\,e_1); e_2) \mapsto [e_2/x]e_1}\,. \tag{8.5d}$$

The bracketed rule and premise are to be included for a *call-by-value* interpretation of function application, and excluded for a *call-by-name* interpretation.[1]

Theorem 8.4 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof The proof is by induction on Rules (8.5), which define the dynamics of the language.

Consider Rule (8.5d),

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

Suppose that $\texttt{ap}(\texttt{lam[}\tau_2\texttt{]}(x\,.\,e_1); e_2) : \tau_1$. By Lemma 8.2 we have $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 8.3 $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 8.5 (Canonical Forms)**.** *If $e : \texttt{arr}(\tau_1; \tau_2)$ and e val, then $e = \lambda\ (x\!:\!\tau_1)\, e_2$ for some variable x and expression e_2 such that $x : \tau_1 \vdash e_2 : \tau_2$.*

Proof By induction on the typing rules, using the assumption e val. □

Theorem 8.6 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof The proof is by induction on Rules (8.4). Note that because we consider only closed terms, there are no hypotheses on typing derivations.

Consider Rule (8.4b) (under the by-name interpretation). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1'; e_2)$. In the former case, we have by Lemma 8.5 that $e_1 = \texttt{lam[}\tau_2\texttt{]}(x\,.\,e)$ for some x and e. But then $\texttt{ap}(e_1; e_2) \mapsto [e_2/x]e$. □

[1] Although the term "call-by-value" is accurately descriptive, the origin of the term "call-by-name" remains shrouded in mystery.

8.3 Evaluation Dynamics and Definitional Equality

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

$$\frac{}{\mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)} \tag{8.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v}. \tag{8.6b}$$

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

Theorem 8.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof In the forward direction we proceed by rule induction on Rules (8.6), following along similar lines as those in the proof of Theorem 7.2.

In the reverse direction we proceed by rule induction on Rules (5.1). The proof relies on an analogue of Lemma 7.4, which states that evaluation is closed under converse execution, which is proved by induction on Rules (8.5). □

Definitional equality for the call-by-name dynamics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\,\rightarrow\}$ is defined by a straightforward extension to Rules (5.10):

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{8.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) \equiv \mathtt{ap}(e_1';e_2') : \tau} \tag{8.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \rightarrow \tau_2}. \tag{8.7c}$$

Definitional equality for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (8.7a) to require that the argument be a value. However, to be fully expressive, we must also widen the concept of a value to include all variables that are in scope, so that Rule (8.7a) would apply even when the argument is a variable. The justification for this is that in call-by-value, the parameter of a function stands for the value of its argument and not for the argument itself. The call-by-value definitional equality judgment has the form

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

where Γ consists of paired hypotheses $x : \tau, x$ val stating, for each variable x in scope, its type and that it is a value. We write $\Gamma \vdash e$ val to indicate that e is a value under these hypotheses, so that, for example, $x : \tau, x$ val $\vdash x$ val. (The typing hypothesis is irrelevant, but harmless, to the value judgment.)

8.4 Dynamic Scope

The dynamics of function application given by Rules (8.5) is defined only for expressions without free variables. When a function is called, the argument is substituted for the function parameter, ensuring that the result remains closed. Moreover, because substitution of closed expressions can never incur capture, the scopes of variables are not disturbed by the dynamics, ensuring that the principles of binding and scope described in Chapter 1 are respected. This treatment of variables is called *static scoping*, or *static binding*, to contrast it with an alternative approach that is now described.

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

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

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

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

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

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

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

One way around this difficulty is to ignore types altogether and rely on run-time checks to ensure that bad things do not happen, despite the evident failure of safety. (See Chapter 18 for a full exploration of this approach.) But even if we ignore worries about safety, we are still left with the serious problem that the names of bound variables matter and cannot be altered without changing the meaning of a program. So, for example, to use expression e', we must bear in mind that the parameter f occurs within the scope of a binder for y, a fact that is not revealed by the type of e' (and certainly not if we disregard types entirely!) If we change e' so that it binds a different variable, say z, then we must correspondingly

change e to ensure that it refers to z, and not to y, in order to preserve the overall behavior of the system of two expressions. This means that e and e' must be developed in tandem, violating a basic principle of modular decomposition. (For more on dynamic scope, please see Chapter 33.)

8.5 Notes

Nearly all programming languages provide some form of function definition mechanism of the kind illustrated here. The main point of the present account is to demonstrate that a more natural and more powerful approach is to separate the generic concept of a definition from the specific concept of a function. Function types codify the general notion in a systematic manner that encompasses function definitions as a special case and, moreover, admits passing functions as arguments and returning them as results without special provision. The essential contribution of Church's λ-calculus (Church, 1941) was to take the notion of function as primary and to demonstrate that nothing more is needed to obtain a fully expressive programming language.

9 Gödel's T

The language $\mathcal{L}\{\texttt{nat} \to\}$, better known as *Gödel's* ***T***, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\texttt{num str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\texttt{nat} \to\}$ provides a general mechanism, called primitive recursion, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an intrinsic termination proof for each program in the language. Consequently we may define only *total* functions in the language, those that always return a value for each argument. In essence every program in $\mathcal{L}\{\texttt{nat} \to\}$ "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 $\mathcal{L}\{\texttt{nat} \to\}$. To do so would require a master termination proof for every possible program in the language, something that we prove does not exist.

9.1 Statics

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

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

We write $\overline{n}$ for the expression $\texttt{s}(\ldots\texttt{s}(\texttt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression $\texttt{rec}(e;e_0;x\,.\,y\,.\,e_1)$ is called *primitive recursion*. It represents the e-fold iteration of the transformation $x\,.\,y\,.\,e_1$ starting from e_0. The bound variable x represents the predecessor and the bound variable y represents the result of the x-fold iteration. The "`with`" clause in the concrete syntax for the recursor binds the variable y to the result of the recursive call, as will become apparent shortly.

Sometimes *iteration*, written as $\mathtt{iter}(e; e_0; y\,.\,e_1)$, is considered as an alternative to primitive recursion. It has essentially the same meaning as primitive recursion, except that only the result of the recursive call is bound to y in e_1, and no binding is made for the predecessor. Clearly iteration is a special case of primitive recursion, because we can always ignore the predecessor binding. Conversely, primitive recursion is definable from iteration, provided that we have product types (Chapter 11) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

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

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

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

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

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

$$\frac{\Gamma, x : \rho \vdash e : \tau}{\Gamma \vdash \mathtt{lam}[\rho](x\,.\,e) : \mathtt{arr}(\rho; \tau)} \tag{9.1e}$$

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

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

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

9.2 Dynamics

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

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

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

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

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

The transition rules for the dynamics of $\mathcal{L}\{\mathtt{nat}\to\}$ are as follows:

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

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

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

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

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

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

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

The bracketed rules and premises are to be 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 will not be evaluated.

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

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

Theorem 9.3 (Safety)**.**

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

9.3 Definability

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

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

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

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

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

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

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

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

To 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:

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

This function grows very quickly. For example, $A(4, 2) \approx 2^{65,536}$, which is often cited as being much larger than the number of atoms in the universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of arguments (m, n). On each recursive call, either m decreases or m remains the same and n decreases, so inductively the recursive calls are well-defined, and hence so is $A(m, n)$.

A *first-order primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using primitive recursion, but without using any higher-order functions. Ackermann's

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

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

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

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

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

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

to be the expression

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

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

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

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

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

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

9.4 Undefinability

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

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

Proof See Corollary 47.15. □

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

Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in $\mathcal{L}\{\mathtt{nat}\to\}$. We make use of a technique, called *Gödel numbering*, that assigns a unique natural number to each closed expression

of $\mathcal{L}\{\mathtt{nat}\to\}$. This allows us to manipulate expressions as data values in $\mathcal{L}\{\mathtt{nat}\to\}$ and hence permits $\mathcal{L}\{\mathtt{nat}\to\}$ to compute with its own programs.[1]

The essence of Gödel numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is slightly more difficult, the main complication being to ensure 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. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let m be the index of o in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of a to be the number

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

where p_k is the kth prime number (so that $p_0 = 2$, $p_1 = 3$, and so on) and $n_1, \ldots, n_k$ are the Gödel numbers of $a_1, \ldots, a_k$, respectively. This assigns a natural number to each ast. Conversely, given a natural number n, we may apply the prime factorization theorem to "parse" n as a unique abstract syntax tree. (If the factorization is not of the appropriate form, which can be only 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, ensures that f_{univ} is a well-defined function. It is called the *universal function* for $\mathcal{L}\{\mathtt{nat}\to\}$ because it specifies the behavior of any expression e of type $\mathtt{nat} \to \mathtt{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the *diagonal function*, $d : \mathbb{N} \to \mathbb{N}$, by the equation $d(m) = f_{univ}(m)(m)$. This function is chosen so that $d(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for this definition will become apparent in a moment.)

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

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

Let e_D be the expression

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

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

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

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

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 evident that f_{univ} is computable in the sense that we can define it

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

[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.

in a sufficiently powerful programming language. But the preceding argument shows that $\mathcal{L}\{\mathtt{nat}\to\}$ is not sufficiently powerful for this task. That is, $\mathcal{L}\{\mathtt{nat}\to\}$ is not universal. By demanding termination we sacrifice expressiveness. The preceding argument shows that this is an inescapable trade-off. If you want universality, you have to give up termination, and if you want termination, then you must give up universality. There is no alternative.

9.5 Notes

$\mathcal{L}\{\mathtt{nat}\to\}$ was introduced by Gödel in his study of the consistency of arithmetic (Gödel, 1980). Gödel showed how to "compile" proofs in arithmetic into well-typed terms of the language $\mathcal{L}\{\mathtt{nat}\to\}$ and to reduce the consistency problem for arithmetic to the termination of programs in $\mathcal{L}\{\mathtt{nat}\to\}$. This was perhaps the first programming language whose design was directly influenced by the verification (of termination) of its programs.

10 Plotkin's PCF

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, also known as *Plotkin's PCF*, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to those in $\mathcal{L}\{\texttt{nat} \to\}$, expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ might not terminate when evaluated: Its definable functions are, in general, partial rather than total. Informally, the difference between $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ and $\mathcal{L}\{\texttt{nat} \to\}$ is that the former moves the proof of termination for an expression from the expression itself into the mind of the programmer. The type system no longer ensures termination, which permits a wider range of functions to be defined in the system, but at the cost of admitting infinite loops when the termination proof is either incorrect or absent.

The crucial concept embodied in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is the fixed point characterization of recursive definitions. In ordinary mathematical practice we may define a function f by *recursion equations* such as these:

$$f(0) \triangleq 1$$
$$f(n+1) \triangleq (n+1) \times f(n).$$

These may be viewed as simultaneous equations in the variable f ranging over functions on the natural numbers. The function we seek is a *solution* to these equations—a function $f : \mathbb{N} \to \mathbb{N}$ such that the preceding conditions are satisfied. We must, of course, show that these equations have a unique solution, which is easily shown by mathematical induction on the argument to f.

The solution to such a system of equations may be characterized as the fixed point of an associated functional (operator mapping functions to functions). To see this, let us rewrite these equations in another form:

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

Rewriting 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 we 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 the argument f to the functional F, and not in terms of f' itself! The function f we seek is then a *fixed point* of F, which is a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words, f is defined to the $\mathit{fix}(F)$, where *fix* is an operator on functionals yielding a fixed point of F.

Why does an operator such as F have a fixed point? Informally, a fixed point may be obtained as the limit of a series of approximations of the desired solution obtained by iterating the functional F. This is where partial functions come into the picture. Let us say that a partial function, ϕ on the natural numbers, is an *approximation* to a total function f if $\phi(m) = n$ implies that $f(m) = n$. Let $\bot : \mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function: $\bot(n)$ is undefined for every $n \in \mathbb{N}$. Intuitively, this is the "worst" approximation to the desired solution f of the recursion equations previously given. Given any approximation ϕ of f, we may "improve" it by considering $\phi' = F(\phi)$. Intuitively, ϕ' is defined on 0 and on $m + 1$ for every $m \geq 0$ on which ϕ is defined. Continuing in this manner, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ' and hence a further improvement on ϕ. If we start with $\bot$ as the initial approximation to f and then pass to the limit

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

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

This fixed point characterization of recursion equations is taken as a primitive concept in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$—we may obtain the least fixed point of *any* functional definable in the language. Using this, we may solve any set of recursion equations we like, with the proviso that there is no guarantee that the solution is a *total* function. Rather, it is guaranteed to be a *partial* function that may be undefined on some, all, or no inputs. This is the price we pay for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like. It is our task as programmers to ensure that the functions defined by recursion are total—all of our loops terminate.

10.1 Statics

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

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

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

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{10.1a}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat} \vdash e_1 : \tau}{\Gamma \vdash \texttt{ifz}(e; e_0; x\,.\,e_1) : \tau} \tag{10.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1]\,(x\,.\,e) : \texttt{parr}(\tau_1; \tau_2)} \tag{10.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{parr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1; e_2) : \tau} \tag{10.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}[\tau]\,(x\,.\,e) : \tau}\,. \tag{10.1g}$$

Rule (10.1g) reflects the self-referential nature of general recursion. To show that $\texttt{fix}[\tau]\,(x\,.\,e)$ has type τ, we *assume* that it is the case by assigning that type to the variable x, which stands for the recursive expression itself, and checking that the body e has type τ under this very assumption.

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

Lemma 10.1. *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

10.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is defined by the judgments $e\ \mathsf{val}$, specifying the closed values, and $e \mapsto e'$, specifying the steps of evaluation.

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

$$\frac{}{\texttt{z}\ \mathsf{val}} \tag{10.2a}$$

$$\frac{[e\ \mathsf{val}]}{\texttt{s}(e)\ \mathsf{val}} \tag{10.2b}$$

$$\frac{}{\texttt{lam}[\tau]\,(x\,.\,e)\ \mathsf{val}}\,. \tag{10.2c}$$

The bracketed premise on Rule (10.2b) is to be included for the *eager* interpretation of the sucessor operation and omitted for the *lazy* interpretation. (See Chapter 37 for a further discussion of laziness.)

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

$$\left[\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right] \tag{10.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e; e_0; x\,.\,e_1) \mapsto \mathtt{ifz}(e'; e_0; x\,.\,e_1)} \tag{10.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z}; e_0; x\,.\,e_1) \mapsto e_0} \tag{10.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}(\mathtt{s}(e); e_0; x\,.\,e_1) \mapsto [e/x]e_1} \tag{10.3d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1'; e_2)} \tag{10.3e}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1; e_2')}\right] \tag{10.3f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}[\tau]\,(x\,.\,e); e_2) \mapsto [e_2/x]e} \tag{10.3g}$$

$$\frac{}{\mathtt{fix}[\tau]\,(x\,.\,e) \mapsto [\mathtt{fix}[\tau]\,(x\,.\,e)/x]e}\,. \tag{10.3h}$$

The bracketed rule, Rule (10.3a), is to be included for an eager interpretation of the successor, and omitted otherwise. Bracketed Rule (10.3f) and the bracketed premise on Rule (10.3g) are to be included for a call-by-value interpretation and omitted for a call-by-name interpretation of function application. Rule (10.3h) implements self-reference by substituting the recursive expression itself for the variable x in its body; this is called *unwinding* the recursion.

Theorem 10.2 (Safety)**.**

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

Proof The proof of preservation is by induction on the derivation of the transition judgment. Consider Rule (10.3h). 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 (10.1g) the result follows immediately because we may make progress by unwinding the recursion. □

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

Definitional equality for the call-by-name variant of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, written as $\Gamma \vdash e_1 \equiv e_2 : \tau$, is defined to be the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{z}; e_0; x\,.\,e_1) \equiv e_0 : \tau} \tag{10.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{s}(e); e_0; x\,.\,e_1) \equiv [e/x]e_1 : \tau} \tag{10.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}[\tau](x\,.\,e) \equiv [\mathtt{fix}[\tau](x\,.\,e)/x]e : \tau} \tag{10.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau_1](x\,.\,e_2); e_1) \equiv [e_1/x]e_2 : \tau}\,. \tag{10.4d}$$

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

10.3 Definability

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

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

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

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

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

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

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

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

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

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

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

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

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

It is informative to classify those partial functions ϕ that are definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. These are the so-called *partial recursive functions*, which are defined to be the primitive recursive functions augmented by the *minimization* operation: Given $\phi(m, n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m', n)$ it is defined and nonzero, and (2) $\phi(m, n) = 0$. If no such m exists, then $\psi(n)$ is undefined.

Theorem 10.3. *A partial function ϕ on the natural numbers is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ iff it is partial recursive.*

Proof sketch Minimization is readily definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with considerable tedium, define an evaluator for expressions of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ as a partial recursive function, using Gödel numbering to represent expressions as numbers. Consequently $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ does not exceed the power of the set of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the set of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language currently available or that will ever be available.[1] Therefore $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

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

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

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

By Church's Law the universal function is definable in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. In contrast, we proved in Chapter 9 that the analogous function is *not* definable in $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ using the technique of diagonalization. It is instructive to examine why that argument does not apply in the

[1] See Chapter 17 for further discussion of Church's Law.

present setting. As in Section 9.4, we may derive the equivalence

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

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

10.4 Notes

The language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is derived from the work by Plotkin (1977). Plotkin introduced PCF to study the relationship between its operational and denotational semantics, but many authors have used PCF as the subject of study for many issues in the design and semantics of languages. In this respect PCF may be thought of as the *E. coli* of programming languages.

PART IV

Finite Data Types

11 Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated eliminatory forms are *projections*, which select the first and second components of a pair. The *nullary product*, or *unit*, type consists solely of the unique "null tuple" of no values and has no associated eliminatory form. The product type admits both a *lazy* and an *eager* dynamics. According to the lazy dynamics, a pair is a value without regard to whether its components are values; they are not evaluated until (if ever) they are accessed and used in another computation. According to the eager dynamics, a pair is a value only if its components are values; they are evaluated when the pair is created.

More generally, we may consider the *finite product* $\langle \tau_i \rangle_{i \in I}$ 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. Similar to binary products, finite products admit both an eager and a lazy interpretation.

11.1 Nullary and Binary Products

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

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

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

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

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \tag{11.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \tag{11.1b}$$

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

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2}\,. \tag{11.1d}$$

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

$$\frac{}{\langle\rangle\ \mathsf{val}} \tag{11.2a}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle\ \mathsf{val}} \tag{11.2b}$$

$$\left[\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\right] \tag{11.2c}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle}\right] \tag{11.2d}$$

$$\frac{e \mapsto e'}{e \cdot \mathtt{l} \mapsto e' \cdot \mathtt{l}} \tag{11.2e}$$

$$\frac{e \mapsto e'}{e \cdot \mathtt{r} \mapsto e' \cdot \mathtt{r}} \tag{11.2f}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{l} \mapsto e_1} \tag{11.2g}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{r} \mapsto e_2}\,. \tag{11.2h}$$

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

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

Theorem 11.1 (Safety)**.**

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

Proof Preservation is proved by induction on transition defined by Rules (11.2). Progress is proved by induction on typing defined by Rules (11.1). □

11.2 Finite Products

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

Sort			Abstract Form	Concrete Form	Description
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 as $\mathtt{pr}[i](e)$, or $e \cdot i$ for short.

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

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

in which 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 so as to enhance readability.

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle} \tag{11.3a}$$

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

In Rule (11.3b) the index $i \in I$ is a *particular* element of the index set I, whereas in Rule (11.3a) the index i ranges over the index set I.

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

$$\frac{[e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}} \tag{11.4a}$$

$$\left[\frac{e_1 \ \mathsf{val} \quad \ldots \quad e_{j-1} \ \mathsf{val} \quad e_j \mapsto e'_j \quad e'_{j+1} = e_{j+1} \quad \ldots \quad e'_n = e_n}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \mapsto \langle i_1 \hookrightarrow e'_1, \ldots, i_n \hookrightarrow e'_n \rangle}\right] \tag{11.4b}$$

$$\frac{e \mapsto e'}{e \cdot i \mapsto e' \cdot i} \tag{11.4c}$$

$$\frac{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \cdot i_k \mapsto e_k}\,. \tag{11.4d}$$

As formulated, Rule (11.4b) specifies that the components of a tuple are to be 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 11.2 (Safety)**.** *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \mapsto e'$.*

Proof The safety theorem may be decomposed into progress and preservation lemmas, which are proved as in Section 11.1. □

11.3 Primitive and Mutual Recursions

In the presence of products we may simplify the primitive recursion construct defined in Chapter 9 so that only the result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\mathtt{iter}(e; e_0; x\,.\,e_1)$, we may define primitive recursion in the sense of Chapter 9 to be the expression $e' \cdot \mathtt{r}$, where e' is the expression

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

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

We may also use product types to implement *mutual recursion*, which allows several mutually recursive computations to be defined simultaneously. For example, consider the following recursion equations defining two mathematical functions on the natural numbers:

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

Intuitively, $E(n)$ is nonzero if and only if n is even, and $O(n)$ is nonzero if and only if n is odd. If we wish to define these functions in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, we immediately face the problem of how to define two functions simultaneously. There is a trick available in this special case that takes advantage of the fact that E and O have the same type: Simply define $\mathtt{eo}$ of type $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ so that $\mathtt{eo}(\bar{0})$ represents E and $\mathtt{eo}(\bar{1})$ represents O. The details are left as an exercise for the reader.)

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

$$\langle \mathtt{even} \hookrightarrow \mathtt{nat} \to \mathtt{nat}, \mathtt{odd} \hookrightarrow \mathtt{nat} \to \mathtt{nat}\rangle.$$

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

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

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

where e_E is the expression

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

and e_O is the expression

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

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

11.4 Notes

Product types are the essence of structured data. All languages have some form of product type, but frequently in a form that is combined with other, separable, concepts. Common manifestations of products include (1) functions with "multiple arguments" or "multple results"; (2) "objects" represented as tuples of mutually recursive functions; and (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 23). Allen et al. (2006) analyzes many of the key ideas in the framework of dependent type theory.

12 Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by *sum types*, specifically the *binary sum*, which offers a choice of two things, and the *nullary sum*, which offers a choice of no things. *Finite sums* generalize nullary and binary sums to permit an arbitrary number of cases indexed by a finite index set. As with products, sums come in both eager and lazy variants, differing in how values of sum type are defined.

12.1 Nullary and Binary Sums

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

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

The nullary sum represents a choice of zero alternatives and hence admits no introductory form. The eliminatory form `abort(`e`)` aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary sum type are labeled to indicate whether they are drawn from the left or the right summand, either `l` $\cdot\, e$ or `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:

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}(e) : \tau} \tag{12.1a}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{l}\cdot e : \tau_1 + \tau_2} \tag{12.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r} \cdot e : \tau_1 + \tau_2} \tag{12.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} : \tau}. \tag{12.1d}$$

For the sake of readability, in Rules (12.1b) and (12.1c) $\mathtt{l} \cdot e$ and $\mathtt{r} \cdot e$ are written in place of the abstract syntax $\mathtt{in}[\tau_1;\tau_2]\,[\mathtt{l}]\,(e)$ and $\mathtt{in}[\tau_1;\tau_2]\,[\mathtt{r}]\,(e)$, which includes the types τ_1 and τ_2 explicitly. In Rule (12.1d) 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:

$$\frac{e \mapsto e'}{\mathtt{abort}(e) \mapsto \mathtt{abort}(e')} \tag{12.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{l} \cdot e\ \mathsf{val}} \tag{12.2b}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{r} \cdot e\ \mathsf{val}} \tag{12.2c}$$

$$\left[\frac{e \mapsto e'}{\mathtt{l} \cdot e \mapsto \mathtt{l} \cdot e'}\right] \tag{12.2d}$$

$$\left[\frac{e \mapsto e'}{\mathtt{r} \cdot e \mapsto \mathtt{r} \cdot e'}\right] \tag{12.2e}$$

$$\frac{e \mapsto e'}{\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} \mapsto \mathtt{case}\, e'\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\}} \tag{12.2f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{l} \cdot e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} \mapsto [e/x_1]e_1} \tag{12.2g}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{r} \cdot e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} \mapsto [e/x_2]e_2}. \tag{12.2h}$$

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

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

Theorem 12.1 (Safety)**.**

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

Proof The proof proceeds by induction on Rules (12.2) for preservation and by induction on Rules (12.1) for progress. □

12.2 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:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	$::=$	$\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$	$[\tau_i]_{i \in I}$	sum
Exp	e	$::=$	$\mathtt{in}[\vec{\tau}]\,[i]\,(e)$	$i \cdot e$	injection
			$\mathtt{case}(e; \{i \hookrightarrow x_i \,.\, e_i\}_{i \in I})$	$\mathtt{case}\, e\, \{i \cdot x_i \Rightarrow e_i\}_{i \in I}$	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 $I = \{l_1, \ldots, l_n\}$, the type of I-classified values may be written as

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

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

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

Finite sums 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 symbolic names for the classes so as to enhance readability.

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

$$\frac{\Gamma \vdash e : \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]} \tag{12.3a}$$

$$\frac{\Gamma \vdash e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n] \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}\, e\, \{i_1 \cdot x_1 \Rightarrow e_1 \mid \ldots \mid i_n \cdot x_n \Rightarrow e_n\} : \tau}. \tag{12.3b}$$

These rules generalize to the finite case of the statics for nullary and binary sums given in Section 12.1.

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

$$\frac{[e\ \mathsf{val}]}{i \cdot e\ \mathsf{val}} \tag{12.4a}$$

$$\left[\frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\right] \tag{12.4b}$$

$$\frac{e \mapsto e'}{\mathtt{case}\, e\, \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto \mathtt{case}\, e'\, \{i \cdot x_i \Rightarrow e_i\}_{i \in I}} \tag{12.4c}$$

$$\frac{i \cdot e\ \mathsf{val}}{\mathtt{case}\, i \cdot e\, \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto [e/x_i]e_i}. \tag{12.4d}$$

These again generalize the dynamics of binary sums given in Section 12.1.

Theorem 12.2 (Safety)**.** *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \mapsto e'$.*

Proof The proof is similar to that for the binary case, as described in Section 12.1. □

12.3 Applications of Sum Types

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

12.3.1 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, it must be `unit`—in other words, e has *no interesting value* (but it could diverge). On the other hand, if e : `void`, then *e must not yield a value*; if it were to have a value, it would have to be a value of type `void`, of which there are none. This shows that what is called the `void` type in many languages is really the type `unit` because it indicates that an expression has no interesting value, not that it has no value at all!

12.3.2 Booleans

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

Sort			**Abstract Form**	**Concrete Form**	**Description**
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:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{12.5a}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{12.5b}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau}. \tag{12.5c}$$

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

$$\frac{}{\texttt{true val}} \tag{12.6a}$$

$$\frac{}{\texttt{false val}} \tag{12.6b}$$

$$\frac{}{\texttt{if true then } e_1 \texttt{ else } e_2 \mapsto e_1} \tag{12.6c}$$

$$\frac{}{\texttt{if false then } e_1 \texttt{ else } e_2 \mapsto e_2} \tag{12.6d}$$

$$\frac{e \mapsto e'}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mapsto \texttt{if } e' \texttt{ then } e_1 \texttt{ else } e_2}\,. \tag{12.6e}$$

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

$$\texttt{bool} = \texttt{unit} + \texttt{unit} \tag{12.7a}$$

$$\texttt{true} = \texttt{l} \cdot \langle\rangle \tag{12.7b}$$

$$\texttt{false} = \texttt{r} \cdot \langle\rangle \tag{12.7c}$$

$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{case } e \,\{\texttt{l} \cdot x_1 \Rightarrow e_1 \mid \texttt{r} \cdot x_2 \Rightarrow e_2\}. \tag{12.7d}$$

In Equation (12.7d) 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 evident statics and dynamics of the type `bool` are engendered by these definitions.

12.3.3 Enumerations

More generally, sum types may be used to define *finite enumeration* types, those whose values are one of an explicitly given finite set and whose elimination form is a case analysis on the elements of that set. For example, the type `suit`, whose elements are ♣, ◇, ♡, and ♠, has as an elimination form the case analysis

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

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

12.3.4 Options

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

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	$\texttt{opt}(\tau)$	$\tau\ \texttt{opt}$	option
Exp	e	::=	$\texttt{null}$	$\texttt{null}$	nothing
			$\texttt{just}(e)$	$\texttt{just}(e)$	something
			$\texttt{ifnull}[\tau](e; e_1; x.e_2)$	$\texttt{check}\, e\, \{\texttt{null} \Rightarrow e_1 \mid \texttt{just}(x) \Rightarrow e_2\}$	null test

The type $\texttt{opt}(\tau)$ represents the type of "optional" values of type τ. The introductory 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]

$$\tau\ \texttt{opt} = \texttt{unit} + \tau \tag{12.8a}$$

$$\texttt{null} = \texttt{l} \cdot \langle\rangle \tag{12.8b}$$

$$\texttt{just}(e) = \texttt{r} \cdot e \tag{12.8c}$$

$$\texttt{check}\, e\, \{\texttt{null} \Rightarrow e_1 \mid \texttt{just}(x_2) \Rightarrow e_2\} = \texttt{case}\, e\, \{\texttt{l} \cdot \_ \Rightarrow e_1 \mid \texttt{r} \cdot x_2 \Rightarrow e_2\}. \tag{12.8d}$$

It is left to the reader to examine the statics and dynamics implied by these definitions.

The option type is the key to understanding a common misconception, the *null pointer fallacy*. This fallacy, which is particularly common in object-oriented languages, is based on two related errors. The first error is to deem the values of certain types to be mysterious entities called *pointers*, based on suppositions about how these values might be represented at run time, rather than on the semantics of the type itself. The second error compounds the first. A particular value of a pointer type is distinguished as the *null pointer*, which, unlike the other elements of that type, does not designate a value of that type at all, but rather rejects all attempts to use it as such.

To help avoid such failures, such languages usually include a function, say $\texttt{null} : \tau \to \texttt{bool}$, that yields $\texttt{true}$ if its argument is null and $\texttt{false}$ otherwise. This allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

$$\texttt{if}\, \texttt{null}(e)\, \texttt{then} \ldots \mathit{error} \ldots\ \texttt{else} \ldots \mathit{proceed} \ldots. \tag{12.9}$$

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 may be traced to the failure to distinguish the type τ from the type $\tau\ \texttt{opt}$. Rather than thinking of the elements of type τ as pointers, and thereby having to worry about the null pointer, we instead distinguish between a *genuine* value of type τ

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

and an *optional* value of type τ. An optional value of type τ may or may not be present, but, if it is, the underlying value is truly a value of type τ (and cannot be null). The elimination form for the option type,

$$\texttt{check}\, e\, \{\texttt{null} \Rightarrow e_{error} \mid \texttt{just}(x) \Rightarrow e_{ok}\}, \tag{12.10}$$

propagates the information that e is present into the nonnull 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 nonnull branch no further null checks, explicit or implicit, are required. Observe that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (12.9); the advantage of option types is precisely that it does so.

12.4 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 25 for more on this correspondence.) The absence of sums is the origin of Hoare's self-described "billion dollar mistake," the null pointer (Hoare, 2009). Bad language designs impose the burden of handling "null" values on programmers, resulting in countless errors that manifest themselves only at run time.

13 Pattern Matching

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

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

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

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

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

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

$$\mathtt{match}\ e\ \{\langle \mathtt{l} \cdot \langle\rangle, x\rangle \Rightarrow x + x \mid \langle \mathtt{r} \cdot \langle\rangle, y\rangle \Rightarrow y * y\}. \tag{13.1}$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 11 and 12.

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

13.1 A Pattern Language

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

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

The operator `match` has two operands, the expression to match and a series of rules. A sequence of rules is constructed using the operator `rules[`n`]`, which has $n \geq 0$ operands. Each rule is constructed by the operator `rule[`k`]`, which specifies that it has two operands, binding k variables in the second.

13.2 Statics

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

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

with almost the same meaning as

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

except that each variable is required to be used *at most once* in p. When reading the judgment $\Lambda \Vdash p : \tau$ it is helpful to think of Λ as an *output* and p and τ as *inputs*. Given p and τ, the rules determine the hypotheses Λ such that $\Lambda \Vdash p : \tau$:

$$\frac{}{x : \tau \Vdash x : \tau} \tag{13.2a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau} \tag{13.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}} \tag{13.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2} \tag{13.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2} \tag{13.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2}\,. \tag{13.2f}$$

Rule (13.2a) states that a variable is a pattern of type τ. Rule (13.2d) states that a pair pattern consists of two patterns with disjoint variables.

The typing judgments for a rule,

$$p \Rightarrow e : \tau \rightsquigarrow \tau',$$

and for a sequence of rules,

$$r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau',$$

specify that rules transform a value of type τ into a value of type τ'. These judgments are inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau \rightsquigarrow \tau'} \tag{13.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau \rightsquigarrow \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau \rightsquigarrow \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau'}\,. \tag{13.3b}$$

Using the typing judgments for rules, we may state the typing rule for a `match` expression quite easily:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau \rightsquigarrow \tau'}{\Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'}\,. \tag{13.4}$$

13.3 Dynamics

A *substitution* θ is a finite mapping from variables to values. If θ is the substitution $\{x_0 \hookrightarrow e_0, \ldots, x_{k-1} \hookrightarrow e_{k-1}\}$, we write $\hat{\theta}(e)$ for $[e_1, \ldots, e_k/x_1, \ldots, x_k]e$. The judgment $\theta : \Lambda$ is inductively defined by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{13.5a}$$

$$\frac{\theta : \Lambda \quad e : \tau}{\theta \otimes x \hookrightarrow e : \Lambda, x : \tau}\,. \tag{13.5b}$$

The judgment $\theta \Vdash p \lhd e$ states that the pattern p matches the value e, as witnessed by the substitution θ defined on the variables of p. This judgment is inductively defined by the following rules:

$$\frac{}{x \hookrightarrow e \Vdash x \lhd e} \tag{13.6a}$$

$$\frac{}{\emptyset \Vdash \_ \lhd e} \tag{13.6b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle} \tag{13.6c}$$

$$\frac{\theta_1 \Vdash p_1 \lhd e_1 \quad \theta_2 \Vdash p_2 \lhd e_2 \quad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2\rangle \lhd \langle e_1, e_2\rangle} \tag{13.6d}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{l} \cdot p \lhd \mathtt{l} \cdot e} \tag{13.6e}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{r} \cdot p \lhd \mathtt{r} \cdot e}\,. \tag{13.6f}$$

These rules simply collect the bindings for the pattern variables required to form a substitution witnessing the success of the matching process.

The judgment $e \perp p$ states that e does not match the pattern p. It is inductively defined by the following rules:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{13.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{13.7b}$$

$$\frac{}{\mathtt{l} \cdot e \perp \mathtt{r} \cdot p} \tag{13.7c}$$

$$\frac{e \perp p}{\mathtt{l} \cdot e \perp \mathtt{l} \cdot p} \tag{13.7d}$$

$$\frac{}{\mathtt{r} \cdot e \perp \mathtt{l} \cdot p} \tag{13.7e}$$

$$\frac{e \perp p}{\mathtt{r} \cdot e \perp \mathtt{r} \cdot p}\,. \tag{13.7f}$$

Neither a variable nor a wild card nor a null-tuple can mismatch any value of appropriate type. A pair can mismatch a pair pattern only because of a mismatch in one of its components. An injection into a sum type can mismatch the opposite injection or it can mismatch the same injection by having its argument mismatch the argument pattern.

Theorem 13.1. *Suppose that $e : \tau$, e val, and $\Lambda \Vdash p : \tau$. Then either there exists θ such that $\theta : \Lambda$ and $\theta \Vdash p \lhd e$, or $e \perp p$.*

Proof By rule induction on Rules (13.2), making use of the canonical forms' lemma to characterize the shape of e based on its type. □

The dynamics of the `match` expression is given in terms of the pattern match and mismatch judgments as follows:

$$\frac{e \mapsto e'}{\mathtt{match}\, e\, \{rs\} \mapsto \mathtt{match}\, e'\, \{rs\}} \tag{13.8a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{match}\,e\,\{\}\ \mathsf{err}} \tag{13.8b}$$

$$\frac{e\ \mathsf{val} \quad \theta \Vdash p_0 \lhd e}{\mathtt{match}\,e\,\{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto \hat{\theta}(e_0)} \tag{13.8c}$$

$$\frac{e\ \mathsf{val} \quad e \perp p_0 \quad \mathtt{match}\,e\,\{rs\} \mapsto e'}{\mathtt{match}\,e\,\{p_0 \Rightarrow e_0 \,|\, rs\} \mapsto e'}\,. \tag{13.8d}$$

Rule (13.8b) specifies that evaluation results in a checked error once all rules are exhausted. Rule (13.8c) specifies that the rules are to be considered in order. If the match value e matches the pattern p_0 of the initial rule in the sequence, then the result is the corresponding instance of e_0; otherwise, matching continues by considering the remaining rules.

Theorem 13.2 (Preservation)**.** *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof By a straightforward induction on the derivation of $e \mapsto e'$. □

13.4 Exhaustiveness and Redundancy

Although it is possible to state and prove a progress theorem for $\mathcal{L}\{pat\}$ as defined in Section 13.1, it would not have much force, because the statics does not rule out pattern-matching failure. What is missing is enforcement of the *exhaustiveness* of a sequence of rules, which ensures that every value of the domain type of a sequence of rules must match some rule in the sequence. In addition it would be useful to rule out *redundancy* of rules, which arises when a rule can match only values that are also matched by a preceding rule. Because pattern matching considers rules in the order in which they are written, such a rule can never be executed, and hence can be safely eliminated.

13.4.1 Match Constraints

To express exhaustiveness and irredundancy, we introduce a language of *match constraints* that identify a subset of the closed values of a type. With each rule we associate a constraint that classifies the values that are matched by that rule. A sequence of rules is *exhaustive* if every value of the domain type of the rules satisfies the match constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match constraint also satisfies the match constraint of some preceding rule.

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

Sort			Abstract Form	Concrete Form	Description
Constr	ξ	::=	`all`$[\tau]$	$\top$	truth
			`and`$(\xi_1;\xi_2)$	$\xi_1 \wedge \xi_2$	conjunction
			`nothing`$[\tau]$	$\bot$	falsity
			`or`$(\xi_1;\xi_2)$	$\xi_1 \vee \xi_2$	disjunction
			`l` $\cdot\, \xi_1$	`l` $\cdot\, \xi_1$	left injection
			`r` $\cdot\, \xi_2$	`r` $\cdot\, \xi_2$	right injection
			`triv`	$\langle\rangle$	unit
			`pair`$(\xi_1;\xi_2)$	$\langle \xi_1, \xi_2 \rangle$	pair

It is easy to define the judgment $\xi : \tau$ specifying that the constraint ξ constrains values of type τ.

The *De Morgan dual* $\overline{\xi}$ of a match constraint ξ is defined by the following rules:

$$\begin{aligned}
\overline{\top} &= \bot \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\bot} &= \top \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\texttt{l} \cdot \xi_1} &= \texttt{l} \cdot \overline{\xi_1} \vee \texttt{r} \cdot \top \\
\overline{\texttt{r} \cdot \xi_1} &= \texttt{r} \cdot \overline{\xi_1} \vee \texttt{l} \cdot \top \\
\overline{\langle\rangle} &= \bot \\
\overline{\langle \xi_1, \xi_2 \rangle} &= \langle \overline{\xi_1}, \xi_2 \rangle \vee \langle \xi_1, \overline{\xi_2} \rangle \vee \langle \overline{\xi_1}, \overline{\xi_2} \rangle.
\end{aligned}$$

Intuitively, the dual of a match constraint expresses the negation of that constraint. In the case of the last four rules it is important to keep in mind that these constraints apply only to specific types.

The *satisfaction* judgment $e \models \xi$ is defined for values e and constraints ξ of the same type by the following rules:

$$\frac{}{e \models \top} \tag{13.9a}$$

$$\frac{e \models \xi_1 \quad e \models \xi_2}{e \models \xi_1 \wedge \xi_2} \tag{13.9b}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{13.9c}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{13.9d}$$

$$\frac{e_1 \models \xi_1}{\texttt{l} \cdot e_1 \models \texttt{l} \cdot \xi_1} \tag{13.9e}$$

$$\frac{e_2 \models \xi_2}{\mathtt{r} \cdot e_2 \models \mathtt{r} \cdot \xi_2} \tag{13.9f}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{13.9g}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2\rangle \models \langle \xi_1, \xi_2\rangle}\,. \tag{13.9h}$$

The De Morgan dual construction negates a constraint.

Lemma 13.3. *If ξ is a constraint on values of type τ, then $e \models \overline{\xi}$ iff and only iff $e \not\models \xi$.*

We define the *entailment* of two constraints $\xi_1 \models \xi_2$ to mean that $e \models \xi_2$ whenever $e \models \xi_1$. By Lemma 13.3 we have that $\xi_1 \models \xi_2$ iff $\models \overline{\xi_1} \vee \xi_2$. We often write $\xi_1, \ldots, \xi_n \models \xi$ for $\xi_1 \wedge \ldots \wedge \xi_n \models \xi$ so that in particular $\models \xi$ means that $e \models \xi$ for every value $e : \tau$.

13.4.2 Enforcing Exhaustiveness and Redundancy

To enforce exhaustiveness and irredundancy, the statics of pattern matching is augmented with constraints that express the set of values matched by a given set of rules. A sequence of rules is *exhaustive* if every value of the domain type of the rule satisfies the match constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match constraint also satisfies the match constraint of some preceding rule. A sequence of rules is *irredundant* iff no rule is redundant relative to the rules that precede it in the sequence.

The judgment $\Lambda \Vdash p : \tau\ [\xi]$ augments the judgment $\Lambda \Vdash p : \tau$ with a match constraint characterizing the set of values of type τ matched by the pattern p. It is inductively defined by the following rules:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top]} \tag{13.10a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau\ [\top]} \tag{13.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}\ [\langle\rangle]} \tag{13.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2\ [\mathtt{l} \cdot \xi_1]} \tag{13.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2\ [\mathtt{r} \cdot \xi_2]} \tag{13.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]}\,. \tag{13.10f}$$

Lemma 13.4. *Suppose that $\Lambda \Vdash p : \tau\ [\xi]$. For every $e : \tau$ such that e val, $e \models \xi$ iff $\theta \Vdash p \triangleleft e$ for some θ, and $e \not\models \xi$ iff $e \perp p$.*

The judgment $\Gamma \vdash r : \tau \rightsquigarrow \tau' \ [\xi]$ augments the formation judgment for a rule with a match constraint characterizing the pattern component of the rule. The judgment $\Gamma \vdash rs : \tau \rightsquigarrow \tau' \ [\xi]$ augments the formation judgment for a sequence of rules with a match constraint characterizing the values matched by some rule in the given rule sequence:

$$\frac{\Lambda \Vdash p : \tau \ [\xi] \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau \rightsquigarrow \tau' \ [\xi]} \tag{13.11a}$$

$$\frac{\begin{array}{c}(\forall 1 \leq i \leq n)\ \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1} \\ \Gamma \vdash r_1 : \tau \rightsquigarrow \tau' \ [\xi_1] \qquad \ldots \qquad \Gamma \vdash r_n : \tau \rightsquigarrow \tau' \ [\xi_n]\end{array}}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau' \ [\xi_1 \vee \ldots \vee \xi_n]}\,. \tag{13.11b}$$

Rule (13.11b) requires that each successive rule not be redundant relative to the preceding rules. The overall constraint associated with the rule sequence specifies that every value of type τ satisfy the constraint associated with some rule.

The typing rule for `match` expressions demands that the rules that comprise it be exhaustive:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau \rightsquigarrow \tau' \ [\xi] \quad \models \xi}{\Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'}\,. \tag{13.12}$$

Rule (13.11b) ensures that ξ is a disjunction of the match constraints associated with the constituent rules of the match expression. The requirement that ξ be valid amounts to requiring that every value of type τ satisfy the constraint of at least one rule of the match.

Theorem 13.5. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof The exhaustiveness check in Rule (13.12) ensures that if e val and $e : \tau$, then $e \models \xi$. The form of ξ given by Rule (13.11b) ensures that $e \models \xi_i$ for some constraint ξ_i corresponding to the ith rule. By Lemma 13.4 the value e must match the ith rule, which is enough to ensure progress. □

13.4.3 Checking Exhaustiveness and Redundancy

Checking exhaustiveness and redundancy reduces to showing that the constraint validity judgment $\models \xi$ is decidable. We prove this by defining a judgment Ξ incon, where Ξ is a finite set of constraints of the same type, with the meaning that no value of this type satisfies all of the constraints in Ξ. We then show that either Ξ incon or not.

The rules defining inconsistency of a finite set Ξ of constraints of the same type are as follows:

$$\frac{\Xi\ \mathsf{incon}}{\Xi, \top\ \mathsf{incon}} \tag{13.13a}$$

$$\frac{\Xi, \xi_1, \xi_2\ \mathsf{incon}}{\Xi, \xi_1 \wedge \xi_2\ \mathsf{incon}} \tag{13.13b}$$

$$\frac{}{\Xi, \bot\ \mathsf{incon}} \tag{13.13c}$$

$$\frac{\Xi, \xi_1 \text{ incon} \quad \Xi, \xi_2 \text{ incon}}{\Xi, \xi_1 \vee \xi_2 \text{ incon}} \tag{13.13d}$$

$$\frac{}{\Xi, \mathtt{l} \cdot \xi_1, \mathtt{r} \cdot \xi_2 \text{ incon}} \tag{13.13e}$$

$$\frac{\Xi \text{ incon}}{\mathtt{l} \cdot \Xi \text{ incon}} \tag{13.13f}$$

$$\frac{\Xi \text{ incon}}{\mathtt{r} \cdot \Xi \text{ incon}} \tag{13.13g}$$

$$\frac{\Xi_1 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \tag{13.13h}$$

$$\frac{\Xi_2 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \,. \tag{13.13i}$$

In Rule (13.13f) we write $\mathtt{l} \cdot \Xi$ for the finite set of constraints $\mathtt{l} \cdot \xi_1, \ldots, \mathtt{l} \cdot \xi_n$, where $\Xi = \xi_1, \ldots, \xi_n$, and similarly in Rules (13.13g), (13.13h), and (13.13i).

Lemma 13.6. *It is decidable whether Ξ incon.*

Proof The premise of each rule involves only constraints that are proper components of the constraints in the conclusion. Consequently we can simplify Ξ by inverting each of the applicable rules until no rule applies; then we determine whether the resulting set Ξ' is contradictory in the sense that it contains $\bot$ or both $\mathtt{l} \cdot \xi$ and $\mathtt{r} \cdot \xi'$ for some ξ and ξ'. □

Lemma 13.7. *Ξ incon iff $\Xi \models \bot$.*

Proof From left to right we proceed by induction on Rules (13.13). From right to left we may show that if Ξ incon is not derivable, then there exists a value e such that $e \models \Xi$, and hence $\Xi \not\models \bot$. □

13.5 Notes

Pattern matching against heterogeneous structured data was first explored in the context of logic programming languages, such as Prolog (Kowalski, 1988; Colmerauer and Roussel, 1993), but with an execution model based on proof search. Pattern matching in the form described here is present in the functional languages Miranda (Turner, 1987), Hope (Burstall et al., 1980), Standard ML (Milner et al., 1997), Caml (Cousineau and Mauny, 1998), and Haskell (Jones, 2003).

14 Generic Programming

14.1 Introduction

Many programs can be seen as instances of a general pattern applied to a particular situation. Very often the pattern is determined by the types of the data involved. For example, in Chapter 9 the pattern of computing by recursion over a natural number is isolated as the defining characteristic of the type of natural numbers. This concept will itself emerge as an instance of the concept of *type-generic*, or just *generic*, programming.

Suppose that we have a function f of type $\rho \to \rho'$ that transforms values of type ρ into values of type ρ'. For example, f might be the doubling function on natural numbers. We wish to extend f to a transformation from type $[\rho/t]\tau$ to type $[\rho'/t]\tau$ by applying f to various spots in the input where a value of type ρ occurs to obtain a value of type ρ', leaving the rest of the data structure alone. For example, τ might be $\mathtt{bool} \times \rho$, in which case f could be extended to a function of type $\mathtt{bool} \times \rho \to \mathtt{bool} \times \rho'$ that sends the pairs $\langle a, b\rangle$ to the pair $\langle a, f\,(b)\rangle$.

This example glosses over a significant problem of ambiguity of the extension. Given a function f of type $\rho \to \rho'$, it is not obvious in general how to extend it to a function mapping $[\rho/t]\tau$ to $[\rho'/t]\tau$. The problem is that it is not clear which of many occurrences of ρ in $[\rho/t]\tau$ are to be transformed by f, even if there is only one occurrence of ρ. To avoid ambiguity we need a way to mark which occurrences of ρ in $[\rho/t]\tau$ are to be transformed, and which are to be left fixed. This can be achieved by isolating the type operator $t\,.\,\tau$, which is a type expression in which a designated variable t marks the spots at which we wish the transformation to occur. Given $t\,.\,\tau$ and $f : \rho \to \rho'$, we can extend f unambiguously to a function of type $[\rho/t]\tau \to [\rho'/t]\tau$.

The technique of using a type operator to determine the behavior of a piece of code is called *generic programming*. The power of generic programming depends on which forms of type operator are considered. The simplest case is that of a polynomial type operator, one constructed from sum and product of types, including their nullary forms. These may be extended to *positive* type operators, which also permit restricted forms of function types.

14.2 Type Operators

A *type operator* is a type equipped with a designated variable whose occurrences mark the positions in the type where a transformation is to be applied. A type operator is represented by an abstractor $t\,.\,\tau$ such that t type $\vdash$ τ type. An example of a type operator is the abstractor

$$t\,.\,\mathtt{unit} + (\mathtt{bool} \times t)$$

in which occurrences of t mark the spots in which a transformation is to be applied. An *instance* of the type operator $t\,.\,\tau$ is obtained by substituting a type, ρ, for the variable t, within the type τ. We sometimes write $\mathtt{Map}[t\,.\,\tau](\rho)$ for the substitution instance $[\rho/t]\tau$.

The *polynomial* type operators are those constructed from the type variable t, the types `void` and `unit`, and the product and sum type constructors $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. It is a straightforward exercise to give inductive definitions of the judgment $t\,.\,\tau$ poly stating that the operator $t\,.\,\tau$ is a polynomial type operator.

14.3 Generic Extension

The *generic extension* primitive has the form

$$\mathtt{map}[t\,.\,\tau]\,(x\,.\,e';e)$$

with statics given by the following rule:

$$\frac{t \text{ type} \vdash \tau \text{ type} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{map}[t\,.\,\tau]\,(x\,.\,e';e) : [\rho'/t]\tau}\,. \tag{14.1}$$

The abstractor $x\,.\,e'$ specifies a transformation from type ρ, the type of x, to type ρ', the type of e'. The expression e of type $[\rho/t]\tau$ determines the value to be transformed to obtain a value of type $[\rho'/t]\tau$. The occurrences of t in τ determine the spots at which the transformation given by $x\,.\,e$ is to be performed.

The dynamics of generic extension is specified by the following rules. We consider here only polynomial type operators, leaving the extension to positive type operators to be considered later.

$$\frac{}{\mathtt{map}[t\,.\,t]\,(x\,.\,e';e) \mapsto [e/x]e'} \tag{14.2a}$$

$$\frac{}{\mathtt{map}[t\,.\,\mathtt{unit}]\,(x\,.\,e';e) \mapsto \langle\rangle} \tag{14.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t\,.\,\tau_1 \times \tau_2]\,(x\,.\,e';e) \\ \mapsto \\ \langle \mathtt{map}[t\,.\,\tau_1]\,(x\,.\,e';e\cdot \mathtt{l}), \mathtt{map}[t\,.\,\tau_2]\,(x\,.\,e';e\cdot \mathtt{r})\rangle\end{array}} \tag{14.2c}$$

$$\frac{}{\texttt{map}[t\,.\,\texttt{void}]\,(x\,.\,e';e) \mapsto \texttt{abort}(e)} \tag{14.2d}$$

$$\frac{}{\begin{array}{c}\texttt{map}[t\,.\,\tau_1+\tau_2]\,(x\,.\,e';e)\\ \mapsto\\ \texttt{case}\,e\,\{\texttt{l}\cdot x_1 \Rightarrow \texttt{l}\cdot\texttt{map}[t\,.\,\tau_1]\,(x\,.\,e';x_1)\mid \texttt{r}\cdot x_2 \Rightarrow \texttt{r}\cdot\texttt{map}[t\,.\,\tau_2]\,(x\,.\,e';x_2)\}.\end{array}} \tag{14.2e}$$

Rule (14.2a) applies the transformation $x\,.\,e'$ to e itself, because the operator $t\,.\,t$ specifies that the transformation is to be perfomed directly. Rule (14.2b) states that the empty tuple is transformed to itself. Rule (14.2c) states that to transform e according to the operator $t\,.\,\tau_1\times\tau_2$, the first component of e is transformed according to $t\,.\,\tau_1$ and the second component of e is transformed according to $t\,.\,\tau_2$. Rule (14.2d) states that the transformation of a value of type `void` aborts, because there can be no such values. Rule (14.2e) states that to transform e according to $t\,.\,\tau_1+\tau_2$, case analyze e and reconstruct it after transforming the injected value according to $t\,.\,\tau_1$ or $t\,.\,\tau_2$.

Consider the type operator $t\,.\,\tau$ given by $t\,.\,\texttt{unit}+(\texttt{bool}\times t)$. Let $x\,.\,e$ be the abstractor $x\,.\,\texttt{s}(x)$, which increments a natural number. Using Rules (14.2) we may derive that

$$\texttt{map}[t\,.\,\tau]\,(x\,.\,e;\texttt{r}\cdot\langle\texttt{true},n\rangle) \mapsto^* \texttt{r}\cdot\langle\texttt{true},n+1\rangle.$$

The natural number in the second component of the pair is incremented, because the type variable t occurs in that position in the type operator $t\,.\,\tau$.

Theorem 14.1 (Preservation)**.** *If* $\texttt{map}[t\,.\,\tau]\,(x\,.\,e';e):\tau'$ *and* $\texttt{map}[t\,.\,\tau]\,(x\,.\,e';e)\mapsto e''$*, then* $e'':\tau'$*.*

Proof By inversion of Rule (14.1) we have

1. t type $\vdash \tau$ type;
2. $x:\rho\vdash e':\rho'$ for some ρ and ρ';
3. $e:[\rho/t]\tau$;
4. τ' is $[\rho'/t]\tau$.

We proceed by cases on Rules (14.2). For example, consider Rule (14.2c). It follows from inversion that $\texttt{map}[t\,.\,\tau_1]\,(x\,.\,e';e\cdot\texttt{l}):[\rho'/t]\tau_1$ and similarly that $\texttt{map}[t\,.\,\tau_2]\,(x\,.\,e';e\cdot\texttt{r}):[\rho'/t]\tau_2$. It is easy to check that

$$\langle\texttt{map}[t\,.\,\tau_1]\,(x\,.\,e';e\cdot\texttt{l}),\texttt{map}[t\,.\,\tau_2]\,(x\,.\,e';e\cdot\texttt{r})\rangle$$

has type $[\rho'/t]\tau_1\times\tau_2$, as required. □

The *positive* type operators extend the polynomial type operators to admit restricted forms of function type. Specifically, $t\,.\,\tau_1\to\tau_2$ is a positive type operator, provided that (1) t *does not occur* in τ_1, and (2) $t\,.\,\tau_2$ is a positive type operator. In general, any occurrences of a type variable t in the domain of a function type are said to be *negative occurrences*,

whereas any occurrences of t within the range of a function type, or within a product or sum type, are said to be *positive occurrences*.[1] A positive type operator is one for which only positive occurrences of the parameter t are permitted.

The generic extension according to a positive type operator is defined similarly to the case of a polynomial type operator, with the following additional rule:

$$\frac{}{\mathtt{map}[t.\tau_1 \to \tau_2]\,(x.e';e) \mapsto \lambda\,(x_1:\tau_1)\,\mathtt{map}[t.\tau_2]\,(x.e';e(x_1))}\,. \tag{14.3}$$

Because t is not permitted to occur within the domain type, the type of the result is $\tau_1 \to [\rho'/t]\tau_2$, assuming that e is of type $\tau_1 \to [\rho/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

It is interesting to consider what goes wrong if we relax the restriction on positive type operators to admit negative, as well as positive, occurrences of the parameter of a type operator. Consider the type operator $t.\tau_1 \to \tau_2$, without restriction on t, and suppose that $x : \rho \vdash e' : \rho'$. The generic extension $\mathtt{map}[t.\tau_1 \to \tau_2]\,(x.e';e)$ should have type $[\rho'/t]\tau_1 \to [\rho'/t]\tau_2$, given that e has type $[\rho/t]\tau_1 \to [\rho/t]\tau_2$. The extension should yield a function of the form

$$\lambda\,(x_1:[\rho'/t]\tau_1)\ldots(e(\ldots(x_1)))$$

in which we apply e to a transformation of x_1 and then transform the result. The trouble is that we are given, inductively, that $\mathtt{map}[t.\tau_1]\,(x.e';-)$ transforms values of type $[\rho/t]\tau_1$ into values of type $[\rho'/t]\tau_1$, but *we need to go the other way around* in order to make x_1 suitable as an argument for e. Unfortunately, there is no obvious way to obtain the required transformation.

One solution to this is to assume that the fundamental transformation $x.e'$ is *invertible* so that we may apply the inverse transformation on x_1 to get an argument of type suitable for e, then apply the forward transformation on the result, just as in the positive case. Because we cannot invert an arbitrary transformation, we must instead pass both the transformation and its inverse to the generic extension operation so that it can "go backwards" as necessary to cover negative occurrences of the type parameter. So the generic extension applies only when we are given a *type isomorphism* (a pair of mutually inverse mappings between two types) and then results in another isomorphism pair.

14.4 Notes

The generic extension of a type operator is an example of the concept of a *functor* in category theory (MacLane, 1998). Generic programming is essentially *functorial* programming, exploiting the functorial action of polynomial type operators (Hinze and Jeuring, 2003).

[1] The origin of this terminology seems to be that a function type $\tau_1 \to \tau_2$ is analogous to the implication $\phi_1 \supset \phi_2$, which is classically equivalent to $\neg\phi_1 \vee \phi_2$, so that occurrences in the domain are under the negation.

PART V

Infinite Data Types

15 Inductive and Coinductive Types

The *inductive* and the *coinductive* types are two important forms of recursive type. Inductive types correspond to *least*, or *initial*, solutions of certain type isomorphism equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, the elements of an inductive type are those that may be obtained by a finite composition of its introductory forms. Consequently, if we specify the behavior of a function on each of the introductory forms of an inductive type, then its behavior is determined for all values of that type. Such a function is called a *recursor*, or *catamorphism*. Dually, the elements of a coinductive type are those that behave properly in response to a finite composition of its elimination forms. Consequently, if we specify the behavior of an element on each elimination form, then we have fully specified that element as a value of that type. Such an element is called an *generator*, or *anamorphism*.

15.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 9. The type `nat` is defined to be the least type containing `z` and closed under `s(-)`. The minimality condition is witnessed by the existence of the recursor, $\texttt{iter}\, e\, \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$, which transforms a natural number into a value of type τ, given its value for zero, and a transformation from its value on a number to its value on the successor of that number. This operation is well-defined precisely because there are no other natural numbers. Put the other way around, the existence of this operation expresses the inductive nature of the type `nat`.

With a view toward deriving the type `nat` as a special case of an inductive type, it is useful to consolidate zero and successor into a single introductory form and to correspondingly consolidate the basis and inductive step of the recursor. The following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \texttt{unit} + \texttt{nat}}{\Gamma \vdash \texttt{fold}_{\texttt{nat}}(e) : \texttt{nat}} \tag{15.1a}$$

$$\frac{\Gamma, x : \texttt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{rec}_{\texttt{nat}}[x.e_1](e_2) : \tau}. \tag{15.1b}$$

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introductory form of the type nat. Using this, we define the expression z as $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$ and $\mathtt{s}(e)$ as $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor $\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2)$ takes as argument the abstractor $x.e_1$ that consolidates the basis and inductive step into a single computation that, given a value of type $\mathtt{unit} + \tau$, yields a value of type τ. Intuitively, if x is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\mathtt{r} \cdot e$, then e_1 computes the inductive step as a function of the result e of the recursive call.

The dynamics of the consolidated representation of natural numbers is given by the following rules:

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathsf{val}} \tag{15.2a}$$

$$\frac{e_2 \mapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2) \mapsto \mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2')} \tag{15.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}[x.e_1](\mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \mapsto \\ [\mathtt{map}[t.\mathtt{unit} + t](y.\mathtt{rec}_{\mathtt{nat}}[x.e_1](y); e_2)/x]e_1.\end{array}} \tag{15.2c}$$

Rule (15.2c) makes use of generic extension (see Chapter 5) to apply the recursor to the predecessor, if any, of a natural number. The idea is that the result of extending the recursor from the type $\mathtt{unit} + \mathtt{nat}$ to the type $\mathtt{unit} + \tau$ is substituted into the inductive step, given by the expression e_1. If we expand the definition of the generic extension in place, we obtain the following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}[x.e_1](\mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \mapsto \\ [\mathtt{case}\, e_2\, \{\mathtt{l} \cdot \_ \Rightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{r} \cdot y \Rightarrow \mathtt{r} \cdot \mathtt{rec}_{\mathtt{nat}}[x.e_1](y)\}/x]e_1.\end{array}}$$

An illustrative example of a coinductive type is the type of *streams* of natural numbers. A stream is an infinite sequence of natural numbers such that an element of the stream can be computed only after computing all preceding elements in that stream. That is, the computations of successive elements of the stream are sequentially dependent in that the computation of one element influences the computation of the next. This characteristic of the introductory form for streams is *dual* to the analogous property of the eliminatory form for natural numbers whereby the result for a number is determined by its result for all preceding numbers.

A stream is characterized by its behavior under the elimination forms for the stream type: $\mathtt{hd}(e)$ returns the next, or head, element of the stream, and $\mathtt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of a recursor, that determines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{nat}} \tag{15.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}(e) : \mathtt{stream}} \tag{15.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\, e\, \{\mathtt{hd}(x) \Rightarrow e_1 \mid \mathtt{tl}(x) \Rightarrow e_2\} : \mathtt{stream}}\,. \tag{15.3c}$$

In Rule (15.3c) the current state of the stream is given by the expression e of some type τ, and the head and tail of the stream are determined by the expressions e_1 and e_2, respectively, as functions of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\, e\, \{\mathtt{hd}(x) \Rightarrow e_1 \mid \mathtt{tl}(x) \Rightarrow e_2\}\ \mathsf{val}} \tag{15.4a}$$

$$\frac{e \mapsto e'}{\mathtt{hd}(e) \mapsto \mathtt{hd}(e')} \tag{15.4b}$$

$$\frac{}{\mathtt{hd}(\mathtt{strgen}\, e\, \{\mathtt{hd}(x) \Rightarrow e_1 \mid \mathtt{tl}(x) \Rightarrow e_2\}) \mapsto [e/x]e_1} \tag{15.4c}$$

$$\frac{e \mapsto e'}{\mathtt{tl}(e) \mapsto \mathtt{tl}(e')} \tag{15.4d}$$

$$\frac{}{\begin{array}{c}\mathtt{tl}(\mathtt{strgen}\, e\, \{\mathtt{hd}(x) \Rightarrow e_1 \mid \mathtt{tl}(x) \Rightarrow e_2\}) \\ \mapsto \\ \mathtt{strgen}\, [e/x]e_2\, \{\mathtt{hd}(x) \Rightarrow e_1 \mid \mathtt{tl}(x) \Rightarrow e_2\}\end{array}}\,. \tag{15.4e}$$

Rules (15.4c) and (15.4e) express the dependency of the head and tail of the stream on its current state. Observe that the tail is obtained by applying the generator to the new state determined by e_2 as a function of the current state.

To derive streams as a special case of a coinductive type, we consolidate the head and the tail into a single eliminatory form, and reorganize the generator correspondingly. This leads to the following statics:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{unfold}_{\mathtt{stream}}(e) : \mathtt{nat} \times \mathtt{stream}} \tag{15.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathtt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2) : \mathtt{stream}}\,. \tag{15.5b}$$

Rule (15.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head $\mathtt{hd}(e)$ and tail $\mathtt{tl}(e)$ of a stream e are defined to be the projections $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{l}$ and $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{r}$, respectively. Rule (15.5b) states that a stream may be generated from the state element e_2 by an expression e_1 that yields the head element and the next state as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)\ \mathsf{val}} \tag{15.6a}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}_{\mathtt{stream}}(e) \mapsto \mathtt{unfold}_{\mathtt{stream}}(e')} \tag{15.6b}$$

$$\frac{}{\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)) \mapsto \mathtt{map}[t.\mathtt{nat} \times t](y.\mathtt{gen}_{\mathtt{stream}}[x.e_1](y); [e_2/x]e_1).} \tag{15.6c}$$

Rule (15.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\frac{}{\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)) \mapsto \langle([e_2/x]e_1) \cdot \mathtt{l}, \mathtt{gen}_{\mathtt{stream}}[x.e_1](([e_2/x]e_1) \cdot \mathtt{r})\rangle.}$$

15.2 Statics

We may now give a fully general account of inductive and coinductive types, which are defined in terms of positive type operators. We consider the language, $\mathcal{L}\{\mu_{\mathrm{i}}\mu_{\mathrm{f}}\}$, with inductive and coinductive types.

15.2.1 Types

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over types. The abstract syntax of inductive and coinductive types is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	$::=$	t	t	self-reference
			$\mathtt{ind}(t.\tau)$	$\mu_{\mathrm{i}}(t.\tau)$	inductive
			$\mathtt{coi}(t.\tau)$	$\mu_{\mathrm{f}}(t.\tau)$	coinductive

Type formation judgements have the form

$$t_1\ \mathsf{type}, \ldots, t_n\ \mathsf{type} \vdash \tau\ \mathsf{type},$$

where $t_1, \ldots, t_n$ are type names. We let Δ range over finite sets of hypotheses of the form $t\ \mathsf{type}$, where t is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{15.7a}$$

$$\frac{}{\Delta \vdash \mathtt{unit}\ \mathsf{type}} \tag{15.7b}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{prod}(\tau_1;\tau_2) \ \mathsf{type}} \tag{15.7c}$$

$$\frac{}{\Delta \vdash \mathtt{void} \ \mathsf{type}} \tag{15.7d}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{sum}(\tau_1;\tau_2) \ \mathsf{type}} \tag{15.7e}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{15.7f}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type} \quad \Delta \vdash t.\tau \ \mathsf{pos}}{\Delta \vdash \mathtt{ind}(t.\tau) \ \mathsf{type}} \tag{15.7g}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type} \quad \Delta \vdash t.\tau \ \mathsf{pos}}{\Delta \vdash \mathtt{coi}(t.\tau) \ \mathsf{type}}\,. \tag{15.8}$$

15.2.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Exp	e	$::=$	$\mathtt{fold}[t.\tau](e)$	$\mathtt{fold}(e)$	constructor
			$\mathtt{rec}[t.\tau][x.e_1](e_2)$	$\mathtt{rec}[x.e_1](e_2)$	recursor
			$\mathtt{unfold}[t.\tau](e)$	$\mathtt{unfold}(e)$	destructor
			$\mathtt{gen}[t.\tau][x.e_1](e_2)$	$\mathtt{gen}[x.e_1](e_2)$	generator

The statics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{ind}(t.\tau)} \tag{15.9a}$$

$$\frac{\Gamma, x : [\rho/t]\tau \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \mathtt{ind}(t.\tau)}{\Gamma \vdash \mathtt{rec}[t.\tau][x.e_1](e_2) : \rho} \tag{15.9b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{unfold}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{15.9c}$$

$$\frac{\Gamma \vdash e_2 : \rho \quad \Gamma, x : \rho \vdash e_1 : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau][x.e_1](e_2) : \mathtt{coi}(t.\tau)}\,. \tag{15.9d}$$

15.3 Dynamics

The dynamics of inductive and coinductive types is given in terms of the generic extension operation described in Chapter 14. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\}$:

$$\frac{}{\texttt{fold}(e)\ \mathsf{val}} \tag{15.10a}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{rec}[x.e_1](e_2) \mapsto \texttt{rec}[x.e_1](e_2')} \tag{15.10b}$$

$$\frac{}{\begin{array}{c}\texttt{rec}[x.e_1](\texttt{fold}(e_2)) \\ \mapsto \\ [\texttt{map}[t.\tau](y.\texttt{rec}[x.e_1](y); e_2)/x]e_1\end{array}} \tag{15.10c}$$

$$\frac{}{\texttt{gen}[x.e_1](e_2)\ \mathsf{val}} \tag{15.10d}$$

$$\frac{e \mapsto e'}{\texttt{unfold}(e) \mapsto \texttt{unfold}(e')} \tag{15.10e}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}(\texttt{gen}[x.e_1](e_2)) \\ \mapsto \\ \texttt{map}[t.\tau](y.\texttt{gen}[x.e_1](y); [e_2/x]e_1).\end{array}} \tag{15.10f}$$

Rule (15.10c) states that to evaluate the recursor on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value and then perform the inductive step on the result. Rule (15.10f) is simply the dual of this rule for coinductive types.

Lemma 15.1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof By rule induction on Rules (15.10). □

Lemma 15.2. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof By rule induction on Rules (15.9). □

15.4 Notes

The treatment of inductive and coinductive types is derived from the work by Mendler (1987), which is based on the categorial analysis of these concepts (MacLane, 1998;

Taylor, 1999). The functorial action of a type constructor (described in Chapter 14) plays a central role. Specifically, inductive types are initial algebras and coinductive types are final coalgebras for a functor given by a composition of type constructors. The positivity requirement imposed on well-formed inductive and coinductive types ensures that the action of the associated type constructor is properly functorial.

16 Recursive Types

Inductive and coinductive types, such as natural numbers and streams, may be seen as examples of *fixed points* of type operators *up to isomorphism*. An isomorphism between two types, τ_1 and τ_2, is given by two expressions,

1. $x_1 : \tau_1 \vdash e_2 : \tau_2$, and
2. $x_2 : \tau_2 \vdash e_1 : \tau_1$

which are mutually inverse to each other.[1] For example, the types $\mathtt{nat}$ and $\mathtt{unit} + \mathtt{nat}$ are isomorphic, as witnessed by the following two expressions:

1. $x : \mathtt{unit} + \mathtt{nat} \vdash \mathtt{case}\, x\, \{\mathtt{l} \cdot \_ \Rightarrow \mathtt{z} \mid \mathtt{r} \cdot x_2 \Rightarrow \mathtt{s}(x_2)\} : \mathtt{nat}$, and
2. $x : \mathtt{nat} \vdash \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{s}(x_2) \Rightarrow \mathtt{r} \cdot x_2\} : \mathtt{unit} + \mathtt{nat}$.

These are called the $\mathtt{fold}$ and $\mathtt{unfold}$ operations, respectively, of the isomorphism $\mathtt{nat} \cong \mathtt{unit} + \mathtt{nat}$. Thinking of $\mathtt{unit} + \mathtt{nat}$ as $[\mathtt{nat}/t](\mathtt{unit} + t)$, this means that $\mathtt{nat}$ is a fixed point of the type operator $t\,.\,\mathtt{unit} + t$.

In this chapter we study the language $\mathcal{L}\{+ \times \rightharpoonup \mu\}$, which provides solutions to all such type equations. The *recursive type* $\mu t\,.\,\tau$ is defined to be a solution to the isomorphism problem

$$\mu t\,.\,\tau \cong [\mu t\,.\,\tau/t]\tau.$$

This is witnessed by the operations

$$x : \mu t\,.\,\tau \vdash \mathtt{unfold}(x) : [\mu t\,.\,\tau/t]\tau$$

and

$$x : [\mu t\,.\,\tau/t]\tau \vdash \mathtt{fold}(x) : \mu t\,.\,\tau,$$

which are mutually inverse to each other.

Requiring solutions to all type equations may seem suspicious, because we know by Cantor's Theorem that an isomorphism such as $X \cong (X \to \mathbf{2})$ is set-theoretically impossible. This negative result tells us not that our requirement is untenable, but rather that *types are not sets*. To permit solutions of arbitrary type equations, we must take into account

[1] To make this precise requires a discussion of equivalence of expressions, which is taken up in Chapter 47. For now we rely on an intuitive understanding of when two expressions are equivalent.

that types describe computations, some of which may not even terminate. Consequently the function space does not coincide with the set-theoretic function space, but rather is analogous to it (in a precise sense that we do not go into here).

16.1 Solving Type Isomorphisms

The recursive type $\mu t.\tau$, where $t.\tau$ is a type operator, represents a solution for t to the isomorphism $t \cong \tau$. The solution is witnessed by two operations, $\texttt{fold}(e)$ and $\texttt{unfold}(e)$, that relate the recursive type $\mu t.\tau$ to its unfolding $[\mu t.\tau/t]\tau$ and serve as its introduction and elimination forms, respectively.

The language $\mathcal{L}\{+\times\rightharpoonup\mu\}$ extends $\mathcal{L}\{\rightharpoonup\}$ with recursive types and their associated operations:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	$::=$	t	t	self-reference
			$\texttt{rec}(t.\tau)$	$\mu t.\tau$	recursive
Exp	e	$::=$	$\texttt{fold}[t.\tau](e)$	$\texttt{fold}(e)$	constructor
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$	destructor

The statics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ consists of two forms of judgment. The first, called *type formation*, is a general hypothetical judgment of the form

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

where Δ has the form $t_1 \ \mathsf{type}, \ldots, t_k \ \mathsf{type}$. Type formation is inductively defined by the following rules:

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

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

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

The second form of judgment of the statics is the *typing judgment*, which is a hypothetical judgment of the form

$$\Gamma \vdash e : \tau,$$

where we assume that $\tau \ \mathsf{type}$. Typing for $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is inductively defined by the following rules:

$$\frac{\Gamma \vdash e : [\texttt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{fold}[t.\tau](e) : \texttt{rec}(t.\tau)} \tag{16.2a}$$

$$\frac{\Gamma \vdash e : \texttt{rec}(t.\tau)}{\Gamma \vdash \texttt{unfold}(e) : [\texttt{rec}(t.\tau)/t]\tau}. \tag{16.2b}$$

The dynamics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is specified by one axiom stating that the elimination form is inverse to the introduction form:

$$\frac{[e\ \mathsf{val}]}{\mathtt{fold}[t.\tau]\,(e)\ \mathsf{val}} \tag{16.3a}$$

$$\left[\frac{e \mapsto e'}{\mathtt{fold}[t.\tau]\,(e) \mapsto \mathtt{fold}[t.\tau]\,(e')}\right] \tag{16.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{16.3c}$$

$$\frac{\mathtt{fold}[t.\tau]\,(e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}[t.\tau]\,(e)) \mapsto e}\,. \tag{16.3d}$$

The bracketed premise and rule are to be included for an *eager* interpretation of the introduction form and omitted for a *lazy* interpretation.

It is a straightforward exercise to prove type safety for $\mathcal{L}\{+\times\rightharpoonup\mu\}$.

Theorem 16.1 (Safety)**.**

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

16.2 Recursive Data Structures

One important application of recursive types is to the representation of inductive data types such as the type of natural numbers. We may think of the type `nat` as a solution (up to isomorphism) of the type equation

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

According to this isomorphism every natural number is either zero or the successor of another natural number. A solution is given by the recursive type

$$\mu t\,.\,[\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t]. \tag{16.4}$$

The introductory forms for the type `nat` are defined by the following equations:

$$\begin{aligned}\mathtt{z} &= \mathtt{fold}(\mathtt{z} \cdot \langle\rangle)\\ \mathtt{s}(e) &= \mathtt{fold}(\mathtt{s} \cdot e).\end{aligned}$$

The conditional branch may then be defined as follows:

$$\mathtt{ifz}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\} \quad = \quad \mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{z} \cdot \_ \Rightarrow e_0 \mid \mathtt{s} \cdot x \Rightarrow e_1\},$$

where the "underscore" indicates a variable that does not occur free in e_0. It is easy to check that these definitions exhibit the expected behavior.

As another example, the type `list` of lists of natural numbers may be represented by the recursive type

$$\mu t\,.\,[\mathtt{n} \hookrightarrow \mathtt{unit}, \mathtt{c} \hookrightarrow \mathtt{nat} \times t]$$

so that we have the isomorphism

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

The list formation operations are represented by the following equations:

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

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned} &\mathtt{case}\, e\, \{\mathtt{nil} \Rightarrow e_0 \mid \mathtt{cons}(x; y) \Rightarrow e_1\} \\ &\quad = \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{n} \cdot \_ \Rightarrow e_0 \mid \mathtt{c} \cdot \langle x, y \rangle \Rightarrow e_1\}, \end{aligned}$$

where we have used an underscore for a "don't care" variable and pattern-matching syntax to bind the components of a pair.

As long as sums and products are evaluated eagerly, there is a natural correspondence between this representation of lists and the conventional "blackboard notation" for linked lists. We may think of `fold` as an abstract heap-allocated pointer to a tagged cell consisting of either (a) the tag `n` with no associated data, or (b) the tag `c` attached to a pair consisting of a natural number and another list, which must be an abstract pointer of the same sort. If sums or products are evaluated lazily, then the blackboard notation breaks down because it is unable to depict the suspended computations that are present in the data structure. In general there is no substitute for the type itself. Drawings can be helpful but the type determines the semantics.

We may also represent coinductive types, such as the type of streams of natural numbers, using recursive types. The representation is particularly natural in the case that `fold(−)` is evaluated lazily, for then we may define the type `stream` to be the recursive type

$$\mu t\,.\,\mathtt{nat} \times t.$$

This states that every stream may be thought of as a computation of a pair consisting of a number and another stream. If `fold(−)` is evaluated eagerly, then we may instead consider the recursive type

$$\mu t\,.\,\mathtt{unit} \to (\mathtt{nat} \times t),$$

which expresses the same representation of streams. In either case streams cannot be easily depicted in blackboard notation, not so much because they are infinite, but because there is no accurate way to depict the delayed computation other than by an expression in the programming language. Here again we see that pictures can be helpful but are not adequate for accurately defining a data structure.

16.3 Self-Reference

In the general recursive expression $\texttt{fix}[\tau](x.e)$, the variable x stands for the expression itself. This is ensured by the unrolling transition

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

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 implicitly instantiated to itself whenever 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 rather than as an ad hoc language feature. The derivation is based on isolating a type of self-referential expressions given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	$\texttt{self}(\tau)$	$\tau\ \texttt{self}$	self-referential type
Exp	e	::=	$\texttt{self}[\tau](x.e)$	$\texttt{self}\ x\ \texttt{is}\ e$	self-referential expression
			$\texttt{unroll}(e)$	$\texttt{unroll}(e)$	unroll self-reference

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

$$\frac{\Gamma, x : \texttt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{self}[\tau](x.e) : \texttt{self}(\tau)} \tag{16.5a}$$

$$\frac{\Gamma \vdash e : \texttt{self}(\tau)}{\Gamma \vdash \texttt{unroll}(e) : \tau}\,. \tag{16.5b}$$

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

$$\frac{}{\texttt{self}[\tau](x.e)\ \texttt{val}} \tag{16.6a}$$

$$\frac{e \mapsto e'}{\texttt{unroll}(e) \mapsto \texttt{unroll}(e')} \tag{16.6b}$$

$$\frac{}{\texttt{unroll}(\texttt{self}[\tau](x.e)) \mapsto [\texttt{self}[\tau](x.e)/x]e}\,. \tag{16.6c}$$

The main difference, compared with general recursion, is that we distinguish a type of self-referential expression, rather than impose self-reference at every type. However, as will become clear shortly, the self-referential type is sufficient to implement general recursion, so the difference is largely one of technique.

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

$$\texttt{self}(\tau) \cong \texttt{self}(\tau) \to \tau.$$

This means that we seek a fixed point of the type operator $t\,.\,t \to \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\texttt{rec}(t\,.\,t \to \tau),$$

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

The self-referential expression $\texttt{self}[\tau](x\,.\,e)$ is then defined to be the expression

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

We may easily check that Rule (16.5a) is derivable according to this definition. The expression $\texttt{unroll}(e)$ is correspondingly defined to be the expression

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

It is easy to check that Rule (16.5b) is derivable from this definition. Moreover, we may check that

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

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

One consequence of admitting the self-referential type $\texttt{self}(\tau)$ is that we may use it to define general recursion for *any* type. To be precise, we may define $\texttt{fix}[\tau](x\,.\,e)$ to stand for the expression

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

in which we have unrolled the recursion at each occurrence of x within e. It is easy to check that this verifies the statics of general recursion given in Chapter 10. Moreover, it also validates the dynamics, as evidenced by the following derivation:

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

It follows that recursive types may be used to define a nonterminating expression of every type, namely $\texttt{fix}[\tau](x\,.\,x)$. Unlike many other type constructs we have considered, recursive types change the meaning of *every* type, not just those that involve recursion. Recursive types are therefore said to be a *nonconservative extension* of languages such as $\mathcal{L}\{\texttt{nat}\to\}$, which otherwise admits no nonterminating computations.

16.4 The Origin of State

The notion of *state* in a computation—which is discussed thoroughly in Part XIII—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, you may be familiar with the concept of a *flip-flop* or a *latch* at the hardware level. These are circuits 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 is fed back into its input so as to convey the current state of the gate to the logic that determines its next state.

One way to model an RS latch using recursive types is to make explicit the passage of time in the determination of the current output of the gate as a function of its inputs and its previous outputs. An RS latch is a value of type τ_{rsl} given by

$$\mu t\,.\,\langle \mathtt{X} \hookrightarrow \mathtt{bool}, \mathtt{Q} \hookrightarrow \mathtt{bool}, \mathtt{N} \hookrightarrow t \rangle.$$

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 $e\,\mathtt{@}\,\mathtt{X}$ to mean $\mathtt{unfold}(e)\cdot\mathtt{X}$ and define $e\,\mathtt{@}\,\mathtt{Q}$ and $e\,\mathtt{@}\,\mathtt{N}$ similarly. The expressions $e\,\mathtt{@}\,\mathtt{X}$ and $e\,\mathtt{@}\,\mathtt{Q}$ evaluate to the "current" outputs of the latch e, and $e\,\mathtt{@}\,\mathtt{N}$ evaluates to another latch representing the "next" state determined as a function of the "current" state.[2]

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

$$\mathtt{fix}\,rsl\,\mathtt{is}\,\lambda\ (o\!:\!\tau_{rsl})\,\mathtt{fix}\,this\,\mathtt{is}\,e_{rsl},$$

where e_{rsl} is given by

$$\mathtt{fold}(\langle \mathtt{X} \hookrightarrow nor(\langle s, o\,\mathtt{@}\,\mathtt{Q}\rangle), \mathtt{Q} \hookrightarrow nor(\langle r, o\,\mathtt{@}\,\mathtt{X}\rangle), \mathtt{N} \hookrightarrow rsl(this)\rangle)$$

and *nor* is the obvious function defined on the Booleans.[3] The outputs of the latch are computed as functions of the r and s inputs and the ouputs 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 *rsl* to the initial state:

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

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

The foregoing implementation of a latch models time explicitly by providing the `N` component of the latch to compute the next state from the current one. It is also possible to model time implicitly by treating the latch as a *transducer* whose inputs and outputs are *signals* that change over time. A signal may be represented by a stream of Booleans (as described in Chapter 15 or using general recursive types as described earlier in this chapter), in which case a transducer is a stream transformer that computes the successive elements of

[2] For simplicity the `R` and `S` inputs are fixed, which amounts to requiring that we build a new latch whenever these are changed. It is straightforward to modify the construction so that new `R` and `S` inputs may be provided whenever the next state of a latch is computed, allowing for these inputs to change over time.

[3] It suffices to require that `fold` be evaluated lazily to ensure that recursion is well-grounded. This assumption is unnecessary if the next state component is abstracted on the `R` and `S` inputs, as suggested earlier.

the outputs from the successive elements of the inputs by applying a function to them. This implicit formulation is arguably more natural than the explicit one previously given, but it nevertheless relies on recursive types and self-reference, just as does the implementation previously given.

16.5 Notes

The systematic study of recursive types in programming was initiated by Scott (1976, 1982) to provide a mathematical model of the untyped λ-calculus. The derivation of recursion from recursive types is essentially an application of Scott's theory to find the interpretation of a fixed point combinator in a model of the λ-calculus given by a recursive type. The 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. Abadi and Cardelli (1996) and Cook (2009), among others, explore similar ideas to model objects. The account of signals as streams is inspired by the pioneering work of Kahn (MacQueen, 2009).

PART VI

Dynamic Types

17 The Untyped λ-Calculus

Types are the central organizing principle in the study of programming languages. Yet many languages of practical interest are said to be untyped. Have we missed something important? The answer is *no*: the supposed opposition between typed and untyped languages is illusory. In fact, untyped languages are special cases of typed languages with a single recursive type. Far from being untyped, such languages are unityped.

In this chapter we study the premier example of a unityped programming language, the *(untyped) λ-calculus*. This formalism was introduced by Church in the 1930s 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, and 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.

17.1 The λ-Calculus

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

Sort			Abstract Form	Concrete Form	Description
Exp	u	$::=$	x	x	variable
			$\lambda(x.u)$	$\lambda\ (x)\ u$	λ-abstraction
			$\mathtt{ap}(u_1;u_2)$	$u_1(u_2)$	application

The statics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical 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 parameters when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{17.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(u_1;u_2)\ \mathsf{ok}} \tag{17.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda(x.u)\ \mathsf{ok}}\,. \tag{17.1c}$$

The dynamics of $\mathcal{L}\{\lambda\}$ is given equationally, rather than via a transition system. Definitional equality for $\mathcal{L}\{\lambda\}$ is a judgment of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok}$ for some $n \geq 0$, and u and u' are terms having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u\ \mathsf{ok} \vdash u \equiv u} \tag{17.2a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{17.2b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{17.2c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2')} \tag{17.2d}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x\,.\,u) \equiv \lambda(x\,.\,u')} \tag{17.2e}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash e_2\ \mathsf{ok} \quad \Gamma \vdash e_1\ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(\lambda(x\,.\,e_2); e_1) \equiv [e_1/x]e_2}\,. \tag{17.2f}$$

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

17.2 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 is certainly true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration in a gravitational field.[1]

We sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 10. The main idea is to show that the PCF primitives for manipulating

[1] Unfortunately, it is common in computer science to put forth as "laws" assertions that are not scientific laws at all. For example, Moore's Law is merely an observation about a near-term trend in microprocessor fabrication that is certainly not valid over the long term, and Amdahl's Law is but a simple truth of arithmetic. Worse, Church's Law, which is a proper scientific law, is usually called *Church's Thesis*, which, to the author's ear, suggests something less than the full force of a scientific law.

the natural numbers are definable in the untyped λ-calculus. This means, in particular, that we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and nonzero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

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

$$\overline{0} \triangleq \lambda\,(b)\,\lambda\,(s)\,b \tag{17.3a}$$

$$\overline{n+1} \triangleq \lambda\,(b)\,\lambda\,(s)\,s(\overline{n}(b)(s)). \tag{17.3b}$$

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, we do not find it difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped λ-terms:

$$\mathtt{succ} \triangleq \lambda\,(x)\,\lambda\,(b)\,\lambda\,(s)\,s(x(b)(s)) \tag{17.4}$$

$$\mathtt{plus} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(x)(\mathtt{succ}) \tag{17.5}$$

$$\mathtt{times} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(\overline{0})(\mathtt{plus}(x)). \tag{17.6}$$

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 $\mathtt{ifz}(u; u_0; x\,.\,u_1)$ requires a bit of ingenuity. We wish to find a term $\mathtt{pred}$ such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{17.7}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{17.8}$$

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

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

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

$$\langle u_1, u_2\rangle \triangleq \lambda\,(f)\; f\,(u_1)\,(u_2) \tag{17.9}$$

$$u \cdot \mathtt{l} \triangleq u\,(\lambda\,(x)\;\lambda\,(y)\;x) \tag{17.10}$$

$$u \cdot \mathtt{r} \triangleq u\,(\lambda\,(x)\;\lambda\,(y)\;y). \tag{17.11}$$

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:

$$u'_p \triangleq \lambda\,(x)\;x\,(\langle\overline{0},\overline{0}\rangle)\,(\lambda\,(y)\;\langle y \cdot \mathtt{r}, \mathtt{succ}\,(y \cdot \mathtt{r})\rangle) \tag{17.12}$$

$$u_p \triangleq \lambda\,(x)\;u'_p(x) \cdot \mathtt{l}. \tag{17.13}$$

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

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

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

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

It is easy to check that

$$\mathbf{Y}(F) \equiv F(\mathbf{Y}(F)).$$

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

Although it is clear that $\mathbf{Y}$ as just defined computes a fixed point of its argument, it is probably less clear why it works or how one might have invented it in the first place. The main idea is actually quite simple. If a function is to be recursive, it is given an additional first argument, which is arranged to stand for the function itself. Whenever the function wishes to call itself, it calls the implicit first argument, which is for this reason often called `this` or `self`. At each call site to a recursive function, the function is applied to itself before being applied to any other argument. This ensures that the argument called `this` actually refers to the function itself.

With this in mind, it is easy to see how to derive the definition of $\mathbf{Y}$. 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 by replacing each use of f in $F(f)$ by $f(f)$. Now observe 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 is

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

To finish the derivation, we need only observe 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, namely $\mathbf{Y}$ as just defined, that computes the fixed point of *any* F.

17.3 Scott's Theorem

Scott's Theorem states that definitional equality for the untyped λ-calculus is undecidable: There is no algorithm to determine whether 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 $\mathcal{B}\ u$ and $\mathcal{A}_1\ u$ implies that it is *not* the case that $\mathcal{B}\ u$. 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 breaks down 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 nontrivial[2] behavioral properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms are *inseparable*.

Lemma 17.1. *For any u there exists v such that $u\,(\overline{\ulcorner v \urcorner}) \equiv v$.*

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

1. $\mathbf{ap}\,(\overline{\ulcorner u_1 \urcorner})\,(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1\,(u_2) \urcorner}$.
2. $\mathbf{nm}\,(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

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

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

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

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

Lemma 17.2. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two nontrivial 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 $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

Proof Suppose there is such an untyped term w. Let v be the untyped term $\lambda\,(x)\,\mathtt{ifz}(w(x); u_1; \_.u_0)$, where $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. By Lemma 17.1 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 17.3. *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 nontrivial behavioral property of untyped terms. It is therefore inseparable from its negation and hence is undecidable. □

17.4 Untyped Means Unityped

The untyped λ-calculus may be faithfully embedded in a typed language with recursive types. This means that every untyped λ-term has a representation as a typed expression in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the λ-calculus in $\mathcal{L}\{+\times\rightharpoonup\mu\}$ (which we could surely do), but rather a direct representation of untyped λ-terms as typed expressions in a language with recursive types.

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

$$D \triangleq \mu t.\, t \to t.$$

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

$$D \cong D \to D.$$

This specifies that D is a type that is isomorphic to the space of functions on D itself, something that is impossible in conventional set theory, but is feasible in the computationally based setting of the λ-calculus.

This isomorphism leads to the following translation, of $\mathcal{L}\{\lambda\}$ into $\mathcal{L}\{+\times\rightharpoonup\mu\}$:

$$x^\dagger \triangleq x \tag{17.14a}$$

$$\lambda\,(x)\,u^\dagger \triangleq \mathtt{fold}(\lambda\,(x:D)\,u^\dagger) \tag{17.14b}$$

$$u_1(u_2)^\dagger \triangleq \mathtt{unfold}(u_1^\dagger)(u_2^\dagger). \tag{17.14c}$$

Observe that the embedding of a λ-abstraction is a value and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

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

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

Thus we see that the canonical *untyped* language, $\mathcal{L}\{\lambda\}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all. Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the expense of incurring substantial dynamic overhead to coerce values to and from the recursive type. In Chapter 18 we take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus this traditional distinction can hardly be considered an opposition, as dynamic languages are but particular forms of static languages in which undue emphasis is placed on a single recursive type.

17.5 Notes

The untyped λ-calculus was introduced by Church (1941) as a codification of the informal concept of a computable function. Unlike the well-known machine models, such as the Turing machine or the random access machine, the λ-calculus directly codifies mathematical and programming practice. 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 (1980) 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 "unityped" rather than as "untyped."

18 Dynamic Typing

We saw in Chapter 17 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped λ-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur that are due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. This property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped λ-calculus (rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument or to add a function to a number. One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be dynamically typed, in purported opposition to statically typed languages. But the supposed opposition is illusory: Just as the so-called untyped λ-calculus turns out to be unityped, so dynamic languages turn out to be but restricted forms of static language. This remark is so important it bears repeating: Every dynamic language is inherently a static language in which we confine ourselves to a (needlessly) restricted type discipline to ensure safety.

18.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, called $\mathcal{L}\{dyn\}$. The abstract syntax of $\mathcal{L}\{dyn\}$ is given by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Exp	d	$::=$	x	x	variable
			$\texttt{num}(n)$	$\overline{n}$	numeral
			$\texttt{zero}$	$\texttt{zero}$	zero
			$\texttt{succ}(d)$	$\texttt{succ}(d)$	successor
			$\texttt{ifz}(d; d_0; x\,.\,d_1)$	$\texttt{ifz}\,d\,\{\texttt{zero} \Rightarrow d_0 \mid \texttt{succ}(x) \Rightarrow d_1\}$	zero test
			$\texttt{fun}(\lambda\,(x)\,d)$	$\lambda\,(x)\,d$	abstraction
			$\texttt{ap}(d_1; d_2)$	$d_1(d_2)$	application
			$\texttt{fix}(x\,.\,d)$	$\texttt{fix}\,x\,\texttt{is}\,d$	recursion

There are two classes of values in $\mathcal{L}\{dyn\}$, the *numbers*, which have the form $\overline{n}$, and the *functions*, which have the form $\lambda\ (x)\ d$. The expressions `zero` and `succ(`d`)` are not values, but rather are operations that evaluate to values. General recursion is definable by use of a fixed point combinator, but is taken as primitive here to simplify the analysis of the dynamics in Section 18.3.

As usual, the abstract syntax of $\mathcal{L}\{dyn\}$ is what matters, but we use the concrete syntax to write examples in a convenient manner. However, it is often the case for dynamic languages, including $\mathcal{L}\{dyn\}$, that the concrete syntax is deceptive in that it obscures an important detail of the abstract syntax, namely that every value is tagged with a classifier that plays a significant role at run time (as we will see shortly). So although the concrete syntax for a number $\overline{n}$ suggests a "bare" representation, the abstract syntax reveals that the number is labelled with the class `num` to indicate that the value is of the numeric class. This is done to distinguish it from a function value, which concretely has the form $\lambda\ (x)\ d$, but whose abstract syntax, `fun(`$\lambda\ (x)\ d$`)`, indicates that it is to be classified with the tag `fun` to distinguish it from a number. As we will see shortly, this tagging is of prime importance in any dynamic language, so it is important to pay close attention to the abstract form in what follows.

The statics of $\mathcal{L}\{dyn\}$ is essentially the same as that of $\mathcal{L}\{\lambda\}$ given in Chapter 17; 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 indicate that d is a closed expression of $\mathcal{L}\{dyn\}$.

The dynamics of $\mathcal{L}\{dyn\}$ must check for errors that would never arise in a language such as $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. 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:

$d\ \mathsf{val}$	d is a (closed) value
$d \mapsto d'$	d evaluates in one step to d'
$d\ \mathsf{err}$	d incurs a run-time error
$d\ \mathsf{is\_num}\ n$	d is of class `num` with value n
$d\ \mathsf{isnt\_num}$	d is not of class `num`
$d\ \mathsf{is\_fun}\ x.d$	d is of class `fun` with body $x.d$
$d\ \mathsf{isnt\_fun}$	d is not of class `fun`

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

The value judgment, $d\ \mathsf{val}$ states that d is a fully evaluated (closed) expression:

$$\frac{}{\mathtt{num(}n\mathtt{)}\ \mathsf{val}} \tag{18.1a}$$

$$\frac{}{\mathtt{fun(}\lambda\ (x)\ d\mathtt{)}\ \mathsf{val}}. \tag{18.1b}$$

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

$$\frac{}{\texttt{num(}n\texttt{)}\ \textsf{is\_num}\ n} \tag{18.2a}$$

$$\frac{}{\texttt{fun(}\lambda\ \texttt{(}x\texttt{)}\ d\texttt{)}\ \textsf{is\_fun}\ x\,.\,d}\,. \tag{18.2b}$$

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

$$\frac{}{\texttt{num(\_)}\ \textsf{isnt\_fun}} \tag{18.3a}$$

$$\frac{}{\texttt{fun(\_)}\ \textsf{isnt\_num}}\,. \tag{18.3b}$$

The transition judgment $d \mapsto d'$ and the error judgment $d\ \textsf{err}$ are defined simultaneously by the following rules:

$$\frac{}{\texttt{zero} \mapsto \texttt{num(z)}} \tag{18.4a}$$

$$\frac{d \mapsto d'}{\texttt{succ(}d\texttt{)} \mapsto \texttt{succ(}d'\texttt{)}} \tag{18.4b}$$

$$\frac{d\ \textsf{err}}{\texttt{succ(}d\texttt{)}\ \textsf{err}} \tag{18.4c}$$

$$\frac{d\ \textsf{is\_num}\ n}{\texttt{succ(}d\texttt{)} \mapsto \texttt{num(s(}n\texttt{))}} \tag{18.4d}$$

$$\frac{d\ \textsf{isnt\_num}}{\texttt{succ(}d\texttt{)}\ \textsf{err}} \tag{18.4e}$$

$$\frac{d \mapsto d'}{\texttt{ifz(}d\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)} \mapsto \texttt{ifz(}d'\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)}} \tag{18.4f}$$

$$\frac{d\ \textsf{err}}{\texttt{ifz(}d\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)}\ \textsf{err}} \tag{18.4g}$$

$$\frac{d\ \textsf{is\_num}\ 0}{\texttt{ifz(}d\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)} \mapsto d_0} \tag{18.4h}$$

$$\frac{d\ \textsf{is\_num}\ n+1}{\texttt{ifz(}d\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)} \mapsto [\texttt{num(}n\texttt{)}/x]d_1} \tag{18.4i}$$

$$\frac{d\ \textsf{isnt\_num}}{\texttt{ifz(}d\texttt{;}\,d_0\texttt{;}\,x\,\texttt{.}\,d_1\texttt{)}\ \textsf{err}} \tag{18.4j}$$

$$\frac{d_1 \mapsto d_1'}{\texttt{ap(}d_1\texttt{;}\,d_2\texttt{)} \mapsto \texttt{ap(}d_1'\texttt{;}\,d_2\texttt{)}} \tag{18.4k}$$

$$\frac{d_1\ \textsf{err}}{\texttt{ap(}d_1\texttt{;}\,d_2\texttt{)}\ \textsf{err}} \tag{18.4l}$$

$$\frac{d_1\ \textsf{is\_fun}\ x\,.\,d}{\texttt{ap(}d_1\texttt{;}\,d_2\texttt{)} \mapsto [d_2/x]d} \tag{18.4m}$$

$$\frac{d_1 \text{ isnt\_fun}}{\texttt{ap}(d_1;d_2) \text{ err}} \tag{18.4n}$$

$$\frac{}{\texttt{fix}(x.d) \mapsto [\texttt{fix}(x.d)/x]d}\,. \tag{18.4o}$$

Rule (18.4i) labels the predecessor with the class num to maintain the invariant that variables are bound to expressions of $\mathcal{L}\{dyn\}$.

Lemma 18.1 (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 a straightforward inspection of the rules defining the class-checking judgments. □

Theorem 18.2 (Progress)**.** *If d ok, then either d val or d err, or there exists d' such that $d \mapsto d'$.*

Proof By induction on the structure of d. For example, if $d = \texttt{succ}(d')$, then we have by induction either d' val, d' err, or $d' \mapsto d''$ for some d''. In last case we have by Rule (18.4b) that $\texttt{succ}(d') \mapsto \texttt{succ}(d'')$, and in the second-to-last case we have by Rule (18.4c) that $\texttt{succ}(d')$ err. Is d' val, then by Lemma 18.1, either d' is_num n or d' isnt_num. In the former case $\texttt{succ}(d') \mapsto \texttt{num}(n+1)$, and in the latter $\texttt{succ}(d')$ err. The other cases are handled similarly. □

Lemma 18.3 (Exclusivity)**.** *For any d in $\mathcal{L}\{dyn\}$, exactly one of the following holds: d val, or d err, or $d \mapsto d'$ for some d'.*

Proof By induction on the structure of d, making reference to Rules (18.4). □

18.2 Variations and Extensions

The dynamic language $\mathcal{L}\{dyn\}$ defined in Section 18.1 closely parallels the static language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 10. One discrepancy, however, is in the treatment of natural numbers. Whereas in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ the zero and successor operations are introductory forms for the type nat, in $\mathcal{L}\{dyn\}$ they are elimination forms that act on separately defined numerals. This is done to ensure that there is a single class of numbers, rather than a separate class for zero and successor.

An alternative is to treat zero and $\texttt{succ}(d)$ as values of two separate classes and to introduce the obvious class-checking judgments for them. This complicates the error checking rules, and admits problematic values such as $\texttt{succ}(\lambda\ (x)\ d)$, but it allows us to

avoid having a class of numbers. When written in this style, the dynamics of the conditional branch is given as follows:

$$\frac{d \mapsto d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto \texttt{ifz}(d'; d_0; x\,.\,d_1)} \tag{18.5a}$$

$$\frac{d\ \mathsf{is\_zero}}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto d_0} \tag{18.5b}$$

$$\frac{d\ \mathsf{is\_succ}\ d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto [d'/x]d_1} \tag{18.5c}$$

$$\frac{d\ \mathsf{isnt\_zero} \quad d\ \mathsf{isnt\_succ}}{\texttt{ifz}(d; d_0; x\,.\,d_1)\ \mathsf{err}}\,. \tag{18.5d}$$

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.

Structured data may be added to $\mathcal{L}\{dyn\}$ by use of similar techniques. The classic example is to introduce a null value and a constructor for combining two values into one:

Sort			Abstract Form	Concrete Form	Description
Exp	d	::=	$\texttt{nil}$	$\texttt{nil}$	null
			$\texttt{cons}(d_1; d_2)$	$\texttt{cons}(d_1; d_2)$	pair
			$\texttt{ifnil}(d; d_0; x, y\,.\,d_1)$	$\texttt{ifnil}\,d\,\{\texttt{nil} \Rightarrow d_0 \mid \texttt{cons}(x; y) \Rightarrow d_1\}$	conditional

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

Lists may be represented by null and pairing. For example, the list consisting of three zeroes is represented by the value

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

But what to make of this beast?

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

This does not correspond to a list, because it does not end with `nil`.

The difficulty with encoding lists by use of null and pair becomes apparent when defining functions that operate on them. For example, here is a possible definition of the function that appends 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 the function does not traverse its second argument; it can be any value at all. For example, we may append a list to a function and obtain the "list" that ends with a λ previously given.

It might be argued that the conditional branch that distinguishes null from a pair is inappropriate in $\mathcal{L}\{dyn\}$, because there are more than just these two classes in the language.

One approach that avoids this criticism is to abandon the idea of pattern matching on the class of data entirely, 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.

In the present case we would reformulate the extension of $\mathcal{L}\{dyn\}$ with null and pairing as follows:

Sort			Abstract Form	Concrete Form	Description
Exp	d	::=	`cond`$(d;d_0;d_1)$	`cond`$(d;d_0;d_1)$	conditional
			`nil?`(d)	`nil?`(d)	nil test
			`cons?`(d)	`cons?`(d)	pair test
			`car`(d)	`car`(d)	first projection
			`cdr`(d)	`cdr`(d)	second projection

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

Written in this form, the append function 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, rather than restrict, the conditional branch so that it includes cases for each possible class of value in the language. So, for example, 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 perform run-time checks. For example, 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. This is an inherent limitation of dynamic languages, which do not permit handling values other than classified values.

[1] Predicates evaluate to the null value to indicate that a condition is false and to some nonnull value to indicate that it is true.

[2] This terminology for the projections is archaic, but firmly established in the literature.

18.3 Critique of Dynamic Typing

The safety theorem for $\mathcal{L}\{dyn\}$ is often promoted as an advantage of dynamic over static typing. Unlike static languages, which rule out some candidate programs as ill-typed, essentially every piece of abstract syntax in $\mathcal{L}\{dyn\}$ is well-formed, and hence, by Theorem 18.2, has a well-defined dynamics. But this can also be seen as a disadvantage, because errors that could be ruled out at compile time by type checking are not signaled until run time in $\mathcal{L}\{dyn\}$. To make this possible, the dynamics of $\mathcal{L}\{dyn\}$ must enforce conditions that need not be checked in a statically typed language.

Consider, for example, the addition function in $\mathcal{L}\{dyn\}$, whose specification is that, when passed two values of class `num`, it returns their sum, which is also of class `num`:[3]

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

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

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

It is deceptive, because 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, observe 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. This means that the dynamic class check incurred by the application of p in the recursive call is guaranteed to succeed. But $\mathcal{L}\{dyn\}$ offers no means of suppressing this redundant check, because it cannot express the invariant that p is always bound to a value of class `fun`.

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

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

Classification is not free—storage is required for the class label, and it takes time to detach the class from a value each time it is used and to attach a class to a value whenever

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

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 nonnegligible and should be eliminated whenever possible. But this is impossible within $\mathcal{L}\{dyn\}$, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

18.4 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 essentially 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 confused by the ellision of the dynamic checks that are made explicit here. Although the surface syntax of dynamic PCF is essentially the same as that for PCF, minus the type annotations, the underlying dynamics is fundamentally different. It is for this reason that static PCF cannot be properly seen as a restriction of dynamic PCF by the imposition of a type system, contrary to what seems to be a widely held belief.

19 Hybrid Typing

A hybrid language is one that combines static and dynamic typing by enriching a statically typed language with a distinguished type `dyn` of dynamic values. The dynamically typed language considered in Chapter 18 may be embedded into the hybrid language by regarding a dynamically typed program as a statically typed program of type `dyn`. This shows that static and dynamic types are not opposed to one another, but may coexist harmoniously.

The notion of a hybrid language, however, is itself illusory, because the type `dyn` is really a particular recursive type. This shows that there is no need for any special mechanisms to support dynamic typing. Rather, they may be derived from the more general concept of a recursive type. This shows that *dynamic typing is but a mode of use of static typing*. The supposed opposition between dynamic and static typing is therefore a fallacy: Dynamic typing can hardly be opposed to that of which it is but a special case.

19.1 A Hybrid Language

Consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$, which extends $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ with the following additional constructs:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`dyn`	`dyn`	dynamic
Exp	e	::=	`new[`l`](`e`)`	l `!` e	construct
			`cast[`l`](`e`)`	e `?` l	destruct
Cls	l	::=	`num`	`num`	number
			`fun`	`fun`	function

The type `dyn` is the type of dynamically classified values. The `new` operation attaches a classifier to a value, and the `cast` operation checks the classifier and returns the associated value.

The statics of $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$ extends that of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ with the following additional rules:

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}(e) : \texttt{dyn}} \tag{19.1a}$$

$$\frac{\Gamma \vdash e : \texttt{dyn} \rightharpoonup \texttt{dyn}}{\Gamma \vdash \texttt{new[fun]}(e) : \texttt{dyn}} \tag{19.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}]\,(e) : \mathtt{nat}} \tag{19.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}]\,(e) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}. \tag{19.1d}$$

The statics ensures that class labels are applied to objects of the appropriate type, namely `num` for natural numbers and `fun` for functions defined over labeled values.

The dynamics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn} \rightharpoonup\}$ extends that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with the following rules:

$$\frac{e\ \mathsf{val}}{\mathtt{new}[l]\,(e)\ \mathsf{val}} \tag{19.2a}$$

$$\frac{e \mapsto e'}{\mathtt{new}[l]\,(e) \mapsto \mathtt{new}[l]\,(e')} \tag{19.2b}$$

$$\frac{e \mapsto e'}{\mathtt{cast}[l]\,(e) \mapsto \mathtt{cast}[l]\,(e')} \tag{19.2c}$$

$$\frac{\mathtt{new}[l]\,(e)\ \mathsf{val}}{\mathtt{cast}[l]\,(\mathtt{new}[l]\,(e)) \mapsto e} \tag{19.2d}$$

$$\frac{\mathtt{new}[l']\,(e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l]\,(\mathtt{new}[l']\,(e))\ \mathsf{err}}. \tag{19.2e}$$

Casting compares the class of the object with the required class, returning the underlying object if these coincide and signaling an error otherwise.

Lemma 19.1 (Canonical Forms)**.** *If $e : \mathtt{dyn}$ and e val, then $e = \mathtt{new}[l]\,(e')$ for some class l and some e' val. If $l = \mathtt{num}$, then $e' : \mathtt{nat}$, and if $l = \mathtt{fun}$, then $e' : \mathtt{dyn} \rightharpoonup \mathtt{dyn}$.*

Proof By a straightforward rule induction on the statics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn} \rightharpoonup\}$. □

Theorem 19.2 (Safety)**.** *The language $\mathcal{L}\{\mathtt{nat}\ \mathtt{dyn} \rightharpoonup\}$ is safe*

1. *if $e : \tau$ and $e \mapsto e'$, then $e' : \tau$,*
2. *if $e : \tau$, then either e val, or e err, or $e \mapsto e'$ for some e'.*

Proof Preservation is proved by rule induction on the dynamics, and progress is proved by rule induction on the statics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for $\mathcal{L}\{\mathit{dyn}\}$—a well-typed cast might fail at run time if the class of the cast does not match the class of the value. □

The type dyn need not be taken as primitive in any language with sum types and recursive types. Specifically, the type dyn is definable in such a language by the following correspondences:[1]

$$\texttt{dyn} \triangleq \mu t\,.\,[\texttt{num} \hookrightarrow \texttt{nat}, \texttt{fun} \hookrightarrow t \rightharpoonup t] \tag{19.3}$$

$$\texttt{new}[\texttt{num}](e) \triangleq \texttt{fold}(\texttt{num} \cdot e) \tag{19.4}$$

$$\texttt{new}[\texttt{fun}](e) \triangleq \texttt{fold}(\texttt{fun} \cdot e) \tag{19.5}$$

$$\texttt{cast}[\texttt{num}](e) \triangleq \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num} \cdot x \Rightarrow x \mid \texttt{fun} \cdot x \Rightarrow \texttt{error}\} \tag{19.6}$$

$$\texttt{cast}[\texttt{fun}](e) \triangleq \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num} \cdot x \Rightarrow \texttt{error} \mid \texttt{fun} \cdot x \Rightarrow x\}. \tag{19.7}$$

Thus there is no need for a primitive notion of dynamic type, provided that sums and recursive types are available.

19.2 Dynamics as Static Typing

The language $\mathcal{L}\{dyn\}$ described in Chapter 18 may be embedded into $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ by a simple translation that makes explicit the class checking in the dynamics of $\mathcal{L}\{dyn\}$. Specifically, we may define a translation $d^\dagger$ of expressions of $\mathcal{L}\{dyn\}$ into expressions of $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ according to the following static correctness criterion:

Theorem 19.3. *If x_1 ok, …, x_n ok $\vdash$ d ok according to the statics of $\mathcal{L}\{dyn\}$, then $x_1 : \texttt{dyn}, \ldots, x_n : \texttt{dyn} \vdash d^\dagger : \texttt{dyn}$ in $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$.*

The proof of Theorem 19.3 is a straightforward induction on the structure of d based on the following translation:

$$\begin{aligned}
x^\dagger &\triangleq x \\
\texttt{num}(n)^\dagger &\triangleq \texttt{new}[\texttt{num}](\overline{n}) \\
\texttt{zero}^\dagger &\triangleq \texttt{new}[\texttt{num}](\texttt{z}) \\
\texttt{succ}(d)^\dagger &\triangleq \texttt{new}[\texttt{num}](\texttt{s}(\texttt{cast}[\texttt{num}](d^\dagger))) \\
\texttt{ifz}(d; d_0; x\,.\,d_1) &\triangleq \texttt{ifz}(\texttt{cast}[\texttt{num}](d^\dagger); d_0^\dagger; x\,.\,[\texttt{new}[\texttt{num}](x)/x]d_1^\dagger) \\
(\lambda\,(x)\,d)^\dagger &\triangleq \texttt{new}[\texttt{fun}](\lambda\,(x{:}\texttt{dyn})\,d^\dagger) \\
(d_1(d_2))^\dagger &\triangleq \texttt{cast}[\texttt{fun}](d_1^\dagger)(d_2^\dagger) \\
\texttt{fix}(x\,.\,d) &\triangleq \texttt{fix}[\texttt{dyn}](x\,.\,d^\dagger).
\end{aligned}$$

Although a rigorous proof requires methods extending those to be developed in Chapter 48, it should be clear that the translation is faithful to the dynamics of $\mathcal{L}\{dyn\}$ given in Chapter 18.

[1] The expression error aborts the computation with an error; this can be accomplished using exceptions, which are described in Chapter 28.

19.3 Optimization of Dynamic Typing

The language $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ combines static and dynamic typing by enriching $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the type dyn of classified values. It is for this reason called a *hybrid* language. Unlike a purely dynamic type system, a hybrid type system can express invariants that are crucial to the optimization of programs in $\mathcal{L}\{dyn\}$.

Consider the addition function in $\mathcal{L}\{dyn\}$ given in Section 18.3, which is transcribed here for easy reference:

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

This function may be regarded as a value of type dyn in $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$, given as follows:

$$\texttt{fun}\,!\,\lambda\,(x\texttt{:dyn})\,\texttt{fix}\,p\texttt{:dyn}\,\texttt{is}\,\texttt{fun}\,!\,\lambda\,(y\texttt{:dyn})\,e_{x,p,y},$$

where

$$x : \texttt{dyn},\, p : \texttt{dyn},\, y : \texttt{dyn} \vdash e_{x,p,y} : \texttt{dyn}$$

is the expression

$$\texttt{ifz}\,(y\,?\,\texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num}\,!\,(\texttt{s}((p\,?\,\texttt{fun})(\texttt{num}\,!\,y')\,?\,\texttt{num}))\}.$$

The embedding into $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ makes explicit the run-time checks that are implicit in the dynamics of $\mathcal{L}\{dyn\}$.

Careful examination of the embedded formulation of addition reveals a great deal of redundancy and overhead that can be eliminated in the statically typed version. Eliminating this redundancy requires a static type discipline, because the intermediate computations involve values of a type other than dyn. Because a dynamic language can express values of only one type, it is impossible to express the optimized form within a dynamic language. This shows that the superficial freedoms offered by dynamic languages supposedly accruing from the omission of types are, in fact, severe restrictions on the expressiveness of the language compared with a static language with a type of dynamic values.

The first redundancy arises from the use of recursion in a dynamic language. In the previous example we use recursion to define the inner loop p of the computation. The value p is, by definition, a λ-abstraction, which is explicitly tagged as a function. Yet the call to p within the loop checks at run time whether p is in fact a function before applying it. Because p is an internally defined function, all of its call sites are under the control of the addition function, which means that there is no need for such pessimism at calls to p, provided that we change its type to $\texttt{dyn} \rightharpoonup \texttt{dyn}$, which directly expresses the invariant that p is a function acting on dynamic values.

Performing this transformation, we obtain the following reformulation of the addition function that eliminates this redundancy:

$$\texttt{fun}\,!\,\lambda\,(x\texttt{:dyn})\,\texttt{fun}\,!\,\texttt{fix}\,p\texttt{:dyn} \rightharpoonup \texttt{dyn}\,\texttt{is}\,\lambda\,(y\texttt{:dyn})\,e'_{x,p,y},$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,?\,\texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num}\,!\,(\texttt{s}(p(\texttt{num}\,!\,y')\,?\,\texttt{num}))\}.$$

We have "hoisted" the function class label out of the loop and suppressed the cast inside the loop. Correspondingly, the type of p has changed to $\texttt{dyn} \rightharpoonup \texttt{dyn}$.

Next we observe that the parameter y of type dyn is cast to a number on each iteration of the loop before it is tested for zero. Because this function is recursive, the bindings of y arise in one of two ways: at the initial call to the addition function and on each recursive call. But the recursive call is made on the predecessor of y, which is a true natural number that is labeled with num at the call site, only to be removed by the class check at the conditional on the next iteration. This suggests that we hoist the check on y outside of the loop and avoid labeling the argument to the recursive call. Doing so changes the type of the function, however, from $\texttt{dyn} \rightharpoonup \texttt{dyn}$ to $\texttt{nat} \rightharpoonup \texttt{dyn}$. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class num, and, if so, the underlying value is incremented and labeled with class num. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the class check is redundant, because we know that it must have class num. But what if the result came from the other branch of the conditional? In that case the function returns x, which need not be of class num because it is provided by the caller of the function. However, we may reasonably insist that it is an error to call addition with a nonnumeric argument. We can enforce this by replacing x in the zero branch of the conditional by $x\ ?\,\texttt{num}$.

Combining these optimizations, we obtain the inner loop e''_x defined as follows:

$$\texttt{fix}\ p\texttt{:nat} \rightharpoonup \texttt{nat}\ \texttt{is}\ \lambda\ \texttt{(}y\texttt{:nat)}\ \texttt{ifz}\ y\ \{\texttt{zero} \Rightarrow x\ ?\,\texttt{num} \mid \texttt{succ(}y'\texttt{)} \Rightarrow \texttt{s(}p\texttt{(}y'\texttt{))}\}.$$

This function has type $\texttt{nat} \rightharpoonup \texttt{nat}$ and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Finally, we recall that the overall goal is to define a version of addition that works on values of type dyn. Thus we require a value of type $\texttt{dyn} \rightharpoonup \texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. This can be converted to the required form by precomposing with a cast to num and postcomposing with a coercion to num:

$$\texttt{fun}\ !\ \lambda\ \texttt{(}x\texttt{:dyn)}\ \texttt{fun}\ !\ \lambda\ \texttt{(}y\texttt{:dyn)}\ \texttt{num}\ !\ \texttt{(}e''_x\texttt{(}y\ ?\,\texttt{num))}.$$

The innermost λ-abstraction converts the function e''_x from type $\texttt{nat} \rightharpoonup \texttt{nat}$ to type $\texttt{dyn} \rightharpoonup \texttt{dyn}$ by composing it with a class check that ensures that y is a natural number at the initial call site and applies a label to the result to restore it to type dyn.

The outcome of these transformations is that the inner loop of the computation runs at "full speed," without any manipulation of tags on functions or numbers. But the outermost form of addition has been retained as a value of type dyn encapsulating a curried function that takes two arguments of type dyn. This preserves the correctness of all calls to addition, which pass and return values of type dyn while optimizing its execution during the computation. Of course, we could strip the class tags from the addition function, changing its type from dyn to the more descriptive $\texttt{dyn} \rightharpoonup \texttt{dyn} \rightharpoonup \texttt{dyn}$, but this imposes the requirement on the caller to treat addition not as a value of type dyn, but rather as a function that must be applied to two successive values of type dyn whose class is num. As long as the call sites to addition are under programmer control, there is no obstacle to effecting this transformation.

It is only when there may be external call sites, not directly under programmer control, that there is any need to package addition as a value of type `dyn`. Applying this principle generally, we see that dynamic typing is only of marginal utility—that is, is used only at the margins of a system where uncontrolled calls arise. Internally to a system there is no benefit, and considerable drawback, to restricting attention to the type `dyn`.

19.4 Static Versus Dynamic Typing

There have been many attempts by advocates of dynamic typing to distinguish dynamic from static languages. It is useful to review the supposed distinctions from the present viewpoint.

1. *Dynamic languages associate types with values, whereas static languages associate types with variables*. But this is nonsense arising from the confusion of types with classes. Dynamic languages associate *classes*, not *types*, with values by tagging them with identifiers such as `num` and `fun`. This form of classification amounts to a use of recursive sum types within a statically typed language, and hence cannot be seen as a distinguishing feature of dynamic languages. Morever, static languages assign types to expressions, not just variables. Because dynamic languages are just particular static languages (with a single type), the same can be said of dynamic languages.
2. *Dynamic languages check types at run time, whereas static languages check types at compile time*. This, too, is erroneous. Dynamic languages are just as surely statically typed as static languages, albeit for a degenerate type system with only one type. As we have seen, dynamic languages do perform class checks at run time, but so too do static languages that admit sum types. The difference is only the extent to which we must use classification: always in a dynamic language, only as necessary in a static language.
3. *Dynamic languages support heterogeneous collections, whereas static languages support homogeneous collections*. But this, too, is in error. Sum types exist to support heterogeneity, and any static language with sums admits heterogenous data structures. A typical example is a list such as

   $$\texttt{cons(num(1);cons(fun(}\lambda\ \texttt{(}x\texttt{)}\ x\texttt{);nil))}.$$

   It is sometimes said that such a list is not representable in a static language because of the disparate nature of its components. Both static and dynamic languages are *type* homogeneous, but may be *class* heterogeneous. All elements of the preceding list are of *type* `dyn`; the first is of *class* `num`, and the second is of *class* `fun`.

What, then, are we to make of the supposed distinction between dynamic and static languages? Rather than being in opposition to each other, it is more accurate to say that *dynamic languages are a mode of use of static languages*. Every dynamic language *is* a static language, albeit one with a paucity of types available to the programmer (only one!).

But as we have just seen, types express and enforce invariants that are crucial to the correctness and efficiency of programs.

19.5 Notes

The concept of a hybrid type system is wholly artificial, serving only as an explanatory bridge between dynamic and static languages. Viewing dynamic languages as static languages with recursive types was first proposed by Scott (1980), who also suggested the term unityped as a more descriptive alternative to untyped.

PART VII

Variable Types

20 Girard's System F

The languages we have considered so far are all *monomorphic* in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in $\mathcal{L}\{\mathtt{nat}\to\}$ there is a *distinct* identity function for each type τ, namely $\lambda\,(x:\tau)\,x$, even though the behavior is the same for each choice of τ. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f:\tau_2\to\tau_3)\,\lambda\,(g:\tau_1\to\tau_2)\,\lambda\,(x:\tau_1)\,f(g(x)).$$

Each choice of the three types requires a *different* program, even though they all exhibit the same behavior when executed.

Obviously it would be useful to capture the general pattern once and for all and to instantiate this pattern each time we need it. The expression patterns codify generic (type-independent) behaviors that are shared by all instances of the pattern. Such generic expressions are said to be *polymorphic*. In this chapter we study a language introduced by Girard under the name *System **F*** and by Reynolds under the name *polymorphic typed λ-calculus*. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 21) 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.)

20.1 System F

*System **F***, or the *polymorphic λ-calculus*, or $\mathcal{L}\{\to\forall\}$, is a minimal functional language that illustrates the core concepts of polymorphic typing and permits us to examine its surprising expressive power in isolation from other language features. The syntax of System **F** is given

by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Typ	τ	$::=$	t	t	variable
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Exp	e	$::=$	x	x	
			$\mathtt{lam}[\tau](x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{Lam}(t.e)$	$\Lambda(t.e)$	type abstraction
			$\mathtt{App}[\tau](e)$	$e[\tau]$	type application

A *type abstraction* $\mathtt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* t standing for an unspecified type within e. A *type application*, or *instantiation*, $\mathtt{App}[\tau](e)$, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type* $\mathtt{all}(t.\tau)$ that determines the type τ of the result as a function of the argument t.

The statics of $\mathcal{L}\{\to\forall\}$ consists of two judgment forms, the *type formation* judgment,

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

and the *typing judgment*,

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

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

The rules defining the type formation judgment are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{20.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{20.1b}$$

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

The rules defining the typing judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.2a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{20.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{20.2c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'}. \tag{20.2e}$$

Lemma 20.1 (Regularity)**.** *If $\Delta\ \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i\ \mathsf{type}$ for each assumption $x_i : \tau_i$ in Γ, then $\Delta \vdash \tau\ \mathsf{type}$.*

Proof By induction on Rules (20.2). □

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 20.2 (Substitution)**.**

1. *If $\Delta, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ and $\Delta \vdash \tau\ \mathsf{type}$, then $\Delta \vdash [\tau/t]\tau'\ \mathsf{type}$.*
2. *If $\Delta, t\ \mathsf{type}\ \Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau\ \mathsf{type}$, then $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.*
3. *If $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta\ \Gamma \vdash e : \tau$, then $\Delta\ \Gamma \vdash [e/x]e' : \tau'$.*

The second part of the lemma requires substitution into the context Γ, as well as into the term and its type, because the type variable t may occur freely in any of these positions.

Returning to the motivating examples from the introduction, the polymorphic identity function I is written as

$$\Lambda(t.\lambda\ (x:t)\ x);$$

it has the polymorphic type

$$\forall(t.t \to t).$$

Instances of the polymorphic identity are written as $I\,[\tau]$, where τ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function C is written as

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda\ (f:t_2 \to t_3)\ \lambda\ (g:t_1 \to t_2)\ \lambda\ (x:t_1)\ f(g(x))))).$$

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 $\mathcal{L}\{\rightarrow\forall\}$ is given as follows:

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

$$\frac{}{\mathtt{Lam}(t\,.\,e)\ \mathsf{val}} \tag{20.3b}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}[\tau_1](x\,.\,e);e_2) \mapsto [e_2/x]e} \tag{20.3c}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{20.3d}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1;e_2')}\right] \tag{20.3e}$$

$$\frac{}{\mathtt{App}[\tau](\mathtt{Lam}(t\,.\,e)) \mapsto [\tau/t]e} \tag{20.3f}$$

$$\frac{e \mapsto e'}{\mathtt{App}[\tau](e) \mapsto \mathtt{App}[\tau](e')}\,. \tag{20.3g}$$

The bracketed premises and rule are to be included for a call-by-value interpretation and omitted for a call-by-name interpretation of $\mathcal{L}\{\rightarrow\forall\}$.

It is a simple matter to prove safety for $\mathcal{L}\{\rightarrow\forall\}$, using familiar methods.

Lemma 20.3 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val;*

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

Proof By rule induction on the statics. □

Theorem 20.4 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof By rule induction on the dynamics. □

Theorem 20.5 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof By rule induction on the statics. □

20.2 Polymorphic Definability

The language $\mathcal{L}\{\to\forall\}$ is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types. This is most naturally expressed using definitional equality, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash \lambda\,(x:\tau)\,e_2(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{20.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.e)\,[\rho] \equiv [\rho/t]e : [\rho/t]\tau}\,. \tag{20.4b}$$

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).

20.2.1 Products and Sums

The nullary product, or unit, type is definable in $\mathcal{L}\{\to\forall\}$ 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 $\mathcal{L}\{\to\forall\}$ by using encoding tricks similar to those described in Chapter 17 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 11 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in $\mathcal{L}\{\to\forall\}$ from these definitions:

$$\langle e_1, e_2\rangle\cdot\mathtt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle\cdot\mathtt{r} \equiv e_2 : \tau_2.$$

The nullary sum, or void, type is definable in $\mathcal{L}\{\to\forall\}$:

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

There is no definitional equality to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}\{\to\forall\}$:

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

provided that the types make sense. It is easy to check that the following equivalences are derivable in $\mathcal{L}\{\to\forall\}$:

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

and

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

Thus the dynamic behavior specified in Chapter 12 is correctly implemented by these definitions.

20.2.2 Natural Numbers

As previously stated, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\to\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

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

Because the result type τ is arbitrary, this means that if we have an iterator, then it can be used to define a function of type

$$\mathtt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument n, yields a polymorphic function that, for any result type t, 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 in this manner, we may simply *identify* a natural number n with the polymorphic iterate-up-to-n function just described. This means that we may define the type of natural numbers

in $\mathcal{L}\{\to\forall\}$ by the following equations:

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

It is easy to check that the statics and dynamics of the natural numbers type given in Chapter 9 are derivable in $\mathcal{L}\{\to\forall\}$ under these definitions.

This shows that $\mathcal{L}\{\to\forall\}$ is *at least as expressive* as $\mathcal{L}\{\texttt{nat}\to\}$. But is it *more* expressive? Yes! It is possible to show that the evaluation function for $\mathcal{L}\{\texttt{nat}\to\}$ is definable in $\mathcal{L}\{\to\forall\}$, even though it is not definable in $\mathcal{L}\{\texttt{nat}\to\}$ itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for $\mathcal{L}\{\to\forall\}$ is not definable in $\mathcal{L}\{\to\forall\}$. We may enrich $\mathcal{L}\{\to\forall\}$ a bit more to define the evaluator for $\mathcal{L}\{\to\forall\}$, but as long as all programs in the enriched language terminate, we once again have an undefinable function, the evaluation function for that extension.

20.3 Parametricity Overview

A remarkable property of $\mathcal{L}\{\to\forall\}$ 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 must be the identity function. Informally, when i is applied to a type τ and an argument of type τ, it must return a value of type τ. But 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 must be either $\Lambda(t\,.\,\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x)$ or $\Lambda(t\,.\,\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y)$. For when b is applied to two arguments of some type, its only choice to return a value of that type is to return one of the two.

What is remarkable is that these properties of i and b have been derived *without knowing anything about the expressions themselves*, but only their types. The theory of parametricity implies that we are able to derive theorems about the behavior of a program knowing only its type. Such theorems are sometimes called *free theorems* because they come "for free" as a consequence of typing and require no program analysis or verification to derive. These theorems underpin the remarkable experience with polymorphic languages that well-typed programs tend to behave as expected when executed. That is, satisfying the type checker is sufficient condition for correctness. Parametricity so constrains the behavior of a program that there are relatively few programs of the same type that exhibit unintended behavior, ruling out a large class of mistakes that commonly arise when writing code. Parametricity also guarantees representation independence for abstract types, a topic that is discussed further in Chapter 21.

20.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of $\mathcal{L}\{\to\forall\}$. These are obtained in one of two ways:

1. restricting type quantification to unquantified types,
2. restricting the occurrence of quantifiers within types.

20.4.1 Predicative Fragment

The remarkable expressive power of the language $\mathcal{L}\{\to\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let τ be the type $\forall(t.t \to t)$, and, assuming that $e : \tau$, we may apply e to its own type, obtaining the expression $e[\tau]$ of type $\tau \to \tau$. Written out in full, this is the type

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

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of e itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to e again, obtaining the expression $e[\tau](e)$, which is again of type τ—the very type of e.

This property of $\mathcal{L}\{\to\forall\}$ is called *impredicativity*;[1] the language $\mathcal{L}\{\to\forall\}$ is said to permit *impredicative (type) quantification*. The distinguishing characteristic of impredicative polymorphism is that it involves a kind of circularity in that the meaning of a quantified type is given in terms of its instances, including the quantified type itself. This quasi-circularity is responsible for the surprising expressive power of $\mathcal{L}\{\to\forall\}$ and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of $\mathcal{L}\{\to\forall\}$ terminate).

Contrast this with $\mathcal{L}\{\to\}$, in which the type of an application of a function is evidently smaller than the type of the function itself. For if $e : \tau_1 \to \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of e. This situation extends to polymorphism, provided that we impose the restriction that a quantified type can be instantiated only by an unquantified type. For in that case passage from $\forall(t.\tau)$ to $[\rho/t]\tau$ decreases the number of quantifiers (even if the size of the type expression viewed as a tree grows). For example, the type $\forall(t.t \to t)$ may be instantiated with the type $u \to u$ to obtain the type $(u \to u) \to (u \to u)$. This type has more symbols in it than τ, but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called *predicative*[2]. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the natural numbers are no longer definable in it.

[1] Pronounced *im-PRED-ic-a-tiv-it-y.*
[2] Pronounced *PRED-i-ca-tive.* polymorphism

20.4.2 Prenex Fragment

A rather more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level—not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is often imposed for the sake of type inference, which permits type annotations to be omitted entirely in the knowledge that they can be recovered from the way the expression is used. Type inference is not discussed here, but a formulation of the prenex fragment of $\mathcal{L}\{\to\forall\}$ is given because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of $\mathcal{L}\{\to\forall\}$ is designated $\mathcal{L}^1\{\to\forall\}$ for reasons that will become clear in the next subsection. It is defined by *stratifying* types into two sorts, the *monotypes* (or *rank*-0 types) and the *polytypes* (or *rank*-1 types). The monotypes are those that do not involve any quantification and may be used to instantiate the polymorphic quantifier. The polytypes include the monotypes, but also permit quantification over monotypes. These classifications are expressed by the judgments $\Delta \vdash \tau \ \mathsf{mono}$ and $\Delta \vdash \tau \ \mathsf{poly}$, where Δ is a finite set of hypotheses of the form $t \ \mathsf{mono}$, where t is a type variable not otherwise declared in Δ. The rules for deriving these judgments are as follows:

$$\frac{}{\Delta, t \ \mathsf{mono} \vdash t \ \mathsf{mono}} \tag{20.5a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{mono} \quad \Delta \vdash \tau_2 \ \mathsf{mono}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{mono}} \tag{20.5b}$$

$$\frac{\Delta \vdash \tau \ \mathsf{mono}}{\Delta \vdash \tau \ \mathsf{poly}} \tag{20.5c}$$

$$\frac{\Delta, t \ \mathsf{mono} \vdash \tau \ \mathsf{poly}}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{poly}}\,. \tag{20.5d}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The statics of $\mathcal{L}^1\{\to\forall\}$ is given by rules for deriving hypothetical judgments of the form $\Delta\ \Gamma \vdash e : \rho$, where Δ consists of hypotheses of the form $t \ \mathsf{mono}$, and Γ consists of hypotheses of the form $x : \rho$, where $\Delta \vdash \rho \ \mathsf{poly}$. The rules defining this judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.6a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)} \tag{20.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{20.6c}$$

$$\frac{\Delta, t \ \mathsf{mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.6d}$$

$$\frac{\Delta \vdash \tau \ \mathsf{mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'}\,. \tag{20.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgments have the form $e : \rho$ for some expression e and polytype ρ.

The restriction on the domain of a λ-abstraction to be a monotype means that a fully general `let` construct is no longer definable—there are no means of binding an expression of polymorphic type to a variable. For this reason it is usual to augment $\mathcal{L}^1\{\rightarrow\forall\}$ with a primitive `let` construct whose statics is as follows:

$$\frac{\Delta \vdash \tau_1 \text{ poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}\,[\tau_1]\,(e_1; x\,.\,e_2) : \tau_2}\ . \tag{20.7}$$

For example, the expression

$$\mathtt{let}\ I : \forall(t\,.\,t \rightarrow t)\ \mathtt{be}\ \Lambda(t\,.\,\lambda\ (x : t)\ x)\ \mathtt{in}\ I\,[\tau \rightarrow \tau]\,(I\,[\tau])$$

has type $\tau \rightarrow \tau$ for any polytype τ.

20.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathcal{L}^1\{\rightarrow\forall\}$ may be generalized to form a hierarchy of languages in which the occurrences of polymorphic types are restricted in relation to function types. The key feature of the prenex fragment is that quantified types are not permitted to occur in the domain of a function type. The prenex fragment also prohibits polymorphic types from the range of a function type, but it would be harmless to admit it, there being no significant difference between the type $\rho \rightarrow \forall(t\,.\,\tau)$ and the type $\forall(t\,.\,\rho \rightarrow \tau)$ (where $t \notin \rho$). This motivates the definition of a hierarchy of fragments of $\mathcal{L}\{\rightarrow\forall\}$ that subsumes the prenex fragment as a special case.

We define a judgment of the form $\tau\ \mathsf{type}\,[k]$, where $k \geq 0$, to mean that τ is a type of rank k. Informally, types of rank 0 have no quantification and types of rank $k+1$ may involve quantification, but the domains of function types are restricted to be of rank k. Thus, in the terminology of Subsection 20.4.2, a monotype is a type of rank 0 and a polytype is a type of rank 1.

The definition of the types of rank k is defined simultaneously for all k by the following rules. These rules involve hypothetical judgments of the form $\Delta \vdash \tau\ \mathsf{type}\,[k]$, where Δ is a finite set of hypotheses of the form $t_i\ \mathsf{type}\,[k_i]$ for some pairwise distinct set of type variables t_i. The rules defining these judgments are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\,[k] \vdash t\ \mathsf{type}\,[k]} \tag{20.8a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[0]}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2)\ \mathsf{type}\,[0]} \tag{20.8b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2)\ \mathsf{type}\,[k+1]} \tag{20.8c}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type}\,[k]}{\Delta \vdash \tau\ \mathsf{type}\,[k+1]} \tag{20.8d}$$

$$\frac{\Delta, t\ \mathsf{type}\,[k] \vdash \tau\ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{all}(t\,.\,\tau)\ \mathsf{type}\,[k+1]}\ . \tag{20.8e}$$

With these restrictions in mind, it is a good exercise to define the statics of $\mathcal{L}^k\{\to\forall\}$, the restriction of $\mathcal{L}\{\to\forall\}$ to types of rank k (or less). It is most convenient to consider judgments of the form $e : \tau\,[k]$ specifying simultaneously that $e : \tau$ and $\tau\ \mathsf{type}\,[k]$. For example, the rank-limited rules for λ-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}\,(x\,.\,e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[0]} \tag{20.9a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}\,(x\,.\,e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[k+1]}\,. \tag{20.9b}$$

The remaining rules follow a similar pattern.

The rank-limited languages $\mathcal{L}^k\{\to\forall\}$ clarify the need for a primitive (as opposed to derived) definition mechanism in $\mathcal{L}^1\{\to\forall\}$. The prenex fragment of $\mathcal{L}\{\to\forall\}$ corresponds to the rank-1 fragment $\mathcal{L}^1\{\to\forall\}$. The `let` construct for rank-1 types is definable in $\mathcal{L}^2\{\to\forall\}$ from λ-abstraction and application. This definition makes sense only at rank 2 because it abstracts over a rank-1 polymorphic type and is therefore not available at lesser rank.

20.5 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 description of parametricity as providing "theorems for free" was popularized by Wadler (1989).

21 Abstract Types

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

Data abstraction may be formalized by extending the language $\mathcal{L}\{\rightarrow\forall\}$ with *existential types*. Interfaces are modeled as existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are modeled as packages, the introductory form for existentials, and clients are modeled as uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is modeled so naturally and directly by the logical concept of existential type quantification. Existential types are closely connected with universal types and hence are often treated together. The superficial reason is that both are forms of type quantification and hence both require the machinery of type variables. The deeper reason is that existentials are *definable* from universals—surprisingly, data abstraction is actually just a form of polymorphism! One consequence of this observation is that representation independence is just a use of the parametricity properties of polymorphic functions discussed in Chapter 20.

21.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

Sort	Abstract Form	Concrete Form	Description
Typ τ ::=	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$	interface
Exp e ::=	$\texttt{pack}[t.\tau][\rho](e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$	implementation
	$\texttt{open}[t.\tau][\rho](e_1; t, x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x:\tau\,\texttt{in}\,e_2$	client

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

The abstract syntax of the `open` construct specifies that the type variable t and the expression variable x are bound within the client. They may be renamed at will by α-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type t may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: The use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to α-equivalence and is not particularly tied to data abstraction.

21.1.1 Statics

The statics of existential types is specified by rules defining when an existential is well-formed and by giving typing rules for the associated introductory and eliminatory forms:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{21.1a}$$

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}[t.\tau]\,[\rho]\,(e) : \texttt{some}(t.\tau)} \tag{21.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) : \tau_2}. \tag{21.1c}$$

Rule (21.1c) 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 21.1 (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 (21.1). □

21.1.2 Dynamics

The dynamics of existential types is specified by the following rules (including the bracketed material for an eager interpretation and omitting it for a lazy interpretation):

$$\frac{[e\ \mathsf{val}]}{\mathtt{pack}[t.\tau]\,[\rho]\,(e)\ \mathsf{val}} \tag{21.2a}$$

$$\left[\frac{e \mapsto e'}{\mathtt{pack}[t.\tau]\,[\rho]\,(e) \mapsto \mathtt{pack}[t.\tau]\,[\rho]\,(e')}\right] \tag{21.2b}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) \mapsto \mathtt{open}[t.\tau]\,[\tau_2]\,(e_1'; t, x.e_2)} \tag{21.2c}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{open}[t.\tau]\,[\tau_2]\,(\mathtt{pack}[t.\tau]\,[\rho]\,(e); t, x.e_2) \mapsto [\rho, e/t, x]e_2}. \tag{21.2d}$$

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

21.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\to\forall\}$ to the new constructs.

Theorem 21.2 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof By rule induction on $e \mapsto e'$, making use of substitution for both expression and type variables. □

Lemma 21.3 (Canonical Forms)**.** *If $e : \mathtt{some}(t.\tau)$ 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, making use of the definition of closed values. □

Theorem 21.4 (Progress)**.** *If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.*

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

21.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of queues of natural numbers supporting three operations:

1. forming the empty queue,
2. inserting an element at the tail of the queue, and
3. removing the head of the queue, which is assumed to be nonempty.

This is clearly a bare-bones interface, but it is sufficient to illustrate the main ideas of data abstraction. Queue elements may be taken to be of any type τ of our choosing; we will not be specific about this choice, since nothing depends on it.

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

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

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

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

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

where

$$e_i : \mathtt{nat} \times \mathtt{list} \to \mathtt{list} = \lambda\,(x{:}\mathtt{nat} \times \mathtt{list})\,e_i',$$

and

$$e_r : \mathtt{list} \to \mathtt{nat} \times \mathtt{list} = \lambda\,(x{:}\mathtt{list})\,e_r'.$$

Here the expression e_i' conses the first component of x, the element, onto the second component of x, the queue. Correspondingly, the expression e_r' reverses its argument and returns the head element paired with the reversal of the tail. These operations "know" that queues are represented as values of type `list` and are programmed accordingly.

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

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

In this case e_i has type

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

and e_r has type

$$(\texttt{list} \times \texttt{list}) \to \texttt{nat} \times (\texttt{list} \times \texttt{list}).$$

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

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

21.3 Definability of Existentials

It turns out that it is not necessary to extend $\mathcal{L}\{\to\forall\}$ with existential types to model data abstraction, because they are already definable using only universal types! Before the details are given, let us consider why this should be possible. The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

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

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

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

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, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type τ of the `open` construct.) Finally, a package consisting of a representation type ρ and an implementation e is a polymorphic function that, when given the result type t and the client x instantiates x with ρ and passes to it the implementation e.

It is then a straightforward exercise to show that this translation correctly reflects the statics and dynamics of existential types.

21.4 Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types. More precisely, there is a criterion, called *bisimilarity*, for relating two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation for another that is bisimilar to it. This leads to a simple methodology for proving the correctness of *candidate* implementation of an abstract type, which is to show that it is bisimilar to an obviously correct *reference* implementation of it. 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 existentials in terms of universals given in Section 21.3. It is an immediate consequence of the definition that the client of an abstract type is polymorphic in the representation of the abstract type. A client c of an abstract type $\exists(t.\tau)$ has type $\forall(t.\tau \to \tau_2)$, where t does not occur free in τ_2 (but may, of course, occur in τ). Applying the parametricity property described informally in Chapter 20 (and developed rigorously in Chapter 49), this says that if R is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on both of them. The fact that t does not occur in the result type ensures that the behavior of the client is independent of the choice of relation between the implementations, provided that this relation is preserved by the operations that implements it.

To see what this means requires that we specify what is meant by a bisimulation. This is best done by example. Consider the existential type $\exists(t.\tau)$, where τ is the 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 the empty queue, an insert operation, and a remove operation, respectively. For the sake of simplicity, the element type is chosen to be the natural numbers. The result of removal is an optional pair, according to whether the queue is empty or not.

Theorem 49.12 ensures that if ρ and ρ' are any two closed types, R is a relation between expressions of these two types; then, if any of the implementations $e : [\rho/x]\tau$ and $e' : [\rho'/x]\tau$ respect R, $c\,[\rho]\,e$ 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 said to be *bisimilar*. The terminology stems from the requirement that the operations of the abstract type preserve the relation: If it holds before an operation is performed, then it must also hold afterward, 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 just specified. For the reference implementation we choose ρ to be the type `list`, and define the empty queue to be the empty list, define insert to add the specified element to the head of the list, and define remove to remove the last element of the list. The code is as follows:

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

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 $\mathtt{list} \times \mathtt{list}$ 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. If it is nonempty, 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:

$$
\begin{aligned}
t &\triangleq \mathtt{list} \times \mathtt{list} \\
\mathtt{emp} &\triangleq \langle \mathtt{nil}, \mathtt{nil}\rangle \\
\mathtt{ins} &\triangleq \lambda\,(x{:}\mathtt{nat})\,\lambda\,(\langle bs, fs\rangle{:}t)\,\langle \mathtt{cons}(x; bs), fs\rangle \\
\mathtt{rem} &\triangleq \lambda\,(\langle bs, fs\rangle{:}t)\,\mathtt{case}\,fs\,\{\mathtt{nil} \Rightarrow e \mid \mathtt{cons}(f; fs') \Rightarrow \langle bs, fs'\rangle\},
\end{aligned}
$$

where $e \triangleq \mathtt{case}\,\mathtt{rev}(bs)\,\{\mathtt{nil} \Rightarrow \mathtt{null} \mid \mathtt{cons}(b; bs') \Rightarrow \mathtt{just}(\langle b, \langle \mathtt{nil}, bs'\rangle\rangle)\}$.

The cost of the occasional reversal may be 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. This reduces to specifying a relation R between the types `list`

and `list` $\times$ `list` such that the three simulation conditions previously given are satisfied by the two implementations just described. The relation in question states that $R(l, \langle b, f \rangle)$ iff the list l is the list `app(`b`)(rev(`f`))`, where `app` is the evident append function on lists. That is, thinking of l as the reference representation of the queue, the candidate must maintain that the elements of b followed by the elements of f in reverse order form precisely the list l. It is easy to check that the implementations just described preserve this relation. Having done so, we are assured that the client c behaves the same regardless of whether we use the reference or the candidate. Because the reference implementation is obviously correct (albeit inefficient), the candidate must also be correct in that the behavior of any client is unaffected by using it instead of the reference.

21.5 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 the work by Reynolds (1974), but the connection with existential types was not explicitly drawn there. The account of representation independence given here is derived from the work by Mitchell (1986).

22 Constructors and Kinds

The types $\texttt{nat} \to \texttt{nat}$ and $\texttt{nat}\,\texttt{list}$ may be thought of as being built from other types by the application of a *type constructor*, or *type operator*. These two examples differ from each other in that the function space type constructor takes two arguments, whereas the list type constructor takes only one. We may, for the sake of uniformity, think of types such as `nat` as being built by a type constructor of *no* arguments. More subtly, we may even think of the types $\forall(t.\tau)$ and $\exists(t.\tau)$ as being built up in the same way by regarding the quantifiers as *higher-order* type operators.

These seemingly disparate cases may be treated uniformly by enriching the syntactic structure of a language with a new layer of *constructors*. To ensure that constructors are used properly (for example, that the list constructor is given only one argument and that the function constructor is given two), we classify constructors by *kinds*. Constructors of a distinguished kind `T` are types, which may be used to classify expressions. To allow for multiargument and higher-order constructors, we also consider finite product and function kinds. (Later we consider even richer kinds.)

The distinction between constructors and kinds on one hand and types and expressions on the other reflects a fundamental separation between the static, and the dynamic *phase* of processing of a programming language, called the *phase distinction*. The static phase implements the statics, and the dynamic phase implements the dynamics. Constructors may be seen as a form of *static data* that are manipulated during the static phase of processing. Expressions are a form of *dynamic data* that are manipulated at run time. Because the dynamic phase follows the static phase (we execute only well-typed programs), we may also manipulate constructors at run time.

Adding constructors and kinds to a language introduces more technical complications than might at first be apparent. The main difficulty is that as soon as we enrich the kind structure beyond the distinguished kind of types, it becomes essential to simplify constructors to determine whether they are equivalent. For example, if we admit product kinds, then a pair of constructors is a constructor of product kind, and projections from a constructor of product kind are also constructors. But what if we form the first projection from the pair consisting of the constructors `nat` and `str`? This should be equivalent to `nat`, because the elimination form is postinverse to the introduction form. Consequently, any expression (say, a variable) of the one type should also be an expression of the other. That is, typing should respect definitional equality of constructors.

There are two main ways to deal with this. One is to introduce a concept of definitional equality for constructors and to demand that the typing judgment for expressions respect

definitional equality of constructors of kind `T`. This means, however, that we must show that definitional equality is decidable if we are to build a complete implementation of the language. The other is to prohibit formation of awkward constructors such as the projection from a pair so that there is never any issue of when two constructors are equivalent (only when they are identical). But this complicates the definition of substitution, because a projection from a constructor variable is well-formed until we substitute a pair for the variable. Both approaches have their benefits, but the second is simplest and is adopted here.

22.1 Statics

The syntax of kinds is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Kind	κ	$::=$	`Type`	`T`	types
			`Unit`	1	nullary product
			`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$	binary product
			`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \to \kappa_2$	function

The kinds consist of the kind of types `T` and the unit kind `Unit`, and are closed under formation of product and function kinds.

The syntax of constructors is divided into two syntactic sorts, the *neutral* and the *canonical*, according to the following grammar:

Sort			Abstract Form	Concrete Form	Description
NCon	a	$::=$	u	u	variable
			`proj[l]`(a)	$a \cdot$ `l`	first projection
			`proj[r]`(a)	$a \cdot$ `r`	second projection
			`app`$(a_1;c_2)$	$a_1[c_2]$	application
CCon	c	$::=$	`atom`(a)	$\widehat{a}$	atomic
			`unit`	$\langle\rangle$	null tuple
			`pair`$(c_1;c_2)$	$\langle c_1, c_2\rangle$	pair
			`lam`$(u.c)$	$\lambda\, u.c$	abstraction

The reason to distinguish neutral from canonical constructors is to ensure that it is impossible to apply an elimination form to an introduction form, which demands an equation to capture the inversion principle. For example, the putative constructor $\langle c_1, c_2\rangle \cdot$ `l`, which would be definitionally equal to c_1, is ill-formed according to the preceding syntax chart. This is because the argument to a projection must be neutral, but a pair is only canonical, not neutral.

The canonical constructor $\widehat{a}$ is the inclusion of neutral constructors into canonical constructors. However, the grammar does not capture a crucial property of the statics that ensures that only neutral constructors of kind `T` may be treated as canonical. This requirement

is imposed to limit the forms of canonical constructors of the other kinds. In particular, variables of function, product, or unit kind will turn out not to be canonical, but only neutral.

The statics of constructors and kinds is specified by the judgments

$$\begin{array}{ll} \Delta \vdash a \Uparrow \kappa & \text{neutral constructor formation} \\ \Delta \vdash c \Downarrow \kappa & \text{canonical constructor formation.} \end{array}$$

In each of these judgments Δ is a finite set of hypotheses of the form

$$u_1 \Uparrow \kappa_1, \ldots, u_n \Uparrow \kappa_n$$

for some $n \geq 0$. The form of the hypotheses expresses the principle that variables are neutral constructors. The formation judgments are to be understood as generic hypothetical judgments with parameters $u_1, \ldots, u_n$ that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow \kappa} \tag{22.1a}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{l} \Uparrow \kappa_1} \tag{22.1b}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{r} \Uparrow \kappa_2} \tag{22.1c}$$

$$\frac{\Delta \vdash a_1 \Uparrow \kappa_2 \to \kappa \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash a_1 [c_2] \Uparrow \kappa} \tag{22.1d}$$

$$\frac{\Delta \vdash a \Uparrow \mathtt{T}}{\Delta \vdash \widehat{a} \Downarrow \mathtt{T}} \tag{22.1e}$$

$$\frac{}{\Delta \vdash \langle\rangle \Downarrow 1} \tag{22.1f}$$

$$\frac{\Delta \vdash c_1 \Downarrow \kappa_1 \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \langle c_1 , c_2 \rangle \Downarrow \kappa_1 \times \kappa_2} \tag{22.1g}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \lambda\, u\, .\, c_2 \Downarrow \kappa_1 \to \kappa_2}\ . \tag{22.1h}$$

Rule (22.1e) specifies that the only neutral constructors that are canonical are those with kind $\mathtt{T}$. This ensures that the language enjoys the following canonical forms property, which is easily proved by inspection of Rules (22.1).

Lemma 22.1. *Suppose that $\Delta \vdash c \Downarrow \kappa$.*

1. *If $\kappa = 1$, then $c = \langle\rangle$.*
2. *If $\kappa = \kappa_1 \times \kappa_2$, then $c = \langle c_1 , c_2 \rangle$ for some c_1 and c_2 such that $\Delta \vdash c_i \Downarrow \kappa_i$ for $i = 1, 2$.*
3. *If $\kappa = \kappa_1 \to \kappa_2$, then $c = \lambda\, u\, .\, c_2$ for some u and c_2 such that $\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2$.*

22.2 Higher Kinds

To equip a language $\mathcal{L}$ with constructors and kinds requires that we augment its statics with hypotheses governing constructor variables and that we relate constructors of kind $\mathtt{T}$ (types as static data) to the classifiers of dynamic expressions (types as classifiers). To achieve this, the statics of $\mathcal{L}$ must be defined to have judgments of the following two forms:

$$\begin{array}{ll} \Delta \vdash \tau\ \mathsf{type} & \text{type formation} \\ \Delta\ \Gamma \vdash e : \tau & \text{expression formation} \end{array}$$

where, as before, Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k$$

for some $k \geq 0$ such that $\Delta \vdash \tau_i\ \mathsf{type}$ for each $1 \leq i \leq k$.

As a general principle, every constructor of kind $\mathtt{T}$ is a classifier:

$$\frac{\Delta \vdash \tau \Uparrow \mathtt{T}}{\Delta \vdash \tau\ \mathsf{type}}\ . \tag{22.2}$$

In many cases this is the sole rule of type formation, so that every classifier is a constructor of kind $\mathtt{T}$. However, this need not be the case. In some situations we may wish to have strictly more classifiers than constructors of the distinguished kind.

To see how this might arise, let us consider two extensions of $\mathcal{L}\{\rightarrow\forall\}$ from Chapter 20. In both cases we extend the universal quantifier $\forall(t\,.\,\tau)$ to admit quantification over an arbitrary kind, written as $\forall\, u :: \kappa\,.\,\tau$, but the two languages differ in what constitutes a constructor of kind $\mathtt{T}$. In one case, the *impredicative*, we admit quantified types as constructors, and in the other, the *predicative*, we exclude quantified types from the domain of quantification.

The impredicative fragment includes the following two constructor constants:

$$\frac{}{\Delta \vdash \rightarrow \Uparrow \mathtt{T} \rightarrow \mathtt{T} \rightarrow \mathtt{T}} \tag{22.3a}$$

$$\frac{}{\Delta \vdash \forall_\kappa \Uparrow (\kappa \rightarrow \mathtt{T}) \rightarrow \mathtt{T}}\ . \tag{22.3b}$$

We regard the classifier $\tau_1 \rightarrow \tau_2$ to be the application $\rightarrow[\tau_1]\,[\tau_2]$. Similarly, we regard the classifier $\forall\, u :: \kappa\,.\,\tau$ to be the application $\forall_\kappa[\lambda\, u\,.\,\tau]$.

The predicative fragment excludes the constant specified by Rule (22.3b) in favor of a separate rule for the formation of universally quantified types:

$$\frac{\Delta, u \Uparrow \kappa \vdash \tau\ \mathsf{type}}{\Delta \vdash \forall\, u :: \kappa\,.\,\tau\ \mathsf{type}}\ . \tag{22.4}$$

The point is that $\forall\, u :: \kappa\,.\,\tau$ is a type (as classifier), but is *not* a constructor of kind type.

The significance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

$$\frac{\Delta, u \Uparrow \kappa \; \Gamma \vdash e : \tau}{\Delta \; \Gamma \vdash \Lambda(u::\kappa.e) : \forall u :: \kappa . \tau} \tag{22.5a}$$

$$\frac{\Delta \; \Gamma \vdash e : \forall u :: \kappa . \tau \quad \Delta \vdash c \Downarrow \kappa}{\Delta \; \Gamma \vdash e[c] : [c/u]\tau} . \tag{22.5b}$$

[Rule (22.5b) makes use of substitution, whose definition requires some care. We return to this point in Section 22.3.]

Rule (22.5b) makes clear that a polymorphic abstraction quantifies over the constructors of kind κ. When κ is $\mathtt{T}$ this kind may or may not include all of the classifiers of the language, according to whether we are working with the impredicative formulation of quantification (in which the quantifiers are distinguished constants for building constructors of kind $\mathtt{T}$) or the predicative formulation (in which quantifiers arise only as classifiers and not as constructors).

The main idea is that *constructors are static data*, so that a constructor abstraction $\Lambda(u::\kappa.e)$ of type $\forall u :: \kappa . \tau$ is a mapping from static data c of kind κ to dynamic data $[c/u]e$ of type $[c/u]\tau$. Rule (22.1e) tells us that every constructor of kind $\mathtt{T}$ determines a classifier, but it may or may not be the case that every classifier arises in this manner.

22.3 Canonizing Substitution

Rule (22.5b) involves substitution of a canonical constructor c of kind κ into a family of types $u \Uparrow \kappa \vdash \tau$ type. This operation is written as $[c/u]\tau$, as usual. Although the intended meaning is clear, it is in fact impossible to interpret $[c/u]\tau$ as the standard concept of substitution defined in Chapter 1. The reason is that to do so would risk violating the distinction between neutral and canonical constructors. Consider, for example, the case of the family of types

$$u \Uparrow \mathtt{T} \to \mathtt{T} \vdash u[d] \Uparrow \mathtt{T},$$

where $d \Uparrow \mathtt{T}$. (It is not important what we choose for d, so we leave it abstract.) Now if $c \Downarrow \mathtt{T} \to \mathtt{T}$, then by Lemma 22.1 we have that c is $\lambda u' . c'$. Thus, if interpreted conventionally, substitution of c for u in the given family yields the "constructor" $(\lambda u' . c')[d]$, which is not well-formed.

The solution is to define a form of *canonizing substitution* that simplifies such "illegal" combinations as it performs the replacement of a variable by a constructor of the same kind. In the case just sketched this means that we must ensure that

$$[\lambda u' . c'/u]u[d] = [d/u']c'.$$

If viewed as a definition this equation is problematic because it switches from substituting for u in the constructor $u\,[d]$ to substituting for u' in the unrelated constructor c'. Why should such a process terminate? The answer lies in the observation that the kind of u' is definitely smaller than the kind of u, because the kind of the former is the domain kind of the latter. In all other cases of substitution (as we will see shortly) the size of the target of the substitution becomes smaller; in the case just cited the size may increase, but the type of the target variable decreases. Therefore by a lexicographic induction on the type of the target variable and the structure of the target constructor, we may prove that canonizing substitution is well-defined.

We now turn to the task of making this precise. We define simultaneously two principal forms of substitution, one of which divides into two cases:

$[c/u:\kappa]a = a'$	canonical into neutral yielding neutral
$[c/u:\kappa]a = c' \Downarrow \kappa'$	canonical into neutral yielding canonical and kind
$[c/u:\kappa]c' = c''$	canonical into canonical yielding canonical.

Substitution into a neutral constructor divides into two cases according to whether the substituted variable u occurs in a *critical position* in a sense subsequently made precise.

These forms of substitution are simultaneously inductively defined by the following rules, which are broken into groups for clarity.

The first set of rules defines substitution of a canonical constructor into a canonical constructor; the result is always canonical:

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\widehat{a'} = \widehat{a''}} \tag{22.6a}$$

$$\frac{[c/u:\kappa]a' = c'' \Downarrow \kappa''}{[c/u:\kappa]\widehat{a'} = c''} \tag{22.6b}$$

$$\frac{}{[u/\langle\rangle:\kappa]=\langle\rangle} \tag{22.6c}$$

$$\frac{[c/u:\kappa]c_1' = c_1'' \quad [c/u:\kappa]c_2' = c_2''}{[c/u:\kappa]\langle c_1', c_2'\rangle = \langle c_1'', c_2''\rangle} \tag{22.6d}$$

$$\frac{[c/u:\kappa]c' = c'' \quad (u \neq u') \quad (u' \notin c)}{[c/u:\kappa]\lambda\, u'.c' = \lambda\, u'.c''}\,. \tag{22.6e}$$

The conditions on variables in Rule (22.6e) may always be met by renaming the bound variable u' of the abstraction.

The second set of rules defines substitution of a canonical constructor into a neutral constructor, yielding another neutral constructor:

$$\frac{(u \neq u')}{[c/u:\kappa]u' = u'} \tag{22.7a}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]a' \cdot \mathtt{1} = a'' \cdot \mathtt{1}} \tag{22.7b}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]a' \cdot \mathtt{r} = a'' \cdot \mathtt{r}} \tag{22.7c}$$

$$\frac{[c/u:\kappa]a_1 = a_1' \quad [c/u:\kappa]c_2 = c_2'}{[c/u:\kappa]a_1\,[c_2] = a_1'\,(c_2')}\,. \tag{22.7d}$$

Rule (22.7a) pertains to a *noncritical* variable, which is not the target of substitution. The remaining rules pertain to situations in which the recursive call on a neutral constructor yields a neutral constructor.

The third set of rules defines substitution of a canonical constructor into a neutral constructor, yielding a canonical constructor and its kind:

$$\frac{}{[c/u:\kappa]u = c \Downarrow \kappa} \tag{22.8a}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]a' \cdot \mathtt{l} = c_1' \Downarrow \kappa_1'} \tag{22.8b}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]a' \cdot \mathtt{r} = c_2' \Downarrow \kappa_2'} \tag{22.8c}$$

$$\frac{[c/u:\kappa]a_1' = \lambda\, u'.\, c' \Downarrow \kappa_2' \to \kappa' \quad [c/u:\kappa]c_2' = c_2'' \quad [c_2''/u':\kappa_2']c' = c''}{[c/u:\kappa]a_1'\,[c_2'] = c'' \Downarrow \kappa'}\,. \tag{22.8d}$$

Rule (22.8a) governs a *critical* variable, which is the target of substitution. The substitution transforms it from a neutral constructor to a canonical constructor. This has a knock-on effect in the remaining rules of the group, which analyze the canonical form of the result of the recursive call to determine how to proceed. Rule (22.8d) is the most interesting rule. In the third premise, all three arguments to substitution change as we substitute the (substituted) argument of the application for the parameter of the (substituted) function into the body of that function. Here we require the type of the function in order to determine the type of its parameter.

Theorem 22.2. *Suppose that* $\Delta \vdash c \Downarrow \kappa$, $\Delta, u \Uparrow \kappa \vdash c' \Downarrow \kappa'$, *and* $\Delta, u \Uparrow \kappa \vdash a' \Uparrow \kappa'$. *There exists a unique* $\Delta \vdash c'' \Downarrow \kappa'$ *such that* $[c/u:\kappa]c' = c''$. *Either there exists a unique* $\Delta \vdash a'' \Uparrow \kappa'$ *such that* $[c/u:\kappa]a' = a''$ *or there exists a unique* $\Delta \vdash c'' \Downarrow \kappa'$ *such that* $[c/u:\kappa]a' = c''$, *but not both.*

Proof Simultaneously by a lexicographic induction with the major index being the structure of the kind κ and with the minor index determined by the formation rules for c' and a'. For all rules except Rule (22.8d), the inductive hypothesis applies to the premise(s) of the relevant formation rules. For Rule (22.8d) we appeal to the major inductive hypothesis applied to κ_2', which is a component of the kind $\kappa_2' \to \kappa'$. □

22.4 Canonization

With canonizing substitution in hand, it is perfectly possible to confine our attention to constructors in canonical form. However, for some purposes it can be useful to admit

a more relaxed syntax in which it is possible to form noncanonical constructors that may be transformed into canonical form. The prototypical example is the constructor $(\lambda\, u\,.\, c_2)\,[c_1]$, which is malformed according to Rules (22.1), because the first argument of an application is required to be in atomic form, whereas the λ-abstraction is in canonical form. However, if c_1 and c_2 are already canonical, then the malformed application may be transformed into the well-formed canonical form $[c_1/u]c_2$, where substitution is as defined in Section 22.3. If c_1 or c_2 is not already canonical, we may inductively put them into canonical form before performing the substitution, resulting in the same canonical form.

A constructor in *general form* is one that is well-formed with respect to Rules (22.1), but disregarding the distinction between atomic and canonical forms. We write $\Delta \vdash c :: \kappa$ to mean that c is a well-formed constructor of kind κ in general form. The difficulty with admitting general-form constructors is that they introduce nontrivial equivalences between constructors. For example, we must ensure that $\langle \mathtt{int}, \mathtt{bool}\rangle \cdot \mathtt{l}$ is equivalent to $\mathtt{int}$ wherever the former may occur. With this in mind we introduce a *canonization* procedure that allows us to define equivalence of general-form constructors, written as $\Delta \vdash c_1 \equiv c_2 :: \kappa$, to mean that c_1 and c_2 have identical canonical forms (up to α-equivalence).

Canonization of general-form constructors is defined by these two judgments:

1. Canonization: $\Delta \vdash c :: \kappa \Downarrow \overline{c}$: transform general-form constructor c of kind κ to canonical form $\overline{c}$.
2. Atomization: $\Delta \vdash c \Uparrow \underline{c} :: \kappa$: transform general-form constructor c to obtain atomic form $\underline{c}$ of kind κ.

These two judgments are defined simultaneously by the following rules. The canonization judgment is used to determine the canonical form of a general-form constructor; the atomization judgment is an auxiliary to the first that transforms constructors into atomic form. The canonization judgment is to be thought of as having mode $(\forall, \forall, \exists)$, whereas the atomization judgment is to be thought of as having mode $(\forall, \exists, \exists)$:

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \mathtt{T}}{\Delta \vdash c :: \mathtt{T} \Downarrow \widehat{\underline{c}}} \tag{22.9a}$$

$$\frac{}{\Delta \vdash c :: 1 \Downarrow \langle\rangle} \tag{22.9b}$$

$$\frac{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1 \Downarrow \overline{c_1} \quad \Delta \vdash c \cdot \mathtt{r} :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \times \kappa_2 \Downarrow \langle \overline{c_1}, \overline{c_2}\rangle} \tag{22.9c}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c\,[u] :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \to \kappa_2 \Downarrow \lambda\, u\,.\, \overline{c_2}} \tag{22.9d}$$

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow u :: \kappa} \tag{22.9e}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{l} \Uparrow \underline{c} \cdot \mathtt{l} :: \kappa_1} \tag{22.9f}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{r} \Uparrow \underline{c} \cdot \mathtt{r} :: \kappa_2} \tag{22.9g}$$

$$\frac{\Delta \vdash c_1 \Uparrow \underline{c_1} :: \kappa_1 \to \kappa_2 \quad \Delta \vdash c_2 :: \kappa_1 \Downarrow \overline{c_2}}{\Delta \vdash c_1\,[c_2] \Uparrow \underline{c_1}\,[\overline{c_2}] :: \kappa_2}\,. \tag{22.9h}$$

The canonization judgment produces canonical forms, and the atomization judgment produces atomic forms.

Lemma 22.3.

1. *If* $\Delta \vdash c :: \kappa \Downarrow \overline{c}$, *then* $\Delta \vdash \overline{c} \Downarrow \kappa$.
2. *If* $\Delta \vdash c \Uparrow \underline{c} :: \kappa$, *then* $\Delta \vdash \underline{c} \Uparrow \kappa$.

Proof By induction on Rules (22.9). □

Theorem 22.4. *If* $\Gamma \vdash c :: \kappa$, *then there exists* $\overline{c}$ *such that* $\Delta \vdash c :: \kappa \Downarrow \overline{c}$.

Proof By induction on the formation rules for general-form constructors, making use of an analysis of the general-form constructors of kind $\mathtt{T}$. □

22.5 Notes

The classical approach is to consider general-form constructors at the outset, for which substitution is readily defined, and then to test equivalence of general-form constructors by reduction to a common irreducible form. Two main lemmas are required for this approach. First, every constructor must reduce in a finite number of steps to an irreducible form; this is called *normalization*. Second, the relation "has a common irreducible form" must be shown to be transitive; this is called *confluence*. Here we have turned the development on its head by considering only canonical constructors in the first place, then defining canonizing substitution introduced by Watkins et al. (2008). It is then straightforward to decide equivalence of general-form constructors by canonization of both sides of a candidate equation.

PART VIII

Subtyping

23 Subtyping

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

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

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

Experience shows that the subsumption principle, 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 determine whether a candidate subtyping relationship is sensible, it suffices to consider whether every *introductory* form of the subtype can be safely manipulated by every *eliminatory* form of the supertype. A subtyping principle makes sense only if it passes this test; the proof of the type safety theorem for a given subtyping relation ensures that this is the case.

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

23.1 Subsumption

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

$$\frac{}{\tau <: \tau} \tag{23.1a}$$

$$\frac{\tau'' <: \tau' \quad \tau' <: \tau}{\tau'' <: \tau}\,. \tag{23.1b}$$

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

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

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash e : \tau}. \tag{23.2}$$

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

23.2 Varieties of Subtyping

In this section we informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 23.4 we examine some of these in more detail from the point of view of type safety.

23.2.1 Numeric Types

For languages with numeric types, our mathematical experience suggests subtyping relationships among them. For example, in a language with types `int`, `rat`, and `real`, representing the integers, the rationals, and the reals, respectively, it is tempting to postulate the subtyping relationships

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

by analogy with the set containments

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

familiar from mathematical experience.

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

For mathematical purposes it is entirely reasonable to overlook fine distinctions such as that between $\mathbb{Z}$ and its embedding within $\mathbb{Q}$. This is justified because the operations on rationals are restricted to the embedding in the expected manner: If we add two integers thought of as rationals in the canonical way, then the result is the rational associated with

their sum. The other operations are similar, provided that we take some care in defining them to ensure that it all works out properly. For the purposes of computing, however, we cannot be quite so cavalier, because we must also take account of algorithmic efficiency and the finiteness of machine representations. Often what are called "real numbers" in a programming language are, in fact, finite precision floating point numbers, a small subset of the rational numbers. Not every rational can be exactly represented as a floating point number, nor is floating point arithmetic restricted to rational arithmetic, even when its arguments are exactly represented as floating point numbers.

23.2.2 Product Types

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

Consider a context in which a value of type $\tau = \langle \tau_j \rangle_{j \in J}$ is required. The statics of finite products [Rules (11.3)] ensures that the only operation we may perform on a value of type τ, other than to bind it to a variable, is to take the jth projection from it for some $j \in J$ to obtain a value of type τ_j. Now suppose that e is of type τ'. If the projection $e \cdot j$ is to be well-formed, then τ' must be a finite-product type $\langle \tau'_i \rangle_{i \in I}$ such that $j \in I$. Moreover, for this to be of type τ_j, it is enough to require that $\tau'_j = \tau_j$. Because $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite-product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j}. \tag{23.3}$$

This rule is sufficient for the required subtyping, but not necessary; we consider a more liberal form of this rule in Section 23.3. The justification for Rule (23.3) is that we may evaluate $e \cdot i$ regardless of the actual form of e, provided only that it has a field indexed by $i \in I$.

23.2.3 Sum Types

By an argument dual to the one given for finite-product types, we may derive a related subtyping rule for finite-sum types. If a value of type $\sum_{j \in J} \tau_j$ is required, the statics of sums (Rules (12.3)) ensures that the only nontrivial operation that we may perform on that value is a J-indexed case analysis. If we provide a value of type $\sum_{i \in I} \tau'_i$ instead, no difficulty will arise so long as $I \subseteq J$ and each τ'_i is equal to τ_i. If the containment is strict, some cases cannot arise, but this does not disrupt safety. This leads to the following subtyping rule for finite sums:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j}. \tag{23.4}$$

Note well the reversal of the containment as compared with Rule (23.3).

23.3 Variance

In addition to basic subtyping principles such as those considered in Section 23.2, it is also important to consider the effect of subtyping on type constructors. A type constructor is said to be *covariant* in an argument if subtyping in that argument is preserved by the constructor. It is said to be *contravariant* if subtyping in that argument is reversed by the constructor. It is said to be *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

23.3.1 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 expected to be of type τ_j, then it is sufficient to require that $j \in I$ and $\tau_j' <: \tau_j$. This is summarized by the following rule:

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\prod_{i \in I} \tau_i' <: \prod_{i \in I} \tau_i}. \tag{23.5}$$

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

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

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\sum_{i \in I} \tau_i' <: \sum_{i \in I} \tau_i}. \tag{23.6}$$

23.3.2 Function Types

The variance of the function type constructor is a bit more subtle. Let us consider first the variance of the function type in its range. Suppose that $e : \tau_1 \to \tau_2'$. This means that if $e_1 : \tau_1$, then $e(e_1) : \tau_2'$. If $\tau_2' <: \tau_2$, then $e(e_1) : \tau_2$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau_2' <: \tau_2}{\tau_1 \to \tau_2' <: \tau_1 \to \tau_2}. \tag{23.7}$$

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

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

of e as having type $\tau_1' \to \tau_2$:

$$\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2}. \tag{23.8}$$

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

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

$$\frac{\tau_1' <: \tau_1 \quad \tau_2' <: \tau_2}{\tau_1 \to \tau_2' <: \tau_1' \to \tau_2}. \tag{23.9}$$

Beware of the reversal of the ordering in the domain!

23.3.3 Quantified Types

The extension of subtyping to quantified types requires a judgment of the form $\Delta \vdash \tau' <: \tau$, where $\Delta \vdash \tau'$ type and $\Delta \vdash \tau$ type. The variance principles for the quantifiers may then be stated so that both are covariant in the quantified type:

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \forall(t.\tau') <: \forall(t.\tau)} \tag{23.10a}$$

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \exists(t.\tau') <: \exists(t.\tau)}. \tag{23.10b}$$

The judgment $\Delta \vdash \tau' <: \tau$ states that τ' is a subtype of τ uniformly in the type variables declared in Δ. Consequently we may derive the principle of substitution:

Lemma 23.1. *If Δ, t type $\vdash \tau' <: \tau$, and $\Delta \vdash \rho$ type, then $\Delta \vdash [\rho/t]\tau' <: [\rho/t]\tau$.*

Proof By induction on the subtyping derivation. □

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

It is natural to extend subtyping to the quantifiers by allowing quantification over all subtypes of a specified type. This is called *bounded quantification*. To express bounded quantification we consider additional hypotheses of the form $t <: \rho$, expressing that t is a variable that may only be instantiated to subtypes of ρ:

$$\frac{}{\Delta, t \text{ type}, t <: \tau \vdash t <: \tau} \tag{23.11a}$$

$$\frac{\Delta \vdash \tau :: \mathtt{T}}{\Delta \vdash \tau <: \tau} \tag{23.11b}$$

$$\frac{\Delta \vdash \tau'' <: \tau' \quad \Delta \vdash \tau' <: \tau}{\Delta \vdash \tau'' <: \tau} \tag{23.11c}$$

$$\frac{\Delta \vdash \rho' <: \rho \quad \Delta, t \text{ type}, t <: \rho' \vdash \tau' <: \tau}{\Delta \vdash \forall t <: \rho . \tau' <: \forall t <: \rho' . \tau'} \tag{23.11d}$$

$$\frac{\Delta \vdash \rho' <: \rho \quad \Delta, t \text{ type}, t <: \rho' \vdash \tau' <: \tau}{\Delta \vdash \exists t <: \rho' . \tau' <: \exists t <: \rho . \tau} . \tag{23.11e}$$

Rule (23.11d) states that the universal quantifier is contravariant in its bound, whereas Rule (23.11e) states that the existential quantifier is covariant in its bound.

23.3.4 Recursive Types

The variance principle for recursive types is rather subtle and has been the source of errors in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mu t . [\texttt{empty} \hookrightarrow \texttt{unit}, \texttt{binode} \hookrightarrow \langle \texttt{data} \hookrightarrow \texttt{nat}, \texttt{lft} \hookrightarrow t, \texttt{rht} \hookrightarrow t \rangle],$$

and the type of "bare" binary trees, without labels on the nodes,

$$\mu t . [\texttt{empty} \hookrightarrow \texttt{unit}, \texttt{binode} \hookrightarrow \langle \texttt{lft} \hookrightarrow t, \texttt{rht} \hookrightarrow t \rangle].$$

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:

$$\mu t . [\texttt{empty} \hookrightarrow \texttt{unit}, \texttt{binode} \hookrightarrow \tau_2, \texttt{trinode} \hookrightarrow \tau_3],$$

where

$$\tau_2 \triangleq \langle \texttt{lft} \hookrightarrow t, \texttt{rht} \hookrightarrow t \rangle,$$

and

$$\tau_3 \triangleq \langle \texttt{lft} \hookrightarrow t, \texttt{mid} \hookrightarrow t, \texttt{rht} \hookrightarrow t \rangle.$$

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 must proceed by three-way case analysis, which covers both forms of binary tree.

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

$$\frac{t \text{ type} \vdash \tau' <: \tau}{\mu t . \tau' <: \mu t . \tau} \ \textbf{??} \tag{23.12}$$

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

Rule (23.12) validates the intuitively plausible subtyping between labeled binary trees 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 (23.12) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\tau' = \mu t\,.\,\langle \mathtt{a} \hookrightarrow t \to \mathtt{nat}, \mathtt{b} \hookrightarrow t \to \mathtt{int} \rangle$$

and

$$\tau = \mu t\,.\,\langle \mathtt{a} \hookrightarrow t \to \mathtt{int}, \mathtt{b} \hookrightarrow t \to \mathtt{int} \rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using Rule (23.12) we may derive $\tau' <: \tau$, which we will show to be 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 $q : \mathtt{nat} \to \mathtt{nat}$ is the discrete square root function. Because $\tau' <: \tau$, it follows that $e : \tau$ as well, and hence

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

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

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

(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

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

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

Rule (23.12) is therefore incorrect. But what has gone wrong? The error lies in the choice of a single parameter to stand for both recursive types, which does not correctly model self-reference. In effect we are regarding two distinct recursive types as equal while checking their bodies for a subtyping relationship. But this is clearly wrong! It fails to take account of the self-referential nature of recursive types. On the left-hand side the bound variable stands for the subtype, whereas on the right-hand side the bound variable stands for the supertype. Confusing them leads to the unsoundness just illustrated.

As is often the case with self-reference, the solution is to *assume* what we are trying to prove and check that this assumption can be maintained by examining the bodies of

the recursive types. To do so we make use of bounded quantification to state the rule of subsumption for recursive types:

$$\frac{\Delta, t\ \mathsf{type}, t'\ \mathsf{type}, t' <: t \vdash \tau' <: \tau \quad \Delta, t'\ \mathsf{type} \vdash \tau'\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mu t'.\tau' <: \mu t.\tau}. \tag{23.13}$$

That is, to check whether $\mu t'.\tau' <: \mu t.\tau$, we assume that $t' <: t$, because t' and t stand for the respective recursive types, and check that $\tau' <: \tau$ under this assumption. It is instructive to check that the unsound subtyping is *not* derivable using this rule: The subtyping assumption is at odds with the contravariance of the function type in its domain.

23.4 Safety

Proving safety for a language with subtyping is considerably more delicate than for languages without. The rule of subsumption means that the static type of an expression reveals only partial information about the underlying value. This changes the proof of the preservation and progress theorems and requires some care in stating and proving the auxiliary lemmas required for the proof.

As a representative case we sketch the proof of safety for a language with subtyping for product types. The subtyping relation is defined by Rules (23.3) and (23.5). We assume that the statics includes subsumption, Rule (23.2).

Lemma 23.2 (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 (11.3), augmented by Rule (23.2). □

Lemma 23.3 (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 (23.2). □

Theorem 23.4 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof By induction on Rules (11.4). For example, consider Rule (11.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$, $e' = e_k$. By Lemma 23.3 we have that $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, $k \in J$, and $\tau_k <: \tau$. By another application of Lemma 23.3 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 23.3 again, we have $J \subseteq I$ and $\tau_j' <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaing cases are similar. □

Lemma 23.5 (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 (11.3) augmented by Rule (23.2). □

Theorem 23.6 (Progress). *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof By induction on Rules (11.3) augmented by Rule (23.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. Rule (11.4d) follows from Lemma 23.5. □

23.5 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 whenever τ' is a subtype of τ, it also weakens the meaning of a type assertion $e : \tau$ to mean that e has some type contained in the type τ. This precludes expressing the requirement that e has *exactly* the type τ, or that two expressions jointly have the *same* type. And it is precisely this weakness that creates so many of the difficulties with subtyping.

Much has been written about subtyping. Standard ML (Milner et al., 1997) is one of the earliest full-scale languages to make essential use of subtyping. The statics of the ML module system makes use of two subtyping relations, called *enrichment* and *realization*, corresponding to product subtyping and type definitions. The first systematic studies of subtyping include those by Mitchell (1984), Reynolds (1980), and Cardelli (1988). Pierce (2002) gives a thorough account of subtyping, especially of recursive and polymorphic types, and proves that subtyping for bounded impredicative universal quantification is undecidable.

24 Singleton Kinds

The expression $\texttt{let}\ e_1 : \tau\ \texttt{be}\ x\ \texttt{in}\ e_2$ is a form of abbreviation mechanism by which we may bind e_1 to the variable x for use within e_2. In the presence of function types this expression is definable as the application $(\lambda\ (x:\tau)\ e_2)(e_1)$, which accomplishes the same thing. It is natural to consider an analogous form of `let` expression that permits a type expression to be bound to a type variable within a specified scope. Using $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ to bind the type variable t to τ within the expression e, we may write expressions such as

$$\texttt{def}\ t\ \texttt{is}\ \texttt{nat} \times \texttt{nat}\ \texttt{in}\ \lambda\ (x:t)\ \texttt{s}(x \cdot \texttt{l}),$$

which introduces a *type abbreviation* within an expression. To ensure that this expression is well-typed, the type variable t must be *synonymous* with the type $\texttt{nat} \times \texttt{nat}$, for otherwise the body of the λ-abstraction is not type correct.

Following the pattern of the expression level `let`, we might guess that $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ abbreviates the polymorphic instantiation $\Lambda(t.e)[\tau]$, which binds t to τ within e. This does indeed capture the dynamics of type abbreviation, but it fails to validate the intended statics. The difficulty is that, according to this interpretation of type definitions, the expression e is type checked in the absence of any knowledge of the binding of t, rather than in the knowledge that t is synonymous with τ. Thus, in the preceding example, the expression $\texttt{s}(x \cdot \texttt{l})$ would fail to type check, unless the binding of t were exposed.

The interpretation of type definition in terms of type abstraction and type application fails. Lacking any other idea, we might argue that type abbreviation ought to be considered as a primitive concept, rather than as a derived notion. The expression $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ would be taken as a primitive form of expression whose statics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e : \tau'}. \tag{24.1}$$

This would address the problem of supporting type abbreviations, but it does so in a rather ad hoc manner. We seek a more principled solution that arises naturally from the type structure of the language.

The methodology of identifying language constructs with type structure suggests that we ask not how to support type abbreviations, but rather what form of type structure gives rise to type abbreviations? Thinking along these lines leads to the concept of *singleton kinds*, which not only account for type abbreviations but also play a crucial role in the design of module systems, as is discussed in detail in Chapters 45 and 46.

24.1 Overview

The central organizing principle of type theory is *compositionality*. To ensure that a program may be decomposed into separable parts, we ensure that the composition of a program from constituent parts is mediated by the types of those parts. Put in other terms, the only thing that one portion of a program "knows" about another is its type. For example, the formation rule for addition of natural numbers depends only on the type of its arguments (both must have type `nat`) and not on their specific form or value. But in the case of a type abbreviation of the form `def` t `is` τ `in` e, the principle of compositionality dictates that the only thing that e knows about the type variable t is its kind, namely `T`, and not its binding, namely τ. This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of t to capture its identity. This may be achieved through the notion of a *singleton kind*. Informally, the kind `S(`τ`)` is the kind of types that is definitionally equal to τ. That is, up to definitional equality, this kind has only one inhabitant, namely τ. Consequently, if u :: `S(`τ`)` is a variable of singleton kind, then within its scope, the variable u is synonymous with τ. Thus we may represent `def` t `is` τ `in` e by Λ`(`t`::S(`τ`).`e`)[`τ`]`, which correctly propagates the identity of t, namely τ, to e during type checking.

A proper treatment of singleton kinds requires some additional machinery at the constructor and kind level. First, we must capture the idea that a constructor of singleton kind is a fortiori a constructor of kind `T` and hence is a type. Otherwise, a variable u of singleton kind cannot be used as a type, even though it is explicitly defined to be one! This may be captured by introducing a *subkinding* relation, κ_1 :<: κ_2, which is analogous to subtyping, except at the kind level. The fundamental axiom of subkinding is `S(`τ`)` :<: `T`, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind `T` within the singleton kind `S(`τ`)`. This intermixing of the constructor and kind level means that singletons are a form of *dependent kind* in that a kind may depend on a constructor. Another way to say the same thing is that `S(`τ`)` represents a *family of kinds* indexed by constructors of kind `T`. This in turn implies that we must generalize the product and function kinds to *dependent products* and *dependent functions*. The dependent product kind, $\Sigma\, u::\kappa_1\,.\,\kappa_2$, classifies pairs $\langle c_1, c_2\rangle$ such that c_1 :: κ_1, as might be expected, and c_2 :: $[c_1/u]\kappa_2$, in which the kind of the second component is sensitive to the first component itself, and not just its kind. The dependent function kind, $\Pi\, u::\kappa_1\,.\,\kappa_2$ classifies functions that, when applied to a constructor c_1 :: κ_1, results in a constructor of kind $[c_1/u]\kappa_2$. The important point is that the kind of the result is sensitive to the argument and not just to its kind.

Third, it is useful to consider singletons not just of kind `T`, but also of higher kinds. To support this we introduce *higher singletons*, written as `S(`c `::` κ`)`, where κ is a kind and c is

a constructor of kind k. These are definable in terms of the basic form of singleton kinds using dependent function and product kinds.

24.2 Singletons

The syntax of singleton kinds is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Kind	κ	$::=$	$\mathtt{S}(c)$	$\mathtt{S}(c)$	singleton

Informally, the singleton kind $\mathtt{S}(c)$ classifies constructors that are equivalent (in a sense to be made precise shortly) to c. For the time being we tacitly include the constructors and kinds given in Chapter 22 (but see Section 24.3).

The following judgment forms comprise the statics of singletons:

$\Delta \vdash \kappa\ \mathsf{kind}$	kind formation
$\Delta \vdash \kappa_1 \equiv \kappa_2$	kind equivalence
$\Delta \vdash c :: \kappa$	constructor formation
$\Delta \vdash c_1 \equiv c_2 :: \kappa$	constructor equivalence
$\Delta \vdash \kappa_1 :<: \kappa_2$	subkinding.

These judgments are defined simultaneously by a collection of rules including the following:

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c)\ \mathsf{kind}} \tag{24.2a}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash c :: \mathtt{S}(c)} \tag{24.2b}$$

$$\frac{\Delta \vdash c :: \mathtt{S}(d)}{\Delta \vdash c \equiv d :: \mathtt{Type}} \tag{24.2c}$$

$$\frac{\Delta \vdash c :: \kappa_1 \quad \Delta \vdash \kappa_1 :<: \kappa_2}{\Delta \vdash c :: \kappa_2} \tag{24.2d}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) :<: \mathtt{Type}} \tag{24.2e}$$

$$\frac{\Delta \vdash c \equiv d :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) \equiv \mathtt{S}(d)} \tag{24.2f}$$

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_2}{\Delta \vdash \kappa_1 :<: \kappa_2} \tag{24.2g}$$

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_2 \quad \Delta \vdash \kappa_2 :<: \kappa_3}{\Delta \vdash \kappa_1 :<: \kappa_3}\,. \tag{24.2h}$$

Omitted for brevity are rules stating that constructor and kind equivalence are reflexive, symmetric, transitive, and preserved by kind and constructor formation.

Rule (24.2b) expresses the principle of "self-recognition," which states that every constructor c of kind $\mathtt{Type}$ also has the kind $\mathtt{S}(c)$. By Rule (24.2c) any constructor of kind $\mathtt{S}(c)$ is definitionally equal to c. Consequently, self-recognition is in this sense an expression of the reflexivity of constructor equivalence. Rule (24.2e) is just the subsumption principle restated at the level of constructors and kinds. Rule (24.2f) states that the singleton kind respects equivalence of its constructors, so that equivalent constructors determine the same singletons. Rules (24.2g) and (24.2h) state that the subkinding relation is a preorder that respects kind equivalence.

To see these rules in action let us consider a few illustrative examples. First, consider the behavior of variables of singleton kind. Suppose that $\Delta \vdash u :: \mathtt{S}(c)$ is such a variable. Then by Rule (24.2c) we may deduce that $\Delta \vdash u \equiv c :: \mathtt{T}$. Thus the declaration of u with a singleton kind serves to define u to be the constructor (of kind $\mathtt{T}$) specified by its kind. Singletons capture the concept of a type definition discussed in the introduction to this chapter.

Taking this a step further, the existential type $\exists\, u :: \mathtt{S}(c)\,.\,\tau$ is the type of packages whose representation type is (equivalent to) c—it is an abstract type whose identity is revealed by assigning it a singleton kind. By the general principles of equivalence we have that the type $\exists\, u :: \mathtt{S}(c)\,.\,\tau$ is equivalent to the type $\exists\, \_ :: \mathtt{S}(c)\,.\,[c/u]\tau$, wherein we have propagated the equivalence of u and c into the type τ. On the other hand we may also "forget" the definition of u, because the subtyping

$$\exists\, u :: \mathtt{S}(c)\,.\,\tau \mathrel{<:} \exists\, u :: \mathtt{T}\,.\,\tau$$

is derivable using the following variance rule for existentials over a kind:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_2 \quad \Delta, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Delta \vdash \exists\, u :: \kappa_1\,.\,\tau_1 <: \exists\, u :: \kappa_2\,.\,\tau_2}\,. \tag{24.3}$$

Similarly, we may derive the subtyping

$$\forall\, u :: \mathtt{T}\,.\,\tau \mathrel{<:} \forall\, u :: \mathtt{S}(c)\,.\,\tau$$

from the following variance rule for universals over a kind:

$$\frac{\Delta \vdash \kappa_2 :<: \kappa_1 \quad \Delta, u :: \kappa_2 \vdash \tau_1 <: \tau_2}{\Delta \vdash \forall\, u :: \kappa_1\,.\,\tau_1 <: \forall\, u :: \kappa_2\,.\,\tau_2}\,. \tag{24.4}$$

Informally, the displayed subtyping states that a polymorphic function that may be applied to *any* type is one that may be applied only to a particular type c.

These examples show that singleton kinds express the idea of a scoped definition of a type variable in a way that is not tied to an ad hoc definition mechanism, but rather arises naturally from general principles of binding and scope. We will see in Chapters 45 and 46 more sophisticated uses of singletons to manage the interaction among program modules.

24.3 Dependent Kinds

Although it is perfectly possible to add singleton kinds to the framework of higher kinds introduced in Chapter 22, to do so would be to shortchange the expressiveness of the language. Using higher kinds we can express the kind of constructors that, when applied to a type, yield a specific type, say `int`, as result, namely $\mathtt{T} \to \mathtt{S(int)}$. But we cannot express the kind of constructors that, when applied to a type, yield *that very type* as result, for there is no way for the result kind to refer to the argument of the function. Similarly, using product kinds, we can express the kind of pairs whose first component is `int` and whose second component is an arbitrary type, namely $\mathtt{S(int)} \times \mathtt{T}$. But we cannot express the kind of pairs whose second component is equivalent to its first component, for there is no way for the kind of the second component to make reference to the first component itself.

To express such concepts requires a generalization of product and function kinds in which the kind of the second component of a pair may mention the first component of that pair, or the kind of the result of a function may mention the argument to which it is applied. Such kinds are called *dependent kinds* because they involve kinds that mention, or depend on, constructors (of kind `T`). The syntax of dependent kinds is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Kind	κ	$::=$	$\mathtt{S}(c)$	$\mathtt{S}(c)$	singleton
			$\Sigma\,(\kappa_1; u\,.\,\kappa_2)$	$\Sigma\,u::\kappa_1\,.\,\kappa_2$	dependent product
			$\Pi\,(\kappa_1; u\,.\,\kappa_2)$	$\Pi\,u::\kappa_1\,.\,\kappa_2$	dependent function
Con	c	$::=$	u	u	variable
			$\mathtt{pair}(c_1; c_2)$	$\langle c_1, c_2\rangle$	pair
			$\mathtt{proj[l]}(c)$	$c \cdot \mathtt{l}$	first projection
			$\mathtt{proj[r]}(c)$	$c \cdot \mathtt{r}$	second projection
			$\mathtt{lam}[\kappa](u\,.\,c)$	$\lambda\,(u::\kappa)\,c$	abstraction
			$\mathtt{app}(c_1; c_2)$	$c_1\,[c_2]$	application

As a notational convenience, when there is no dependency in a kind we write $\kappa_1 \times \kappa_2$ for $\Sigma\,\_::\kappa_1\,.\,\kappa_2$, and $\kappa_1 \to \kappa_2$ for $\Pi\,\_::\kappa_1\,.\,\kappa_2$, where the "blank" stands for an irrelevant variable.

The syntax of dependent kinds differs from that given in Chapter 22 for higher kinds in that we do not draw a distinction between atomic and canonical constructors and consider that substitution as defined conventionally, rather than hereditarily. This simplifies the syntax, but at the expense of leaving open the decidability of constructor equivalence. The method of hereditary substitution considered in Chapter 22 may be extended to singleton kinds, but we do not develop this extension here. Instead we simply assert without proof that equivalence of well-formed constructors is decidable.

The dependent product kind $\Sigma\,u::\kappa_1\,.\,\kappa_2$ classifies pairs $\langle c_1, c_2\rangle$ of constructors in which c_1 has kind κ_1 and c_2 has kind $[c_1/u]\kappa_2$. For example, the kind $\Sigma\,u::\mathtt{T.S}(u)$ classifies pairs $\langle c, c\rangle$, where c is a constructor of kind `T`. More generally, this kind classifies pairs of the form $\langle c_1, c_2\rangle$, where c_1 and c_2 are equivalent, but not necessarily identical, constructors.

The dependent function kind $\Pi\, u::\kappa_1.\kappa_2$ classifies constructors c that, when applied to a constructor c_1 of kind κ_1 yield a constructor of kind $[c_1/u]\kappa_2$. For example, the kind $\Pi\, u::\mathtt{T.S}(u)$ classifies constructors that, when applied to a constructor c, yield a constructor equivalent to c; a constructor of this kind is essentially the identity function. We may, of course, combine these to form kinds such as

$$\Pi\, u::\mathtt{T} \times \mathtt{T.S}(u \cdot \mathtt{r}) \times \mathtt{S}(u \cdot \mathtt{l}),$$

which classifies functions that swap the components of a pair of types. (Such examples may lead us to surmise that the behavior of any constructor may be pinned down precisely by use of dependent kinds. We see in Section 24.4 that this is indeed the case.)

The formation, introduction, and elimination rules for the product kind are as follows:

$$\frac{\Delta \vdash \kappa_1\ \mathsf{kind} \quad \Delta, u :: \kappa_1 \vdash \kappa_2\ \mathsf{kind}}{\Delta \vdash \Sigma\, u::\kappa_1.\kappa_2\ \mathsf{kind}} \tag{24.5a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: [c_1/u]\kappa_2}{\Delta \vdash \langle c_1, c_2\rangle :: \Sigma\, u::\kappa_1.\kappa_2} \tag{24.5b}$$

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1.\kappa_2}{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1} \tag{24.5c}$$

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1.\kappa_2}{\Delta \vdash c \cdot \mathtt{r} :: [c_1/u]\kappa_2}\,. \tag{24.5d}$$

In Rule (24.5a), observe that the variable u may occur in the kind κ_2 by appearing in a singleton kind. Correspondingly, Rules (24.5b), (24.5c), and (24.5d) substitute a constructor for this variable.

Constructor equivalence is defined to be an equivalence relation that is compatible with all forms of constructors and kinds, so that a constructor may always be replaced by an equivalent constructor and the result will be equivalent. The following equivalence axioms govern the constructors associated with the dependent product kind:

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2\rangle \cdot \mathtt{l} \equiv c_1 :: \kappa_1} \tag{24.6a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2\rangle \cdot \mathtt{r} \equiv c_2 :: \kappa_2}\,. \tag{24.6b}$$

The subkinding rule for the dependent product kind specifies that it is covariant in both positions:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 :<: \kappa_2'}{\Delta \vdash \Sigma\, u::\kappa_1.\kappa_2 :<: \Sigma\, u::\kappa_1'.\kappa_2'}\,. \tag{24.7}$$

The congruence rule for equivalence of dependent product kinds is formally similar:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \Sigma\, u::\kappa_1.\kappa_2 \equiv \Sigma\, u::\kappa_1'.\kappa_2'}\,. \tag{24.8}$$

Notable consequences of these rules include the subkindings

$$\Sigma\, u::\mathtt{S(int)}.\mathtt{S}(u) :<: \Sigma\, u::\mathtt{T}.\mathtt{S}(u)$$

$$\Sigma\, u::\mathtt{T}.\mathtt{S}(u) :<: \mathtt{T} \times \mathtt{T}$$

and the equivalence

$$\Sigma\, u::\mathtt{S(int)}.\mathtt{S}(u) \equiv \mathtt{S(int)} \times \mathtt{S(int)}.$$

Subkinding is used to "forget" information about the identity of the components of a pair, and equivalence is used to propagate such information within a kind.

The formation, introduction, and elimination rules for dependent function kinds are as follows:

$$\frac{\Delta \vdash \kappa_1 \text{ kind} \quad \Delta, u :: \kappa_2 \vdash \kappa_2 \text{ kind}}{\Delta \vdash \Pi\, u::\kappa_1.\kappa_2 \text{ kind}} \tag{24.9a}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2}{\Delta \vdash \lambda\,(u::\kappa_1)\, c :: \Pi\, u::\kappa_1.\kappa_2} \tag{24.9b}$$

$$\frac{\Delta \vdash c :: \Pi\, u::\kappa_1.\kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash c\,[c_1] :: [c_1/u]\kappa_2}. \tag{24.9c}$$

Rule (24.9b) specifies that the result kind of a λ-abstraction depends uniformly on the argument u. Correspondingly, Rule (24.9c) specifies that the kind of an application is obtained by substitution of the argument into the result kind of the function itself.

The following rule of equivalence governs the constructors associated with the dependent product kind:

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash (\lambda\,(u::\kappa_1)\, c)\,[c_1] \equiv [c_1/u]c :: \kappa_2}. \tag{24.10}$$

The subkinding rule for the dependent function kind specifies that it is contravariant in its domain and covariant in its range:

$$\frac{\Delta \vdash \kappa_1' :<: \kappa_1 \quad \Delta, u :: \kappa_1' \vdash \kappa_2 :<: \kappa_2'}{\Delta \vdash \Pi\, u::\kappa_1.\kappa_2 :<: \Pi\, u::\kappa_1'.\kappa_2'}. \tag{24.11}$$

The equivalence rule is similar, except that the symmetry of equivalence obviates a choice of variance:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \Pi\, u::\kappa_1.\kappa_2 \equiv \Pi\, u::\kappa_1'.\kappa_2'}. \tag{24.12}$$

Rule (24.11) gives rise to the subkinding

$$\Pi\, u::\mathtt{T}.\mathtt{S(int)} :<: \Pi\, u::\mathtt{S(int)}.\mathtt{T},$$

which illustrates the covariance and contravariance of the dependent function kind. In particular, a function that takes any type and delivers the type `int` is also a function that takes the type `int` and delivers a type. Rule (24.12) gives rise to the equivalence

$$\Pi\, u::\mathtt{S(int)}.\mathtt{S}(u) \equiv \mathtt{S(int)} \to \mathtt{S(int)},$$

which propagates information about the argument into the range kind. Combining these two rules, we may derive the subkinding

$$\Pi\, u::\mathtt{T}.\mathtt{S}(u) :<: \mathtt{S}(\mathtt{int}) \to \mathtt{S}(\mathtt{int}).$$

Intuitively, a constructor function that yields its argument is, in particular, a constructor function that may be applied only to `int` and yields `int`. Formally, by contravariance we have the subkinding

$$\Pi\, u::\mathtt{T}.\mathtt{S}(u) :<: \Pi\, u::\mathtt{S}(\mathtt{int}).\mathtt{S}(u),$$

and by sharing propagation we may derive the indicated superkind.

24.4 Higher Singletons

Although singletons are restricted to constructors of kind `T`, we may use dependent product and function kinds to define singletons of every kind. Specifically, we wish to define the kind $\mathtt{S}(c::\kappa)$, where c is of kind κ, that classifies constructors equivalent to c. When $\kappa = \mathtt{T}$ this is, of course, just $\mathtt{S}(c)$; the problem is to define singletons for the higher kinds $\Sigma\, u::\kappa_1.\kappa_2$ and $\Pi\, u::\kappa_1.\kappa_2$.

To see what is involved, suppose that $c :: \kappa_1 \times \kappa_2$. The singleton kind $\mathtt{S}(c::\kappa_1 \times \kappa_2)$ should classify constructors equivalent to c. If we assume, inductively, that singletons have been defined for κ_1 and κ_2, then we need only observe that c is equivalent to $\langle c\cdot\mathtt{l}, c\cdot\mathtt{r}\rangle$. For then the singleton $\mathtt{S}(c::\kappa_1\times\kappa_2)$ may be defined to be $\mathtt{S}(c\cdot\mathtt{l}::\kappa_1)\times\mathtt{S}(c\cdot\mathtt{r}::\kappa_2)$. Similarly, suppose that $c :: \kappa_1 \to \kappa_2$. Using the equivalence of c and $\lambda\,(u::\kappa_1\to\kappa_2)\,c[u]$, we may define $\mathtt{S}(c::\kappa_1\to\kappa_2)$ to be $\Pi\, u::\kappa_1.\mathtt{S}(c[u]::\kappa_2)$.

In general the kind $\mathtt{S}(c::\kappa)$ is defined by induction on the structure of κ by the following kind equivalences:

$$\frac{\Delta \vdash c :: \mathtt{S}(c')}{\Delta \vdash \mathtt{S}(c::\mathtt{S}(c')) \equiv \mathtt{S}(c)} \tag{24.13a}$$

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1.\kappa_2}{\Delta \vdash \mathtt{S}(c::\Sigma\, u::\kappa_1.\kappa_2) \equiv \Sigma\, u::\mathtt{S}(c\cdot\mathtt{l}::\kappa_1).\mathtt{S}(c\cdot\mathtt{r}::\kappa_2)} \tag{24.13b}$$

$$\frac{\Delta \vdash c :: \Pi\, u::\kappa_1.\kappa_2}{\Delta \vdash \mathtt{S}(c::\Pi\, u::\kappa_1.\kappa_2) \equiv \Pi\, u::\kappa_1.\mathtt{S}(c[u]::\kappa_2)}\,. \tag{24.13c}$$

The sensibility of these equations relies on Rule (24.2c) together with the following principles of constructor equivalence, called *extensionality principles*:

$$\frac{\Delta \vdash c :: \Sigma\, u::\kappa_1.\kappa_2}{\Delta \vdash c \equiv \langle c\cdot\mathtt{l}, c\cdot\mathtt{r}\rangle :: \Sigma\, u::\kappa_1.\kappa_2} \tag{24.14a}$$

$$\frac{\Delta \vdash c :: \Pi\, u::\kappa_1.\kappa_2}{\Delta \vdash c \equiv \lambda\,(u::\kappa_1)\,c[u] :: \Pi\, u::\kappa_1.\kappa_2}\,. \tag{24.14b}$$

Rule (24.2c) states that the only constructors of kind $\mathtt{S}(c')$ are those equivalent to c', and Rules (24.14a) and (24.14b) state that the only members of the dependent product and function types are pairs and λ-abstractions, respectively, of the appropriate kinds.

Finally, the following *self-recognition* rules are required to ensure that Rule (24.2b) may be extended to higher kinds:

$$\frac{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1 \quad \Delta \vdash c \cdot \mathtt{r} :: [c \cdot \mathtt{l}/u]\kappa_2}{\Delta \vdash c :: \Sigma\, u::\kappa_1 . \kappa_2} \tag{24.15a}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c\,[u] :: \kappa_2}{\Delta \vdash c :: \Pi\, u::\kappa_1 . \kappa_2}\,. \tag{24.15b}$$

An illustrative case arises when u is a constructor variable of kind $\Sigma\, v::\mathtt{T.S}(v)$. We may derive $u \cdot \mathtt{l} :: \mathtt{S}(u \cdot \mathtt{l})$ by using Rule (24.2b). We may also derive $u \cdot \mathtt{r} :: \mathtt{S}(u \cdot \mathtt{l})$ by using Rule (24.5d). Therefore, by Rule (24.15a), we may derive $u :: \Sigma\, v::\mathtt{S}(u \cdot \mathtt{l}).\mathtt{S}(u \cdot \mathtt{l})$, which is a subkind of $\Sigma\, v::\mathtt{T.S}(v)$. This more precise kind is a correct kinding for u, because the first component of u is indeed $u \cdot \mathtt{l}$ and the second component of u is equivalent to the first component, and hence is also $u \cdot \mathtt{l}$. But without Rule (24.15a) it is impossible to derive this fact.

The point of introducing higher singletons is to ensure that every constructor may be classified by a kind that determines it up to definitional equality. We would expect that higher singletons, viewed as an extension of singleton types, enjoy similar properties. This is captured by the following theorem:

Theorem 24.1. *If* $\Delta \vdash c :: \kappa$*, then* $\Delta \vdash \mathtt{S}(c :: \kappa) :<: \kappa$ *and* $\Delta \vdash c :: \mathtt{S}(c :: \kappa)$*.*

The proof of this theorem is surprisingly intricate; the reader is referred to the references in the next section for details.

24.5 Notes

Singleton kinds were introduced by Stone and Harper (2006) to isolate the concept of type sharing that arises in the ML module system (Milner et al., 1997; Harper and Lillibridge, 1994; Leroy, 1994). Crary (2009) extends the concept of hereditary substitution discussed in Chapter 22 to singleton kinds.

PART IX

Classes and Methods

25 Dynamic Dispatch

It frequently arises that the values of a type are partitioned into a variety of *classes*, each classifying data with a distinct internal structure. A good example is provided by the type of points in the plane, which may be classified according to whether they are represented in Cartesian or polar form. Both are represented by a pair of real numbers, but in the Cartesian case these are the x and y coordinates of the point, whereas in the polar case these are its distance r from the origin and its angle θ with the polar axis. A classified value is said to be an *object*, or *instance*, of its class. The class determines the type of the classified data, which are called the *instance type* of the class. The classified data itself is called the *instance data* of the object.

Functions that act on classified values are sometimes called *methods*. The behavior of a method is determined by the class of its argument. The method is said to *dispatch* on the class of the argument.[1] Because it happens at run time, this is called *dynamic dispatch*. For example, the squared distance of a point from the origin is calculated differently according to whether the point is represented in Cartesian or polar form. In the former case the required distance is $x^2 + y^2$, whereas in the latter it is simply r itself. Similarly, the quadrant of a Cartesian point may be determined by examining the sign of its x and y coordinates, and the quadrant of a polar point may be calculated by taking the integral part of the angle θ divided by $\pi/2$.

Dynamic dispatch is often described in terms of a particular implementation strategy, which we call the class-based organization. In this organization each object is represented by a vector of methods specialized to the class of that object. We may equivalently use a method-based organization in which each method branches on the class of an object to determine its behavior. Regardless of the organization used, the fundamental idea is that (a) objects are classified and (b) methods dispatch on the class of an object. The class-based and method-based organizations are interchangeable and, in fact, are related by a natural duality between sum and product types. We elucidate this symmetry by focusing first on the behavior of each method on each object, which is given by a dispatch matrix. From this we derive both a class-based and a method-based organization in such a way that their equivalence is evident.

[1] More generally, we may dispatch on the class of multiple arguments simultaneously. We concentrate on single dispatch for the sake of simplicity.

25.1 The Dispatch Matrix

Because each method acts by dispatch on the class of its argument, we may envision the entire system of classes and methods as a matrix e_{dm}, called the *dispatch matrix*, whose rows are classes, whose columns are methods, and whose (c, d) entry defines the behavior of method d acting on an argument of class c, expressed as a function of the instance data of the object. Thus the dispatch matrix has a type of the form

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

where C is the set of class names, D is the set of method names, τ^c is the instance type associated with class c, and ρ_d is the result type of method d. The instance type is the same for all methods acting on a given class, and the result type is the same for all classes acted on by a given method.

As an illustrative example, let us consider the type of points in a plane classified into two classes, `cart` and `pol`, corresponding to the Cartesian and polar representations, respectively. The instance data for a Cartesian point have the type

$$\tau^{\mathtt{cart}} = \langle \mathtt{x} \hookrightarrow \mathtt{real}, \mathtt{y} \hookrightarrow \mathtt{real} \rangle,$$

and the instance data for a polar point have the type

$$\tau^{\mathtt{pol}} = \langle \mathtt{r} \hookrightarrow \mathtt{real}, \mathtt{th} \hookrightarrow \mathtt{real} \rangle.$$

Consider two methods acting on points `dist` and `quad`, which compute the squared distance of a point from the origin and the quadrant of a point, respectively. The squared distance method is given by the tuple $e_{\mathtt{dist}} = \langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}_{\mathtt{dist}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}}_{\mathtt{dist}} \rangle$ of type

$$\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{dist}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{dist}} \rangle,$$

where $\rho_{\mathtt{dist}} = \mathtt{real}$ is the result type,

$$e^{\mathtt{cart}}_{\mathtt{dist}} = \lambda\ (u : \tau^{\mathtt{cart}})\ (u \cdot \mathtt{x})^2 + (u \cdot \mathtt{y})^2$$

is the squared distance computation for a Cartesian point, and

$$e^{\mathtt{pol}}_{\mathtt{dist}} = \lambda\ (v : \tau^{\mathtt{pol}})\ (v \cdot \mathtt{r})^2$$

is the squared distance computation for a polar point. Similarly, the quadrant method is given by the tuple $e_{\mathtt{quad}} = \langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}_{\mathtt{quad}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}}_{\mathtt{quad}} \rangle$ of type

$$\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{quad}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{quad}} \rangle,$$

where $\rho_{\mathtt{quad}} = [\mathtt{I}, \mathtt{II}, \mathtt{III}, \mathtt{IV}]$ is the type of quadrant and $e^{\mathtt{cart}}_{\mathtt{quad}}$ and $e^{\mathtt{pol}}_{\mathtt{quad}}$ are expressions that compute the quadrant of a point in rectangular and polar forms, respectively.

Now let $C = \{\, \mathtt{cart}, \mathtt{pol} \,\}$ and let $D = \{\, \mathtt{dist}, \mathtt{quad} \,\}$, and define the dispatch matrix e_{dm} to be the value of type

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

such that, for each class c and method d,

$$e_{\mathsf{dm}} \cdot c \cdot d \mapsto^* e_d^c.$$

That is, the entry in the dispatch matrix e_{dm} for class c and method d defines the behavior of that method acting on an object of that class.

Dynamic dispatch is an abstraction given by the following components:

- A type `obj` of *objects*, which are classified by the classes on which the methods act.
- An operation `new[`c`](`e`)` of type `obj` that creates an object of the class c with instance data given by the expression e of type τ^c.
- An operation $e \Leftarrow d$ of type ρ_d that invokes method d on the object given by the expression e of type `obj`.

These operations are required to satisfy the defining characteristic of dynamic dispatch,

$$(\texttt{new[}c\texttt{]}(e)) \Leftarrow d \mapsto^* e_d^c(e),$$

which states that invoking method d on an object of class c with instance data e amounts to applying e_d^c, the code in the dispatch matrix for class c and method d, to the instance data e.

There are two main ways to implement this abstraction. One formulation, called the *class-based* organization, defines objects as tuples of methods and creates objects by specializing the methods to the given instance data. Another formulation, called the *method-based* organization, creates objects by tagging the instance data with the class and defines methods by dispatch on the class of the object. These two organizations are isomorphic to one another and hence may be interchanged at will. Nevertheless, many languages favor one representation over the other, asymmetrizing an inherently symmetric situation.

25.2 Class-Based Organization

The class-based organization starts with the observation that the dispatch matrix may be reorganized to "factor out" the instance data for each method acting on that class to obtain the *class vector* e_{cv} of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \to (\prod_{d \in D} \rho_d)).$$

Each row of the class vector consists of a *constructor* that determines the result of each of the methods when acting on given instance data.

An object has the type $\rho = \prod_{d \in D} \rho_d$ consisting of the product over the methods of the result types of the methods. For example, in the case of points in the plane, the type ρ is the product type

$$\langle \texttt{dist} \hookrightarrow \rho_{\texttt{dist}}, \texttt{quad} \hookrightarrow \rho_{\texttt{quad}} \rangle.$$

Each component specifies the result of each of the methods acting on that object.

The message send operation $e \Leftarrow d$ is just the projection $e \cdot d$. So, in the case of points in the plane, $e \Leftarrow \mathtt{dist}$ is the projection $e \cdot \mathtt{dist}$ and similarly $e \Leftarrow \mathtt{quad}$ is the projection $e \cdot \mathtt{quad}$.

The class-based organization consolidates the implementation of each class into a class vector e_{cv}, a tuple of type τ_{cv} consisting of the *constructor* of type $\tau^c \to \rho$ for each class $c \in C$. The class vector is defined by $e_{\mathsf{cv}} = \langle e^c \rangle_{c \in C}$, where for each $c \in C$ the expression e^c is

$$\lambda\,(u : \tau^c)\,\langle e_{\mathsf{dm}} \cdot c \cdot d(u) \rangle_{d \in D}.$$

For example, the constructor for the class `cart` is the function $e^{\mathtt{cart}}$ given by the expression

$$\lambda\,(u : \tau^{\mathtt{cart}})\,\langle \mathtt{dist} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{dist}(u), \mathtt{quad} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{quad}(u) \rangle.$$

Similarly, the constructor for the class `pol` is the function $e^{\mathtt{pol}}$ given by the expression

$$\lambda\,(u : \tau^{\mathtt{pol}})\,\langle \mathtt{dist} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{dist}(u), \mathtt{quad} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{quad}(u) \rangle.$$

The class vector e_{cv} in this case is the tuple $\langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}} \rangle$ of type $\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho \rangle$.

We obtain an object of a class by applying the constructor for that class to the instance data:

$$\mathtt{new}[c]\,(e) \triangleq e_{\mathsf{cv}} \cdot c(e).$$

For example, a Cartesian point is obtained by writing $\mathtt{new}[\mathtt{cart}]\,(\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{cart}\,(\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle).$$

Similarly, we obtain a polar point by writing $\mathtt{new}[\mathtt{pol}]\,(\mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{pol}\,(\langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0 \rangle).$$

It is easy to check for this organization of points that for each class c and method d, we may derive

$$\begin{aligned} (\mathtt{new}[c]\,(e)) \Leftarrow d &\mapsto^* (e_{\mathsf{cv}} \cdot c(e)) \cdot d \\ &\mapsto^* e_{\mathsf{dm}} \cdot c \cdot d(e). \end{aligned}$$

That is, the message send evokes the behavior of the given method on the instance data of the given object.

25.3 Method-Based Organization

The method-based organization starts with the *transpose* of the dispatch matrix, which has the type

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

By observing that each row of the transposed dispatch matrix determines a method, we obtain the *method vector* e_{mv} of type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d\in D}(\sum_{c\in C}\tau^c) \to \rho_d.$$

Each entry of the method vector consists of a *dispatcher* that determines the result as a function of the instance data associated with a given object.

An object is a value of type $\tau = \sum_{c\in C}\tau^c$, the sum over the classes of the instance types. For example, the type of points in the plane is the sum type

$$[\mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}}].$$

Each point is labeled with its class, specifying its representation as having either Cartesian or polar form.

An object of a class c is just the instance data labeled with its class to form an element of the object type:

$$\mathtt{new}[c](e) \triangleq c \cdot e.$$

For example, a Cartesian point with coordinates x_0 and y_0 is given by the expression

$$\mathtt{new}[\mathtt{cart}](\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0\rangle) \triangleq \mathtt{cart} \cdot \langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0\rangle.$$

Similarly, a polar point with distance r_0 and angle θ_0 is given by the expression

$$\mathtt{new}[\mathtt{pol}](\langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0\rangle) \triangleq \mathtt{pol} \cdot \langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0\rangle.$$

The method-based organization consolidates the implementation of each method into the method vector e_{mv} of type τ_{mv}, defined by $\langle e_d\rangle_{d\in D}$, where for each $d \in D$ the expression $e_d : \tau \to \rho_d$ is

$$\lambda\,(\mathit{this}:\tau)\ \mathtt{case}\ \mathit{this}\ \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d(u)\}_{c\in C}.$$

Each entry in the method vector may be thought of as a *dispatch function* that determines the action of that method on each class of object.

In the case of points in the plane, the method vector has the product type

$$\langle \mathtt{dist} \hookrightarrow \tau \to \rho_{\mathtt{dist}}, \mathtt{quad} \hookrightarrow \tau \to \rho_{\mathtt{quad}}\rangle.$$

The dispatch function for the `dist` method has the form

$$\lambda\,(\mathit{this}:\tau)\ \mathtt{case}\ \mathit{this}\ \{\mathtt{cart} \cdot u \Rightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{dist}(u) \mid \mathtt{pol} \cdot v \Rightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{dist}(v)\},$$

and the dispatch function for the quad method has the similar form

$$\lambda\,(\mathit{this}:\tau)\ \mathtt{case}\ \mathit{this}\,\{\mathtt{cart}\cdot u \Rightarrow e_{\mathsf{dm}}\cdot\mathtt{cart}\cdot\mathtt{quad}(u)\,|\,\mathtt{pol}\cdot v \Rightarrow e_{\mathsf{dm}}\cdot\mathtt{pol}\cdot\mathtt{quad}(v)\}.$$

The *message send* operation $e \Leftarrow d$ applies the dispatch function for method d to the object e:

$$e \Leftarrow d \triangleq e_{\mathsf{mv}}\cdot d(e).$$

Thus we have, for each class c and method d,

$$\begin{aligned}(\mathtt{new}[c]\,(e)) \Leftarrow d &\mapsto^* e_{\mathsf{mv}}\cdot d(c\cdot e)\\ &\mapsto^* e_{\mathsf{dm}}\cdot c\cdot d(e).\end{aligned}$$

The result is, of course, the same as for the class-based organization.

25.4 Self-Reference

It is often useful to allow methods to create new objects or to send messages to objects. This is not possible using the simple dispatch matrix described in Section 25.1, for the simple reason that there is no provision for self-reference within its entries. This deficiency may be remedied by changing the type of the entries of the dispatch matrix to account for sending messages and creating objects, as follows:

$$\prod_{c\in C}\prod_{d\in D}\forall(t\,.\,\tau_{\mathsf{cv}}\to\tau_{\mathsf{mv}}\to\tau^c\to\rho_d).$$

The type variable t is an abstract type representing the object type. The types τ_{cv} and τ_{mv} the type of the class and method vectors, respectively, defined in terms of the abstract type t. They are defined by the equations

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

and

$$\tau_{\mathsf{mv}} \triangleq \prod_{d\in D}(t\to\rho_d).$$

The component of the class vector corresponding to a class c is a constructor that builds a value of the abstract object type t from the instance data for c. The component of the method vector corresponding to a method d is a dispatcher that yields a result of type ρ_d when applied to a value of the abstract object type t.

In accordance with the revised type of the dispatch matrix, the behavior associated with class c and method d has the form

$$\Lambda(t\,.\,\lambda\,(cv:\tau_{\mathsf{cv}})\ \lambda\,(mv:\tau_{\mathsf{mv}})\ \lambda\,(u:\tau^c)\ e_d^c).$$

The arguments cv and mv are used to create new objects and to send messages to objects. Within the expression e_d^c an object of class c' with instance data e' is created by writing

$cv \cdot c'(e')$, which selects the appropriate constructor from the class vector cv and applies it to the given instance data. The class c' may well be the class c itself; this is one form of self-reference within e_d^c. Similarly, within e a method d' is invoked on e' by writing $mv \cdot d'(e')$. The method d' may well be the method d itself; this is another aspect of self-reference within e_d^c.

To account for self-reference, the method vector will be defined to have the self-referential type $\mathtt{self}([\tau/t]\tau_{\mathsf{mv}})$ in which the object type τ is, as before, the sum of the instance types of the classes, $\sum_{c\in C} \tau^c$. The method vector is defined by the following expression:

$$\mathtt{self}\ mv\ \mathtt{is}\ \langle d \hookrightarrow \lambda\ (this\mathtt{:}\tau)\ \mathtt{case}\ this\ \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d\,[\tau]\,(e'_{\mathsf{cv}})\,(e'_{\mathsf{mv}})\,(u)\}_{c\in C}\rangle_{d\in D},$$

where

$$e'_{\mathsf{cv}} \triangleq \langle c \hookrightarrow \lambda\ (u\mathtt{:}\tau^c)\ c \cdot u\rangle_{c\in C}$$

and

$$e'_{\mathsf{mv}} \triangleq \mathtt{unroll}(mv).$$

Object creation is defined by the equation

$$\mathtt{new}[c]\,(e) \triangleq c \cdot e$$

and message send is defined by the equation

$$e \Leftarrow d \triangleq \mathtt{unroll}(e_{\mathsf{mv}}) \cdot d\,(e).$$

To account for self-reference in the class-based organization, the class vector will be defined to have the type $\mathtt{self}([\rho/t]\tau_{\mathsf{cv}})$ in which the object type ρ is, as before, the product of the result types of the methods, $\prod_{d\in D} \rho_d$. The class vector is defined by the following expression:

$$\mathtt{self}\ cv\ \mathtt{is}\ \langle c \hookrightarrow \lambda\ (u\mathtt{:}\tau^c)\ \langle d \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d\,[\rho]\,(e''_{\mathsf{cv}})\,(e''_{\mathsf{mv}})\,(u)\rangle_{d\in D}\rangle_{c\in C},$$

where

$$e''_{\mathsf{cv}} \triangleq \mathtt{unroll}(cv)$$

and

$$e''_{\mathsf{mv}} \triangleq \langle d \hookrightarrow \lambda\ (this\mathtt{:}\rho)\ this \cdot d\rangle_{d\in D}.$$

Object creation is defined by the equation

$$\mathtt{new}[c]\,(e) \triangleq \mathtt{unroll}(e_{\mathsf{cv}}) \cdot c\,(e)$$

and message send is defined by the equation

$$e \Leftarrow d \triangleq e \cdot d.$$

The symmetries between the two organizations are striking. They are a reflection of the fundamental symmetries between sum and product types.

25.5 Notes

The term "object-oriented" means many things to many people, but certainly dynamic dispatch, the association of "methods" with "classes," is one of its central concepts. According to the present development, these characteristic features emerge as instances of the more general concepts of sum, product, and function types, which are useful, alone and in combination, in a wide variety of circumstances. A bias toward either a class- or a method-based organization of functions defined on sums seems misplaced in view of the inherent symmetries of the situation. Either formulation may be readily combined with recursive types and self-reference as described in Chapter 16 to account for methods that return objects as results.

The literature on object-oriented programming, of which dynamic dispatch is one aspect, is extensive. Abadi and Cardelli (1996) and Pierce (2002) provide a thorough account of the foundations of the subject.

26 Inheritance

In this chapter we build on Chapter 25 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 each of the classes. The definitions of these behaviors may 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 may 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 *submethod* of a given *supermethod*. (It is also possible to admit multiple superclasses or multiple supermethods, but we confine our attention to *single*, rather than *multiple*, inheritance.) For simplicity we restrict attention to the simple, non-self-referential case in the following development.

26.1 Class and Method Extension

We begin by considering the extension of 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 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 $\tau^{c^*} \to \rho_d$.

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

This 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 manner 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 ensures that the inherited behavior may 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 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 submethod 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 is sensible only 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 ensures that the result of the old behavior is sufficient for the new behavior.

We now consider how inheritance relates to the method- and class-based organizations of dynamic dispatch considered in Chapter 25.

26.2 Class-Based Inheritance

Recall that the class-based organization given in Chapter 25 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 26.1. 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 as $(\,)^\dagger$, between τ^*_{cv} and the type

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

which may 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 is just an extension of 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. In short, 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 26.1. 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 as $(\,)^\ddagger$, between ρ^* and the type $\rho \times \rho_{d^*}$, where ρ is the old object type. By use of this, the new class vector e^*_{cv} may be 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 redefined 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. This means that any objects with the new method may be used in situations expecting an object without the new method, as might be expected. To avoid the redefinition of old classes when a new method is introduced, we may restrict inheritance so that new methods are added only to new subclasses. Subclasses may then have more methods than superclasses, and objects of the subclass may be provided when an object of the superclass is required.

26.3 Method-Based Inheritance

The situation with the method-based organization is dual to that of the class-based organization. Recall that the method-based organization given in Chapter 25 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 26.1. 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 as $(\,)^\ddagger$, between τ^*_{mv} and the type

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

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

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

The old method vector is reused intact, extended with an additional 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 additional method is ignored.

Finally, let us consider the effect of adding a new class c^*, as described in Section 26.1. 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 supertype of the old object type τ. There is an isomorphism, written as $(\,)^\dagger$, between τ^* and the sum type $\tau + \tau^{c^*}$, which we may use to define the new method vector e^*_{mv} as follows:

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

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

26.4 Notes

Advocates of object-oriented programming differ on the importance of inheritance. Philosophers tend to stress inheritance, but practitioners commonly avoid it or reduce it to vestigial

form. The most common restrictions amount to a reformulation of some of the modularity mechanisms that are discussed in Chapters 45 and 46.

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 the methodological requirement that subclasses respect the behavior of the superclass whenever inheritance is used.

PART X

Exceptions and Continuations

27 Control Stacks

The technique of structural dynamics is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of "search rules" requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record "where we are" in the expression so that we may resume from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a *control stack*, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit, the transition rules avoid the need for any premises—every rule is an axiom. This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it. This chapter introduces an abstract machine $\mathcal{K}\{\mathtt{nat}{\rightharpoonup}\}$ for the language $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. The purpose of this machine is to make control flow explicit by introducing a control stack that maintains a record of the pending subcomputations of a computation. We then prove the equivalence of $\mathcal{K}\{\mathtt{nat}{\rightharpoonup}\}$ with the structural dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

27.1 Machine Definition

A state s of $\mathcal{K}\{\mathtt{nat}{\rightharpoonup}\}$ consists of a *control stack* k and a closed expression, e. States may take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression e relative to a control stack k
2. A *return* state of the form $k \triangleleft e$, where e val, corresponds to the evaluation of a stack k relative to a closed value e.

As an aid to memory, note that the separator "points to" the focal entity of the state, the expression in an evaluation state and the stack in a return state.

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is to be returned. Formally, a control stack is a list of *frames*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{27.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k\,;f\ \mathsf{stack}}\,. \tag{27.1b}$$

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{27.2a}$$

$$\frac{}{\mathtt{ifz}(-;e_1;x\,.\,e_2)\ \mathsf{frame}} \tag{27.2b}$$

$$\frac{}{\mathtt{ap}(-;e_2)\ \mathsf{frame}}\,. \tag{27.2c}$$

The frames correspond to search rules in the dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. Thus, instead of relying on the structure of the transition derivation to maintain a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgment between states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers:

$$\frac{}{k \triangleright \mathtt{z} \mapsto k \triangleleft \mathtt{z}} \tag{27.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \mapsto k;\mathtt{s}(-) \triangleright e} \tag{27.3b}$$

$$\frac{}{k;\mathtt{s}(-) \triangleleft e \mapsto k \triangleleft \mathtt{s}(e)}\,. \tag{27.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor and evaluate e; when that returns with e', we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}(e;e_1;x\,.\,e_2) \mapsto k;\mathtt{ifz}(-;e_1;x\,.\,e_2) \triangleright e} \tag{27.4a}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x\,.\,e_2) \triangleleft \mathtt{z} \mapsto k \triangleright e_1} \tag{27.4b}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x\,.\,e_2) \triangleleft \mathtt{s}(e) \mapsto k \triangleright [e/x]e_2}\,. \tag{27.4c}$$

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined, we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions and recursion:

$$\frac{}{k \triangleright \mathtt{lam}[\tau](x\,.\,e) \mapsto k \triangleleft \mathtt{lam}[\tau](x\,.\,e)} \tag{27.5a}$$

$$\frac{}{k \triangleright \mathtt{ap}(e_1;e_2) \mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1} \tag{27.5b}$$

$$\frac{}{k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau](x\,.\,e) \mapsto k \triangleright [e_2/x]e} \tag{27.5c}$$

$$\frac{}{k \triangleright \mathtt{fix}[\tau](x\,.\,e) \mapsto k \triangleright [\mathtt{fix}[\tau](x\,.\,e)/x]e}\,. \tag{27.5d}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated. Note that evaluation of general recursion requires no stack space! (But see Chapter 37 for more on the evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \tag{27.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}}\,. \tag{27.6b}$$

27.2 Safety

To define and prove safety for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ requires that we introduce a new typing judgment $k : \tau$, which states that the stack k expects a value of type τ. This judgment is inductively defined by the following rules:

$$\frac{}{\epsilon : \tau} \tag{27.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k ; f : \tau}\,. \tag{27.7b}$$

This definition makes use of an auxiliary judgment $f : \tau \Rightarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ':

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{27.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-; e_1; x\,.\,e_2) : \mathtt{nat} \Rightarrow \tau} \tag{27.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-; e_2) : \mathtt{arr}(\tau_2; \tau) \Rightarrow \tau}\,. \tag{27.8c}$$

The states of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are well-formed if their stack and expression components match:

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{27.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}}\,. \tag{27.9b}$$

The proof of safety of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ is left to the reader.

Theorem 27.1 (Safety)**.**

1. *If s ok and $s \mapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

27.3 Correctness of the Control Machine

If we evaluate a given expression, e, using $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, and vice versa?

The answer to this question breaks down into two conditions relating $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ to $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$:

Completeness. If $e \mapsto^* e'$, where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$.
Soundness. If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness, a plausible first step is to consider a proof by induction on the definition of multistep transition, which reduces the theorem to the following two lemmas:

1. If e val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e$.
2. If $e \mapsto e'$, then, for every v val, if $\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$.

The first can be proved easily by induction on the structure of e. The second requires an inductive analysis of the derivation of $e \mapsto e'$, giving rise to two complications that must be accounted for in the proof. The first complication is that we cannot restrict attention to the empty stack, for if e is, say, $\texttt{ap}(e_1; e_2)$, then the first step of the machine is

$$\epsilon \triangleright \texttt{ap}(e_1; e_2) \mapsto \epsilon; \texttt{ap}(-; e_2) \triangleright e_1,$$

and so we must consider evaluation of e_1 on a nonempty stack.

A generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \texttt{ap}(e_1; e_2)$, $e' = \texttt{ap}(e_1'; e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \texttt{ap}(e_1'; e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \texttt{ap}(e_1; e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \texttt{ap}(e_1'; e_2) \mapsto k; \texttt{ap}(-; e_2) \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a v_1 such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

This means that we must consider the ultimate value of each subexpression of an expression in order to complete the proof. This information is provided by the evaluation dynamics described in Chapter 7, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 27.2. *If $e \Downarrow v$, then for every k* stack, *$k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analog of Theorem 7.2 for $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, which states that $e \Downarrow v$ iff $e \mapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ machine state as encoding an expression and show that $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ transitions under this encoding.

Specifically, we define a 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 $s \mapsto^* s'$, where s' final, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then e' val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \leftarrow\!\!\shortmid e$ and s final, then e val.
2. If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, $s' \leftarrow\!\!\shortmid e'$, and $e' \mapsto^* v$, where v val, then $e \mapsto^* v$.

The first is quite simple; we need only observe that the unraveling of a final state is a value. For the second, it is enough to show the following lemma.

Lemma 27.3. *If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then $e \mapsto^* e'$.*

Corollary 27.4. *$e \mapsto^* \overline{n}$ iff $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft \overline{n}$.*

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

27.3.1 Completeness

Proof of Lemma 27.2 The proof is by induction on an evaluation dynamics for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v}. \tag{27.10}$$

For an arbitrary control stack k, we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1;e_2) &\mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1 \\ &\mapsto^* k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

27.3.2 Soundness

The judgment $s \mapsto e'$, 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:

$$\frac{k \bowtie e = e'}{k \triangleright e \mapsto e'} \tag{27.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \mapsto e'}\,. \tag{27.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{27.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k\,;\mathtt{s}(-) \bowtie e = e'} \tag{27.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1; e_2; x\,.\,e_3) = e'}{k\,;\mathtt{ifz}(-; e_2; x\,.\,e_3) \bowtie e_1 = e'} \tag{27.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{k\,;\mathtt{ap}(-; e_2) \bowtie e_1 = e}\,. \tag{27.12d}$$

These judgments both define total functions.

Lemma 27.5. *The judgment $s \mapsto e$ has mode $(\forall, \exists!)$, and the judgment $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique e' such that $k \bowtie e = e'$.

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

Lemma 27.6. *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

Proof The proof is by rule induction on the transition $e \mapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, k.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (27.12) that $k\,;\mathtt{ap}(-; e_2) \bowtie e_1 = d$ and $k\,;\mathtt{ap}(-; e_2) \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x\,.\,e); e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k'\,;\mathtt{ap}(-; c_2)$. It follows from

Rules (27.12) that $k' \bowtie \mathtt{ap}(e; c_2) = d$ and $k' \bowtie \mathtt{ap}(e'; c_2) = d'$. But by the structural dynamics $\mathtt{ap}(e; c_2) \mapsto \mathtt{ap}(e'; c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 27.3.

Proof of Lemma 27.3 The proof is by case analysis on the transitions of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$. In each case, after unraveling, the transition will correspond to zero or one transition of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = k;\mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $k;\mathtt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k;\mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k;\mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}[\tau](x.e_1); e_2) = e'$. The result follows from Lemma 27.6. □

27.4 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 may be seen as a linearization of a structural operational semantics (Plotkin, 1981). An advantage of a machine model is that the explicit treatment of control is required for languages that allow the control state to be explicitly manipulated (see Chapter 29 for a prime example). A disadvantage is that we are required to make explicit the control state of the computation, rather than leave it implicit as in structural operational semantics. Which is better depends wholly on the situation at hand, though historically there has been greater emphasis on abstract machines than on structural semantics.

28 Exceptions

Exceptions effect a nonlocal transfer of control from the point at which the exception is *raised* to an enclosing handler for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, 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 direct and immediate, rather than indirect via a series of explicit checks.

28.1 Failures

A *failure* is a control mechanism that permits a computation to refuse to return a value to the point of its evaluation. A failure is detected by *catching* it, which means to divert evaluation to a *handler* that turns the failure into a success (unless the handler itself fails).

The following grammar defines the syntax of failures:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Exp	e	::=	`fail`	`fail`	failure
			`catch`$(e_1;e_2)$	`catch` e_1 `ow` e_2	handler

The expression `fail` aborts the current evaluation, and the expression `catch`$(e_1;e_2)$ handles any failure in e_1 by evaluating e_2 instead.

The statics of failures is straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{28.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1;e_2) : \tau} \tag{28.1b}$$

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 failures may be given using *stack unwinding*. Evaluation of a `catch` installs a handler on the control stack. Evaluation of a `fail` unwinds the control stack by popping frames until it reaches the nearest enclosing handler, to which control is passed.

The handler is evaluated in the context of the surrounding control stack, so that failures within it propagate further up the stack.

Stack unwinding can be defined directly using structural dynamics, but we prefer to make use of the stack machine defined in Chapter 27. In addition to states of the form $k \triangleright e$, which evaluates the expression e on the stack k, and $k \triangleleft e$, which passes the value e to the stack k, we make use of an additional form of state, $k \blacktriangleleft$, which passes a failure up the stack to the nearest enclosing handler.

The set of frames defined in Chapter 27 is extended with the additonal form $\mathtt{catch}(-;e_2)$. The transition rules given in Chapter 27 are extended with the following additional rules:

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \tag{28.2a}$$

$$\frac{}{k \triangleright \mathtt{catch}(e_1;e_2) \mapsto k;\mathtt{catch}(-;e_2) \triangleright e_1} \tag{28.2b}$$

$$\frac{}{k;\mathtt{catch}(-;e_2) \triangleleft v \mapsto k \triangleleft v} \tag{28.2c}$$

$$\frac{}{k;\mathtt{catch}(-;e_2) \blacktriangleleft \mapsto k \triangleright e_2} \tag{28.2d}$$

$$\frac{}{k;f \blacktriangleleft \mapsto k \blacktriangleleft}\,. \tag{28.2e}$$

As a notational convenience, we require that Rule (28.2e) apply only if none of the preceding rules applies. Evaluating $\mathtt{fail}$ propagates a failure up the stack. The act of raising an exception may raise an exception. Evaluating $\mathtt{catch}(e_1;e_2)$ consists of pushing the handler onto the control stack and evaluating e_1. If a value is propagated to the handler, the handler is removed and the value continues to propagate upward. If a failure is propagated to the handler, the stored expression is evaluated with the handler removed from the control stack. All other frames propagate failures.

The definition of initial state remains the same as for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{28.3a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}}\,. \tag{28.3b}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is easy to extend the definition of stack typing given in Chapter 27 to account for the new forms of frame, and then to prove safety in the usual way. However, the meaning of progress must be weakened to take account of failure: A well-typed expression is a value, or may take a step, or may signal failure.

Theorem 28.1 (Safety)**.**

1. *If s ok and $s \mapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

28.2 Exceptions

Failures are simplistic in that they do not distinguish different causes and hence do not permit handlers to react differently to different circumstances. An *exception* is a generalization of a failure that associates a value with the failure. This value is passed to the handler, allowing it to discriminate between various forms of failures and to pass data appropriate to that form of failure. The type of values associated with exceptions is discussed in Section 28.3. For now, we simply assume that there is some type τ_{exn} of values associated with a failure.

The syntax of exceptions is given by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Exp	e	$::=$	$\texttt{raise}[\tau](e)$	$\texttt{raise}(e)$	exception
			$\texttt{handle}(e_1; x\,.\,e_2)$	$\texttt{handle}\,e_1\,\texttt{ow}\,x \Rightarrow e_2$	handler

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\texttt{handle}(e_1; x\,.\,e_2)$ binds a variable x in the handler e_2 to which the associated value of the exception is bound, should an exception be raised during the execution of e_1.

The statics of exceptions extends the statics of failures to account for the type of the value carried with the exception:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau} \tag{28.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1; x\,.\,e_2) : \tau}\,. \tag{28.4b}$$

The dynamics of exceptions is a mild generalization of the dynamics of failures in which we generalize the failure state $k \blacktriangleleft$ to the exception state, $k \blacktriangleleft e$, which passes a value of type τ_{exn} along with the failure. The syntax of stack frames is extended to include $\texttt{raise}[\tau](-)$ and $\texttt{handle}(-; x\,.\,e_2)$. The dynamics of exceptions is specified by the following rules:

$$\overline{k \triangleright \texttt{raise}[\tau](e) \mapsto k;\texttt{raise}[\tau](-) \triangleright e} \tag{28.5a}$$

$$\overline{k;\texttt{raise}[\tau](-) \triangleleft e \mapsto k \blacktriangleleft e} \tag{28.5b}$$

$$\overline{k;\texttt{raise}[\tau](-) \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.5c}$$

$$\overline{k \triangleright \texttt{handle}(e_1; x\,.\,e_2) \mapsto k;\texttt{handle}(-; x\,.\,e_2) \triangleright e_1} \tag{28.5d}$$

$$\overline{k;\texttt{handle}(-; x\,.\,e_2) \triangleleft e \mapsto k \triangleleft e} \tag{28.5e}$$

$$\overline{k;\texttt{handle}(-; x\,.\,e_2) \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{28.5f}$$

$$\frac{(f \neq \texttt{handle}(-; x\,.\,e_2))}{k; f \blacktriangleleft e \mapsto k \blacktriangleleft e}\,. \tag{28.5g}$$

It is a straightforward exercise to extend the safety theorem given in Section 28.1 to exceptions.

28.3 Exception Type

The statics of exceptions is parameterized by a type τ_{exn} of exception values. There is no restriction on the choice of this type, but it must be one and the same for all exceptions in a program. For otherwise an exception handler cannot analyze the value associated with the exception without risking type safety.

But how is τ_{exn} to be chosen? A very naïve choice would be to take it to be the type `str` of strings. This allows us to associate an "explanation" with an exception. For example, we may write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. This is fine as far as it goes, but a handler for such an exception would have to interpret the string if it is to distinguish one exception from another, and this is clearly impractical and inconvenient.

Another popular choice is to take τ_{exn} to be `nat`, so that exceptional conditions are encoded as *error numbers* that describe the source of the error.[1] By dispatching on the numeric code of the exception the handler can determine how to recover from it. But the trouble is that we must establish a globally agreed-on system of numbering, which is clearly untenable and incompatible with modular decomposition and component reuse. Moreover, it is practically impossible to associate meaningful data with an exceptional condition, information that might well be useful to a handler.

The latter concern—how to associate data specific to the exceptional condition—can be addressed by taking τ_{exn} to be a sum type whose classes are the exceptional conditions. The instance type of the class determines the data associated with the exception. For example, the type τ_{exn} might be chosen to be a sum type of the form

$$[\texttt{div} \hookrightarrow \texttt{unit}, \texttt{fnf} \hookrightarrow \texttt{string}, \ldots].$$

The class `div` might represent an arithmetic fault, with no associated data, and the class `fnf` might represent a "file not found" error, with associated data being the name of the file.

Using a sum type for τ_{exn} makes it easy for the handler to discriminate on the source of the failure and to recover the associated data without fear of a type safety violation. For example, we might write

```
try e1 ow x ⇒
  match x {
    div ⟨⟩ ⇒ e_div
  | fnf s ⇒ e_fnf }
```

to handle the exceptions specified by the sum type given in the preceding paragraph.

The problem with choosing a sum type for τ_{exn} is that it imposes a *static classification* of the sources of failure in a program. There must be one, globally agreed-on type that classifies

[1] This convention is used in the Unix operating system, for example.

all possible forms of failure and specifies their associated data. Using sums in this manner impedes modular development and evolution, because all of the modules comprising a system must agree on the one, central type of exception values. A better approach is to use *dynamic classification* for exception values by choosing τ_{exn} to be an *extensible sum*, one to which new classes may be added at execution time. This allows separate program modules to introduce their own failure classification scheme without worrying about interference with one another; the initialization of the module generates new classes at run time that are guaranteed to be distinct from all other classes previously or subsequently generated. (See Chapter 34 for more on dynamic classification.)

28.4 Encapsulation of Exceptions

It is sometimes useful to distinguish expressions that can fail or raise an exception from those that cannot. An expression is called *fallible*, or *exceptional*, if it can fail or raise an exception during its evaluation, and is *infallible*, or *unexceptional*, otherwise. The concept of fallibility is intentionally permissive in that an infallible expression may be considered to be (vacuously) fallible, whereas infallibility is intended to be strict in that an infallible expression cannot fail. Consequently, if e_1 and e_2 are two infallible expressions whose values are required in a computation, we may evaluate them in either order without affecting the outcome. If, on the other hand, one or both are fallible, then the outcome of the computation is sensitive to the evaluation order (whichever fails first determines the overall result).

To formalize this distinction we distinguish two *modes* of expression, the fallible and the infallible, linked by a *modality* classifying the fallible expressions of a type:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Type	τ	::=	`fallible(`τ`)`	τ `fallible`	fallible
Fall	f	::=	`fail`	`fail`	failure
			`ok(`e`)`	`ok(`e`)`	success
			`try(`e`;`x`.`f_1`;`f_2`)`	`let fall(`x`) be `e` in `f_1` ow `f_2	handler
Infall	e	::=	x	x	variable
			`fall(`f`)`	`fall(`f`)`	fallible
			`try(`e`;`x`.`e_1`;`e_2`)`	`let fall(`x`) be `e` in `e_1` ow `e_2	handler

The type `fallible(`τ`)` is the type of encapsulated fallible expressions of type τ. Fallible expressions include failures, successes (infallible expressions thought of as vacuously fallible), and handlers that intercept failures, but which may themselves fail. Infallible expressions include variables, encapsulated fallible expressions, and handlers that intercepts failures, always yielding an infallible result.

The statics of encapsulated failures consists of two judgment forms, $\Gamma \vdash e : \tau$ for infallible expressions and $\Gamma \vdash f \sim \tau$ for fallible expressions. These judgments are defined

by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{28.6a}$$

$$\frac{\Gamma \vdash f \sim \tau}{\Gamma \vdash \mathtt{fall}(f) : \mathtt{fallible}(\tau)} \tag{28.6b}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{try}(e; x\,.\,e_1; e_2) : \tau'} \tag{28.6c}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} \sim \tau} \tag{28.6d}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ok}(e) \sim \tau} \tag{28.6e}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash f_1 \sim \tau' \quad \Gamma \vdash f_2 \sim \tau'}{\Gamma \vdash \mathtt{try}(e; x\,.\,f_1; f_2) \sim \tau'}\,. \tag{28.6f}$$

Rule (28.6c) specifies that a handler may be used to turn a fallible expression (encapsulated by e) into an infallible computation, provided that the result is infallible regardless of whether the encapsulated expression succeeds or fails.

The dynamics of encapsulated failures is readily derived, though some care must be taken with the elimination form for the modality:

$$\frac{}{\mathtt{fall}(f)\ \mathsf{val}} \tag{28.7a}$$

$$\frac{}{k \triangleright \mathtt{try}(e; x\,.\,e_1; e_2) \mapsto k; \mathtt{try}(-; x\,.\,e_1; e_2) \triangleright e} \tag{28.7b}$$

$$\frac{}{k; \mathtt{try}(-; x\,.\,e_1; e_2) \triangleleft \mathtt{fall}(f) \mapsto k; \mathtt{try}(-; x\,.\,e_1; e_2); \mathtt{fall}(-) \triangleright f} \tag{28.7c}$$

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \tag{28.7d}$$

$$\frac{}{k \triangleright \mathtt{ok}(e) \mapsto k; \mathtt{ok}(-) \triangleright e} \tag{28.7e}$$

$$\frac{}{k; \mathtt{ok}(-) \triangleleft e \mapsto k \triangleleft \mathtt{ok}(e)} \tag{28.7f}$$

$$\frac{e\ \mathsf{val}}{k; \mathtt{try}(-; x\,.\,e_1; e_2); \mathtt{fall}(-) \triangleleft \mathtt{ok}(e) \mapsto k \triangleright [e/x]e_1} \tag{28.7g}$$

$$\frac{}{k; \mathtt{try}(-; x\,.\,e_1; e_2); \mathtt{fall}(-) \blacktriangleleft \mapsto k \triangleright e_2}\,. \tag{28.7h}$$

The rules for the fallible form of handler have been omitted; they are similar to Rules (28.7b)–(28.7c) and (28.7g)–(28.7h), albeit with infallible subexpressions e_1 and e_2 replaced by fallible subexpressions f_1 and f_2.

An initial state has the form $k \triangleright e$, where e is an infallible expression and k is a stack of suitable type. Consequently, a fallible expression f can be evaluated only on a stack of the form

$$k; \mathtt{try}(-; x\,.\,e_1; e_2); \mathtt{fall}(-)$$

in which a handler for any failure that may arise from f is present. Therefore a final state has the form $\epsilon \lhd e$, where e val; no uncaught failure can arise.

28.5 Notes

Various forms of exceptions were explored in various dialects of Lisp (for example, Steele, 1990). The original formulation of ML (Gordon et al., 1979) as a metalanguage for mechanized logic made extensive use of exceptions, called "failures," to implement tactics and tacticals. These days most languages include an exception mechanism of the kind considered here.

The essential distinction between the exception *mechanism* and exception *values* is often misunderstood. Exception values are often dynamically classified (in the sense used in Chapter 34), but dynamic classification has many more uses than just exception values. Another common misconception is to link exceptions erroneously with fluid binding (Chapter 33).

29 Continuations

The semantics of many control constructs (such as exceptions and coroutines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *continuations*; they are values that can be passed and returned at will in a computation. Continuations never "expire," and it is always sensible to reinstate a continuation without compromising safety. Thus continuations support unlimited "time travel"—we can go back to a previous point in the computation and then return to some point in its future, at will.

Why are continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact, this is precisely what is necessary to implement *threads* (concurrently executing programs)—the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor.

29.1 Informal Overview

We extend $\mathcal{L}\{\rightarrow\}$ with the type `cont(`τ`)` of continuations accepting values of type τ. The introduction form for `cont(`τ`)` is `letcc[`τ`](`x`.`e`)`, which binds the *current continuation* (that is, the current control stack) to the variable x and evaluates the expression e. The corresponding elimination form is `throw[`τ`](`e_1`;`e_2`)`, which restores the value of e_1 to the control stack that is the value of e_2.

To illustrate the use of these primitives, consider the problem of multiplying the first n elements of an infinite sequence q of natural numbers, where q is represented by a function of type `nat` $\rightarrow$ `nat`. If zero occurs among the first n elements, we would like to effect an "early return" with the value zero rather than perform the remaining multiplications. This problem can be solved by use of exceptions (this is left as an exercise), but a solution is given that uses continuations in preparation for what follows.

Here is the solution in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, without shortcutting:

```
fix ms is
  λ q : nat ⇀ nat.
    λ n : nat.
      case n {
        z ⇒ s(z)
      | s(n') ⇒ (q z) × (ms (q ∘ succ) n')
      }
```

The recursive call composes q with the successor function to shift the sequence by one step.

Here is the version with shortcutting:

```
λ q : nat ⇀ nat.
  λ n : nat.
    letcc ret : nat cont in
      let ms be
        fix ms is
          λ q : nat ⇀ nat.
            λ n : nat.
              case n {
                z ⇒ s(z)
              | s(n') ⇒
                case q z {
                  z ⇒ throw z to ret
                | s(n'') ⇒ (q z) × (ms (q ∘ succ) n')
                }
              }
      in
        ms q n
```

The `letcc` binds the return point of the function to the variable `ret` for use within the main loop of the computation. If zero is encountered, control is thrown to `ret`, effecting an early return with the value zero.

Let us look at another example: Given a continuation k of type τ `cont` and a function f of type $\tau' \to \tau$, return a continuation k' of type τ' `cont` with the following behavior: throwing a value v' of type τ' to k' throws the value $f(v')$ to k. This is called *composition of a function with a continuation*. We wish to fill in the following template:

```
fun compose(f:τ' → τ,k:τ cont):τ' cont = ....
```

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. This is the continuation that, when given a value v',

applies f to it and throws the result to k. We can seize this continuation by using `letcc`, writing

```
throw f(letcc x:τ′ cont in ...) to k
```

At the point of the ellipsis, the variable x is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting the preceding evaluation! We do not want to actually throw a value to this continuation (yet); instead we wish to abort it and return it as the result. Here is the final code:

```
fun compose (f:τ′ → τ, k:τ cont):τ′ cont =
  letcc ret:τ′ cont cont in
    throw (f (letcc r in throw r to ret)) to k
```

The type of `ret` is that of a continuation-expecting continuation.

29.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\rightarrow\}$ expressions with these additional forms:

Sort			Abstract Form	Concrete Form	Description
Type	τ	::=	`cont(`τ`)`	τ `cont`	continuation
Expr	e	::=	`letcc[`τ`](`$x.e$`)`	`letcc` x `in` e	mark
			`throw[`τ`](`$e_1;e_2$`)`	`throw` e_1 `to` e_2	goto
			`cont(`k`)`	`cont(`k`)`	continuation

The expression `cont(`k`)` is a reified control stack, which arises during evaluation.

The statics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1;e_2) : \tau'}. \tag{29.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The statics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)}. \tag{29.2}$$

A continuation value `cont(`k`)` has type `cont(`τ`)` exactly if it is a stack accepting values of type τ.

To define the dynamics we extend $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{}{\mathtt{throw}[\tau]\,(-;e_2)\ \mathsf{frame}} \tag{29.3a}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{throw}[\tau]\,(e_1;-)\ \mathsf{frame}}\,. \tag{29.3b}$$

Every reified control stack is a value:

$$\frac{k\ \mathsf{stack}}{\mathtt{cont}(k)\ \mathsf{val}}\,. \tag{29.4}$$

The transition rules for the continuation constructs are as follows:

$$\frac{}{k \triangleright \mathtt{letcc}[\tau]\,(x\,.\,e) \mapsto k \triangleright [\mathtt{cont}(k)/x]e} \tag{29.5a}$$

$$\frac{}{k \triangleright \mathtt{throw}[\tau]\,(e_1;e_2) \mapsto k\,;\mathtt{throw}[\tau]\,(-;e_2) \triangleright e_1} \tag{29.5b}$$

$$\frac{e_1\ \mathsf{val}}{k\,;\mathtt{throw}[\tau]\,(-;e_2) \triangleleft e_1 \mapsto k\,;\mathtt{throw}[\tau]\,(e_1;-) \triangleright e_2} \tag{29.5c}$$

$$\frac{}{k\,;\mathtt{throw}[\tau]\,(v;-) \triangleleft \mathtt{cont}(k') \mapsto k' \triangleleft v}\,. \tag{29.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of $\mathcal{L}\{\rightarrow\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ given in Chapter 27.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau']\,(-;e_2) : \tau \Rightarrow \tau'} \tag{29.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\mathtt{throw}[\tau']\,(e_1;-) : \mathtt{cont}(\tau) \Rightarrow \tau'}\,. \tag{29.6b}$$

The rest of the definitions remain as in Chapter 27.

Lemma 29.1 (Canonical Forms)**.** *If $e : \mathtt{cont}(\tau)$ and e val, then $e = \mathtt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 29.2 (Safety)**.**

1. *If s ok and $s \mapsto s'$, then s' ok.*
2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

29.3 Coroutines

The distinction between a routine and a subroutine is the distinction between a manager and a worker. The routine calls on the subroutine to accomplish a piece of work, and the subroutine returns to the routine when its work is done. The relationship is asymmetric in that there is a clear distinction between the *caller*, the main routine, and the *callee*, the subroutine. Often it is useful to consider a symmetric situation in which two routines each call the other to help accomplish a task. Such pairs of routines are called *coroutines*; their relationship to one another is symmetric rather than hierarchical.

The key to implementing a subroutine is for the caller to pass to the callee a continuation representing the return point of the subroutine call. When the subroutine is finished, it calls the continuation passed to it by the calling routine. Because the subroutine is finished at that point, there is no need for the callee to pass a continuation back to the caller. The key to implementing coroutines is to have each routine treat the other as a subroutine of itself. In particular, whenever a coroutine cedes control to its caller, it provides a continuation that the caller may use to cede control back to the callee, in the process providing a continuation for itself. (This raises an interesting question of how the whole process gets started. We will return to this shortly.)

To see how a pair of coroutines is implemented, let us consider the type of each routine in the pair. A routine is a continuation accepting two arguments, a datum to be passed to that routine when it is resumed and a continuation to be resumed when the routine has finished its task. The datum represents the state of the computation, and the continuation is a coroutine that accepts arguments of the same form. Thus, the type of a coroutine must satisfy the type isomorphism

$$\tau\ \texttt{coro} \cong (\tau \times \tau\ \texttt{coro})\ \texttt{cont}.$$

So we may take τ `coro` to be the recursive type

$$\tau\ \texttt{coro} \triangleq \mu t\,.\,(\tau \times t)\ \texttt{cont}.$$

Up to isomorphism, the type τ `coro` is the type of continuation that accepts a value of type τ, representing the state of the coroutine, and the partner coroutine, a value of the same type.

A coroutine r passes control to another coroutine r' by evaluating the expression `resume`$(\langle s, r'\rangle)$, where s is the current state of the computation. Doing so creates a new coroutine whose entry point is the return point (calling site) of the application of `resume`. Therefore the type of `resume` is

$$\tau \times \tau\ \texttt{coro} \to \tau \times \tau\ \texttt{coro}.$$

The definition of `resume` is as follows:

$$\lambda\,(\langle s, r'\rangle : \tau \times \tau\ \texttt{coro})\ \texttt{letcc}\ k\ \texttt{in}\ \texttt{throw}\ \langle s, \texttt{fold}(k)\rangle\ \texttt{to}\ \texttt{unfold}(r').$$

When applied, `resume` seizes the current continuation, and passes the state s and the seized continuation (packaged as a coroutine) to the called coroutine.

But how do we create a system of coroutines in the first place? Because the state is explicitly passed from one routine to the other, a coroutine may be defined as a state transformation function that, when activated with the current state, determines the next state of the computation. A system of coroutines is created by establishing a joint exit point to which the result of the system is thrown, and creating a pair of coroutines that iteratively transforms the state and passes control to the partner routine. If either routine wishes to terminate the computation, it does so by throwing a result value to their common exit point. Thus a coroutine may be specified by a function of type

$$(\rho, \tau)\,\texttt{rout} \triangleq \rho\,\texttt{cont} \to \tau \to \tau,$$

where ρ is the result type and τ is the state type of the system of coroutines.

To set up a system of coroutines, we define a function `run` that, given two routines, creates a function of type $\tau \to \rho$ that, when applied to the initial state, computes a result of type ρ. The computation consists of a cooperating pair of routines that share a common exit point. The definition of `run` begins as follows:

$$\lambda\,(\langle r_1, r_2\rangle)\,\lambda\,(s_0)\,\texttt{letcc}\,x_0\,\texttt{in}\,\texttt{let}\,r_1'\,\texttt{be}\,r_1(x_0)\,\texttt{in}\,\texttt{let}\,r_2'\,\texttt{be}\,r_2(x_0)\,\texttt{in}\ldots.$$

Given two routines, `run` establishes their common exit point and passes this continuation to both routines. By throwing to this continuation either routine may terminate the computation with a result of type ρ. The body of the `run` function continues as follows:

$$\texttt{rep}(r_2')(\texttt{letcc}\,k\,\texttt{in}\,\texttt{rep}(r_1')(\langle s_0, \texttt{fold}(k)\rangle)).$$

The auxiliary function `rep` creates an infinite loop that transforms the state and passes control to the other routine:

$$\lambda\,(t)\,\texttt{fix}\,l\,\texttt{is}\,\lambda\,(\langle s, r\rangle)\,l(\texttt{resume}(\langle t(s), r\rangle)).$$

The system is initialized by starting routine r_1 with the initial state and arranging that, when it cedes control to its partner, it starts routine r_2 with the resulting state. At that point the system is bootstrapped: Each routine will resume the other on each iteration of the loop.

A good example of coroutining arises whenever we wish to interleave input and output in a computation. We may achieve this using a coroutine between a *producer* routine and a *consumer* routine. The producer emits the next element of the input, if any, and passes control to the consumer with that element removed from the input. The consumer processes the next data item, and returns control to the producer, with the result of processing attached to the output. The input and output are modeled as lists of type $\tau_i\,\texttt{list}$ and $\tau_o\,\texttt{list}$, respectively, which are passed back and forth between the routines.[1] The routines exchange messages according to the following protocol. The message $\texttt{OK}(\langle i, o\rangle)$ is sent from the consumer to producer to acknowledge receipt of the previous message and to pass back the current state of the input and output channels. The message $\texttt{EMIT}(\langle v, \langle i, o\rangle\rangle)$, where v is a value of type $\tau_i\,\texttt{opt}$, is sent from the producer to the consumer to emit the next value (if any) from the input and to pass the current state of the input and output channels to the consumer.

[1] In practice the input and output states are implicit, but it is better to make them explicit for the sake of clarity.

This leads to the following implementation of the producer–consumer model. The type τ of the state maintained by the routines is the labeled sum type

$$[\mathtt{OK} \hookrightarrow \tau_i\ \mathtt{list} \times \tau_o\ \mathtt{list}, \mathtt{EMIT} \hookrightarrow \tau_i\ \mathtt{opt} \times (\tau_i\ \mathtt{list} \times \tau_o\ \mathtt{list})].$$

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer P is defined by the expression

$$\lambda\ (x_0)\ \lambda\ (msg)\ \mathtt{case}\ msg\ \{b_1 \mid b_2 \mid b_3\},$$

where the first branch b_1 is

$$\mathtt{OK} \cdot \langle \mathtt{nil}, os \rangle \Rightarrow \mathtt{EMIT} \cdot \langle \mathtt{null}, \langle \mathtt{nil}, os \rangle \rangle$$

the second branch b_2 is

$$\mathtt{OK} \cdot \langle \mathtt{cons}(i; is), os \rangle \Rightarrow \mathtt{EMIT} \cdot \langle \mathtt{just}(i), \langle is, os \rangle \rangle,$$

and the third branch b_3 is

$$\mathtt{EMIT} \cdot \_ \Rightarrow \mathtt{error}.$$

In words, if the input is exhausted, the producer emits the value `null`, along with the current channel state. Otherwise, it emits $\mathtt{just}(i)$, where i is the first remaining input, and removes that element from the passed channel state. The producer cannot see an `EMIT` message, and signals an error if it should occur.

The consumer C is defined by the expression

$$\lambda\ (x_0)\ \lambda\ (msg)\ \mathtt{case}\ msg\ \{b'_1 \mid b'_2 \mid b'_3\},$$

where the first branch b'_1 is

$$\mathtt{EMIT} \cdot \langle \mathtt{null}, \langle \_, os \rangle \rangle \Rightarrow \mathtt{throw}\ os\ \mathtt{to}\ x_0,$$

the second branch b'_2 is

$$\mathtt{EMIT} \cdot \langle \mathtt{just}(i), \langle is, os \rangle \rangle \Rightarrow \mathtt{OK} \cdot \langle is, \mathtt{cons}(f(i); os) \rangle,$$

and the third branch b'_3 is

$$\mathtt{OK} \cdot \_ \Rightarrow \mathtt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to x_0 as the ultimate value of the computation. If it is present, the function f (unspecified here) of type $\tau_i \to \tau_o$ is applied to transform the input to the output, and the result is added to the output channel. If the message `OK` is received, the consumer signals an error, as the producer never produces such a message.

The initial state s_0 has the form $\mathtt{OK} \cdot \langle is, os \rangle$, where is and os are the initial input and output channel states, respectively. The computation is created by the expression

$$\mathtt{run}(\langle P, C \rangle)(s_0),$$

which sets up the coroutines, as described earlier.

Although it is relatively easy to visualize and implement coroutines involving only two partners, it is more complex, and less useful, to consider a similar pattern of control among more than 2 participants. In such cases it is more common to structure the interaction as a collection of n routines, each of which is a coroutine of a central *scheduler*. When a routine resumes its partner, it passes control to the scheduler, which determines which routine to execute next, again as a coroutine of itself. When structured as coroutines of a scheduler, the individual routines are called *threads*. A thread *yields* control by resuming its partner, the scheduler, which then determines which thread to execute next as a coroutine of itself. This pattern of control is called *cooperative multithreading*, because it is based on explicit yields rather than on implicit yields imposed by asynchronous events such as timer interrupts.

29.4 Notes

Continuations are a ubiquitous notion in programming languages. Reynolds (1993) provides an excellent account of the multiple discoveries of continuations. The formulation given here is inspired by Felleisen and Hieb (1992), who pioneered the development of linguistic theories of control and state.

PART XI

Types and Propositions

30 Constructive Logic

Constructive logic codifies the principles of mathematical reasoning as they are actually practiced. In mathematics a proposition may be judged to be true exactly when it has a proof and may be judged to be false exactly when it has a refutation. Because there are, and always will be, unsolved problems, we cannot expect in general that a proposition is either true or false, for in most cases we have neither a proof nor a refutation of it. Constructive logic may be described as *logic as if people matter*, as distinct from classical logic, which may be described as *the logic of the mind of god*. From a constructive viewpoint the judgment "ϕ true" means that "there is a proof of ϕ."

What constitutes a proof is a social construct, an agreement among people as to what a valid argument is. The rules of logic codify a set of principles of reasoning that may be used in a valid proof. The valid forms of proof are determined by the outermost structure of the proposition whose truth is asserted. For example, a proof of a conjunction consists of a proof of each of its conjuncts, and a proof of an implication consists of a transformation of a proof of its antecedent to a proof of its consequent. When spelled out in full, the forms of proof are seen to correspond exactly to the forms of expression of a programming language. With each proposition is associated the type of its proofs; a proof is an expression of the associated type. This association between programs and proofs induces a dynamics on proofs, for they are but programs of some type. In this way proofs in constructive logic have *computational content*, which is to say that they may be interpreted as executable programs of the associated type. Conversely, programs have *mathematical content* as proofs of the proposition associated with their type.

This unification of logic and programming is called the *propositions as types* principle. It is the central organizing principle of the theory of programming languages. Propositions are identified with types, and proofs are identified with programs. A programming technique corresponds to a method of proof; a proof technique corresponds to a method of programming. Viewing types as behavioral specifications of programs, we may see propositions as problem statements whose proofs are solutions that implement the specification.

30.1 Constructive Semantics

Constructive logic is concerned with two judgments, ϕ prop, stating that ϕ expresses a proposition, and ϕ true, stating that ϕ is a true proposition. What distinguishes constructive

from nonconstructive logic is that a proposition is not conceived of as merely a truth value, but instead as a *problem statement* whose solution, if it has one, is given by a proof. A proposition is said to be *true* exactly when it has a proof, in keeping with ordinary mathematical practice. In practice, there is no other criterion of truth than the existence of a proof.

This principle has important, possibly surprising, consequences, the most important of which is that we cannot say, in general, that a proposition is either true or false. If for a proposition to be true means to have a proof of it, what does it mean for a proposition to be false? It means that we have a *refutation* of it, showing that it cannot be proved. That is, a proposition is false if we can show that the assumption that it is true (has a proof) contradicts known facts. In this sense constructive logic is a logic of *positive*, or *affirmative*, *information*—we must have explicit evidence in the form of a proof in order to affirm the truth or falsity of a proposition.

In light of this, it should be clear that not every proposition is either true or false. For if ϕ expresses an unsolved problem, such as the famous $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem, then we have neither a proof nor a refutation of it (the mere absence of a proof not being a refutation). Such a problem is *undecided*, precisely because it is unsolved. Because there will always be unsolved problems (there being infinitely many propositions, but only finitely many proofs at a given point in the evolution of our knowledge), we cannot say that every proposition is *decidable*, that is, either true or false.

Of course, some propositions are decidable and hence may be considered to be either true or false. For example, if ϕ expresses an inequality between natural numbers, then ϕ is decidable, because we can always work out, for given natural numbers m and n, whether $m \leq n$ or $m \not\leq n$—we can either prove or refute the given inequality. This argument does not extend to the real numbers. To get an idea of why not, consider the presentation of a real number by its decimal expansion. At any finite time we will have explored only a finite initial segment of the expansion, which is not enough to determine if it is, say, less than 1. For if we have determined the expansion to be $0.99\ldots9$, we cannot decide at any time, short of infinity, whether the number is 1. (This argument is not a proof, because we may wonder whether there is some other representation of real numbers that admits such a decision to be made finitely, but it turns out that this is not the case.)

The constructive attitude is simply to accept the situation as inevitable and make our peace with that. When faced with a problem we have no choice but to roll up our sleeves and try to prove it or refute it. There is no guarantee of success! Life is hard, but we muddle through somehow.

30.2 Constructive Logic

The judgments ϕ prop and ϕ true of constructive logic are rarely of interest by themselves, but rather in the context of a hypothetical judgment of the form

$$\phi_1 \text{ true}, \ldots, \phi_n \text{ true} \vdash \phi \text{ true}.$$

This judgment expresses that the proposition ϕ is true (has a proof) under the assumptions that each of $\phi_1, \ldots, \phi_n$ is also true (has a proof). Of course, when $n = 0$ this is just the same as the judgment ϕ true.

The structural properties of the hypothetical judgment, when specialized to constructive logic, define what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{30.1a}$$

$$\frac{\Gamma \vdash \phi_1 \text{ true} \quad \Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma \vdash \phi_2 \text{ true}} \tag{30.1b}$$

$$\frac{\Gamma \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{30.1c}$$

$$\frac{\Gamma, \phi_1 \text{ true}, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{30.1d}$$

$$\frac{\Gamma_1, \phi_2 \text{ true}, \phi_1 \text{ true}, \Gamma_2 \vdash \phi \text{ true}}{\Gamma_1, \phi_1 \text{ true}, \phi_2 \text{ true}, \Gamma_2 \vdash \phi \text{ true}}. \tag{30.1e}$$

The last two rules are implicit in that we regard Γ as a set of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

30.2.1 Provability

The syntax of propositional logic is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Prop	ϕ	::=	$\top$	$\top$	truth
			$\bot$	$\bot$	falsity
			$\wedge(\phi_1;\phi_2)$	$\phi_1 \wedge \phi_2$	conjunction
			$\vee(\phi_1;\phi_2)$	$\phi_1 \vee \phi_2$	disjunction
			$\supset(\phi_1;\phi_2)$	$\phi_1 \supset \phi_2$	implication

The connectives of propositional logic are given meaning by rules that determine (a) what constitutes a "direct" proof of a proposition formed from a given connective, and (b) how to exploit the existence of such a proof in an "indirect" proof of another proposition. These are called the *introduction* and *elimination* rules for the connective. The principle of conservation of proof states that these rules are inverse to one another—the elimination rule cannot extract more information (in the form of a proof) than was put into it by the introduction rule, and the introduction rules can be used to reconstruct a proof from the information extracted from them by the elimination rules.

Truth. Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it:

$$\frac{}{\Gamma \vdash \top \text{ true}} \tag{30.2a}$$

$$\textit{(no elimination rule)} \tag{30.2b}$$

Conjunction. Conjunction expresses the truth of both of its conjuncts:

$$\frac{\Gamma \vdash \phi_1 \ \mathsf{true} \quad \Gamma \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}} \tag{30.3a}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \ \mathsf{true}} \tag{30.3b}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_2 \ \mathsf{true}}\,. \tag{30.3c}$$

Implication. Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{true}} \tag{30.4a}$$

$$\frac{\Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{true} \quad \Gamma \vdash \phi_1 \ \mathsf{true}}{\Gamma \vdash \phi_2 \ \mathsf{true}} \tag{30.4b}$$

Falsehood. Falsehood expresses the trivially false (refutable) proposition:

$$\textit{(no introduction rule)} \tag{30.5a}$$

$$\frac{\Gamma \vdash \bot \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}}\,. \tag{30.5b}$$

Disjunction. Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi_1 \ \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{30.6a}$$

$$\frac{\Gamma \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{30.6b}$$

$$\frac{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true} \quad \Gamma, \phi_1 \ \mathsf{true} \vdash \phi \ \mathsf{true} \quad \Gamma, \phi_2 \ \mathsf{true} \vdash \phi \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}}\,. \tag{30.6c}$$

Negation. The negation $\neg\phi$ of a proposition ϕ may be defined as the implication $\phi \supset \bot$. This means that $\neg\phi \ \mathsf{true}$ if $\phi \ \mathsf{true} \vdash \bot \ \mathsf{true}$, which is to say that the truth of ϕ is *refutable* in that we may derive a proof of falsehood from any purported proof of ϕ. Because constructive truth is identified with the existence of a proof, the implied semantics of negation is rather strong. In particular, a problem ϕ is *open* exactly when we can neither affirm nor refute it. This is in contrast to the classical conception of truth, which assigns a fixed truth value to each proposition, so that every proposition is either true or false.

30.2.2 Proof Terms

The key to the propositions-as-types principle is to make explict the forms of proof. The basic judgment $\phi \ \mathsf{true}$, which states that ϕ has a proof, is replaced by the judgment $p : \phi$,

stating that p is a proof of ϕ. (Sometimes p is called a "proof term," but we simply call p a "proof.") The hypothetical judgment is modified correspondingly, with variables standing for the presumed, but unknown, proofs:

$$x_1 : \phi_1, \ldots, x_n : \phi_n \vdash p : \phi.$$

We again let Γ range over such hypothesis lists, subject to the restriction that no variable occurs more than once.

The syntax of proof terms is given by the following grammar:

Sort		Abstract Form	Concrete Form	Description
Prf	$p ::=$	$\top\mathtt{I}$	$\langle\rangle$	truth intro
		$\wedge\mathtt{I}(p_1; p_2)$	$\langle p_1, p_2\rangle$	conj. intro
		$\wedge\mathtt{E[l]}(p)$	$p \cdot \mathtt{l}$	conj. elim
		$\wedge\mathtt{E[r]}(p)$	$p \cdot \mathtt{r}$	conj. elim
		$\supset\mathtt{I}[\phi](x\,.\,p)$	$\lambda\,(x{:}\phi)\;p$	impl. intro
		$\bot\mathtt{E}(p)$	$\mathtt{abort}(p)$	false elim
		$\vee\mathtt{I[l]}(p)$	$\mathtt{l} \cdot p$	disj. intro
		$\vee\mathtt{I[r]}(p)$	$\mathtt{r} \cdot p$	disj. intro
		$\vee\mathtt{E}(p; x_1\,.\,p_1; x_2\,.\,p_2)$	$\mathtt{case}\; p\,\{\mathtt{l} \cdot x_1 \Rightarrow p_1 \mid \mathtt{r} \cdot x_2 \Rightarrow p_2\}$	disj. elim

The concrete syntax of proof terms is chosen to stress the correspondence between propositions and types, as discussed in Section 30.4.

The rules of constructive propositional logic may be restated using proof terms, as follows:

$$\frac{}{\Gamma \vdash \langle\rangle : \top} \tag{30.7a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2\rangle : \phi_1 \wedge \phi_2} \tag{30.7b}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash p_1 \cdot \mathtt{l} : \phi_1} \tag{30.7c}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash p_1 \cdot \mathtt{r} : \phi_2} \tag{30.7d}$$

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Gamma \vdash \lambda\,(x{:}\phi_1)\;p_2 : \phi_1 \supset \phi_2} \tag{30.7e}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2 \quad \Gamma \vdash p_1 : \phi_1}{\Gamma \vdash p(p_1) : \phi_2} \tag{30.7f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathtt{abort}(p) : \phi} \tag{30.7g}$$

$$\frac{\Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \mathtt{l} \cdot p_1 : \phi_1 \vee \phi_2} \tag{30.7h}$$

$$\frac{\Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \mathtt{r} \cdot p_2 : \phi_1 \vee \phi_2} \tag{30.7i}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \phi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \phi}{\Gamma \vdash \mathtt{case}\, p\, \{\mathtt{l} \cdot x_1 \Rightarrow p_1 \mid \mathtt{r} \cdot x_2 \Rightarrow p_2\} : \phi}. \tag{30.7j}$$

30.3 Proof Dynamics

Proof terms in constructive logic are equipped with a dynamics by *Gentzen's Principle*, which states that the eliminatory forms are to be thought of as inverse to the introductory forms. One aspect of Gentzen's Principle is the principle of *conservation of proof*, which states that the information introduced into a proof of a proposition may be extracted without loss by elimination. So, for example, we may state that conjunction elimination is postinverse to conjunction introduction by the definitional equivalences

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2 \rangle \cdot \mathtt{l} \equiv p_1 : \phi_1} \tag{30.8a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2 \rangle \cdot \mathtt{r} \equiv p_2 : \phi_2}. \tag{30.8b}$$

Another aspect of Gentzen's Principle is that principle of *reversability of proof*, which states that every proof may be reconstructed from the information that may be extracted from it by elimination. In the case of conjunction this may be stated by the definitional equivalence

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p \cdot \mathtt{l}, p \cdot \mathtt{r} \rangle \equiv p : \phi_1 \wedge \phi_2}. \tag{30.9}$$

Similar equivalences may be stated for the other connectives. For example, the conservation and reversability principles for implication are given by these rules:

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash (\lambda\ (x{:}\phi_1)\ p_2)(p_1) \equiv [p_1/x]p_2 : \phi_2} \tag{30.10a}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2}{\Gamma \vdash \lambda\ (x{:}\phi_1)\ (p(x)) \equiv p : \phi_1 \supset \phi_2}. \tag{30.10b}$$

The corresponding rules for disjunction and falsehood are given as follows:

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \psi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \psi}{\Gamma \vdash \mathtt{case}\, \mathtt{l} \cdot p\, \{\mathtt{l} \cdot x_1 \Rightarrow p_1 \mid \mathtt{r} \cdot x_2 \Rightarrow p_2\} \equiv [p/x_1]p_1 : \psi} \tag{30.11a}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \psi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \psi}{\Gamma \vdash \mathtt{case}\, \mathtt{r} \cdot p\, \{\mathtt{l} \cdot x_1 \Rightarrow p_1 \mid \mathtt{r} \cdot x_2 \Rightarrow p_2\} \equiv [p/x_2]p_2 : \psi} \tag{30.11b}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x : \phi_1 \vee \phi_2 \vdash q : \psi}{\Gamma \vdash [p/x]q \equiv \mathtt{case}\ p\,\{\mathtt{l} \cdot x_1 \Rightarrow [\mathtt{l} \cdot x_1/x]q \mid \mathtt{r} \cdot x_2 \Rightarrow [\mathtt{r} \cdot x_2/x]q\} : \psi} \tag{30.11c}$$

$$\frac{\Gamma \vdash p : \bot \quad \Gamma, x : \bot \vdash q : \psi}{\Gamma \vdash [p/x]q \equiv \mathtt{abort}(p) : \psi}\,. \tag{30.11d}$$

30.4 Propositions as Types

Reviewing the statics and dynamics of proofs in constructive logic reveals a striking similarity to the statics and dynamics of expressions of various types. For example, the introduction rule for conjunction specifies that a proof of a conjunction consists of a pair of proofs, one for each conjunct, and the elimination rule inverts this, allowing us to extract a proof of each conjunct from any proof of a conjunction. There is an obvious analogy with the static semantics of product types, whose introductory form is a pair and whose eliminatory forms are projections. Gentzen's Principle extends the analogy to the dynamics as well, so that, for example, the elimination forms for conjunction amount to projections that extract the appropriate components from an ordered pair.

The correspondence between propositions and types and between proofs and programs, is summarized by the following chart:

Prop	**Type**
$\top$	`unit`
$\bot$	`void`
$\phi_1 \wedge \phi_2$	$\tau_1 \times \tau_2$
$\phi_1 \supset \phi_2$	$\tau_1 \to \tau_2$
$\phi_1 \vee \phi_2$	$\tau_1 + \tau_2$

The correspondence between propositions and types is a cornerstone of the theory of programming languages. It exposes a deep connection between computation and deduction and serves as a framework for the analysis of language constructs and reasoning principles by relating them to one another.

30.5 Notes

The propositions-as-types principle has its origins in the semantics of intuitionistic logic developed by Brouwer, according to which the truth of a proposition is witnessed by a construction providing computable evidence for it. The forms of evidence are determined by the form of the proposition, so that, for example, evidence for an implication is a computable function transforming evidence for the hypothesis into evidence for the conclusion. An explicit formulation of this semantics was introduced by Heyting and further developed by

a number of authors, including de Bruijn, Curry, Gentzen, Girard, Howard, Kolmogorov, Martin-Löf, and Tait. The propositions-as-types correspondence is sometimes called the *Curry–Howard Isomorphism*, but this terminology neglects the crucial contributions of the others just mentioned. Moreover, the correspondence is not, in general, an isomorphism; rather, it is an expression of Brouwer's Dictum that the concept of proof is best explained by the more general concept of construction (program).

31 Classical Logic

In constructive logic a proposition is true exactly when it has a proof, a derivation of it from axioms and assumptions, and is false exactly when it has a refutation, a derivation of a contradiction from the assumption that it is true. Constructive logic is a logic of positive evidence. To affirm or deny a proposition requires a proof, either of the proposition itself or of a contradiction, under the assumption that it has a proof. We are not always in a position to affirm or deny a proposition. An open problem is one for which we have neither a proof nor a refutation—so that, constructively speaking, it is neither true nor false.

In contrast, classical logic (the one we learned in school) is a logic of perfect information in which every proposition is either true or false. We may say that classical logic corresponds to "god's view" of the world—there are no open problems; rather, all propositions are either true or false. Put another way, to assert that every proposition is either true or false is to weaken the notion of truth to encompass all that is not false, dually to the constructively (and classically) valid interpretation of falsity as all that is not true. The symmetry between truth and falsity is appealing, but there is a price to pay for this: The meanings of the logical connectives are weaker in the classical case than in the constructive.

A prime example is provided by the *law of the excluded middle*, the assertion that $\phi \vee \neg\phi$ true is valid for all propositions ϕ. Constructively, this principle is not universally valid, because it would mean that every proposition has either a proof or a refutation, which is manifestly not the case. Classically, however, the law of the excluded middle is valid, because every proposition is considered to be either false or not false (which is identified with being true in classical logic, in contrast to constructive logic). Nevertheless, classical logic is consistent with constructive logic in that constructive logic does not refute classical logic. As we have seen, constructive logic proves that the law of the excluded middle is positively not refuted (its double negation is constructively true). This shows that constructive logic is stronger (more expressive) than classical logic because it can express more distinctions (namely, between affirmation and irrefutability) and because it is consistent with classical logic (the law of the excluded middle can be added without fear of contradiction).

Proofs in constructive logic have computational content: They can be executed as programs, and their behavior is constrained by their type. Proofs in classical logic also have computational content, but in a weaker sense than in classical logic. Rather than positively affirming a proposition, a proof in classical logic is a computation that cannot be refuted. Computationally, a refutation consists of a continuation, or control stack, that takes a proof of a proposition and derives a contradiction from it. So a proof of a proposition in classical

logic is a computation that, when given a refutation of that proposition deriving a contradiction, witnesses the impossibility of refuting it. In this sense the law of the excluded middle has a proof, precisely because it is irrefutable.

31.1 Classical Logic

In constructive logic a connective is defined by giving its introduction and elimination rules. In classical logic a connective is defined by giving its truth and falsity conditions. The connective's truth rules correspond to introduction and its falsity rules to elimination. The symmetry between truth and falsity is expressed by the principle of indirect proof. To show that ϕ true it is enough to show that ϕ false entails a contradiction, and, conversely, to show that ϕ false it is enough to show that ϕ true leads to a contradiction. Although the second of these is constructively valid, the first is fundamentally classical, expressing the principle of indirect proof.

31.1.1 Provability and Refutability

There are three basic judgment forms in classical logic:

1. ϕ true, stating that the proposition ϕ is provable;
2. ϕ false, stating that the proposition ϕ is refutable;
3. #, stating that a contradiction has been derived.

These are extended to hypothetical judgments in which we admit both provability and refutability assumptions:

$$\phi_1 \text{ false}, \ldots, \phi_m \text{ false } \psi_1 \text{ true}, \ldots, \psi_n \text{ true} \vdash J.$$

The hypotheses are divided into two zones, one for falsity assumptions Δ and one for truth assumptions Γ.

The rules of classical logic are organized around the symmetry between truth and falsity, which is mediated by the contradiction judgment.

The hypothetical judgment is reflexive:

$$\frac{}{\Delta, \phi \text{ false } \Gamma \vdash \phi \text{ false}} \tag{31.1a}$$

$$\frac{}{\Delta\ \Gamma, \phi \text{ true} \vdash \phi \text{ true}}\,. \tag{31.1b}$$

The remaining rules are stated so that the structural properties of weakening, contraction, and transitivity are admissible.

A contradiction arises when a proposition is judged to be both true and false. A proposition is true if its falsity is absurd and is false if its truth is absurd:

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{false} \quad \Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \#} \tag{31.1c}$$

$$\frac{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}} \tag{31.1d}$$

$$\frac{\Delta\ \Gamma, \phi\ \mathsf{true} \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{false}}\,. \tag{31.1e}$$

Truth is trivially true, and cannot be refuted:

$$\frac{}{\Delta\ \Gamma \vdash \top\ \mathsf{true}}\,. \tag{31.1f}$$

A conjunction is true if both conjuncts are true and is false if either conjunct is false:

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}} \tag{31.1g}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{false}} \tag{31.1h}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{false}}\,. \tag{31.1i}$$

Falsity is trivially false and cannot be proved:

$$\frac{}{\Delta\ \Gamma \vdash \bot\ \mathsf{false}}\,. \tag{31.1j}$$

A disjunction is true if either disjunct is true and is false if both disjuncts are false:

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{true}} \tag{31.1k}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{true}} \tag{31.1l}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{false} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2\ \mathsf{false}}\,. \tag{31.1m}$$

Negation inverts the sense of each judgment:

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{false}}{\Delta\ \Gamma \vdash \neg\phi\ \mathsf{true}} \tag{31.1n}$$

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \neg\phi\ \mathsf{false}}\,. \tag{31.1o}$$

An implication is true if its conclusion is true whenever the assumption is true and is false if its conclusion is false yet its assumption is true:

$$\frac{\Delta\ \Gamma, \phi_1\ \mathsf{true} \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2\ \mathsf{true}} \tag{31.1p}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2\ \mathsf{false}}\,. \tag{31.1q}$$

31.1.2 Proofs and Refutations

To explain the dynamics of classical proofs, an explicit syntax for proofs and refutations is introduced first. We define three hypothetical judgments for classical logic with explicit derivations:

1. $\Delta\ \Gamma \vdash p : \phi$, stating that p is a proof of ϕ;
2. $\Delta\ \Gamma \vdash k \div \phi$, stating that k is a refutation of ϕ;
3. $\Delta\ \Gamma \vdash k\ \#\ p$, stating that k and p are contradictory.

The falsity assumptions Δ are represented by a context of the form

$$u_1 \div \phi_1, \ldots, u_m \div \phi_m,$$

where $m \geq 0$, in which the variables $u_1, \ldots, u_n$ stand for refutations. The truth assumptions Γ are represented by a context of the form

$$x_1 : \psi_1, \ldots, x_n : \psi_n,$$

where $n \geq 0$, in which the variables $x_1, \ldots, x_n$ stand for proofs.

The syntax of proofs and refutations is given by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Prf	p	::=	$\top T$	$\langle\rangle$	truth
			$\wedge T(p_1; p_2)$	$\langle p_1, p_2\rangle$	conjunction
			$\vee T\,\mathtt{[l]}(p)$	$\mathtt{l} \cdot p$	disjunction left
			$\vee T\,\mathtt{[r]}(p)$	$\mathtt{r} \cdot p$	disjunction right
			$\neg T(k)$	$\mathtt{not}(k)$	negation
			$\supset T(\phi; x\,.\,p)$	$\lambda\,(x{:}\phi)\ p$	implication
Ref	k	::=	$\bot F$	`abort`	falsehood
			$\wedge F\,\mathtt{[l]}(k)$	$\mathtt{fst}\ ;\ k$	conjunction left
			$\wedge F\,\mathtt{[r]}(k)$	$\mathtt{snd}\ ;\ k$	conjunction right
			$\vee F(k_1; k_2)$	$\mathtt{case}(k_1; k_2)$	disjunction
			$\neg F(p)$	$\mathtt{not}(p)$	negation
			$\supset F(p; k)$	$\mathtt{ap}(p)\ ;\ k$	implication

Proofs serve as evidence for truth judgments, and refutations serve as evidence for false judgments. Contradictions are witnessed by the juxtaposition of a proof and a refutation.

A contradiction arises whenever a proposition is both true and false:

$$\frac{\Delta\ \Gamma \vdash k \div \phi \quad \Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash k \# p}\,. \tag{31.2a}$$

Truth and falsity are defined symmetrically in terms of contradiction:

$$\frac{\Delta, u \div \phi\ \Gamma \vdash k \# p}{\Delta\ \Gamma \vdash \mathtt{ccr}(u \div \phi.k \# p) : \phi} \tag{31.2b}$$

$$\frac{\Delta\ \Gamma, x : \phi \vdash k \# p}{\Delta\ \Gamma \vdash \mathtt{ccp}(x : \phi.k \# p) \div \phi}\,. \tag{31.2c}$$

Reflexivity corresponds to the use of a variable hypothesis:

$$\frac{}{\Delta, u \div \phi\ \Gamma \vdash u \div \phi} \tag{31.2d}$$

$$\frac{}{\Delta\ \Gamma, x : \phi \vdash x : \phi}\,. \tag{31.2e}$$

The other structure properties are admissible.

Truth is trivially true, and cannot be refuted:

$$\frac{}{\Delta\ \Gamma \vdash \langle\rangle : \top}\,. \tag{31.2f}$$

A conjunction is true if both conjuncts are true and is false if either conjunct is false:

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \langle p_1, p_2\rangle : \phi_1 \wedge \phi_2} \tag{31.2g}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1}{\Delta\ \Gamma \vdash \mathtt{fst}\,;k_1 \div \phi_1 \wedge \phi_2} \tag{31.2h}$$

$$\frac{\Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{snd}\,;k_2 \div \phi_1 \wedge \phi_2}\,. \tag{31.2i}$$

Falsity is trivially false and cannot be proved:

$$\frac{}{\Delta\ \Gamma \vdash \mathtt{abort} \div \bot}\,. \tag{31.2j}$$

A disjunction is true if either disjunct is true and is false if both disjuncts are false:

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1}{\Delta\ \Gamma \vdash \mathtt{l} \cdot p_1 : \phi_1 \vee \phi_2} \tag{31.2k}$$

$$\frac{\Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \mathtt{r} \cdot p_2 : \phi_1 \vee \phi_2} \tag{31.2l}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{case}(k_1;k_2) \div \phi_1 \vee \phi_2}\,. \tag{31.2m}$$

Negation inverts the sense of each judgment:

$$\frac{\Delta\ \Gamma \vdash k \div \phi}{\Delta\ \Gamma \vdash \mathtt{not}(k) : \neg\phi} \tag{31.2n}$$

$$\frac{\Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash \mathtt{not}(p) \div \neg\phi}\,. \tag{31.2o}$$

An implication is true if its conclusion is true whenever the assumption is true and is false if its conclusion is false yet its assumption is true:

$$\frac{\Delta\ \Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \lambda\ (x{:}\phi_1)\ p_2 : \phi_1 \supset \phi_2} \tag{31.2p}$$

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{ap}(p_1)\ ;\ k_2 \div \phi_1 \supset \phi_2}\,. \tag{31.2q}$$

31.2 Deriving Elimination Forms

The price of achieving a symmetry between truth and falsity in classical logic is that we must very often rely on the principle of indirect proof: To show that a proposition is true, we often must derive a contradiction from the assumption of its falsity. For example, a proof of

$$(\phi \wedge (\psi \wedge \theta)) \supset (\theta \wedge \phi)$$

in classical logic has the form

$$\lambda\ (w{:}\phi \wedge (\psi \wedge \theta))\ \mathtt{ccr}(u \div \theta \wedge \phi\,.\,k\ \#\ w),$$

where k is the refutation

$$\mathtt{fst}\ ;\ \mathtt{ccp}(x : \phi\,.\,\mathtt{snd}\ ;\ \mathtt{ccp}(y : \psi \wedge \theta\,.\,\mathtt{snd}\ ;\ \mathtt{ccp}(z : \theta\,.\,u\ \#\ \langle z, x\rangle)\ \#\ y)\ \#\ w).$$

And yet in constructive logic this proposition has a direct proof that avoids the circumlocutions of proof by contradiction:

$$\lambda\ (w{:}\phi \wedge (\psi \wedge \theta))\ \langle w \cdot \mathtt{r} \cdot \mathtt{r}, w \cdot \mathtt{l}\rangle.$$

But this proof cannot be expressed (as it is) in classical logic, because classical logic lacks the elimination forms of constructive logic.

However, we may package the use of indirect proof into a slightly more palatable form by deriving the elimination rules of constructive logic. For example, the rule

$$\frac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}$$

is derivable in classical logic:

$$\dfrac{\dfrac{\dfrac{}{\Delta, \phi \ \mathsf{false}\ \Gamma \vdash \phi \ \mathsf{false}} \quad \dfrac{\Delta\ \Gamma \vdash \phi \wedge \psi \ \mathsf{true}}{\Delta, \phi \ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi \ \mathsf{true}}}{\Delta, \phi \ \mathsf{false}\ \Gamma \vdash \#}}{\Delta\ \Gamma \vdash \phi \ \mathsf{true}}.$$

The other elimination forms are derivable in a similar manner, in each case relying on indirect proof to construct a proof of the truth of a proposition from a derivation of a contradiction from the assumption of its falsity.

The derivations of the elimination forms of constructive logic are most easily exhibited by use of proof and refutation expressions, as follows:

$$\begin{aligned}
\mathtt{abort}(p) &= \mathtt{ccr}(u \div \phi\,.\mathtt{abort} \mathbin{\#} p)\\
p \cdot \mathtt{l} &= \mathtt{ccr}(u \div \phi\,.\mathtt{fst}\,;\, u \mathbin{\#} p)\\
p \cdot \mathtt{r} &= \mathtt{ccr}(u \div \psi\,.\mathtt{snd}\,;\, u \mathbin{\#} p)\\
p_1(p_2) &= \mathtt{ccr}(u \div \psi\,.\mathtt{ap}(p_2)\,;\, u \mathbin{\#} p_1)\\
\mathtt{case}\, p_1\, \{\mathtt{l} \cdot x \Rightarrow p_2 \mid \mathtt{r} \cdot y \Rightarrow p\} &= \mathtt{ccr}(u \div \gamma\,.\mathtt{case}(\mathtt{ccp}(x : \phi\,.u \mathbin{\#} p_2); \mathtt{ccp}(y : \psi\,.u \mathbin{\#} p)) \mathbin{\#} p_1).
\end{aligned}$$

It is straightforward to check that the expected elimination rules hold. For example, the rule

$$\dfrac{\Delta\ \Gamma \vdash p_1 : \phi \supset \psi \quad \Delta\ \Gamma \vdash p_2 : \phi}{\Delta\ \Gamma \vdash p_1(p_2) : \psi} \tag{31.3}$$

is derivable from the definition of $p_1(p_2)$ previously given. By suppressing proof terms, we may derive the corresponding provability rule:

$$\dfrac{\Delta\ \Gamma \vdash \phi \supset \psi \ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi \ \mathsf{true}}{\Delta\ \Gamma \vdash \psi \ \mathsf{true}}\ . \tag{31.4}$$

31.3 Proof Dynamics

The dynamics of classical logic arises from the simplification of the contradiction between a proof and a refutation of a proposition. To make this explicit we define a transition system whose states are contradictions $k \mathbin{\#} p$ consisting of a proof p and a refutation k of the same proposition. The steps of the computation consist of simplifications of the contradictory state based on the form of p and k.

The truth and falsity rules for the connectives play off one another in a pleasing manner:

$$\mathtt{fst}\,;k \mathbin{\#} \langle p_1, p_2\rangle \mapsto k \mathbin{\#} p_1 \tag{31.5a}$$

$$\mathtt{snd}\,;k \mathbin{\#} \langle p_1, p_2\rangle \mapsto k \mathbin{\#} p_2 \tag{31.5b}$$

$$\mathtt{case}(k_1;k_2) \mathbin{\#} \mathtt{l}\cdot p_1 \mapsto k_1 \mathbin{\#} p_1 \tag{31.5c}$$

$$\mathtt{case}(k_1;k_2) \mathbin{\#} \mathtt{r}\cdot p_2 \mapsto k_2 \mathbin{\#} p_2 \tag{31.5d}$$

$$\mathtt{not}(p) \mathbin{\#} \mathtt{not}(k) \mapsto k \mathbin{\#} p \tag{31.5e}$$

$$\mathtt{ap}(p_1)\,;k \mathbin{\#} \lambda\,(x{:}\phi)\; p_2 \mapsto k \mathbin{\#} [p_1/x]p_2. \tag{31.5f}$$

The rules of indirect proof give rise to the following transitions:

$$\mathtt{ccp}(x:\phi.k_1 \mathbin{\#} p_1) \mathbin{\#} p_2 \mapsto [p_2/x]k_1 \mathbin{\#} [p_2/x]p_1 \tag{31.5g}$$

$$k_1 \mathbin{\#} \mathtt{ccr}(u \div \phi.k_2 \mathbin{\#} p_2) \mapsto [k_1/u]k_2 \mathbin{\#} [k_1/u]p_2. \tag{31.5h}$$

The first of these defines the behavior of the refutation of ϕ that proceeds by contradicting the assumption that ϕ is true. This refutation is activated by presenting it with a proof of ϕ, which is then substituted for the assumption in the new state. Thus, "ccp" stands for "call with current proof." The second transition defines the behavior of the proof of ϕ that proceeds by contradicting the assumption that ϕ is false. This proof is activated by presenting it with a refutation of ϕ, which is then substituted for the assumption in the new state. Thus, "ccr" stands for "call with current refutation."

Rules (31.5g) and (31.5h) overlap in that there are two possible transitions for a state of the form

$$\mathtt{ccp}(x:\phi.k_1 \mathbin{\#} p_1) \mathbin{\#} \mathtt{ccr}(u \div \phi.k_2 \mathbin{\#} p_2),$$

one to the state $[p/x]k_1 \mathbin{\#} [p/x]p_1$, where p is $\mathtt{ccr}(u \div \phi.k_2 \mathbin{\#} p_2)$, and one to the state $[k/u]k_2 \mathbin{\#} [k/u]p_2$, where k is $\mathtt{ccp}(x:\phi.k_1 \mathbin{\#} p_1)$. The dynamics of classical logic is therefore nondeterministic. To avoid this we may impose a priority ordering among the two cases, preferring one transition over the other when there is a choice. Preferring the first corresponds to a "lazy" dynamics for proofs, because we pass the unevaluated proof p to the refutation on the left-hand side, which is thereby activated. Preferring the second corresponds to an "eager" dynamics for proofs, in which we pass the unevaluated refutation k to the proof, which is thereby activated.

Theorem 31.1 (Preservation)**.** *If $k \div \phi$, $p : \phi$, and $k \mathbin{\#} p \mapsto k' \mathbin{\#} p'$, then there exists ϕ' such that $k' \div \phi'$ and $p' : \phi'$.*

Proof By rule induction on the dynamics of classical logic. □

Theorem 31.2 (Progress)**.** *If $k \div \phi$ and $p : \phi$, then either $k \mathbin{\#} p$ final or $k \mathbin{\#} p \mapsto k' \mathbin{\#} p'$.*

Proof By rule induction on the statics of classical logic. □

To initiate computation we postulate that `halt` is a refutation of any proposition. The initial and final states of a computation are defined as follows:

$$\frac{}{\texttt{halt} \mathbin{\#} p\ \mathsf{initial}} \tag{31.6a}$$

$$\frac{p\ \mathsf{canonical}}{\texttt{halt} \mathbin{\#} p\ \mathsf{final}}\,. \tag{31.6b}$$

The judgment p canonical states that p is a canonical proof, which is defined to be any proof other than an indirect proof.

31.4 Law of the Excluded Middle

The law of the excluded middle is derivable in classical logic:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi\ \mathsf{true}}}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi \vee \neg\phi\ \mathsf{true}} \quad \dfrac{}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi \vee \neg\phi\ \mathsf{false}}}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \#}}{\dfrac{\dfrac{\dfrac{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi\ \mathsf{false}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \neg\phi\ \mathsf{true}}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi \vee \neg\phi\ \mathsf{true}} \quad \dfrac{}{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi \vee \neg\phi\ \mathsf{false}}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \#}}{\phi \vee \neg\phi\ \mathsf{true}}\,.$$

When written out using explicit proofs and refutations, we obtain the proof term $p_0 : \phi \vee \neg\phi$:

$$\texttt{ccr}(u \div \phi \vee \neg\phi\,.\,u \mathbin{\#} \texttt{r} \cdot \texttt{not}(\texttt{ccp}(x : \phi\,.\,u \mathbin{\#} \texttt{l} \cdot x))).$$

To understand the computational meaning of this proof, let us juxtapose it with a refutation $k \div \phi \vee \neg\phi$ and simplify it by using the dynamics given in Section 31.3. The first step is the transition

$$k \mathbin{\#} \texttt{ccr}(u \div \phi \vee \neg\phi\,.\,u \mathbin{\#} \texttt{r} \cdot \texttt{not}(\texttt{ccp}(x : \phi\,.\,u \mathbin{\#} \texttt{l} \cdot x)))$$
$$\mapsto$$
$$k \mathbin{\#} \texttt{r} \cdot \texttt{not}(\texttt{ccp}(x : \phi\,.\,k \mathbin{\#} \texttt{l} \cdot x)),$$

wherein we have replicated k so that it occurs in two places in the result state. By virtue of its type the refutation k must have the form $\texttt{case}(k_1; k_2)$, where $k_1 \div \phi$ and $k_2 \div \neg\phi$. Continuing the reduction, we obtain:

$$\texttt{case}(k_1; k_2) \mathbin{\#} \texttt{r} \cdot \texttt{not}(\texttt{ccp}(x : \phi\,.\,\texttt{case}(k_1; k_2) \mathbin{\#} \texttt{l} \cdot x))$$
$$\mapsto$$
$$k_2 \mathbin{\#} \texttt{not}(\texttt{ccp}(x : \phi\,.\,\texttt{case}(k_1; k_2) \mathbin{\#} \texttt{l} \cdot x)).$$

By virtue of its type, k_2 must have the form $\mathtt{not}(p_2)$, where $p_2 : \phi$, and hence the transition proceeds as follows:

$$\mathtt{not}(p_2) \mathbin{\#} \mathtt{not}(\mathtt{ccp}(x : \phi.\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x)) \mapsto \mathtt{ccp}(x : \phi.\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x) \mathbin{\#} p_2.$$

Observe that p_2 is a valid proof of ϕ. Proceeding, we obtain

$$\mathtt{ccp}(x : \phi.\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x) \mathbin{\#} p_2 \mapsto \mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot p_2 \mapsto k_1 \mathbin{\#} p_2.$$

The first of these two steps is the crux of the matter: The refutation $k = \mathtt{case}(k_1; k_2)$, which was replicated at the outset of the derivation, is re-used, but with a different argument. At the first use, the refutation k, which is provided by the context of use of the law of the excluded middle, is presented with a proof $\mathtt{r} \cdot p_1$ of $\phi \vee \neg\phi$. That is, the proof behaves as though the right disjunct of the law is true, which is to say that ϕ is false. If the context is such that it inspects this proof, it can only be by providing the proof p_2 of ϕ that refutes the claim that ϕ is false. Should this occur, the proof of the law of the excluded middle "backtracks" the context, providing instead the proof $\mathtt{l} \cdot p_2$ to k, which then passes p_2 to k_1 without further incident. The proof of the law of the excluded middle boldly asserts $\neg\phi$ true, regardless of the form of ϕ. Then, if caught in its lie by the context providing a proof of ϕ, it "changes its mind" and asserts ϕ to the original context k after all. No further reversion is possible, because the context has itself provided a proof p_2 of ϕ.

The law of the excluded middle illustrates that classical proofs are to be thought of as interactions between proofs and refutations, which is to say interactions between a proof and the context in which it is used. In programming terms this corresponds to an abstract machine with an explicit control stack, or continuation, representing the context of evaluation of an expression. That expression may access the context (stack, continuation) to effect backtracking as necessary to maintain the perfect symmetry between truth and falsity. The penalty is that a closed proof of a disjunction no longer need reveal which disjunct it proves, for as we have just seen, it may, on further inspection, "change its mind."

31.5 The Double-Negation Translation

One consequence of the greater expressiveness of constructive logic is that classical proofs may be translated systematically into constructive proofs of a classically equivalent proposition. This means that by systematically reorganizing the classical proof we may, without

changing its meaning from a classical perspective, turn it into a constructive proof of a constructively weaker proposition. This shows that there is no loss in adhering to constructive proofs, because every classical proof can be seen as a constructive proof of a constructively weaker, but classically equivalent, proposition. Moreover, it proves that classical logic is weaker (less expressive) than constructive logic, contrary to a naïve interpretation that would say that the additional reasoning principles, such as the law of the excluded middle, afforded by classical logic make it stronger. In programming language terms adding a "feature" does not necessarily strengthen (improve the expressive power of) your language; on the contrary, it may weaken it.

We define a translation ϕ^* of propositions that provides an interpretation of classical into constructive logic according to the following correspondences:

Classical	**Constructive**	
$\Delta\ \Gamma \vdash \phi\ \mathsf{true}$	$\neg\Delta^*\ \Gamma^* \vdash \neg\neg\phi^*\ \mathsf{true}$	truth
$\Delta\ \Gamma \vdash \phi\ \mathsf{false}$	$\neg\Delta^*\ \Gamma^* \vdash \neg\phi^*\ \mathsf{true}$	falsity
$\Delta\ \Gamma \vdash \#$	$\neg\Delta^*\ \Gamma^* \vdash \bot\ \mathsf{true}$	contradiction

Classical truth is weakened to constructive irrefutability; classical falsehood is represented as constructive refutability; classical contradiction is represented by constructive falsehood. Falsity assumptions are negated after translation to express their falsehood; truth assumptions are merely translated as is. Because the double negations are classically cancellable, the translation will be easily seen to yield a classically equivalent proposition. But because $\neg\neg\phi$ is constructively weaker than ϕ, we also see that a proof in classical logic is translated to a constructive proof of a weaker statement.

Many choices for the translation of propositions are available; we have chosen one that makes the proof of the correspondence between classical and constructive logic go smoothly:

$$\begin{aligned}
\top^* &= \top \\
(\phi_1 \wedge \phi_2)^* &= \phi_1^* \wedge \phi_2^* \\
\bot^* &= \bot \\
(\phi_1 \vee \phi_2)^* &= \phi_1^* \vee \phi_2^* \\
(\phi_1 \supset \phi_2)^* &= \phi_1^* \supset \neg\neg\phi_2^* \\
(\neg\phi)^* &= \neg\phi^*.
\end{aligned}$$

It is straightforward to show by induction on the rules of classical logic that the correspondences just summarized hold. Some simple lemmas are required. For example, we must show that the entailment

$$\neg\neg\phi\ \mathsf{true}\ \neg\neg\psi\ \mathsf{true} \vdash \neg\neg(\phi \wedge \psi)\ \mathsf{true}$$

is derivable in constructive logic.

31.6 Notes

The computational interpretation of classical logic was first explored by Griffin (1990) and Murthy (1991). The account given here was influenced by the work of Wadler (2003), transposed by Nanevski from sequent calculus to natural deduction by using multiple forms of judgment. The terminology is inspired by the work of Lakatos (1976), an insightful and inspiring analysis of the discovery of proofs and refutations of conjectures in mathematics. Versions of the double-negation translation were originally given by Gödel and Gentzen, and have been extended and modified in numerous other studies. The computational content of the double-negation translation was first elucidated by Murthy (1991), who established the connection with the continuation-passing transformation used in compilers.

PART XII

Symbols

32 Symbols

A *symbol* is an atomic datum with no internal structure. Whereas a variable is given meaning by substitution, a symbol is given meaning by a family of operations indexed by symbols. A symbol is therefore just a name, or index, for an instance of a family of operations. Many different interpretations may be given to symbols according to the operations we choose to consider, giving rise to concepts such as fluid binding, dynamic classification, mutable storage, and communication channels. With each symbol is associated a type whose interpretation depends on the particular application. The type of a symbol influences the type of its associated operations under each interpretation. For example, in the case of mutable storage, the type of symbol constrains the contents of the cell named by that symbol to values of that type. It is important to bear in mind that a symbol is *not* a value of its associated type, but only a constraint on how that symbol may be interpreted by the operations associated with it.

In this chapter we consider two constructs for computing with symbols. The first is a means of *declaring* new symbols for use within a specified *scope*. The expresssion $\nu\, a : \rho\, \mathtt{in}\, e$ introduces a "new" symbol a with associated type ρ for use within e. The declared symbol a is new in the sense that it is bound by the declaration within e, and so may be renamed at will to ensure that it differs from any finite set of active symbols. Whereas the statics determines the scope of a declared symbol, its range of significance, or *extent*, is determined by the dynamics. There are two different dynamic interpretations of symbols, the *scoped* and the *free* (short for *scope–free*) dynamics. The scoped dynamics limits the extent of the symbol to its scope; the lifetime of the symbol is restricted to the evaluation of its scope. Alternatively, under the free dynamics the extent of a symbol exceeds its scope, extending to the entire computation of which it is a part. We may say that in the free dynamics a symbol "escapes its scope," but it is more accurate to say that its scope widens to encompass the rest of the computation.

The second construct associated with symbols is the concept of a symbolic reference, a form of expression whose sole purpose is to refer to a particular symbol. Symbolic references are values of a type $\rho\, \mathtt{sym}$ and have the form $\mathtt{\&}\, a$ for some symbol a with associated type ρ. The eliminatory form for the type $\rho\, \mathtt{sym}$ is a conditional branch that determines whether a symbolic reference refers to a statically specified symbol. Crucially, the statics of the eliminatory form is carefully designed so that, in the positive case, the type associated with the referenced symbol is made manifest, whereas in the negative case, no type information is gleaned because the referenced symbol could be of any type.

32.1 Symbol Declaration

The ability to declare a new symbol is shared by all applications of symbols in subsequent chapters. The syntax for symbol declaration is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Exp	e	$::=$	$\texttt{new[}\tau\texttt{](}a\texttt{.}e\texttt{)}$	$\nu\, a\!:\!\tau\ \texttt{in}\ e$	generation

The statics of symbol declaration makes use of a *signature*, or *symbol context*, that associates a type with each of a finite set of symbols. We use the letter Σ to range over signatures, which are finite sets of pairs $a \sim \tau$, where a is a symbol and τ is a type. The typing judgment $\Gamma \vdash_\Sigma e : \tau$ is parameterized by a signature Σ, associating types with symbols.

The statics of symbol declaration makes use of a judgment $\tau\ \mathsf{mobile}$, whose definition depends on whether the dynamics is scoped. In a scoped dynamics, mobility is defined so that the computed value of a mobile type cannot depend on any symbol. By constraining the scope of a declaration to have mobile type, we can, under this interpretation, ensure that the extent of a symbol is confined to its scope. In a free dynamics, every type is deemed mobile, because the dynamics ensures that the scope of a symbol is widened to accommodate the possibility that the value returned from the scope of a declaration may depend on the declared symbol. The term "mobile" reflects the informal idea that symbols may or may not be "moved" from the scope of their declaration according to the dynamics given to them. A free dynamics allows symbols to be moved freely, whereas a scoped dynamics limits their range of motion.

The statics of symbol declaration itself is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma, a\sim\rho} e : \tau \quad \tau\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \texttt{new[}\rho\texttt{](}a\texttt{.}e\texttt{)} : \tau}\,. \tag{32.1}$$

As mentioned, the condition on τ is to be chosen so as to ensure that the returned value is meaningful in which dynamics we are using.

32.1.1 Scoped Dynamics

The scoped dynamics of symbol declaration is given by a transition judgment of the form $e \underset{\Sigma}{\mapsto} e'$ indexed by a signature Σ, specifying the active symbols of the transition. Either e or e' may involve the symbols declared in Σ, but no others:

$$\frac{e \underset{\Sigma, a\sim\rho}{\longmapsto} e'}{\texttt{new[}\rho\texttt{](}a\texttt{.}e\texttt{)} \underset{\Sigma}{\mapsto} \texttt{new[}\rho\texttt{](}a\texttt{.}e'\texttt{)}} \tag{32.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{new[}\rho\texttt{](}a\texttt{.}e\texttt{)} \underset{\Sigma}{\mapsto} e}\,. \tag{32.2b}$$

Rule (32.2a) specifies that evaluation takes place within the scope of the declaration of a symbol. Rule (32.2b) specifies that the declared symbol is "forgotten" once its scope has been evaluated.

The definition of the judgment τ mobile must be chosen to ensure that the following *mobility condition* is satisfied:

If τ *mobile*, $\vdash_{\Sigma,a\sim\rho} e : \tau$, *and* e $\mathsf{val}_{\Sigma,a\sim\rho}$, *then* $\vdash_{\Sigma} e : \tau$ *and* e val_{Σ}.

For example, in the presence of symbolic references (see Section 32.2), a function type cannot be deemed mobile, because a function may contain a reference to a local symbol. The type `nat` may be deemed mobile only if the successor is evaluated eagerly, for otherwise a symbolic reference may occur within a value of this type, invalidating the condition.

Theorem 32.1 (Preservation). *If* $\vdash_{\Sigma} e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$, *then* $\vdash_{\Sigma} e' : \tau$.

Proof By induction on the dynamics of symbol declaration. Rule (32.2a) follows directly by induction, applying Rule (32.1). Rule (32.2b) follows directly from the condition on mobility. □

Theorem 32.2 (Progress). *If* $\vdash_{\Sigma} e : \tau$, *then either* $e \underset{\Sigma}{\mapsto} e'$, *or* e val_{Σ}.

Proof There is only one rule to consider, Rule (32.1). By induction we have either $e \underset{\Sigma,a\sim\rho}{\longmapsto} e'$, in which case Rule (32.2a) applies, or e $\mathsf{val}_{\Sigma,a\sim\rho}$, in which case by the mobility condition we have e val_{Σ}, and hence Rule (32.2b) applies. □

32.1.2 Scope-Free Dynamics

The scope–free dynamics of symbols is defined by a transition system between states of the form $\nu\ \Sigma\ \{\, e \,\}$, where Σ is a signature and e is an expression over this signature. The judgment $\nu\ \Sigma\ \{\, e \,\} \mapsto \nu\ \Sigma'\ \{\, e' \,\}$ states that evaluation of e relative to symbols Σ results in the expression e' in the extension Σ' of Σ:

$$\frac{}{\nu\ \Sigma\ \{\, \mathtt{new}[\rho]\,(a.e) \,\} \mapsto \nu\ \Sigma, a \sim \rho\ \{\, e \,\}}\,. \tag{32.3}$$

Rule (32.3) specifies that symbol generation enriches the signature with the newly introduced symbol by extending the signature for all future transitions.

All other rules of the dynamics must be changed accordingly to account for the allocated symbols. For example, the dynamics of function application cannot simply be inherited from Chapter 10, but must be reformulated as follows:

$$\frac{\nu\ \Sigma\ \{\, e_1 \,\} \mapsto \nu\ \Sigma'\ \{\, e_1' \,\}}{\nu\ \Sigma\ \{\, e_1(e_2) \,\} \mapsto \nu\ \Sigma'\ \{\, e_1'(e_2) \,\}} \tag{32.4a}$$

$$\frac{}{\nu\ \Sigma\ \{\, \lambda\,(x:\tau)\,e(e_2) \,\} \mapsto \nu\ \Sigma\ \{\, [e_2/x]e \,\}}\,. \tag{32.4b}$$

These rules shuffle around the signature so as to account for symbol declarations within the constituent expressions of the application. Similar rules must be given for all other constructs of the language.

Theorem 32.3 (Preservation). *If* $\nu\ \Sigma\ \{\, e \,\} \mapsto \nu\ \Sigma'\ \{\, e' \,\}$ *and* $\vdash_\Sigma e : \tau$, *then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$.

Proof There is only one rule to consider, Rule (32.3), which is easily handled by inversion of Rule (32.1). □

Theorem 32.4 (Progress). *If* $\vdash_\Sigma e : \tau$, *then either* $e\ \mathit{val}_\Sigma$ *or* $\nu\ \Sigma\ \{\, e \,\} \mapsto \nu\ \Sigma'\ \{\, e' \,\}$ *for some* Σ' *and* e'.

Proof Immediate, by Rule (32.3). □

32.2 Symbolic References

Symbols are not themselves values, but they may be used to form values. One useful example is provided by the type $\tau\ \mathtt{sym}$ of *symbolic references*. A value of this type has the form $\mathtt{\&}\,a$, where a is a symbol in the signature. To compute with a reference we may branch according to whether it is a reference to a specified symbol. The syntax of symbolic references is given by the following chart:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	$\mathtt{sym}(\tau)$	$\tau\ \mathtt{sym}$	symbols
Exp	e		$\mathtt{sym}[a]$	$\mathtt{\&}\,a$	reference
			$\mathtt{is}[a]\,[t.\tau]\,(e; e_1; e_2)$	$\mathtt{if}\ e\ \mathtt{is}\ a\ \mathtt{then}\ e_1\ \mathtt{ow}\ e_2$	comparison

The expression $\mathtt{sym}[a]$ is a reference to the symbol a, a value of type $\mathtt{sym}(\tau)$. The expression $\mathtt{is}[a]\,[t.\tau]\,(e; e_1; e_2)$ compares the value of e, which must be a reference to some symbol b with the given symbol a. If b is a, the expression evaluates to e_1 and otherwise to e_2.

32.2.1 Statics

The typing rules for symbolic references are as follows:

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\rho} \mathtt{sym}[a] : \mathtt{sym}(\rho)} \tag{32.5a}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\rho} e : \mathtt{sym}(\rho') \quad \Gamma \vdash_{\Sigma, a\sim\rho} e_1 : [\rho/t]\tau \quad \Gamma \vdash_{\Sigma, a\sim\rho} e_2 : [\rho'/t]\tau}{\Gamma \vdash_{\Sigma, a\sim\rho} \mathtt{is}[a]\,[t.\tau]\,(e; e_1; e_2) : [\rho'/t]\tau}. \tag{32.5b}$$

Rule (32.5a) is the introduction rule for the type $\mathtt{sym}(\rho)$. It states that if a is a symbol with associated type ρ, then $\mathtt{sym}[a]$ is an expression of type $\mathtt{sym}(\rho)$. Rule (32.5b) is the elimination rule for the type $\mathtt{sym}(\rho)$. The type associated with the given symbol a is not required to be the same as the type of the symbol referred to by the expression e.

If e evaluates to a reference to a, then these types will coincide, but if it refers to another symbol, $b \neq a$, then these types may well differ.

With this in mind, let us examine carefully Rule (32.5b). A priori there is a discrepancy between the type ρ of a and the type ρ' of the symbol referred to by e. This discrepancy is mediated by the type operator $t\,.\,\tau$.[1] Regardless of the outcome of the comparison, the overall type of the expression is $[\rho'/t]\tau$. To ensure safety, we must ensure that this is a valid type for the result, regardless of whether the comparison succeeds or fails. If e evaluates to the symbol a, then we "learn" that the types ρ' and ρ coincide, because the specified and referenced symbol coincide. This is reflected by the type $[\rho/t]\tau$ for e_1. If e evaluates to some other symbol, $a' \neq a$, then the comparison evaluates to e_2, which is required to have type $[\rho'/t]\tau$; no further information about the type of the symbol is acquired in this branch.

32.2.2 Dynamics

The (scoped) dynamics of symbolic references is given by the following rules:

$$\frac{}{\mathtt{sym}[a]\ \mathsf{val}_{\Sigma,a\sim\rho}} \tag{32.6a}$$

$$\frac{}{\mathtt{is}[a]\,[t\,.\,\tau]\,(\mathtt{sym}[a]; e_1; e_2) \underset{\Sigma,a\sim\rho}{\longmapsto} e_1} \tag{32.6b}$$

$$\frac{(a \neq a')}{\mathtt{is}[a]\,[t\,.\,\tau]\,(\mathtt{sym}[a']; e_1; e_2) \underset{\Sigma,a\sim\rho,a'\sim\rho'}{\longmapsto} e_2} \tag{32.6c}$$

$$\frac{e \underset{\Sigma,a\sim\rho}{\longmapsto} e'}{\mathtt{is}[a]\,[t\,.\,\tau]\,(e; e_1; e_2) \underset{\Sigma,a\sim\rho}{\longmapsto} \mathtt{is}[a]\,[t\,.\,\tau]\,(e'; e_1; e_2)}\,. \tag{32.6d}$$

Rules (32.6b) and (32.6c) specify that $\mathtt{is}[a]\,[t\,.\,\tau]\,(e; e_1; e_2)$ branches according to whether the value of e is a reference to the symbol a.

32.2.3 Safety

To ensure that the mobility condition is satisfied, it is important that symbolic reference types *not* be deemed mobile.

Theorem 32.5 (Preservation)**.** *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$*, then* $\vdash_\Sigma e' : \tau$.

Proof By rule induction on Rules (32.6). The most interesting case is Rule (32.6b). When the comparison is positive, the types ρ and ρ' must be the same, because each symbol has at most one associated type. Therefore e_1, which has type $[\rho'/t]\tau$, also has type $[\rho/t]\tau$, as required. □

[1] See Chapter 14 for a discussion of type operators.

Lemma 32.6 (Canonical Forms)**.** *If* $\vdash_\Sigma e : \mathtt{sym}(\rho)$ *and* e val_Σ*, then* $e = \mathtt{sym}[a]$ *for some* a *such that* $\Sigma = \Sigma', a \sim \rho$.

Proof By rule induction on Rules (32.5), taking account of the definition of values. □

Theorem 32.7 (Progress)**.** *Suppose that* $\vdash_\Sigma e : \tau$*. Then either* e val_Σ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.

Proof By rule induction on Rules (32.5). For example, consider Rule (32.5b), in which we have that $\mathtt{is}[a][t.\tau](e; e_1; e_2)$ has some type τ and that $e : \mathtt{sym}(\rho)$ for some ρ. By induction either Rule (32.6d) applies or else we have that e val_Σ, in which case we are assured by Lemma 32.6 that e is $\mathtt{sym}[a]$ for some symbol b of type ρ declared in Σ. But then progress is ensured by Rules (32.6b) and (32.6c), because equality of symbols is decidable (either a is b or it is not). □

32.3 Notes

The concept of a symbol in a programming language was considered by McCarthy in the original formulation of Lisp (McCarthy, 1965). Unfortunately, symbols were, and often continue to be, confused with variables, as they were in the original formulation of Lisp. Although symbols are frequently encountered in dynamically typed languages, the formulation given here makes clear that they are equally sensible in statically typed languages. The present account was influenced by the work of Pitts and Stark (1993) and on the declaration of names in the π-calculus (Milner, 1999).

33 Fluid Binding

In this chapter we return to the concept of dynamic scoping of variables that was criticized in Chapter 8. There it was observed that dynamic scoping is problematic for at least two reasons:

- A bound variable may not always be renamed in an expression without changing its meaning.
- Because the scope of a variable is resolved dynamically, type safety is compromised.

These violations of the expected behavior of variables are intolerable, because they are at variance with mathematical practice and because they compromise modularity.

It is possible, however, to recover a type-safe analog of dynamic scoping by divorcing it from the concept of a variable and instead introducing a new mechanism, called *fluid*, or *dynamic*, *binding* of a symbol. Fluid binding associates with a symbol a value of its associated type within a specified scope. On exiting that scope, the binding is dropped (or, more accurately, reverted to its binding in the surrounding context).

33.1 Statics

The language $\mathcal{L}\{\texttt{fluid}\}$ extends the language $\mathcal{L}\{\texttt{sym}\}$ defined in Chapter 32 with the following additional constructs:

Sort			Abstract Form	Concrete Form	Description
Exp	e	$::=$	$\texttt{put}[a]\,(e_1; e_2)$	$\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2$	binding
			$\texttt{get}[a]$	$\texttt{get}\,a$	retrieval

As in Chapter 32, we use a to stand for some unspecified *symbol*. The expression $\texttt{get}[a]$ evaluates to the value of the current binding of a, if it has one, and is stuck otherwise. The expression $\texttt{put}[a]\,(e_1; e_2)$ binds the symbol a to the value e_1 for the duration of the evaluation of e_2, at which point the binding of a reverts to what it was prior to the execution. The symbol a is not bound by the put expression, but is instead a parameter of it.

The statics of $\mathcal{L}\{\texttt{fluid}\}$ is defined by judgments of the form

$$\Gamma \vdash_\Sigma e : \tau,$$

where Σ is a finite set of symbol declarations of the form $a \sim \tau$ such that no symbol is declared more than once.

The statics of $\mathcal{L}\{\texttt{fluid}\}$ extends that of $\mathcal{L}\{\texttt{sym}\}$ (see Chapter 32) with the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \texttt{get}[a] : \tau} \tag{33.1a}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau_1} e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a\sim\tau_1} e_2 : \tau_2}{\Gamma \vdash_{\Sigma, a\sim\tau_1} \texttt{put}[a]\,(e_1; e_2) : \tau_2}\,. \tag{33.1b}$$

Rule (33.1b) specifies that the symbol a is a parameter of the expression that must be declared in Σ.

33.2 Dynamics

We assume a stacklike dynamics for symbols, as described in Chapter 32. The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ maintains an association of values with symbols that changes in a stacklike manner during execution. We define a family of transition judgments of the form $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$, where Σ is as in the statics and μ is a finite function mapping some subset of the symbols declared in Σ to values of appropriate type. If μ is defined for some symbol a, then it has the form $\mu' \otimes a \hookrightarrow e$ for some μ' and value e. If, however, μ is undefined for some symbol a, we may regard it as having the form $\mu' \otimes a \hookrightarrow \bullet$. We write $a \hookrightarrow \_$ to stand ambiguously for either $a \hookrightarrow \bullet$ or $a \hookrightarrow e$ for some expression e.

The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma, a\sim\tau}}{\texttt{get}[a] \overset{\mu\otimes a\hookrightarrow e}{\underset{\Sigma, a\sim\tau}{\longmapsto}} e} \tag{33.2a}$$

$$\frac{e_1 \overset{\mu}{\underset{\Sigma}{\mapsto}} e_1'}{\texttt{put}[a]\,(e_1; e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{put}[a]\,(e_1'; e_2)} \tag{33.2b}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma, a\sim\tau} \quad e_2 \overset{\mu\otimes a\hookrightarrow e_1}{\underset{\Sigma, a\sim\tau}{\longmapsto}} e_2'}{\texttt{put}[a]\,(e_1; e_2) \overset{\mu\otimes a\hookrightarrow \_}{\underset{\Sigma, a\sim\tau}{\longmapsto}} \texttt{put}[a]\,(e_1; e_2')} \tag{33.2c}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma, a\sim\tau} \quad e_2\ \mathsf{val}_{\Sigma, a\sim\tau}}{\texttt{put}[a]\,(e_1; e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} e_2}\,. \tag{33.2d}$$

Rule (33.2a) specifies that $\texttt{get}[a]$ evaluates to the current binding of a, if any. Rule (33.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (33.2c) evaluates e_2 in an environment in which the symbol a is bound to the value e_1,

regardless of whether a is already bound in the environment. Rule (33.2d) eliminates the fluid binding for a once evaluation of the extent of the binding has been completed.

According to the dynamics defined by Rules (33.2), there is no transition of the form $\mathtt{get}[a] \overset{\mu}{\underset{\Sigma}{\mapsto}} e$ if $\mu(a) = \bullet$. The judgment $e\ \mathsf{unbound}_\Sigma$ states that execution of e will lead to such a "stuck" state and is inductively defined by the following rules:

$$\frac{\mu(a) = \bullet}{\mathtt{get}[a]\ \mathsf{unbound}_\mu} \tag{33.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\mu}{\mathtt{put}[a]\,(e_1; e_2)\ \mathsf{unbound}_\mu} \tag{33.3b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{unbound}_\mu}{\mathtt{put}[a]\,(e_1; e_2)\ \mathsf{unbound}_\mu}\,. \tag{33.3c}$$

In a larger language it would also be necessary to include error propagation rules of the sort discussed in Chapter 6.

33.3 Type Safety

Define the auxiliary judgment $\mu : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{33.4a}$$

$$\frac{\vdash_\Sigma e : \tau \quad \mu : \Sigma}{\mu \otimes a \hookrightarrow e : \Sigma, a \sim \tau} \tag{33.4b}$$

$$\frac{\mu : \Sigma}{\mu \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau}\,. \tag{33.4c}$$

These rules specify that if a symbol is bound to a value, then that value must be of the type associated with the symbol by Σ. No demand is made in the case in which the symbol is unbound (equivalently, bound to a "black hole").

Theorem 33.1 (Preservation). *If $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$, where $\mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*

Proof By rule induction on Rules (33.2). Rule (33.2a) is handled by the definition of $\mu : \Sigma$. Rule (33.2b) follows immediately by induction. Rule (33.2d) is handled by inversion of Rules (33.1). Finally, Rule (33.2c) is handled by inversion of Rules (33.1) and induction. □

Theorem 33.2 (Progress). *If $\vdash_\Sigma e : \tau$ and $\mu : \Sigma$, then either $e\ \mathsf{val}_\Sigma$, or $e\ \mathsf{unbound}_\mu$, or there exists e' such that $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$.*

Proof By induction on Rules (33.1). For Rule (33.1a), we have $\Sigma \vdash a \sim \tau$ from the premise of the rule, and hence, because $\mu : \Sigma$, we have either $\mu(a) = \bullet$ or $\mu(a) = e$ for some e such that $\vdash_\Sigma e : \tau$. In the former case we have $e\ \mathsf{unbound}_\mu$, and in the latter we have $\mathtt{get}\,[a] \overset{\mu}{\underset{\Sigma}{\mapsto}} e$. For Rule (33.1b), we have by induction that $e_1\ \mathsf{val}_\Sigma$, $e_1\ \mathsf{unbound}_\mu$, or $e_1 \overset{\mu}{\underset{\Sigma}{\mapsto}} e_1'$. In the latter two cases we may apply Rule (33.2b) or Rule (33.3b), respectively. If $e_1\ \mathsf{val}_\Sigma$, we apply induction to obtain that $e_2\ \mathsf{val}_\Sigma$, in which case Rule (33.2d) applies; $e_2\ \mathsf{unbound}_\mu$, in which case Rule (33.3c) applies; or $e_2 \overset{\mu}{\underset{\Sigma}{\mapsto}} e_2'$, in which case Rule (33.2c) applies. □

33.4 Some Subtleties

The value of $\mathtt{put}\,e_1\,\mathtt{for}\,a\,\mathtt{in}\,e_2$ is the value of e_2, calculated in a context in which a is bound to the value of e_1. If e_2 is of a basic type, such as `nat`, then the reversion of the binding of a cannot influence the meaning of the result.[1]

But what if the type of $\mathtt{put}\,e_1\,\mathtt{for}\,a\,\mathtt{in}\,e_2$ is a function type, so that the returned value is a λ-abstraction? The body of the returned λ may refer to the binding of a, which is reverted on return from the `put`. For example, consider the expression

$$\mathtt{put}\,17\,\mathtt{for}\,a\,\mathtt{in}\,\lambda\,(x\mathtt{:nat})\,x + \mathtt{get}\,a, \tag{33.5}$$

which has type $\mathtt{nat} \to \mathtt{nat}$, given that a is a symbol of type `nat`. Let us assume, for the sake of discussion, that a is unbound at the point at which this expression is evaluated. Evaluating the `put` binds a to the number 17 and returns the function $\lambda\,(x\mathtt{:nat})\,x + \mathtt{get}\,a$. But because a is reverted to its unbound state on exiting the `put`, applying this function to an argument will result in an error, unless a binding for a is provided. Thus, if f is bound to the result of evaluating (33.5), then the expression

$$\mathtt{put}\,21\,\mathtt{for}\,a\,\mathtt{in}\,f\,(7) \tag{33.6}$$

will evaluate successfully to 28, whereas evaluation of $f\,(7)$ in the absence of a surrounding binding for a will incur an error.

Contrast this with the superficially similar expression

$$\mathtt{let}\,y\,\mathtt{be}\,17\,\mathtt{in}\,\lambda\,(x\mathtt{:nat})\,x + y, \tag{33.7}$$

in which we have replaced the fluid-bound symbol a by a statically bound variable y. This expression evaluates to $\lambda\,(x\mathtt{:nat})\,x + 17$, which adds 17 to its argument when applied. There is no possibility of an unbound symbol arising at execution time, precisely because variables are interpreted by substitution.

One way to think about this situation is to consider that fluid-bound symbols serve as an alternative to passing additional arguments to a function to specialize its value whenever it

[1] As long as the the successor is evaluated eagerly; if not, the following examples may be adapted to situations in which the value of e_2 is a lazily evaluated number.

is called. To see this, let e stand for the value of expression (33.5), a λ-abstraction whose body is dependent on the binding of the symbol a. To use this function safely, it is necessary that the programmer provide a binding for a prior to calling it. For example, the expression

$$\texttt{put}\,7\,\texttt{for}\,a\,\texttt{in}\,(e(9))$$

evaluates to 16, and the expression

$$\texttt{put}\,8\,\texttt{for}\,a\,\texttt{in}\,(e(9))$$

evaluates to 17. Writing just $e(9)$, without a surrounding binding for a, results in a run-time error attempting to retrieve the binding of the unbound symbol a.

This behavior may be simulated by adding an additional argument to the function value that will be bound to the current binding of the symbol a at the point where the function is called. Instead of using fluid binding, we would provide an additional argument at each call site, writing

$$e'(7)\,(9)$$

and

$$e'(8)\,(9),$$

respectively, where e' is the λ-abstraction

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

Additional arguments can be cumbersome, though, especially when several call sites provide the same binding for a. Using fluid binding, we may write

$$\texttt{put}\,7\,\texttt{for}\,a\,\texttt{in}\,\langle e(8), e(9)\rangle,$$

whereas by using an additional argument we must write

$$\langle e'(7)\,(8), e'(7)\,(9)\rangle.$$

However, such redundancy can be mitigated by simply factoring out the common part, writing

$$\texttt{let}\,f\,\texttt{be}\,e'(7)\,\texttt{in}\,\langle f(8), f(9)\rangle.$$

The awkwardness of this simulation is usually taken as an argument in favor of including fluid binding in a language. The drawback, which is often perceived as an advantage, is that nothing in the type of a function reveals its dependency on the binding of a symbol. It is therefore quite easy to forget that such a binding is required, leading to run-time failures that might better be caught at compile time.

33.5 Fluid References

The `get` and `put` operations for fluid binding are indexed by a symbol that must be given as part of the syntax of the operator. Rather than insist that the target symbol be given

statically, it is useful to be able to defer until run time the choice of fluid on which a `get` or `put` acts. This may be achieved by introducing *references* to fluids, which allow the name of a fluid to be represented as a value. References come equipped with analogs of the `get` and `put` primitives, but for a dynamically determined symbol.

The syntax of references as an extension to $\mathcal{L}\{\texttt{fluid}\}$ is given by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Typ	τ	$::=$	$\texttt{fluid}(\tau)$	$\tau\ \texttt{fluid}$	fluid
Exp	e	$::=$	$\texttt{fl}[a]$	$\texttt{fl}[a]$	reference
			$\texttt{getfl}(e)$	$\texttt{getfl}\ e$	retrieval
			$\texttt{putfl}(e;e_1;e_2)$	$\texttt{putfl}\ e\ \texttt{is}\ e_1\ \texttt{in}\ e_2$	binding

The expression $\texttt{fl}[a]$ is the symbol a considered as a value of type $\texttt{fluid}(\tau)$. The expressions $\texttt{getfl}(e)$ and $\texttt{putfl}(e;e_1;e_2)$ are analogs of the `get` and `put` operations for fluid-bound symbols.

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

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \texttt{fl}[a] : \texttt{fluid}(\tau)} \tag{33.8a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid}(\tau)}{\Gamma \vdash_{\Sigma} \texttt{getfl}(e) : \tau} \tag{33.8b}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid}(\tau) \quad \Gamma \vdash_{\Sigma} e_1 : \tau \quad \Gamma \vdash_{\Sigma} e_2 : \tau_2}{\Gamma \vdash_{\Sigma} \texttt{putfl}(e;e_1;e_2) : \tau_2}\,. \tag{33.8c}$$

Because we are using a scoped dynamics, references to fluids cannot be deemed mobile.

The dynamics of references consists of resolving the referent and deferring to the underlying primitives acting on symbols.

$$\frac{}{\texttt{fl}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{33.9a}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\texttt{getfl}(e) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{getfl}(e')} \tag{33.9b}$$

$$\frac{}{\texttt{getfl}(\texttt{fl}[a]) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{get}[a]} \tag{33.9c}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'}{\texttt{putfl}(e;e_1;e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{putfl}(e';e_1;e_2)} \tag{33.9d}$$

$$\frac{}{\texttt{putfl}(\texttt{fl}[a];e_1;e_2) \overset{\mu}{\underset{\Sigma}{\mapsto}} \texttt{put}[a](e_1;e_2)}\,. \tag{33.9e}$$

33.6 Notes

The concept of dynamic binding arose from the confusion of variables and symbols in early dialects of Lisp. When properly separated, variables retain their substitutive meaning, and symbols give rise to a separate concept of fluid binding. Allen (1978) provides a thorough discussion of the implementation of fluid binding. The formulation given here also draws on the work of Nanevski (2003).

34 Dynamic Classification

In Chapters 12 and 25 we investigated the use of sums for the classification of values of disparate type. Every value of a classified type is labeled with a symbol that determines the type of the instance data. A classified value is decomposed by pattern matching against a known class, which reveals the type of the instance data.

Under this representation the possible classes of an object are fully determined *statically* by its type. However, it is sometimes useful to allow the possible classes of data value to be determined *dynamically*. There are many uses for such a capability, some less apparent than others. The most obvious is simply extensibility, that we wish to introduce new classes of data during execution (and, presumably, define how methods act on values of those new classes).

A less obvious application exploits the fact that the new class is guaranteed to be distinct from any other class that has already been introduced. The class itself is a kind of "secret" that can be disclosed only if the computation that creates the class discloses its existence to another computation. In particular, the class is opaque to any computation to which this disclosure has not been explicitly made. This capability has a number of practical applications.

One application is to use dynamic classification as a "perfect encryption" mechanism that guarantees that a value cannot be determined without access to the appropriate "keys." Keys are, in this scenario, pattern-matching functions that are created when the class is defined. No party that lacks access to the matcher for that class can recover its underlying instance data, and so we may think of that value as encrypted.[1] This can be useful when building programs that communicate over an insecure network: Dynamic classification allows us to build private channels between two parties in the computation (see Chapter 42 for more on this application).

Another application is to exception handling within a program. Exception handling may be seen as a communication between two agents, one that may raise an exception and one that may handle it. We wish to ensure that an exception can be caught only by a designated handler, without fear that any intervening handler may intercept it. This can be achieved by dynamic class allocation. A new class is declared, with the capability to create an instance given only to the raising agent and the capability to match an instance given only to the handler. The exception value cannot be intercepted by any other handler, because no other handler is capable of matching it.

[1] In practice this is implemented by probabilistic techniques to avoid the need for a central arbiter of unicity of symbol names. However, such methods require a source of randomness, which may be seen as just such an arbiter in disguise. There is no free lunch.

34.1 Dynamic Classes

A dynamic class is a symbol that may be generated at run time. A classified value consists of a symbol of type τ together with a value of that type. To compute with a classified value, it is compared with a known class. If the value is of this class, the underlying instance data are passed to the positive branch; otherwise the negative branch is taken, where it may be matched against other known classes.

34.1.1 Statics

The syntax of the language `clsfd` of dynamic classification is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	$::=$	`clsfd`	`clsfd`	classified
Exp	e	$::=$	`in[`a`](`e`)`	$a \cdot e$	instance
			`isin[`a`](`e`;`x`.`e_1`;`e_2`)`	`match` e `as` $a \cdot x \Rightarrow e_1$ `ow` $\Rightarrow e_2$	comparison

The expression `in[`a`](`e`)` is a classified value with class a and underlying value e. The expression `isin[`a`](`e`;`x`.`e_1`;`e_2`)` checks whether the class of the value given by e is a. If so, the classified value is passed to e_1; if not, the expression e_2 is evaluated instead.

The statics of `clsfd` is defined by the following rules:

$$\frac{\Gamma \vdash_{\Sigma, a \sim \rho} e : \rho}{\Gamma \vdash_{\Sigma, a \sim \rho} \texttt{in[}a\texttt{](}e\texttt{)} : \texttt{clsfd}} \tag{34.1a}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \rho} e : \texttt{clsfd} \quad \Gamma, x : \rho \vdash_{\Sigma, a \sim \rho} e_1 : \tau \quad \Gamma \vdash_{\Sigma, a \sim \rho} e_2 : \tau}{\Gamma \vdash_{\Sigma, a \sim \rho} \texttt{isin[}a\texttt{](}e\texttt{;}x\texttt{.}e_1\texttt{;}e_2\texttt{)} : \tau}. \tag{34.1b}$$

The type associated with the symbol in the signature determines the type of the instance data.

34.1.2 Dynamics

To maximize the flexibility in the use of dynamic classification, we give only a free dynamics for symbol generation. Within this framework the dynamics of classification is given by the following rules:

$$\frac{e \ \mathsf{val}_\Sigma}{\texttt{in[}a\texttt{](}e\texttt{)} \ \mathsf{val}_\Sigma} \tag{34.2a}$$

$$\frac{\nu \ \Sigma \ \{ e \} \mapsto \nu \ \Sigma' \ \{ e' \}}{\nu \ \Sigma \ \{ \texttt{in[}a\texttt{](}e\texttt{)} \} \mapsto \nu \ \Sigma' \ \{ \texttt{in[}a\texttt{](}e'\texttt{)} \}} \tag{34.2b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\,\mathtt{isin}[a]\,(\mathtt{in}[a]\,(e);x\,.\,e_1;e_2)\,\}\mapsto \nu\ \Sigma\ \{\,[e/x]e_1\,\}} \tag{34.2c}$$

$$\frac{(a\neq a')}{\nu\ \Sigma\ \{\,\mathtt{isin}[a]\,(\mathtt{in}[a']\,(e');x\,.\,e_1;e_2)\,\}\mapsto \nu\ \Sigma\ \{\,e_2\,\}} \tag{34.2d}$$

$$\frac{\nu\ \Sigma\ \{\,e\,\}\mapsto \nu\ \Sigma'\ \{\,e'\,\}}{\nu\ \Sigma\ \{\,\mathtt{isin}[a]\,(e;x\,.\,e_1;e_2)\,\}\mapsto \nu\ \Sigma'\ \{\,\mathtt{isin}[a]\,(e';x\,.\,e_1;e_2)\,\}}\,. \tag{34.2e}$$

Throughout, if the states involved are well-formed, then there will be a declaration $a\sim\tau$ for some type τ in Σ.

The dynamics of the elimination form for the type `clsfd` relies on *disequality* of names [specifically, Rule (34.2d)]. Because disequality is not preserved under substitution, it is not sensible to consider any language construct whose dynamics relies on such a substitution. To see what goes wrong, consider the expression

$$\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,a\cdot\_\Rightarrow\mathtt{true}\,\mathtt{ow}\Rightarrow\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{false}\,\mathtt{ow}\Rightarrow\mathtt{true}.$$

This is easily seen to evaluate to `false`, because the outer conditional is on the class a, which is a priori different from b. However, if we substitute b for a in this expression we obtain

$$\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{true}\,\mathtt{ow}\Rightarrow\mathtt{match}\,b\cdot\langle\rangle\,\mathtt{as}\,b\cdot\_\Rightarrow\mathtt{false}\,\mathtt{ow}\Rightarrow\mathtt{true},$$

which evaluates to `true`, because now the outer conditional governs the evaluation.

34.1.3 Safety

Theorem 34.1 (Safety)**.**

1. *If* $\vdash_\Sigma e:\tau$ *and* $\nu\ \Sigma\ \{\,e\,\}\mapsto \nu\ \Sigma'\ \{\,e'\,\}$*, then* $\Sigma'\supseteq\Sigma$ *and* $\vdash_{\Sigma'} e':\tau$.
2. *If* $\vdash_\Sigma e:\tau$*, then either* $e\ \mathsf{val}_\Sigma$ *or* $\nu\ \Sigma\ \{\,e\,\}\mapsto \nu\ \Sigma'\ \{\,e'\,\}$ *for some* e' *and* Σ'.

Proof Similar to the safety proofs given in Chapters 12, 13, and 32. □

34.2 Class References

The type `class(`τ`)` has as values references to classes:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`class(`τ`)`	τ `class`	class reference
Exp	e	::=	`cls[`a`]`	`&`a	reference
			`mk(`e_1`;`e_2`)`	`mk(`e_1`;`e_2`)`	instance
			`isofcls(`e_0`;`e_1`;`x`.`e_2`;`e_3`)`	`isofcls(`e_0`;`e_1`;`x`.`e_2`;`e_3`)`	dispatch

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

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{cls}[a] : \mathtt{class}(\tau)} \tag{34.3a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{class}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{mk}(e_1; e_2) : \mathtt{clsfd}} \tag{34.3b}$$

$$\frac{\Gamma \vdash_\Sigma e_0 : \mathtt{class}(\rho) \quad \Gamma \vdash_\Sigma e_1 : \mathtt{clsfd} \quad \Gamma, x : \rho \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma e_3 : \tau}{\Gamma \vdash_\Sigma \mathtt{isofcls}(e_0; e_1; x\,.\,e_2; e_3) : \tau}\,. \tag{34.3c}$$

The corresponding dynamics is given by these rules:

$$\frac{\nu\ \Sigma\ \{\, e_1 \,\} \mapsto \nu\ \Sigma'\ \{\, e_1' \,\}}{\nu\ \Sigma\ \{\, \mathtt{mk}(e_1; e_2) \,\} \mapsto \nu\ \Sigma'\ \{\, \mathtt{mk}(e_1'; e_2) \,\}} \tag{34.4a}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad \nu\ \Sigma\ \{\, e_2 \,\} \mapsto \nu\ \Sigma'\ \{\, e_2' \,\}}{\nu\ \Sigma\ \{\, \mathtt{mk}(e_1; e_2) \,\} \mapsto \nu\ \Sigma'\ \{\, \mathtt{mk}(e_1; e_2') \,\}} \tag{34.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, \mathtt{mk}(\mathtt{cls}[a]; e) \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{in}[a]\,(e) \,\}} \tag{34.4c}$$

$$\frac{\nu\ \Sigma\ \{\, e_0 \,\} \mapsto \nu\ \Sigma'\ \{\, e_0' \,\}}{\nu\ \Sigma\ \{\, \mathtt{isofcls}(e_0; e_1; x\,.\,e_2; e_3) \,\} \mapsto \nu\ \Sigma'\ \{\, \mathtt{isofcls}(e_0'; e_1; x\,.\,e_2; e_3) \,\}} \tag{34.4d}$$

$$\frac{}{\nu\ \Sigma\ \{\, \mathtt{isofcls}(\mathtt{cls}[a]; e_1; x\,.\,e_2; e_3) \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{isin}[a]\,(e_1; x\,.\,e_2; e_3) \,\}}\,. \tag{34.4e}$$

Rules (34.4d) and (34.4e) specify that the first argument is evaluated to determine the target class, which is then used to check whether the second argument, a classified data value, is of the target class. This may be seen as a two-stage pattern-matching process in which evaluation of e_0 determines the pattern against which to match the classified value of e_1.

34.3 Definability of Dynamic Classes

The type `clsfd` may be defined in terms of symbolic references, product types, and existential types by the type expression

$$\mathtt{clsfd} \triangleq \exists(t\,.\,t\ \mathtt{sym} \times t)\,.$$

The introductory form $\mathtt{in}[a]\,(e)$, where a is a symbol whose associated type is ρ and e is an expression of type ρ, is defined to be the package

$$\mathtt{pack}\ \rho\ \mathtt{with}\ \langle \&\, a, e \rangle\ \mathtt{as}\ \exists(t\,.\,t\ \mathtt{sym} \times t)\,.$$

The eliminatory form $\mathtt{isin}[a]\,(e; x\,.\,e_1; e_2)$ is defined in terms of symbol comparison as defined in Chapter 32. Suppose that the overall type of the conditional is τ and that the type associated with the symbol a is ρ. The type of e must be `clsfd`, as previously defined, and

the type of the branches e_1 and e_2 must be τ, with x assumed to be of type ρ in e_1. The conditional is defined to be the expression

$$\texttt{open}\, e\, \texttt{as}\, t\, \texttt{with}\, \langle x, y\rangle : t\, \texttt{sym} \times t\, \texttt{in}\, (e_{body}(y)),$$

where e_{body} is an expression to be defined shortly. The comparison opens the package e, representing the classified value, and decomposes it into a type t, a symbol x of type t `sym` and an underlying value y of type t. The expression e_{body} will have the type $t \to \tau$, so that the application to y is type correct.

The expression e_{body} compares the symbolic reference x with the symbol a of type ρ and yields a value of type $t \to \tau$ regardless of the outcome. It is therefore defined to be the expression

$$\texttt{is}[a]\,[u.u \to \tau]\,(x; e_1'; e_2')$$

where, in accordance with Rule (32.5b), e_1' has type $[\rho/u](u \to \tau) = \rho \to \tau$, and e_2' has type $[t/u](u \to \tau) = t \to \tau$. The expression e_1' "knows" that the abstract type t is ρ, the type associated with the symbol a, because the comparison has come out positively. However, e_2' does not "learn" anything about the identity of t.

It remains to choose the expressions e_1' and e_2'. In the case of a positive comparison, we wish to pass the classified value to the expression e_1 by substitution for the variable x. This is accomplished by defining e_1' to be the expression

$$\lambda\,(x:\rho)\, e_1 : \rho \to \tau.$$

In the case of a negative comparison no value is to be propagated to e_2. We therefore define e_2' to be the expression

$$\lambda\,(\_:t)\, e_2 : t \to \tau.$$

We may then check that the statics and dynamics given in Section 34.1 are derivable, given the definitions of the type of classified values and their introductory and eliminator forms.

34.4 Classifying Secrets

Dynamic classification may be used to enforce *confidentiality* and *integrity* of data values in a program. A value of type `clsfd` may be constructed only by *sealing* it with some class a and may be deconstructed only by a case analysis that includes a branch for a. By controlling which parties in a multiparty interaction have access to the classifier a, we may control how classified values are created (ensuring their integrity) and how they are inspected (ensuring their confidentiality). Any party that lacks access to a cannot decipher a value classified by a, nor may it create a classified value with this class. Because classes are dynamically generated symbols, they provide an absolute confidentiality guarantee among parties in a computation.[2]

[2] Of course, this guarantee is for programs written in conformance with the statics given here. If the abstraction imposed by the type system is violated, no guarantees of confidentiality can be made.

Consider the following simple protocol for controlling the integrity and confidentiality of data in a program. A fresh symbol a is introduced, and we return a pair of functions of type

$$(\tau \to \texttt{clsfd}) \times (\texttt{clsfd} \to \tau\ \texttt{opt}),$$

called the *constructor* and *destructor* functions for that class, which is accomplished by writing

```
newsym a:τ in
  ⟨ λ(x:τ) a·x,
    λ(x:clsfd) match x as a·y ⇒ just(y) ow ⇒ null ⟩.
```

The first function creates a value classified by a, and the second function recovers the instance data of a value classified by a. Outside of the scope of the declaration the symbol a is an absolutely unguessable secret.

To enforce the integrity of a value of type τ, it is sufficient to ensure that only trusted parties have access to the constructor. To enforce the confidentiality of a value of type τ, it is sufficient to ensure that only trusted parties have access to the destructor. Ensuring the integrity of a value amounts to associating an invariant with it that is maintained by the trusted parties that may create an instance of that class. Ensuring the confidentiality of a value amounts to propagating the invariant to parties that may decipher it.

34.5 Notes

Dynamic classification appears in Standard ML (Milner et al., 1997) as the type `exn` of exception values. The utility of this type is obscured by a too-close association with its application to exception values. The use of dynamic classification to control information flow was popularized in the π-calculus (Milner, 1999) in the form of "channel passing." (See Chapter 42 for more on this correspondence.)

PART XIII

State

35 Modernized Algol

Modernized Algol, or $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$, is an imperative, block-structured programming language based on the classic language Algol. $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ may be seen as an extension to $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with a new syntactic sort of commands that act on assignables by retrieving and altering their contents. Assignables are introduced by *declaring* them for use within a specified scope; this is the essence of block structure. Commands may be combined by sequencing and may be iterated by recursion.

$\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ maintains a careful separation between *pure* expressions, whose meaning does not depend on any assignables, and *impure* commands, whose meaning is given in terms of assignables. This ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, and allows for expressions to be manipulated, much as in PCF. Commands, on the other hand, have a tightly constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is that it adheres to the stack discipline, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline. This avoids the need for more complex forms of storage management, at the expense of reducing the expressiveness of the language.

35.1 Basic Commands

The syntax of the language $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ of modernized Algol distinguishes pure *expressions* from impure *commands*. The expressions include those of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ (as described in Chapter 10), augmented with one additional construct, and the commands are those of a simple imperative programming language based on assignment. The language maintains a sharp distinction between *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 are, for the time being, restricted to natural numbers. Expressions evaluate to values and have no effect on assignables. Commands are executed for their effect on assignables and also return a value. Composition of commands not only sequences their execution order, but also passes the value returned by the first to the second before it is executed. The returned value of a command is, for the time being, restricted to the natural numbers. (But see Section 35.3 for the general case.)

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is given by the following grammar, from which repetition of the expression syntax of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ has been omitted for the sake of brevity:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`cmd`	`cmd`	command
Exp	e	::=	`cmd(`m`)`	`cmd` m	encapsulation
Cmd	m	::=	`ret(`e`)`	`ret` e	return
			`bnd(`e`;`x`.`m`)`	`bnd` $x \leftarrow e$ `;` m	sequence
			`dcl(`e`;`a`.`m`)`	`dcl` $a := e$ `in` m	new assignable
			`get[`a`]`	`@`a	fetch
			`set[`a`](`e`)`	$a := e$	assign

The expression `cmd(`m`)` consists of the unevaluated command m, thought of as a value of type `cmd`. The command `ret(`e`)` returns the value of the expression e without having any effect on the assignables. The command `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 `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 `get[`a`]` returns the current contents of the assignable a, and the command `set[`a`](`e`)` changes the contents of the assignable a to the value of e, and returns that value.

35.1.1 Statics

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ consists of two forms of judgment:

1. Expression typing: $\Gamma \vdash_\Sigma e : \tau$.
2. Command formation: $\Gamma \vdash_\Sigma m$ ok.

The context Γ specifies the types of variables, as usual, and the signature Σ consists of a finite set of assignables. These judgments are inductively defined by the following rules:

$$\frac{\Gamma \vdash_\Sigma m \text{ ok}}{\Gamma \vdash_\Sigma \texttt{cmd(}m\texttt{)} : \texttt{cmd}} \tag{35.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{nat}}{\Gamma \vdash_\Sigma \texttt{ret(}e\texttt{)} \text{ ok}} \tag{35.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd} \quad \Gamma, x : \texttt{nat} \vdash_\Sigma m \text{ ok}}{\Gamma \vdash_\Sigma \texttt{bnd(}e\texttt{;}x\texttt{.}m\texttt{)} \text{ ok}} \tag{35.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{nat} \quad \Gamma \vdash_{\Sigma,a} m \text{ ok}}{\Gamma \vdash_\Sigma \texttt{dcl(}e\texttt{;}a\texttt{.}m\texttt{)} \text{ ok}} \tag{35.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \texttt{get[}a\texttt{]} \text{ ok}} \tag{35.1e}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \texttt{nat}}{\Gamma \vdash_{\Sigma,a} \texttt{set[}a\texttt{](}e\texttt{)} \text{ ok}}\,. \tag{35.1f}$$

Rule (35.1a) is the introductory rule for the type `cmd`, and Rule (35.1c) is the corresponding eliminatory form. Rule (35.1d) introduces a new assignable for use within a specified command. The name a of the assignable is bound by the declaration and hence may be renamed to satisfy the implicit constraint that it not already be present in Σ. Rule (35.1e) states that the command to retrieve the contents of an assignable a returns a natural number. Rule (35.1f) states that we may assign a natural number to an assignable.

35.1.2 Dynamics

The dynamics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ is defined in terms of a *memory* μ, a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgment forms:

1. $e\ \mathsf{val}_\Sigma$, stating that e is a value relative to Σ,
2. $e \underset{\Sigma}{\mapsto} e'$, stating that the expression e steps to the expression e'.

These judgments are inductively defined by the following rules, together with the rules defining the dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ (see Chapter 10). It is important, however, that the successor operation be given an *eager*, rather than *lazy*, dynamics so that a closed value of type `nat` is a numeral (for reasons that are explained in Section 35.3).

$$\frac{}{\texttt{cmd}(m)\ \mathsf{val}_\Sigma}\,. \tag{35.2}$$

Rule (35.2) states that an encapsulated command is a value.

The dynamics of commands is defined in terms of states $m \parallel \mu$, where μ is a memory mapping assignables to values and m is a command. There are two judgments governing such states:

1. $m \parallel \mu\ \mathsf{final}_\Sigma$. The state $m \parallel \mu$ is fully executed.
2. $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$. The state $m \parallel \mu$ steps to the state $m' \parallel \mu'$; the set of active assignables is given by the signature Σ.

These judgments are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{ret}(e) \parallel \mu\ \mathsf{final}_\Sigma} \tag{35.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{ret}(e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{ret}(e') \parallel \mu} \tag{35.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{bnd}(e; x\,.\,m) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{bnd}(e'; x\,.\,m) \parallel \mu} \tag{35.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{bnd(cmd(ret(}e\mathtt{))};x\,.\,m\mathtt{)} \parallel \mu \underset{\Sigma}{\mapsto} [e/x]m \parallel \mu} \tag{35.3d}$$

$$\frac{m_1 \parallel \mu \underset{\Sigma}{\mapsto} m_1' \parallel \mu'}{\mathtt{bnd(cmd(}m_1\mathtt{);}x\,.\,m_2\mathtt{)} \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{bnd(cmd(}m_1'\mathtt{);}x\,.\,m_2\mathtt{)} \parallel \mu'} \tag{35.3e}$$

$$\frac{}{\mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} \mathtt{ret(}e\mathtt{)} \parallel \mu \otimes a \hookrightarrow e} \tag{35.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{set}[a]\mathtt{(}e\mathtt{)} \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{set}[a]\mathtt{(}e'\mathtt{)} \parallel \mu} \tag{35.3g}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{set}[a]\mathtt{(}e\mathtt{)} \parallel \mu \otimes a \hookrightarrow \_ \underset{\Sigma}{\mapsto} \mathtt{ret(}e\mathtt{)} \parallel \mu \otimes a \hookrightarrow e} \tag{35.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{dcl(}e\mathtt{;}a\,.\,m\mathtt{)} \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl(}e'\mathtt{;}a\,.\,m\mathtt{)} \parallel \mu} \tag{35.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad m \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes a \hookrightarrow e'}{\mathtt{dcl(}e\mathtt{;}a\,.\,m\mathtt{)} \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl(}e'\mathtt{;}a\,.\,m'\mathtt{)} \parallel \mu'} \tag{35.3j}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad e'\ \mathsf{val}_{\Sigma,a}}{\mathtt{dcl(}e\mathtt{;}a\,.\,\mathtt{ret(}e'\mathtt{))} \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret(}e'\mathtt{)} \parallel \mu}\,. \tag{35.3k}$$

Rule (35.3a) specifies that a `ret` command is final if its argument is a value. Rules (35.3c)–(35.3e) specify the dynamics of sequential composition. The expression e must, by virtue of the type system, evaluate to an encapsulated command, which is to be executed to determine its return value, which is then substituted into m before executing it.

Rules (35.3i)–(35.3k) define the concept of *block structure* in a programming language. Declarations adhere to the *stack discipline* in that an assignable is allocated for the duration of evaluation of the body of the declaration and deallocated after evaluation of the body is complete. Therefore the lifetime of an assignable can be identified with its scope, and hence we may visualize the dynamic lifetimes of assignables as being nested inside one another, in the same manner as their static scopes are nested inside one another. This stacklike behavior of assignables is a characteristic feature of what are known as *Algol-like languages*.

35.1.3 Safety

The judgment $m \parallel \mu\ \mathsf{ok}_\Sigma$ is defined by the rule

$$\frac{\vdash_\Sigma m\ \mathsf{ok} \quad \mu : \Sigma}{m \parallel \mu\ \mathsf{ok}_\Sigma} \tag{35.4}$$

where the auxiliary judgment $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e \ \mathsf{val}_\emptyset \text{ and } \vdash_\emptyset e : \mathtt{nat}}{\mu : \Sigma}. \tag{35.5}$$

That is, the memory must bind a number to each location in Σ.

Theorem 35.1 (Preservation)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$.
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$*, with* $\vdash_\Sigma m \ \mathsf{ok}$ *and* $\mu : \Sigma$*, then* $\vdash_\Sigma m' \ \mathsf{ok}$ *and* $\mu' : \Sigma$.

Proof Simultaneously, by induction on Rules (35.2) and (35.3).

Consider Rule (35.3j). 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 (35.3k). Assume that $\vdash_\Sigma \mathtt{dcl}(e; a.\mathtt{ret}(e')) \ \mathsf{ok}$ and $\mu : \Sigma$. By inversion we have $\vdash_\Sigma e : \mathtt{nat}$, $\vdash_{\Sigma,a} \mathtt{ret}(e') \ \mathsf{ok}$, and hence that $\vdash_{\Sigma,a} e' : \mathtt{nat}$. But because $e' \ \mathsf{val}_{\Sigma,a}$, e' is a numeral, and hence we also have $\vdash_\Sigma e' : \mathtt{nat}$, as required. □

Theorem 35.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* $e \ \mathsf{val}_\Sigma$*, or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.
2. *If* $\vdash_\Sigma m \ \mathsf{ok}$ *and* $\mu : \Sigma$*, then either* $m \parallel \mu \ \mathsf{final}_\Sigma$ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'.

Proof Simultaneously, by induction on Rules (35.1). Consider Rule (35.1d). By the first inductive hypothesis we have either $e \underset{\Sigma}{\mapsto} e'$ or $e \ \mathsf{val}_\Sigma$. In the former case Rule (35.3i) applies. In the latter, we have by the second inductive hypothesis,

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

In the former case we apply Rule (35.3k), and in the latter, Rule (35.3j). □

35.2 Some Programming Idioms

The language $\mathcal{L}\{\mathtt{nat\ cmd} \rightharpoonup\}$ is designed to expose the elegant interplay between the execution of an expression for its value and the execution of a command for its effect on assignables. This section shows how to derive several standard idioms of imperative programming in $\mathcal{L}\{\mathtt{nat\ cmd} \rightharpoonup\}$.

We define the *sequential composition* of commands, written as $\{x \leftarrow m_1 \,;\, m_2\}$, to stand for the command $\texttt{bnd}\, x \leftarrow \texttt{cmd}\,(m_1)\,;\, m_2$. This generalizes to an n-ary form by defining

$$\{x_1 \leftarrow m_1 \,;\, \ldots x_{n-1} \leftarrow m_{n-1} \,;\, m_n\}$$

to stand for the iterated composition

$$\{x_1 \leftarrow m_1 \,;\, \ldots \{x_{n-1} \leftarrow m_{n-1} \,;\, m_n\}\}.$$

We sometimes write just $\{m_1 \,;\, m_2\}$ for the composition $\{\_ \leftarrow m_1 \,;\, m_2\}$ in which the returned value from m_1 is ignored; this generalizes in the obvious way to an n-ary form.

A related idiom, the command $\texttt{do}\, e$, 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:

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

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 nonzero, 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
    }
    ; @ r
  }
}
```

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 on completion of the loop the assignable a contains 0 and r contains the factorial of x, as required.

35.3 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 lies in the details of the proof of Theorem 35.1. The proof of this theorem relies on an eager interpretation of the successor to ensure that the value is well-typed even in the absence of the locally declared assignable a. The proof breaks down, and indeed the preservation theorem is false, when the return type of a command or the contents type of an assignable is unrestricted.

For example, if we may return values of procedure type, then 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 arbitrary type is also unsound. For example, suppose that b is an assignable whose contents are of type $\texttt{nat} \to \texttt{unit}$, and consider the command

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

We assign to b a procedure that uses a locally declared assignable a and then leaves the scope of the declaration. If we then call the procedure stored in b, execution will get stuck attempting to assign to the nonexistent assignable a.

To admit declarations to return values and to admit assignables of types other than nat, we must rework the statics of $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup\}$ to record the returned type of a command and to record the type of the contents of each assignable. First, we generalize the finite set Σ of active assignables to assign a type to each active assignable so that Σ has the form of a finite set of assumptions of the form $a \sim \tau$, where a is an assignable. Second, we replace the judgment $\Gamma \vdash_\Sigma m\ \texttt{ok}$ by the more general form $\Gamma \vdash_\Sigma m \sim \tau$, stating that m is a well-formed command returning a value of type τ. Third, the type cmd must be generalized to $\texttt{cmd}(\tau)$, which is written in examples as τ cmd, to specify the return type of the encapsulated command.

The statics given in Subsection 35.1.1 may be generalized to admit typed commands and typed assignables, as follows:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}(\tau)} \tag{35.6a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \sim \tau} \tag{35.6b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau) \quad \Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x\,.\,m) \sim \tau'} \tag{35.6c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \ \mathsf{mobile} \quad \Gamma \vdash_{\Sigma, a \sim \tau} m \sim \tau'}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a\,.\,m) \sim \tau'} \tag{35.6d}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] \sim \tau} \tag{35.6e}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{set}[a](e) \sim \tau}\,. \tag{35.6f}$$

Apart from the generalization to track returned types and content types, the most important change is to require that the types of assignables and of returned values be mobile.

As in Chapter 32, these rules make use of the judgment τ mobile, which states that the type τ is *mobile*. The definition of this judgment is guided by the following *mobility condition*:

$$\textit{if } \tau \ \mathsf{mobile},\ \vdash_\Sigma e : \tau \textit{ and } e\ \mathsf{val}_\Sigma,\ \textit{then } \vdash_\emptyset e : \tau \textit{ and } e\ \mathsf{val}_\emptyset. \tag{35.7}$$

That is, a value of mobile type does not depend on any active assignables.

Because the successor is evaluated eagerly, the type `nat` may be deemed mobile:

$$\frac{}{\mathtt{nat}\ \mathsf{mobile}}\,. \tag{35.8}$$

Because the body of a procedure may involve an assignable, no procedure type may be considered mobile, nor may the type of commands returning a given type, for similar reasons. However, a product of mobile types may safely be deemed mobile, provided that pairing is evaluated eagerly:

$$\frac{\tau_1 \ \mathsf{mobile} \quad \tau_2 \ \mathsf{mobile}}{\tau_1 \times \tau_2 \ \mathsf{mobile}}\,. \tag{35.9}$$

Similarly, sums may be deemed mobile so long as the injections are evaluated eagerly:

$$\frac{\tau_1 \ \mathsf{mobile} \quad \tau_2 \ \mathsf{mobile}}{\tau_1 + \tau_2 \ \mathsf{mobile}}\,. \tag{35.10}$$

Laziness defeats mobility, because values may contain suspended computations that depend on an assignable. For example, if the successor operation for the natural numbers were evaluated lazily, then $\mathtt{s}(e)$ would be a value for any expression e, including one that refers to an assignable a. Similarly, if pairing were lazy, then products may not be deemed mobile, and if injections were evaluated lazily, then sums may not either.

What about function types other than procedure types? We may at first think they are mobile, because a pure expression cannot depend on an assignable. Although this is indeed the case, the mobility condition need not hold. For example, consider the following value of type $\texttt{nat} \rightharpoonup \texttt{nat}$:

$$\lambda\,(x\!:\!\texttt{nat})\,(\lambda\,(\_\!:\!\texttt{cmd})\,\texttt{z})(\texttt{cmd}\,\{@\,a\}).$$

Although the assignable a is not actually needed to compute the result, it nevertheless occurs in the value, in violation of the safety condition.

The mobility restriction on the statics of assignable declaration 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 35.3 (Preservation for Typed Commands)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_\Sigma e : \tau$*, then* $\vdash_\Sigma e' : \tau$*.*
2. *If* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$*, with* $\vdash_\Sigma m \sim \tau$ *and* $\mu : \Sigma$*, then* $\vdash_\Sigma m' \sim \tau$ *and* $\mu' : \Sigma$*.*

Theorem 35.4 (Progress for Typed Commands)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* e val_Σ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$*.*
2. *If* $\vdash_\Sigma m \sim \tau$ *and* $\mu : \Sigma$*, then either* $m \parallel \mu$ final_Σ *or* $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ *for some* μ' *and* m'*.*

The proofs of Theorems 35.3 and 35.4 follow very closely the proofs of Theorems 35.1 and 35.2. 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.

35.4 Notes

Modernized Algol is essentially a reformulation of Idealized Algol (Reynolds, 1981) in which we have maintained a clearer separation between computations that depend on the store and those that do not. The modal distinction between expressions and commands was present in the original formulation of Algol 60. The same separation has been popularized by Haskell, under the rubric "the `IO` monad."

What are called here *assignables* are regrettably called *variables* in the programming language literature. This clash of terminology is the source of considerable confusion and misunderstanding. It is preferable to retain the well-established meaning of a variable as standing for an unspecified object of a specified type, but to do so requires that we invent a new word for the name of a piece of mutable storage. The word assignable seems apt, and equally as convenient as the misappropriated word variable.

In Idealized Algol, as in the original, an assignable may be used as a form of expression standing for its current contents. Although syntactically convenient, this convention

introduces an unfortunate dependency of expression evaluation on the store that we avoid here.

The concept of mobility of a type was introduced in the ML5 language for distributed computing (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.

36 Assignable References

A reference to an assignable a is a value, written as $\&a$, of *reference type* that uniquely 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 at the point at which it is used. Two references may also 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 said to be *aliases*. The possibility of aliasing complicates reasoning about the correctness of code that uses references, for we must always consider for any two references whether they might be aliases.

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 must be limited to the scope of the assignable to which it refers. This may be achieved by declaring that reference types are immobile, so that they cannot be returned from the body of a declaration or stored in an assignable. Although ensuring adherence to the stack discipline, this restriction precludes the use of 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 the creation of cyclic (self-referential) structures. 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 may be regarded as mobile, and hence values of any type may be stored in assignables or returned from commands.

36.1 Capabilities

The commands $\texttt{get}[a]$ and $\texttt{set}[a](e)$ in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$ operate on a statically specified assignable a. To even write these commands requires that the assignable a be in scope at the point 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 easily 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 provide the procedure with 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

$$\tau\ \texttt{cmd} \times (\tau \rightharpoonup \tau\ \texttt{cmd})$$

given by the pair

$$\langle \texttt{cmd}\ (\texttt{@}\, a), \texttt{proc}\ (x : \tau)\ a := x \rangle.$$

Because a capability type is a product of a command type and a procedure type, no capability type is mobile. This means, in particular, that a capability cannot be returned from a command or stored into an assignable. This is as it should be, for otherwise we would violate the stack discipline for allocating assignables.

Using capabilities, we may program the proposed generic doubling procedure as follows:

$$\texttt{proc}\ (\langle \mathit{get}, \mathit{set} \rangle : \texttt{nat cmd} \times (\tau \rightharpoonup \tau\ \texttt{cmd}))\ \{x \leftarrow \texttt{do}\ \mathit{get}\ ;\ y \leftarrow \texttt{do}\ (\mathit{set}(x + x))\ ;\ \texttt{ret}\ x\}.$$

The procedure is to be 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 directly accessible by this procedure; the capability provided by the caller comprises the commands required to get and set a.

36.2 Scoped Assignables

A weakness of using a capability to provide indirect access to an assignable is that there is no guarantee that a given getter–setter pair is in fact the capability for a particular assignable. For example, we might (deliberately or accidentally) pair the getter for a with the setter for b, leading to unexpected behavior. There is nothing in the type system that prevents the creation of 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 may be compared for equality to determine whether or not they act on the same assignable.[1] The *reference type* $\texttt{ref}\,(\tau)$ has as values references to assignables

[1] The getter and setter are not quite sufficient to define equality, because not all types admit a run-time equality test. When they do, and when there are at least two distinct values of the contents type, we can determine whether they are aliases by assigning to one and checking whether the contents of the other is changed.

of type τ. The introduction and elimination forms for this type are given by the following syntax chart:

Sort			Abstract Form	Concrete Form	Description
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

The statics of reference types is defined by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)} \tag{36.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{ref}(\tau)}{\Gamma \vdash_\Sigma \mathtt{getref}(e) \sim \tau} \tag{36.1b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{ref}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{setref}(e_1;e_2) \sim \tau}\,. \tag{36.1c}$$

Rule (36.1a) specifies that the name of any active assignable is an expression of type $\mathtt{ref}(\tau)$.

The dynamics of reference types simply defers to the corresponding operations on assignables and does not alter the underlying dynamics of assignables:

$$\frac{}{\mathtt{ref}[a]\ \mathsf{val}_{\Sigma,a}} \tag{36.2a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{getref}(e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{getref}(e') \parallel \mu} \tag{36.2b}$$

$$\frac{}{\mathtt{getref}(\mathtt{ref}[a]) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{get}[a] \parallel \mu} \tag{36.2c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{setref}(e_1;e_2) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{setref}(e_1';e_2) \parallel \mu} \tag{36.2d}$$

$$\frac{}{\mathtt{setref}(\mathtt{ref}[a];e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{set}[a](e) \parallel \mu} \tag{36.2e}$$

A reference to an assignable is a value. The `getref` and `setref` operations on references defer to the corresponding operations on assignables once the reference has been determined.

Because references give rise to capabilities, the reference type is deemed to be immobile. Consequently, references cannot be stored in assignables or returned from commands. This ensures safety, as may be readily verified by extending the proof given in Chapter 35.

As an example of the use of references, the generic doubling procedure discussed in the preceding section may be programmed with 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. When handling two or more references at the same time, we must consider the possibility that they are aliases, which is to say that both refer to the same underlying assignable. Consider, for example, a procedure that, when given two references x and y adds twice the contents of y to the contents of x. One way to write this code creates no complications:

$$\lambda\,(x\!:\!\mathtt{nat\,ref})\;\lambda\,(y\!:\!\mathtt{nat\,ref})\;\mathtt{cmd}\,\{x' \leftarrow *\,x\;;\; y' \leftarrow *\,y\;;\; x := x' + y' + y'\}.$$

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 orginal contents and twice the contents of the assignable referenced by y.

But now consider the following apparently equivalent implementation of this procedure:

$$\lambda\,(x\!:\!\mathtt{nat\,ref})\;\lambda\,(y\!:\!\mathtt{nat\,ref})\;\mathtt{cmd}\,\{x \mathtt{+=} y\;;\; x \mathtt{+=} y\},$$

where x += 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 properly provided that x and y do not refer to the same assignable. For if they both reference the same assignable a, with contents n, the result is that a is to set $4 \times n$, instead of the intended $3 \times n$, because the second get of y is influenced by the first assignment to x.

In this case it is entirely obvious how to avoid the problem: Use the first implementation, rather than the second. But the difficulty is not in fixing the problem once it has been uncovered, but rather 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 at a given point of a computation. The problem is that the number of possible aliasing relationships among n references grows quadratically in n, because we must consider all possible pairings.

36.3 Free Assignables

Although it is interesting to note that references and capabilities are compatible with the stack discipline, this is achieved at the expense of their utility. Because references are not mobile, it is not possible to build a data structure containing references internally. In particular, this restriction precludes programming mutable data structures (those whose structure changes during execution).

To allow for more flexible uses of references, we must relax the requirement that assignables be stack-allocated and instead arrange that the lifetime of an assignable extends beyond the scope of its declaration. Such assignables are called *scope–free*, or just *free*,

assignables. If all assignables are scope–free, then every type may safely be deemed mobile. Consequently, references may be used to implement mutable and cyclic data structures.

Supporting free assignables amounts to changing the dynamics so that allocation of assignables persists across transitions. This is achieved by employing transition judgments of the form

$$\nu\ \Sigma\ \{\, m \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, m' \parallel \mu' \,\}.$$

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:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, \mathtt{ret}(e) \parallel \mu \,\}\ \mathsf{final}} \tag{36.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{ret}(e) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{ret}(e') \parallel \mu \,\}} \tag{36.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{bnd}(e; x\,.\,m) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{bnd}(e'; x\,.\,m) \parallel \mu \,\}} \tag{36.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, \mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e)); x\,.\,m) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, [e/x]m \parallel \mu \,\}} \tag{36.3d}$$

$$\frac{\nu\ \Sigma\ \{\, m_1 \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, m_1' \parallel \mu' \,\}}{\nu\ \Sigma\ \{\, \mathtt{bnd}(\mathtt{cmd}(m_1); x\,.\,m_2) \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, \mathtt{bnd}(\mathtt{cmd}(m_1'); x\,.\,m_2) \parallel \mu' \,\}} \tag{36.3e}$$

$$\frac{}{\nu\ \Sigma, a \sim \tau\ \{\, \mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e \,\} \mapsto \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{36.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{set}[a](e) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{set}[a](e') \parallel \mu \,\}} \tag{36.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a \sim \tau}}{\nu\ \Sigma, a \sim \tau\ \{\, \mathtt{set}[a](e) \parallel \mu \otimes a \hookrightarrow \_ \,\} \mapsto \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{36.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{dcl}(e; a\,.\,m) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{dcl}(e'; a\,.\,m) \parallel \mu \,\}} \tag{36.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, \mathtt{dcl}(e; a\,.\,m) \parallel \mu \,\} \mapsto \nu\ \Sigma, a \sim \tau\ \{\, m \parallel \mu \otimes a \hookrightarrow e \,\}}\,. \tag{36.3j}$$

The language $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd}\ \mathtt{ref} \rightharpoonup\}$ extends $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup\}$ with references to free assignables. Its dynamics is similar to that of references to scoped assignables given earlier:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\, \mathtt{getref}(e) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{getref}(e') \parallel \mu \,\}} \tag{36.4a}$$

$$\frac{}{\nu\ \Sigma\ \{\, \mathtt{getref}(\mathtt{ref}[a]) \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, \mathtt{get}[a] \parallel \mu \,\}} \tag{36.4b}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\nu\ \Sigma\ \{\,\mathtt{setref}(e_1;e_2)\ \|\ \mu\,\} \mapsto \nu\ \Sigma\ \{\,\mathtt{setref}(e_1';e_2)\ \|\ \mu\,\}} \tag{36.4c}$$

$$\frac{}{\nu\ \Sigma\ \{\,\mathtt{setref}(\mathtt{ref}[a];e_2)\ \|\ \mu\,\} \mapsto \nu\ \Sigma\ \{\,\mathtt{set}[a](e_2)\ \|\ \mu\,\}}\,. \tag{36.4d}$$

Observe that the evaluation of expressions cannot alter or extend the memory; only commands may do this.

As an illustrative example of the use of references to scope–free assignables, consider the command $\mathtt{newref}[\tau](e)$ defined by

$$\mathtt{dcl}\,a := e\,\mathtt{in}\,\mathtt{ret}\,(\&a). \tag{36.5}$$

This command allocates a fresh assignable and returns a reference to it. Its static and dynamics may be derived from the foregoing rules, as follows:

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{newref}[\tau](e) \sim \mathtt{ref}(\tau)} \tag{36.6}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\ \Sigma\ \{\,\mathtt{newref}[\tau](e)\ \|\ \mu\,\} \mapsto \nu\ \Sigma\ \{\,\mathtt{newref}[\tau](e')\ \|\ \mu\,\}} \tag{36.7a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\,\mathtt{newref}[\tau](e)\ \|\ \mu\,\} \mapsto \nu\ \Sigma, a \sim \tau\ \{\,\mathtt{ret}(\mathtt{ref}[a])\ \|\ \mu \otimes a \hookrightarrow e\,\}}\,. \tag{36.7b}$$

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.

36.4 Safety for Free Assignables

Although the proof of safety for references to scoped assignables presents few difficulties, the safety for free assignables is suprisingly tricky. The main difficulty is to account for the possibility of cyclic dependencies of data structures in memory. The contents of one assignable may contain a reference to itself or a reference to another assignable that contains a reference to it, and so forth. For example, consider the following procedure e of type $\mathtt{nat} \to \mathtt{nat}\,\mathtt{cmd}$:

$$\mathtt{proc}\,(x\!:\!\mathtt{nat})\,\{\mathtt{if}\,(x)\,\mathtt{ret}\,(1)\,\mathtt{else}\,\{f \leftarrow @a\,;\,y \leftarrow f(x-1)\,;\,\mathtt{ret}\,(x * y)\}\}.$$

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 nonzero argument, it will "call itself" by indirect reference through a.

The possibility of cyclic dependencies means that some care in the definition of the judgment $\mu : \Sigma$ is required. The following rule defines the well-formed states:

$$\frac{\vdash_\Sigma m \sim \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\ \Sigma\ \{m\ \|\ \mu\}\ \mathsf{ok}}\,. \tag{36.8}$$

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 the whole of* Σ so that cyclic dependencies are permitted. The judgment $\vdash_{\Sigma'} \mu : \Sigma$ is defined as follows:

$$\frac{\forall a \sim \rho \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \rho}{\vdash_{\Sigma'} \mu : \Sigma} \tag{36.9}$$

Theorem 36.1 (Preservation)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$*, then* $\vdash_\Sigma e' : \tau$.
2. *If* $\nu\ \Sigma\ \{ m \parallel \mu \}$ *ok and* $\nu\ \Sigma\ \{ m \parallel \mu \} \mapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$*, then* $\nu\ \Sigma'\ \{ m' \parallel \mu' \}$ *ok.*

Proof Simultaneously, by induction on transition. We prove the following stronger form of the second statement:

> If $\nu\ \Sigma\ \{ m \parallel \mu \} \mapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$, where $\vdash_\Sigma m \sim \tau$, $\vdash_\Sigma \mu : \Sigma$, then Σ' extends Σ, and $\vdash_{\Sigma'} m' \sim \tau$, and $\vdash_{\Sigma'} \mu' : \Sigma'$.

Consider, for example, the transition

$$\nu\ \Sigma\ \{ \mathtt{dcl}(e; a\,.\,m) \parallel \mu \} \mapsto \nu\ \Sigma, a \sim \rho\ \{ m \parallel \mu \otimes a \hookrightarrow e \}$$

where $e\ \mathsf{val}_\Sigma$. By assumption and inversion of Rule (35.6d) we have ρ such that $\vdash_\Sigma e : \rho$, $\vdash_{\Sigma, a\sim\rho} m \sim \tau$, 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 (36.9) 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 36.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* $e\ \mathsf{val}_\Sigma$ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.
2. *If* $\nu\ \Sigma\ \{ m \parallel \mu \}$ *ok then either* $\nu\ \Sigma\ \{ m \parallel \mu \}$ *final or* $\nu\ \Sigma\ \{ m \parallel \mu \} \mapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$ *for some* Σ'*,* μ'*, and* m'.

Proof Simultaneously, by induction on typing. For the second statement we prove the stronger form:

> If $\vdash_\Sigma m \sim \tau$ and $\vdash_\Sigma \mu : \Sigma$, then either $\nu\ \Sigma\ \{ m \parallel \mu \}$ final, or $\nu\ \Sigma\ \{ m \parallel \mu \} \mapsto \nu\ \Sigma'\ \{ m' \parallel \mu' \}$ for some Σ', μ', and m'.

Consider, for example, the typing rule

$$\frac{\Gamma \vdash_\Sigma e : \rho \quad \Gamma \vdash_{\Sigma, a\sim\rho} m \sim \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a\,.\,m) \sim \tau}\,.$$

We have by the first inductive hypothesis that either $e\ \mathsf{val}_\Sigma$ or $e \underset{\Sigma}{\mapsto} e'$ for some e'. In the latter case we have by Rule (36.3i)

$$\nu\ \Sigma\ \{ \mathtt{dcl}(e; a\,.\,m) \parallel \mu \} \mapsto \nu\ \Sigma\ \{ \mathtt{dcl}(e'; a\,.\,m) \parallel \mu \}.$$

In the former case we have by Rule (36.3j) that

$$\nu\ \Sigma\ \{\,\mathtt{dcl}(e; a\,.\,m)\ \|\ \mu\,\} \mapsto \nu\ \Sigma, a \sim \rho\ \{\,m\ \|\ \mu \otimes a \hookrightarrow e\,\}.$$

As another example, consider the typing rule

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{get}[a] \sim \tau}\,.$$

By assumption $\vdash_{\Sigma,a\sim\tau}\ \mu\ :\ \Sigma, a \sim \tau$, and hence there exists e $\mathsf{val}_{\Sigma,a\sim\tau}$ such that $\mu = \mu' \otimes a \hookrightarrow e$ and $\vdash_{\Sigma,a\sim\tau}\ e : \tau$. By Rule (36.3f),

$$\nu\ \Sigma, a \sim \tau\ \{\,\mathtt{get}[a]\ \|\ \mu' \otimes a \hookrightarrow e\,\} \mapsto \nu\ \Sigma, a \sim \tau\ \{\,\mathtt{ret}(e)\ \|\ \mu' \otimes a \hookrightarrow e\,\},$$

as required. The other cases are handled similarly. □

36.5 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 advantageous in many, perhaps most, situations, it also precludes programming techniques that use storage effects to implement purely functional behavior. A prime example is memorization in a lazy language (which is described in detail in Chapter 37.) 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. Another class of examples is self-adjusting data structures, which use state internally to improve their efficiency without changing their overall purely functional behavior. For example, a splay tree is a binary search tree that uses mutation internally to rebalance the tree as elements are inserted, deleted, and retrieved. This ensures that, for example, lookup operations take 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. Because values are forms of expression, it is essential to relax the strict separation between expressions and commands that characterizes the modal type system for storage effects described in Chapter 35. Although several ad hoc methods have been considered in the literature, the most general approach is to simply do away with the distinction entirely, 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, in addition, engender storage effects during evaluation. The benefit of this approach is that it is straightforward to implement benign effects that are impossible to implement when a strict modal separation between expressions and commands is maintained.

The language $\mathcal{L}\{\mathtt{nat\,ref} \rightharpoonup\}$ is a reformulation of $\mathcal{L}\{\mathtt{nat\,cmd\,ref} \rightharpoonup\}$ in which commands are integrated with expressions. For example, the following rules illustrate the

structure of the statics of $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$:

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a \sim \tau_1} e_2 : \tau_2}{\Gamma \vdash_\Sigma \texttt{dcl}(e_1; a.e_2) : \tau_2} \tag{36.10a}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{get}[a] : \tau} \tag{36.10b}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{set}[a](e) : \tau}. \tag{36.10c}$$

Correspondingly, the dynamics of $\mathcal{L}\{\texttt{nat ref} \rightharpoonup\}$ is given by transitions of the form

$$\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, e' \parallel \mu' \,\},$$

where e is an expression rather than a command. The rules defining the dynamics are very similar to those for $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$, but with commands and expressions integrated into a single category.

To illustrate the concept of a benign effect, consider the technique of *backpatching* to implement recursion. Here is a formulation of the factorial function in $\mathcal{L}\{\texttt{nat ref} \rightharpoonup\}$ in which recursive calls are mediated by an assignable containing the function itself:

```
dcl a := λn:nat.0 in
 { f ← a := λn:nat.ifz(n, 1, n′.n×(@a)(n′))
 ; ret(f)
 }
```

This expression returns a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$ that is obtained by (a) allocating a free assignable initialized arbitrarily (and immaterially) with a function of this type, (b) defining a λ-abstraction in which each "recursive call" consists of retrieving and applying the function stored in that assignable, (c) assigning this function to the assignable, and (d) returning that function. The result is a function on the natural numbers, even though it uses state internally to its implementation.

Backpatching is impossible in $\mathcal{L}\{\texttt{nat cmd ref} \rightharpoonup\}$, because it makes explicit the reliance on state. To see this, let us consider recoding the preceding example in the language with a command modality:

```
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 rather than an expression. Moreover, the result of the command is of procedure type $\texttt{nat} \rightarrow \texttt{(nat cmd)}$ rather than function type

$\texttt{nat} \to \texttt{nat}$. This means that we cannot use the factorial "function" (so implemented) in an expression, but must instead execute it as a command, so that we compute the factorial of n by writing

$$\{\ \texttt{f} \leftarrow \mathit{fact}\texttt{;}\ x \leftarrow \texttt{f(}n\texttt{);}\ \texttt{ret(}x\texttt{)}\ \}.$$

In short, the use of storage effects is exposed, rather than hidden, as is possible in $\mathcal{L}\{\texttt{nat ref} \rightharpoonup\}$.

36.6 Notes

Reynolds (1981) uses capabilities to provide indirect access to assignables; it is a short step from there to references in the form considered here. Often references are considered only for free assignables, but this is not essential. It is perfectly possible to have references to scoped assignables as well, provided that suitable mobility restrictions are imposed to ensure adherence to the stack discipline. The proof of safety of free references given here is inspired by Wright and Felleisen (1994) and Harper (1994).

Benign effects are central to the distinction between Haskell, which is based on an Algol-like separation between commands and expressions, and ML, which is based on the integration of evaluation with execution. Each approach has its advantages and complementary disadvantages; neither is uniformly superior to the other.

PART XIV

Laziness

37 Lazy Evaluation

Lazy evaluation refers to a variety of concepts that seek to *defer* evaluation of an expression until it is definitely required and to *share* the results of any such evaluation among all instances of a single deferred computation. The net result is that a computation is performed at most once among all of its instances. Laziness manifests itself in a number of ways.

One form of laziness is the *by-need* evaluation strategy for function application. Recall from Chapter 8 that the by-name evaluation order passes the argument to a function in unevaluated form so that it is evaluated only if it is actually used. But because the argument is replicated by substitution, it may be evaluated more than once. By-need evaluation ensures that the argument to a function is evaluated at most once by ensuring that all copies of an argument share the result of evaluating any one copy.

Another form of laziness is the concept of a *lazy data structure*. As we have seen in Chapters 11, 12, and 16, we may choose to defer evaluation of the components of a data structure until they are actually required rather than when the data structure is created. But if a component is required more than once, then the same computation will, without further provision, be repeated on each use. To avoid this, the deferred portions of a data structure are shared so an access to one will propagate its result to all occurrences of the same computation.

Yet another form of laziness arises from the concept of general recursion considered in Chapter 10. Recall that the dynamics of general recursion is given by unrolling, which replicates the recursive computation on each use. It would be preferable to share the results of such computation across unrollings. A lazy implementation of recursion avoids such replications by sharing the unrollings.

Traditionally, languages are biased toward either eager or lazy evaluation. Eager languages use a by-value dynamics for function applications and evaluate the components of data structures when they are created. Lazy languages adopt the opposite strategy, preferring a by-name dynamics for functions, and a lazy dynamics for data structures. The overhead of laziness is mitigated by managing sharing to avoid redundancy. Experience has shown, however, that the distinction is better drawn at the level of types rather than at the level of a language design. What is important is to have available both lazy and eager types, so that the programmer may choose which to use in a given situation, rather than having the choice forced by the language designer.

In this chapter we make precise the means by which sharing of computations is achieved in the implementation of laziness. We then isolate these mechanisms into a type of suspended computations whose results are shared across all copies of a given suspension.

37.1 By-Need Dynamics

By-need evaluation of functions uses *memoization* to record the value of a computation so that any future use of the same computation may return the previously computed value (or compute it from scratch if there is none). This is achieved by "naming" each deferred computation with a symbol, which is then used to access its value whenever it is used. A memo table records the deferred computation associated to each symbol until such time as it is evaluated, after which it records the value of that computation. Thus naming implements sharing, and the memo table ensures irredundancy.

The by-need dynamics for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is based on a transition system with states of the form $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$, where Σ is a finite set of hypotheses $a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$ associating types with symbols, e is an expression that may involve the symbols in Σ, and μ maps each symbol declared in Σ to either an expression or a special symbol, $\bullet$, called the *black hole*. (The role of the black hole is subsequently made clear.) As a notational convenience, we employ a bit of legerdemain with the concrete syntax similar to that employed in Chapter 35. Specifically, the concrete syntax for the expression $\texttt{get}[a]$, which fetches the contents of the assignable a, is just $\texttt{@}\,a$, omitting explicit mention of the "get" operation.

The by-need dynamics consists of the following two forms of judgment:

1. $e\ \mathsf{val}_\Sigma$, stating that e is a value that may involve the symbols in Σ.
2. $\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}$, stating that one step of evaluation of the expression e relative to memo table μ with the symbols declared in Σ results in the expression e' relative to the memo table μ' with symbols declared in Σ'.

The dynamics is defined so that the collection of active symbols grows monotonically and so the type of a symbol never changes. The memo table may be altered destructively during execution to reflect progress in the evaluation of the expression associated with a given symbol.

The judgment $e\ \mathsf{val}_\Sigma$, expressing that e is a closed value, is defined by the following rules:

$$\frac{}{\texttt{z}\ \mathsf{val}_\Sigma} \tag{37.1a}$$

$$\frac{}{\texttt{s}(a)\ \mathsf{val}_{\Sigma, a\sim\texttt{nat}}} \tag{37.1b}$$

$$\frac{}{\lambda\ (x:\tau)\ e\ \mathsf{val}_\Sigma}\,. \tag{37.1c}$$

Rules (37.1a)–(37.1c) specify that $\texttt{z}$ is a value, any expression of the form $\texttt{s}(a)$, where a is a symbol, is a value, and that any λ-abstraction, possibly containing symbols, is a value. It is important that symbols themselves are not values, rather they stand for (possibly unevaluated) expressions as specified by the memo table. The expression $\texttt{@}\,a$, which is short for get[a] is *not* closed. Rather, it must be evaluated to determine, and possibly update, the binding of the symbol a in memory.

The initial and final states of evaluation are defined as follows:

$$\frac{}{\nu\,\emptyset\,\{\,e \parallel \emptyset\,\}\ \mathsf{initial}} \tag{37.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\,e \parallel \mu\,\}\ \mathsf{final}}\,. \tag{37.2b}$$

Rule (37.2a) specifies that an initial state consists of an expression evaluated relative to an empty memo table. Rule (37.2b) specifies that a final state has the form $\nu\ \Sigma\ \{\,e \parallel \mu\,\}$, where e is a value relative to Σ.

The transition judgment for the by-need dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\nu\ \Sigma, a\sim\tau\,\{\,a \parallel \mu \otimes a \hookrightarrow e\,\} \mapsto \nu\ \Sigma, a\sim\tau\,\{\,e \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.3a}$$

$$\frac{\nu\ \Sigma, a\sim\tau\,\{\,e \parallel \mu \otimes a \hookrightarrow \bullet\,\} \mapsto \nu\ \Sigma', a\sim\tau\,\{\,e' \parallel \mu' \otimes a \hookrightarrow \bullet\,\}}{\nu\ \Sigma, a\sim\tau\,\{\,a \parallel \mu \otimes a \hookrightarrow e\,\} \mapsto \nu\ \Sigma', a\sim\tau\,\{\,a \parallel \mu' \otimes a \hookrightarrow e'\,\}} \tag{37.3b}$$

$$\frac{}{\nu\ \Sigma\,\{\,\mathtt{s}(e) \parallel \mu\,\} \mapsto \nu\ \Sigma, a\sim\mathtt{nat}\,\{\,\mathtt{s}(a) \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.3c}$$

$$\frac{\nu\ \Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\ \Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\ \Sigma\,\{\,\mathtt{ifz}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\}\parallel \mu\,\} \mapsto \nu\ \Sigma'\,\{\,\mathtt{ifz}\,e'\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\}\parallel \mu'\,\}} \tag{37.3d}$$

$$\frac{}{\nu\ \Sigma\,\{\,\mathtt{ifz}\,\mathtt{z}\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\}\parallel \mu\,\} \mapsto \nu\ \Sigma\,\{\,e_0 \parallel \mu\,\}} \tag{37.3e}$$

$$\frac{}{\left\{\begin{array}{c}\nu\ \Sigma, a\sim\mathtt{nat}\,\{\,\mathtt{ifz}\,\mathtt{s}(a)\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\}\parallel \mu\otimes a\hookrightarrow e\,\}\\ \mapsto\\ \nu\ \Sigma, a\sim\mathtt{nat}\,\{\,[a/x]e_1 \parallel \mu\otimes a\hookrightarrow e\,\}\end{array}\right\}} \tag{37.3f}$$

$$\frac{\nu\ \Sigma\,\{\,e_1 \parallel \mu\,\} \mapsto \nu\ \Sigma'\,\{\,e_1' \parallel \mu'\,\}}{\nu\ \Sigma\,\{\,e_1(e_2) \parallel \mu\,\} \mapsto \nu\ \Sigma'\,\{\,e_1'(e_2) \parallel \mu'\,\}} \tag{37.3g}$$

$$\frac{}{\left\{\begin{array}{c}\nu\ \Sigma\,\{\,\lambda\,(x:\tau)\,e(e_2) \parallel \mu\,\}\\ \mapsto\\ \nu\ \Sigma, a\sim\tau\,\{\,[a/x]e \parallel \mu\otimes a\hookrightarrow e_2\,\}\end{array}\right\}} \tag{37.3h}$$

$$\frac{}{\nu\ \Sigma\,\{\,\mathtt{fix}\,x:\tau\,\mathtt{is}\,e \parallel \mu\,\} \mapsto \nu\ \Sigma, a\sim\tau\,\{\,a \parallel \mu\otimes a\hookrightarrow [a/x]e\,\}}\,. \tag{37.3i}$$

Rule (37.3a) governs a symbol whose associated expression is a value; the value of the symbol is the value associated with that symbol in the memo table. Rule (37.3b) specifies that if the expression associated with a symbol is not a value, then it is evaluated "in place"

until such time as Rule (37.3a) applies. This is achieved by switching the focus of evaluation to the associated expression, while at the same time associating the black hole with that symbol. The black hole represents the absence of a value for that symbol, so that any attempt to access it during evaluation of its associated expression cannot make progress. This signals a circular dependency that, if not caught using a black hole, would initiate an infinite regress. We may therefore think of the black hole as catching a particular form of nontermination that arises when the value of an expression associated with a symbol depends on the symbol itself.

Rule (37.3c) specifies that evaluation of $\mathtt{s}(e)$ allocates a fresh symbol a for the expression e and yields the value $\mathtt{s}(a)$. The value of e is not determined until such time as the predecessor is required in a subsequent computation. This implements a lazy dynamics for the successor. Rule (37.3f), which governs a conditional branch on a successor, substitutes the symbol a for the variable x when computing the predecessor of a nonzero number, ensuring that all occurrences of x share the same predecessor computation.

Rule (37.3g) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (37.3h) specifies that to evaluate an application of a λ-abstraction we allocate a fresh symbol for the argument and substitute this symbol for the parameter of the function. The argument is evaluated only if it is needed in the subsequent computation, and then that value is shared among all occurrences of the parameter in the body of the function.

General recursion is implemented by Rule (37.3i). Recall from Chapter 10 that the expression $\mathtt{fix}\,x\mathtt{:}\tau\ \mathtt{is}\,e$ stands for the solution of the recursion equation $x = e$. Rule (37.3i) computes this solution by associating a fresh symbol a with the body e, substituting a for x within e to effect the self-reference. It is this substitution that permits a named expression to depend on its own name. For example, the expression $\mathtt{fix}\,x\mathtt{:}\tau\ \mathtt{is}\,x$ associates the expression a with a in the memo table and returns a. The next step of evaluation is stuck, because it seeks to evaluate a with a bound to the black hole. In contrast, an expression such as $\mathtt{fix}\,f\mathtt{:}\rho \to \tau\ \mathtt{is}\,\lambda\,(x\mathtt{:}\rho)\,e$ does not get stuck, because the self-reference is "hidden" within the λ-abstraction, and hence need not be evaluated to determine the value of the binding.

37.2 Safety

We write $\Gamma \vdash_\Sigma e : \tau$ to mean that e has type τ under the assumptions Γ, treating symbols declared in Σ as expressions of their associated type. The rules are as in Chapter 10, with the addition of the following rule for symbols:

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} a : \tau}\,. \tag{37.4}$$

This rule amounts to an implicit coercion that turns a symbol into a form of expression. The expression involves a tacit operation to obtain the binding of a symbol.

The judgment $\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{ok}$ is defined by the following rules:

$$\frac{\vdash_\Sigma e : \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{ok}} \tag{37.5a}$$

$$\frac{\forall a \sim \tau \in \Sigma \quad \mu(a) = e \neq \bullet \Longrightarrow \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma}\,. \tag{37.5b}$$

Rule (37.5b) permits self-reference through the memo table by allowing the expression associated with a symbol a to contain a, or, more generally, to contain a symbol whose associated expression contains a, and so on through any finite chain of such dependencies. Moreover, a symbol that is bound to the black hole is deemed to be of any type.

Theorem 37.1 (Preservation)**.** *Suppose that* $\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}$. *If* $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ *ok, then* $\nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}$ *ok.*

Proof We prove by induction on Rules (37.3) that if $\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}$ and $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e' : \tau$.

Consider Rule (37.3b), for which we have $e = e' = a$, $\mu = \mu_0 \otimes a \hookrightarrow e_0$, $\mu' = \mu'_0 \otimes a \hookrightarrow e'_0$, and

$$\nu\ \Sigma, a \sim \tau\ \{\, e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet \,\} \mapsto \nu\ \Sigma', a \sim \tau\ \{\, e'_0 \parallel \mu'_0 \otimes a \hookrightarrow \bullet \,\}.$$

Assume that $\vdash_{\Sigma, a\sim\tau} \mu : \Sigma, a \sim \tau$. It follows that $\vdash_{\Sigma,a\sim\tau} e_0 : \tau$ and $\vdash_{\Sigma,a\sim\tau} \mu_0 : \Sigma$, and hence that

$$\vdash_{\Sigma,a\sim\tau} \mu_0 \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau.$$

We have by induction that $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma',a\sim\tau} e'_0 : \tau'$ and

$$\vdash_{\Sigma',a\sim\tau} \mu_0 \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau.$$

But then

$$\vdash_{\Sigma',a\sim\tau} \mu' : \Sigma', a \sim \tau,$$

which suffices for the result.

Consider Rule (37.3g), so that e is the application $e_1(e_2)$ and

$$\nu\ \Sigma\ \{\, e_1 \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, e'_1 \parallel \mu' \,\}.$$

Suppose that $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$. By inversion of typing $\vdash_\Sigma e_1 : \tau_2 \to \tau$ for some type τ_2 such that $\vdash_\Sigma e_2 : \tau_2$. By induction $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e'_1 : \tau_2 \to \tau$. By weakening we have $\vdash_{\Sigma'} e_2 : \tau_2$, so that $\vdash_{\Sigma'} e'_1(e_2) : \tau$, which is enough for the result. □

The statement of the progress theorem allows for the possibility of encountering a black hole, representing a checkable form of nontermination. The judgment $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ loops,

stating that e diverges by virtue of encountering the black hole, is defined by the following rules:

$$\frac{}{\nu\ \Sigma, a \sim \tau\ \{\, a \parallel \mu \otimes a \hookrightarrow \bullet \,\}\ \mathsf{loops}} \tag{37.6a}$$

$$\frac{\nu\ \Sigma, a \sim \tau\ \{\, e \parallel \mu \otimes a \hookrightarrow \bullet \,\}\ \mathsf{loops}}{\nu\ \Sigma, a \sim \tau\ \{\, a \parallel \mu \otimes a \hookrightarrow e \,\}\ \mathsf{loops}} \tag{37.6b}$$

$$\frac{\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{loops}}{\nu\ \Sigma\ \{\, \mathtt{ifz}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\} \parallel \mu \,\}\ \mathsf{loops}} \tag{37.6c}$$

$$\frac{\nu\ \Sigma\ \{\, e_1 \parallel \mu \,\}\ \mathsf{loops}}{\nu\ \Sigma\ \{\, e_1(e_2) \parallel \mu \,\}\ \mathsf{loops}}\,. \tag{37.6d}$$

Theorem 37.2 (Progress)**.** *If* $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ *ok, then* $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ *final,* $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ *loops, or there exists* μ' *and* e' *such that* $\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto \nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}$.

Proof We proceed by induction on the derivations of $\vdash_\Sigma e : \tau$ and $\vdash_\Sigma \mu : \Sigma$ implicit in the derivation of $\nu\ \Sigma\ \{\, e \parallel \mu \,\}$ ok.

Consider Rule (10.1a), where the variable a is declared in Σ. Thus $\Sigma = \Sigma_0, a \sim \tau$ and $\vdash_\Sigma \mu : \Sigma$. It follows that $\mu = \mu_0 \otimes a \hookrightarrow e_0$ with $\vdash_\Sigma \mu_0 : \Sigma_0$ and $\vdash_\Sigma e_0 : \tau$. Note that $\vdash_\Sigma \mu_0 \otimes a \hookrightarrow \bullet : \Sigma$. Applying induction to the derivation of $\vdash_\Sigma e_0 : \tau$, we consider three cases:

1. $\nu\ \Sigma\ \{\, e_0 \parallel \mu \otimes a \hookrightarrow \bullet \,\}$ final. By inversion of Rule (37.2b) we have $e_0\ \mathsf{val}_\Sigma$, and hence by Rule (37.3a) we obtain $\nu\ \Sigma\ \{\, a \parallel \mu \,\} \mapsto \nu\ \Sigma\ \{\, e_0 \parallel \mu \,\}$.
2. $\nu\ \Sigma\ \{\, e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet \,\}$ loops. By applying Rule (37.6b) we obtain $\nu\ \Sigma\ \{\, a \parallel \mu \,\}$ loops.
3. $\nu\ \Sigma\ \{\, e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet \,\} \mapsto \nu\ \Sigma'\ \{\, e_0' \parallel \mu_0' \otimes a \hookrightarrow \bullet \,\}$. By applying Rule (37.3b) we obtain

$$\nu\ \Sigma\ \{\, a \parallel \mu \otimes a \hookrightarrow e_0 \,\} \mapsto \nu\ \Sigma'\ \{\, a \parallel \mu' \otimes a \hookrightarrow e_0' \,\}.$$

□

37.3 Lazy Data Structures

The by-need dynamics extends to product, sum, and recursive types in a straightforward manner. For example, the by-need dynamics of lazy product types is given by the following rules:

$$\frac{}{\langle a_1, a_2 \rangle\ \mathsf{val}_{\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2}} \tag{37.7a}$$

$$\left\{\begin{array}{c} \nu\ \Sigma\ \{\ \langle e_1, e_2\rangle \parallel \mu\ \} \\ \mapsto \\ \nu\ \Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\ \{\ \langle a_1, a_2\rangle \parallel \mu \otimes a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2\ \} \end{array}\right\} \tag{37.7b}$$

$$\frac{\nu\ \Sigma\ \{\ e \parallel \mu\ \} \mapsto \nu\ \Sigma'\ \{\ e' \parallel \mu'\ \}}{\nu\ \Sigma\ \{\ e \cdot \mathtt{l} \parallel \mu\ \} \mapsto \nu\ \Sigma'\ \{\ e' \cdot \mathtt{l} \parallel \mu'\ \}} \tag{37.7c}$$

$$\frac{\nu\ \Sigma\ \{\ e \parallel \mu\ \}\ \mathsf{loops}}{\nu\ \Sigma\ \{\ e \cdot \mathtt{l} \parallel \mu\ \}\ \mathsf{loops}} \tag{37.7d}$$

$$\left\{\begin{array}{c} \nu\ \Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\ \{\ \langle a_1, a_2\rangle \cdot \mathtt{l} \parallel \mu\ \} \\ \mapsto \\ \nu\ \Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\ \{\ a_1 \parallel \mu\ \} \end{array}\right\} \tag{37.7e}$$

$$\frac{\nu\ \Sigma\ \{\ e \parallel \mu\ \} \mapsto \nu\ \Sigma'\ \{\ e' \parallel \mu'\ \}}{\nu\ \Sigma\ \{\ e \cdot \mathtt{r} \parallel \mu\ \} \mapsto \nu\ \Sigma'\ \{\ e' \cdot \mathtt{r} \parallel \mu'\ \}} \tag{37.7f}$$

$$\frac{\nu\ \Sigma\ \{\ e \parallel \mu\ \}\ \mathsf{loops}}{\nu\ \Sigma\ \{\ e \cdot \mathtt{r} \parallel \mu\ \}\ \mathsf{loops}} \tag{37.7g}$$

$$\left\{\begin{array}{c} \nu\ \Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\ \{\ \langle a_1, a_2\rangle \cdot \mathtt{r} \parallel \mu\ \} \\ \mapsto \\ \nu\ \Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\ \{\ a_2 \parallel \mu\ \} \end{array}\right\}. \tag{37.7h}$$

A pair is considered a value only if its arguments are symbols [Rule (37.7a)], which are introduced when the pair is created [Rule (37.7b)]. The first and second projections evaluate to one or the other symbol in the pair, inducing a demand for the value of that component [Rules (37.7e) and (37.7h)].

Similar techniques may be used to give a by-need dynamics to sums and recursive types.

37.4 Suspensions

Another way to introduce laziness is to consolidate the machinery of the by-need dynamics into a single type whose values are possibly unevaluated, memoized computations. The type of *suspensions* of type τ, written $\tau\ \mathtt{susp}$, has as introductory form $\mathtt{susp}\, x : \tau\ \mathtt{is}\, e$ representing the suspended, possibly self-referential, computation e of type τ, and as eliminatory form the operation $\mathtt{force}(e)$ that evaluates the suspended computation presented by e, records the value in a memo table and returns that value as result.

Using suspension types, we may construct other lazy types according to our needs in a particular program. For example, the type of lazy pairs with components of type τ_1 and τ_2 is expressible as the type

$$\tau_1 \,\mathtt{susp} \times \tau_2 \,\mathtt{susp}$$

and the type of by-need functions with domain τ_1 and range τ_2 is expressible as the type

$$\tau_1 \,\mathtt{susp} \to \tau_2.$$

We may also express more complex combinations of eagerness and laziness, such as the type of "lazy lists" consisting of computations that, when forced, evaluate either to the empty list or to a nonempty list consisting of a natural number and another lazy list:

$$\mu t\,.\,(\mathtt{unit} + (\mathtt{nat} \times t))\,\mathtt{susp}.$$

This type should be contrasted with the type

$$\mu t\,.\,(\mathtt{unit} + (\mathtt{nat} \times t\,\mathtt{susp}))$$

whose values are the empty list and a pair consisting of a natural number and a computation of another such value.

The syntax of suspensions is given by the following grammar:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Typ	τ	::=	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$	suspension
Exp	e	::=	$\mathtt{susp}[\tau](x\,.\,e)$	$\mathtt{susp}\,x : \tau\,\mathtt{is}\,e$	delay
			$\mathtt{force}(e)$	$\mathtt{force}(e)$	force
			$\mathtt{susp}[a]$	$\mathtt{susp}[a]$	self-reference

Suspensions are self-referential; the bound variable x refers to the suspension itself. The expression $\mathtt{susp}[a]$ is a reference to the suspension named a.

The statics of the suspension type is given using a judgment of the form $\Gamma \vdash_\Sigma e : \tau$, where Σ assigns types to the names of suspensions. It is defined by the following rules:

$$\frac{\Gamma, x : \mathtt{susp}(\tau) \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{susp}[\tau](x\,.\,e) : \mathtt{susp}(\tau)} \tag{37.8a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{susp}(\tau)}{\Gamma \vdash_\Sigma \mathtt{force}(e) : \tau} \tag{37.8b}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{susp}[a] : \mathtt{susp}(\tau)}\,. \tag{37.8c}$$

Rule (37.8a) checks that the expression e has type τ under the assumption that x, which stands for the suspension itself, has type $\mathtt{susp}(\tau)$.

The by-need dynamics of suspensions is defined by the following rules:

$$\frac{}{\mathtt{susp}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{37.9a}$$

$$\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\mathtt{susp}[\tau]\,(x\,.\,e)\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma, a\sim\tau\,\{\,\mathtt{susp}[a]\parallel\mu\otimes a\hookrightarrow[a/x]e\,\} \end{array}\right\} \tag{37.9b}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,e'\parallel\mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{force}(e)\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,\mathtt{force}(e')\parallel\mu'\,\}} \tag{37.9c}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a\sim\tau\,\{\,\mathtt{force}(\mathtt{susp}[a])\parallel\mu\otimes a\hookrightarrow e\,\} \\ \mapsto \\ \nu\,\Sigma, a\sim\tau\,\{\,e\parallel\mu\otimes a\hookrightarrow e\,\} \end{array}\right\}} \tag{37.9d}$$

$$\frac{\begin{array}{c}\nu\,\Sigma, a\sim\tau\,\{\,e\parallel\mu\otimes a\hookrightarrow\bullet\,\} \\ \mapsto \\ \nu\,\Sigma', a\sim\tau\,\{\,e'\parallel\mu'\otimes a\hookrightarrow\bullet\,\}\end{array}}{\left\{\begin{array}{c} \nu\,\Sigma, a\sim\tau\,\{\,\mathtt{force}(\mathtt{susp}[a])\parallel\mu\otimes a\hookrightarrow e\,\} \\ \mapsto \\ \nu\,\Sigma', a\sim\tau\,\{\,\mathtt{force}(\mathtt{susp}[a])\parallel\mu'\otimes a\hookrightarrow e'\,\} \end{array}\right\}}. \tag{37.9e}$$

Rule (37.9a) specifies that a reference to a suspension is a value. Rule (37.9b) specifies that evaluation of a delayed computation consists of allocating a fresh symbol for it in the memo table and returning a reference to that suspension. Rules (37.9c)–(37.9e) specify that demanding the value of a suspension forces evaluation of the suspended computation, which is then stored in the memo table and returned as the result.

37.5 Notes

The by-need dynamics given here is inspired by the work of Ariola and Felleisen (1997), but with the crucial distinction that by-need cells are regarded as assignables rather than as variables, in keeping with the principle that variables are given meaning by substitution. The goal of by-need evaluation is to limit substitution in the interest of avoiding redundant computations. As such it cannot properly be modeled by use of variables, but rather requires a form of assignable (introduced in Chapter 35) to which at most one assignment is ever performed.

38 Polarization

Up to this point we have frequently encountered arbitrary choices in the dynamics of various language constructs. For example, when specifying the dynamics of pairs, we must choose, rather arbitrarily, between the *lazy* dynamics, in which all pairs are values regardless of the value status of their components, and the *eager* dynamics, in which a pair is a value only if its components are both values. We could even consider a *half-eager* (or, equivalently, *half-lazy*) dynamics, in which a pair is a value only if, say, the first component is a value, but without regard to the second.

Similar questions arise with sums (all injections are values, or only injections of values are values), recursive types (all folds are values, or only folds of values are values), and function types (functions should be called by-name or by-value). Whole languages are built around adherence to one policy or another. For example, Haskell decrees that products, sums, and recursive types are to be lazy and functions are to be called by name, whereas ML decrees the exact opposite policy. Not only are these choices arbitrary, but it is also unclear why they should be linked. For example, we could very sensibly decree that products, sums, and recursive types are lazy, yet impose a call-by-value discipline on functions. Or we could have eager products, sums, and recursive types, yet insist on call-by-name. It is not at all clear which of these points in the space of choices is right; each has its adherents, and each has its detractors.

Are we therefore stuck in a tarpit of subjectivity? No! The way out is to recognize that these distinctions should not be imposed by the language designer, but rather are choices that are to be made by the programmer. This may be achieved by recognizing that differences in dynamics reflect fundamental *type distinctions* that are being obscured by languages that impose one policy or another. We can have both eager and lazy pairs in the same language by simply distinguishing them as two distinct types, and similarly we can have both eager and lazy sums in the same language, and both by-name and by-value function spaces, by providing sufficient type distinctions as to make the choice available to the programmer.

Eager and lazy types are distinguished by their *polarity*, which is either *positive* or *negative* according to whether the type is defined by the values that inhabit the type or the behavior of expressions of that type. Positive types are *eager*, or *inductive*, in that they are defined by their values. Negative types are *lazy*, or *coinductive*, in that they are defined by the behavior of their elements. The polarity of types is made explicit using a technique called *focusing*. A focused presentation of a programming language distinguishes three general forms of expression, *(positive and negative) values*, *(positive and negative) continuations*, and *(neutral) computations*.

38.1 Positive and Negative Types

Polarization consists of distinguishing positive from negative types according to the following two principles:

1. A positive type is defined by its introduction rules, which specify the *values* of that type in terms of other values. The elimination rules are *inversions* that specify a computation by pattern matching on values of that type.
2. A negative type is defined by its elimination rules, which specify the *observations* that may be performed on elements of that type. The introduction rules specify the *values* of that type by specifying how they respond to observations.

From this characterization we can anticipate that the type of natural numbers would be positive, because it is defined by zero and successor, whereas function types would be negative, because they are characterized by their behavior when applied, and not by their internal structure.

The language $\mathcal{L}^{\pm}\{\mathtt{nat}\rightharpoonup\}$ is a polarized formulation of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. The syntax of types in this language is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
PTyp	τ^+	$::=$	$\mathtt{dn}(\tau^-)$	$\downarrow\tau^-$	suspension
			$\mathtt{nat}$	$\mathtt{nat}$	naturals
NTyp	τ^-	$::=$	$\mathtt{up}(\tau^+)$	$\uparrow\tau^+$	inclusion
			$\mathtt{parr}(\tau_1^+;\tau_2^-)$	$\tau_1^+ \rightharpoonup \tau_2^-$	partial function

The types $\downarrow\tau^-$ and $\uparrow\tau^+$ effect a *polarity shift* from negative to positive and positive to negative, respectively. Intuitively, the shifted type $\uparrow\tau^+$ is just the inclusion of positive into negative values, whereas the shifted type $\downarrow\tau^-$ represents the type of suspended computations of negative type.

The domain of the negative function type is required to be positive, but its range is negative. This allows us to form right-iterated function types,

$$\tau_1^+ \rightharpoonup (\tau_2^+ \rightharpoonup (\ldots(\tau_{n-1}^+ \rightharpoonup \tau_n^-)))$$

directly, but to form a left-iterated function type requires shifting,

$$\downarrow(\tau_1^+ \rightharpoonup \tau_2^-) \rightharpoonup \tau^-,$$

to turn the negative function type into a positive type. Conversely, shifting is needed to define a function whose range is positive, $\tau_1^+ \rightharpoonup \uparrow\tau_2^+$.

38.2 Focusing

The syntax of $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ is motivated by the polarization of its types. For each polarity we have a sort of values and a sort of continuations with which we may create (neutral) computations:

Sort			Abstract Form	Concrete Form	Description
PVal	v^+	::=	$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(v^+)$	$\mathtt{s}(v^+)$	successor
			$\mathtt{del}^-(e)$	$\mathtt{del}^-(e)$	delay
PCont	k^+	::=	$\mathtt{ifz}(e_0; x.e_1)$	$\mathtt{ifz}(e_0; x.e_1)$	conditional
			$\mathtt{force}^-(k^-)$	$\mathtt{force}^-(k^-)$	evaluate
NVal	v^-	::=	$\mathtt{lam}[\tau^+](x.e)$	$\lambda\,(x:\tau^+)\,e$	abstraction
			$\mathtt{del}^+(v^+)$	$\mathtt{del}^+(v^+)$	inclusion
			$\mathtt{fix}(x.v^-)$	$\mathtt{fix}\,x\,\mathtt{is}\,v^-$	recursion
NCont	k^-	::=	$\mathtt{ap}(v^+; k^-)$	$\mathtt{ap}(v^+; k^-)$	application
			$\mathtt{force}^+(x.e)$	$\mathtt{force}^+(x.e)$	evaluate
Comp	e	::=	$\mathtt{ret}(v^-)$	$\mathtt{ret}(v^-)$	return
			$\mathtt{cut}^+(v^+; k^+)$	$v^+ \triangleright k^+$	cut
			$\mathtt{cut}^-(v^-; k^-)$	$v^- \triangleright k^-$	cut

The positive values include the numerals, and the negative values include functions. In addition we may delay a computation of a negative value to form a positive value using $\mathtt{del}^-(e)$, and we may consider a positive value to be a negative value using $\mathtt{del}^+(v^+)$. The positive continuations include the conditional branch, sans argument, and the negative continuations include application sites for functions consisting of a positive argument value and a continuation for the negative result. In addition we include positive continuations to force the computation of a suspended negative value and to extract an included positive value. Computations, which correspond to machine states, consist of returned negative values (these are final states), states passing a positive value to a positive continuation, and states passing a negative value to a negative continuation. General recursion appears as a form of negative value; the recursion is unrolled when it is made the subject of an observation.

38.3 Statics

The statics of $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ consists of a collection of rules for deriving judgments of the following forms:

- Positive values: $\Gamma \vdash v^+ : \tau^+$.
- Positive continuations: $\Gamma \vdash k^+ : \tau^+ > \gamma^-$.

- Negative values: $\Gamma \vdash v^- : \tau^-$.
- Negative continuations: $\Gamma \vdash k^- : \tau^- > \gamma^-$.
- Computations: $\Gamma \vdash e : \gamma^-$.

Throughout Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1^+, \ldots, x_n : \tau_n^+,$$

for some $n \geq 0$, and γ^- is any negative type.

The typing rules for continuations specify both an argument type (on which values they act) and a result type (of the computation resulting from the action on a value). The typing rules for computations specify that the outcome of a computation is a negative type. All typing judgments specify that variables range over positive types. (These restrictions may always be met by appropriate use of shifting.)

The statics of positive values consists of the following rules:

$$\frac{}{\Gamma, x : \tau^+ \vdash x : \tau^+} \tag{38.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{38.1b}$$

$$\frac{\Gamma \vdash v^+ : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(v^+) : \mathtt{nat}} \tag{38.1c}$$

$$\frac{\Gamma \vdash e : \tau^-}{\Gamma \vdash \mathtt{del}^-(e) : \downarrow \tau^-}. \tag{38.1d}$$

Rule (38.1a) specifies that variables range over positive values. Rules (38.1b) and (38.1c) specify that the values of type `nat` are just the numerals. Rule (38.1d) specifies that a suspended computation (necessarily of negative type) is a positive value.

The statics of positive continuations consists of the following rules:

$$\frac{\Gamma \vdash e_0 : \gamma^- \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \gamma^-}{\Gamma \vdash \mathtt{ifz}(e_0; x\,.\,e_1) : \mathtt{nat} > \gamma^-} \tag{38.2a}$$

$$\frac{\Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash \mathtt{force}^-(k^-) : \downarrow \tau^- > \gamma^-}. \tag{38.2b}$$

Rule (38.2a) governs the continuation that chooses between two computations according to whether a natural number is zero or non-zero. Rule (38.2b) specifies the continuation that forces a delayed computation with the specified negative continuation.

The statics of negative values is defined by these rules:

$$\frac{\Gamma, x : \tau_1^+ \vdash e : \tau_2^-}{\Gamma \vdash \lambda\,(x : \tau_1^+)\, e : \tau_1^+ \rightharpoonup \tau_2^-} \tag{38.3a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+}{\Gamma \vdash \mathtt{del}^+(v^+) : \uparrow \tau^+} \tag{38.3b}$$

$$\frac{\Gamma, x : \downarrow \tau^- \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{fix}\, x\, \mathtt{is}\, v^- : \tau^-}. \tag{38.3c}$$

Rule (38.3a) specifies the statics of a λ-abstraction whose argument is a positive value and whose result is a computation of negative type. Rule (38.3b) specifies the inclusion of positive values as negative values. Rule (38.3c) specifies that negative types admit general recursion.

The statics of negative continuations is defined by these rules:

$$\frac{\Gamma \vdash v_1^+ : \tau_1^+ \quad \Gamma \vdash k_2^- : \tau_2^- > \gamma^-}{\Gamma \vdash \mathtt{ap}(v_1^+; k_2^-) : \tau_1^+ \rightharpoonup \tau_2^- > \gamma^-} \tag{38.4a}$$

$$\frac{\Gamma, x : \tau^+ \vdash e : \gamma^-}{\Gamma \vdash \mathtt{force}^+(x.e) : \uparrow \tau^+ > \gamma^-}\,. \tag{38.4b}$$

Rule (38.4a) is the continuation representing the application of a function to the positive argument v_1^+ and executing the body with negative continuation k_2^-. Rule (38.4b) specifies the continuation that passes a positive value, viewed as a negative value, to a computation.

The statics of computations is given by these rules:

$$\frac{\Gamma \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{ret}(v^-) : \tau^-} \tag{38.5a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+ \quad \Gamma \vdash k^+ : \tau^+ > \gamma^-}{\Gamma \vdash v^+ \triangleright k^+ : \gamma^-} \tag{38.5b}$$

$$\frac{\Gamma \vdash v^- : \tau^- \quad \Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash v^- \triangleright k^- : \gamma^-}\,. \tag{38.5c}$$

Rule (38.5a) specifies the basic form of computation that simply returns the negative value v^-. Rules (38.5b) and (38.5c) specify computations that pass a value to a continuation of appropriate polarity.

38.4 Dynamics

The dynamics of $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ is given by a transition system $e \mapsto e'$ specifying the steps of computation. The rules are all axioms; no premises are required because the continuation is used to manage pending computations.

The dynamics consists of the following rules:

$$\overline{\mathtt{z} \triangleright \mathtt{ifz}(e_0; x.e_1) \mapsto e_0} \tag{38.6a}$$

$$\overline{\mathtt{s}(v^+) \triangleright \mathtt{ifz}(e_0; x.e_1) \mapsto [v^+/x]e_1} \tag{38.6b}$$

$$\overline{\mathtt{del}^-(e) \triangleright \mathtt{force}^-(k^-) \mapsto e\,;k^-} \tag{38.6c}$$

$$\overline{\lambda\,(x{:}\tau^+)\,e \triangleright \mathtt{ap}(v^+; k^-) \mapsto [v^+/x]e\,;k^-} \tag{38.6d}$$

$$\overline{\mathtt{del}^+(v^+) \triangleright \mathtt{force}^+(x.e) \mapsto [v^+/x]e} \tag{38.6e}$$

$$\overline{\mathtt{fix}\,x\,\mathtt{is}\,v^- \triangleright k^- \mapsto [\mathtt{del}^-(\mathtt{fix}\,x\,\mathtt{is}\,v^-)/x]v^- \triangleright k^-}\,. \tag{38.6f}$$

These rules specify the interaction between values and continuations.

Rules (38.6) make use of two forms of substitution, $[v^+/x]e$ and $[v^+/x]v^-$, which are defined as in Chapter 1. They also employ a new form of *composition*, written as $e;k_0^-$, which composes a computation with a continuation by attaching k_0^- to the end of the computation specified by e. This composition is defined mutually recursive with the compositions $k^+;k_0^-$ and $k^-;k_0^-$, which essentially concatenate continuations (stacks).

$$\overline{\mathtt{ret}(v^-);k_0^- = v^- \triangleright k_0^-} \tag{38.7a}$$

$$\frac{k^-;k_0^- = k_1^-}{(v^- \triangleright k^-);k_0^- = v^- \triangleright k_1^-} \tag{38.7b}$$

$$\frac{k^+;k_0^- = k_1^+}{(v^+ \triangleright k^+);k_0^- = v^+ \triangleright k_1^+} \tag{38.7c}$$

$$\frac{e_0;k^- = e_0' \quad x \mid e_1;k^- = e_1'}{\mathtt{ifz}(e_0;x.e_1);k^- = \mathtt{ifz}(e_0';x.e_1')} \tag{38.7d}$$

$$\frac{k^-;k_0^- = k_1^-}{\mathtt{force}^-(k^-);k_0^- = \mathtt{force}^-(k_1^-)} \tag{38.7e}$$

$$\frac{k^-;k_0^- = k_1}{\mathtt{ap}(v^+;k^-);k_0^- = \mathtt{ap}(v^+;k_1^-)} \tag{38.7f}$$

$$\frac{x \mid e;k_0^- = e'}{\mathtt{force}^+(x.e);k_0^- = \mathtt{force}^+(x.e')}\,. \tag{38.7g}$$

Rules (38.7d) and (38.7g) make use of the generic hypothetical judgment defined in Chapter 3 to express that the composition is defined uniformly in the bound variable.

38.5 Safety

The proof of preservation for $\mathcal{L}^{\pm}\{\mathtt{nat}{\rightharpoonup}\}$ reduces to the proof of the typing properties of substitution and composition.

Lemma 38.1 (Substitution)**.** *Suppose that* $\Gamma \vdash v^+ : \rho^+$.

1. *If* $\Gamma, x : \rho^+ \vdash e : \gamma^-$, *then* $\Gamma \vdash [v^+/x]e : \gamma^-$.
2. *If* $\Gamma, x : \rho^+ \vdash v^- : \tau^-$, *then* $\Gamma \vdash [v^+/x]v^- : \tau^-$.
3. *If* $\Gamma, x : \rho^+ \vdash k^+ : \tau^+ > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^+ : \tau^+ > \gamma^-$.
4. *If* $\Gamma, x : \rho^+ \vdash v_1^+ : \tau^+$, *then* $\Gamma \vdash [v^+/x]v_1^+ : \tau^+$.
5. *If* $\Gamma, x : \rho^+ \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^- : \tau^- > \gamma^-$.

Proof Simultaneously, by induction on the derivation of the typing of the target of the substitution. □

Lemma 38.2 (Composition)**.**

1. *If* $\Gamma \vdash e : \tau^- $ *and* $\Gamma \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash e \,;\, k^- : \tau^- > \gamma^-$.
2. *If* $\Gamma \vdash k_0^+ : \tau^+ > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^+ \,;\, k_1^- : \tau^+ > \gamma_1^-$.
3. *If* $\Gamma \vdash k_0^- : \tau^- > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^- \,;\, k_1^- : \tau^- > \gamma_1^-$.

Proof Simultaneously, by induction on the derivations of the first premises of each clause of the lemma. □

Theorem 38.3 (Preservation)**.** *If* $\Gamma \vdash e : \gamma^-$ *and* $e \mapsto e'$, *then* $\Gamma \vdash e' : \gamma^-$.

Proof By induction on transition, appealing to inversion for typing and Lemmas 38.1 and 38.2. □

The progress theorem reduces to the characterization of the values of each type. Focusing makes the required properties evident, because it defines directly the values of each type.

Theorem 38.4 (Progress)**.** *If* $\Gamma \vdash e : \gamma^-$, *then either* $e = \mathtt{ret}(v^-)$ *for some* v^- *or there exists* e' *such that* $e \mapsto e'$.

38.6 Notes

The concept of polarization originates with Andreoli (1992), who introduced focusing as a technique for proof search in linear logic. The formulation given here is inspired by Zeilberger (2008), wherein focusing is related to evaluation order in programming languages.

PART XV

Parallelism

39 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 subcomputations, the greater the opportunities for parallelism. This argues for functional models of computation, because the possibility of mutation of shared assignables imposes sequentialization constraints on imperative code.

In this chapter we discuss *nested parallelism* in which we nest parallel computations within one another in a hierarchical manner. Nested parallelism is sometimes called *fork–join* parallelism to emphasize the hierarchical structure arising from *forking* two (or more) parallel computations, then *joining* these computations to combine their results before proceeding. We 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 *depth*) of a parallel program by associating a *series-parallel graph* with each computation.

39.1 Binary Fork–Join

We begin with a parallel language whose sole source of parallelism is the simultaneous evaluation of two variable bindings. This is modeled by a construct of the form $\texttt{par}\,x_1 = e_1\,\texttt{and}\,x_2 = e_2\,\texttt{in}\,e$, in which we bind two variables x_1 and x_2 to two expressions e_1 and e_2, respectively, for use within a single expression e. This represents a simple fork–join primitive in which e_1 and e_2 may be evaluated independently of one another, with their results combined by the expression e. Some other forms of parallelism may be defined in terms of this primitive. As an example, *parallel pairing* may be defined as the expression

$$\texttt{par}\,x_1 = e_1\,\texttt{and}\,x_2 = e_2\,\texttt{in}\,\langle x_1, x_2\rangle,$$

which evaluates the components of the pair in parallel, then constructs the pair itself from these values.

The abstract syntax of the parallel binding construct is given by the abstract binding tree

$$\texttt{par}(e_1; e_2; x_1.x_2.e),$$

which makes clear that the variables x_1 and x_2 are bound *only* within e and not within their bindings. This ensures that evaluation of e_1 is independent of evaluation of e_2, and vice versa. The typing rule for an expression of this form is given as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{par}(e_1; e_2; x_1.x_2.e) : \tau}. \tag{39.1}$$

Although we emphasize the case of binary parallelism, it should be clear that this construct easily generalizes to n-way parallelism for any *static* value of n. We may also define an n-way parallel `let` construct from the binary parallel `let` by cascading binary splits. (For a treatment of n-way parallelism for a *dynamic* value of n, see Section 39.3.)

Both a *sequential* dynamics and a *parallel* dynamics of the parallel `let` construct are given. The definition of the sequential dynamics as a transition judgment of the form $e \mapsto_{\mathsf{seq}} e'$ is entirely straightforward:

$$\frac{e_1 \mapsto e_1'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \texttt{par}(e_1'; e_2; x_1.x_2.e)} \tag{39.2a}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \texttt{par}(e_1; e_2'; x_1.x_2.e)} \tag{39.2b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} [e_1, e_2/x_1, x_2]e}. \tag{39.2c}$$

The parallel dynamics is given by a transition judgment of the form $e \mapsto_{\mathsf{par}} e'$, defined as follows:

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1'; e_2'; x_1.x_2.e)} \tag{39.3a}$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1'; e_2; x_1.x_2.e)} \tag{39.3b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1; e_2'; x_1.x_2.e)} \tag{39.3c}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} [e_1, e_2/x_1, x_2]e}. \tag{39.3d}$$

The parallel dynamics is idealized in that it abstracts away from any limitations on parallelism that would necessarily be imposed in practice by the availability of computing resources.

An important advantage of the present approach is captured by the *implicit parallelism theorem*, which states that the sequential dynamics and the parallel dynamics coincide.

This means that we need never be concerned with the *semantics* of a parallel program (its meaning is determined by the sequential dynamics), but only with its *efficiency*. As a practical matter, this means that a program may be developed on a sequential platform, even if it is intended 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 clearly distinguishes deterministic parallelism, the subject of this chapter, from nondeterministic concurrency, the subject of Chapters 41 and 42.

A proof of the implicit parallelism theorem may be given by giving an evaluation dynamics, $e \Downarrow v$, in the style of Chapter 7, and showing that

$$e \mapsto^*_{\mathsf{par}} v \quad \text{iff} \quad e \Downarrow v \quad \text{iff} \quad e \mapsto^*_{\mathsf{seq}} v$$

where v is a closed expression such that v val. The crucial rule of the evaluation dynamics is the one governing the parallel `let` construct:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow v}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \Downarrow v}. \tag{39.4}$$

It is easy to show that the sequential dynamics agrees with the evaluation dynamics by a straightforward extension of the proof of Theorem 7.2.

Lemma 39.1. *$e \mapsto^*_{\mathsf{seq}} v$ iff $e \Downarrow v$.*

Proof It suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{seq}} v_1$ and $e_2 \mapsto^*_{\mathsf{seq}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{seq}} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{seq}} v.$$

□

By a similar argument we may show that the parallel dynamics also agrees with the evaluation dynamics and hence with the sequential dynamics.

Lemma 39.2. *$e \mapsto^*_{\mathsf{par}} v$ iff $e \Downarrow v$.*

Proof It suffices to show that if $e \mapsto_{\mathsf{par}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{par}} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{par}} v.$$

The proof of the first is by a straightforward induction on the parallel dynamics. The proof of the second proceeds by simultaneous induction on the derivations of $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$. If $e_1 = v_1$ with v_1 val and $e_2 = v_2$ with v_2 val, then the result follows immediately from the third premise. If $e_2 = v_2$ but $e_1 \mapsto_{\mathsf{par}} e_1' \mapsto^*_{\mathsf{par}} v_1$, then by induction we have that $\mathtt{par}\, x_1 = e_1' \,\mathtt{and}\, x_2 = v_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{par}} v$, and hence the result follows by an application of Rule (39.3b). The symmetric case follows similarly by an application of Rule (39.3c), and in the case that both e_1 and e_2 take a step, the result follows by induction and Rule (39.3a). □

Theorem 39.3 (Implicit Parallelism)**.** *The sequential dynamics and parallel dynamics coincide: For all* v val, $e \mapsto^*_{seq} v$ *iff* $e \mapsto^*_{par} v$.

Proof By Lemmas 39.1 and 39.2. □

The implicit parallelism theorem states that parallelism does not affect the semantics of a program, only the efficiency of its execution. Correctness concerns are factored out, focusing attention on complexity.

39.2 Cost Dynamics

This section defines a *parallel cost dynamics* that assigns a *cost graph* to the evaluation of an expression. Cost graphs are defined by the following grammar:

Cost	c	::=	$\mathbf{0}$	zero cost
			$\mathbf{1}$	unit cost
			$c_1 \otimes c_2$	parallel combination
			$c_1 \oplus c_2$	sequential combination

A cost graph is a form of *series-parallel* directed acyclic graph, with a designated *source* node and *sink* node. For $\mathbf{0}$ the graph consists of one node and no edges, with the source and sink both being the node itself. For $\mathbf{1}$ the graph consists of two nodes and one edge directed from the source to the sink. For $c_1 \otimes c_2$, if g_1 and g_2 are the graphs of c_1 and c_2, respectively, then the graph has two additional nodes, a source node with two edges to the source nodes of g_1 and g_2, and a sink node, with edges from the sink nodes of g_1 and g_2 to it. Finally, for $c_1 \oplus c_2$, where g_1 and g_2 are the graphs of c_1 and c_2, the graph has as source node the source of g_1, as sink node the sink of g_2, and an edge from the sink of g_1 to the source of g_2.

The intuition behind a cost graph is that nodes represent subcomputations of an overall computation, and edges represent *sequentiality constraints* stating that one computation depends on the result of another, and hence cannot be started before the one on which it depends completes. The product of two graphs represents *parallelism opportunities* in which there are no sequentiality constraints between the two computations. The assignment of source and sink nodes reflects the overhead of *forking* two parallel computations and *joining* them after they have both completed.

We associate with each cost graph two numeric measures, the *work* $wk(c)$ and the *depth* $dp(c)$. The work is defined by the following equations:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases}. \tag{39.5}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases}. \tag{39.6}$$

Informally, the work of a cost graph determines the total number of computation steps represented by the cost graph and thus corresponds to the *sequential complexity* of the computation. The depth of the cost graph determines the *critical path length*, the length of the longest dependency chain within the computation, which imposes a lower bound on the *parallel complexity* of a computation. The critical path length is the least number of sequential steps that can be taken, even if we have unlimited parallelism available to us, because of steps that can be taken only after the completion of another.

In Chapter 7 we introduced *cost dynamics* as a means of assigning time complexity to evaluation. The proof of Theorem 7.7 shows that $e \Downarrow^k v$ iff $e \mapsto^k v$. That is, the step complexity of an evaluation of e to a value v is just the number of transitions required to derive $e \mapsto^* v$. Here we use cost graphs as the measure of complexity, then relate these cost graphs to the structural dynamics given in Section 39.1.

The judgment $e \Downarrow^c v$, where e is a closed expression, v is a closed value, and c is a cost graph, specifies the cost dynamics. By definition we arrange that $e \Downarrow^{\mathbf{0}} e$ when e val. The cost assignment for `let` is given by the following rule:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\texttt{par}(e_1; e_2; x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v}. \tag{39.7}$$

The cost assignment specifies that, under ideal conditions, e_1 and e_2 are to be evaluated in parallel, and that their results are to be propagated to e. The cost of fork and join is implicit in the parallel combination of costs and assigns unit cost to the substitution because we expect it to be implemented in practice by a constant-time mechanism for updating an environment. The cost dynamics of other language constructs is specified in a similar manner, using only sequential combination so as to isolate the source of parallelism to the `let` construct.

Two simple facts about the cost dynamics are important to keep in mind. First, the cost assignment does not influence the outcome.

Lemma 39.4. *$e \Downarrow v$ iff $e \Downarrow^c v$ for some c.*

Proof From right to left, erase the cost assignments to obtain an evaluation derivation. From left to right, decorate the evaluation derivations with costs as determined by the rules defining the cost dynamics. □

Second, the cost of evaluating an expression is uniquely determined.

Lemma 39.5. *If $e \Downarrow^c v$ and $e \Downarrow^{c'} v$, then c is c'.*

Proof A routine induction on the derivation of $e \Downarrow^c v$. □

The link between the cost dynamics and the structural dynamics given in the preceding section is established by the following theorem, which states that the work cost is the sequential complexity and the depth cost is the parallel complexity of the computation.

Theorem 39.6. *If $e \Downarrow^c v$, then $e \mapsto^w_{seq} v$ and $e \mapsto^d_{par} v$, where $w = wk(c)$ and $d = dp(c)$. Conversely, if $e \mapsto^w_{seq} v$, then there exists c such that $e \Downarrow^c v$ with $wk(c) = w$, and if $e \mapsto^d_{par} v'$, then there exists c' such that $e \Downarrow^{c'} v'$ with $dp(c') = d$.*

Proof The first part is proved by induction on the derivation of $e \Downarrow^c v$, the interesting case being Rule (39.7). By induction we have $e_1 \mapsto^{w_1}_{\mathsf{seq}} v_1$, $e_2 \mapsto^{w_2}_{\mathsf{seq}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto^w_{\mathsf{seq}} v$, where $w_1 = wk(c_1)$, $w_2 = wk(c_2)$, and $w = wk(c)$. By pasting together derivations we obtain a derivation

$$\begin{aligned}
\mathtt{par}(e_1; e_2; x_1.x_2.e) &\mapsto^{w_1}_{\mathsf{seq}} \mathtt{par}(v_1; e_2; x_1.x_2.e) \\
&\mapsto^{w_2}_{\mathsf{seq}} \mathtt{par}(v_1; v_2; x_1.x_2.e) \\
&\mapsto_{\mathsf{seq}} [v_1, v_2/x_1, x_2]e \\
&\mapsto^{w}_{\mathsf{seq}} v.
\end{aligned}$$

Noting that $wk((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \mapsto^{d_1}_{\mathsf{par}} v_1$, $e_2 \mapsto^{d_2}_{\mathsf{par}} v_2$, and $e \mapsto^d_{\mathsf{par}} v$, where $d_1 = dp(c_1)$, $d_2 = dp(c_2)$, and $d = dp(c)$. Assume, without loss of generality, that $d_1 \leq d_2$ (otherwise simply swap the roles of d_1 and d_2 in what follows). We may paste together derivations as follows:

$$\begin{aligned}
\mathtt{par}(e_1; e_2; x_1.x_2.e) &\mapsto^{d_1}_{\mathsf{par}} \mathtt{par}(v_1; e_2'; x_1.x_2.e) \\
&\mapsto^{d_2-d_1}_{\mathsf{par}} \mathtt{par}(v_1; v_2; x_1.x_2.e) \\
&\mapsto_{\mathsf{par}} [v_1, v_2/x_1, x_2]e \\
&\mapsto^{d}_{\mathsf{par}} v.
\end{aligned}$$

Calculating $dp((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1, d_2) + 1 + d$ completes the proof.

Turning to the second part, it suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^c v$ with $wk(c) = wk(c') + 1$, and if $e \mapsto_{\mathsf{par}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^c v$ with $dp(c) = dp(c') + 1$.

Suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ with e_1 val and e_2 val. Then $e \mapsto_{\mathsf{seq}} e'$, where $e = [e_1, e_2/x_1, x_2]e_0$ and there exists c' such that $e' \Downarrow^{c'} v$. But then $e \Downarrow^c v$, where $c = (\mathbf{0} \otimes \mathbf{0}) \oplus \mathbf{1} \oplus c'$, and a simple calculation shows that $wk(c) = wk(c') + 1$, as required. Similarly, $e \mapsto_{\mathsf{par}} e'$ for e' as previously shown, and hence $e \Downarrow^c v$ for some c such that $dp(c) = dp(c') + 1$, as required.

Suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ and $e \mapsto_{\mathsf{seq}} e'$, where $e' = \mathtt{par}(e_1'; e_2; x_1.x_2.e_0)$ and $e_1 \mapsto_{\mathsf{seq}} e_1'$. From the assumption that $e' \Downarrow^{c'} v$, we have by inversion that $e_1' \Downarrow^{c_1'} v_1$, $e_2 \Downarrow^{c_2'} v_2$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0'} v$, with $c' = (c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0'$. By induction there exists c_1 such that $wk(c_1) = 1 + wk(c_1')$ and $e_1 \Downarrow^{c_1} v_1$. But then $e \Downarrow^c v$, with $c = (c_1 \otimes c_2') \oplus \mathbf{1} \oplus c_0'$.

By a similar argument, suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ and $e \mapsto_{\mathsf{par}} e'$, where $e' = \mathtt{par}(e_1'; e_2'; x_1.x_2.e_0)$ and $e_1 \mapsto_{\mathsf{par}} e_1'$, $e_2 \mapsto_{\mathsf{par}} e_2'$, and $e' \Downarrow^{c'} v$. Then by inversion

$e_1' \Downarrow^{c_1'} v_1, e_2' \Downarrow^{c_2'} v_2, [v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. But then $e \Downarrow^{c} v$, where $c = (c_1 \otimes c_2) \oplus \mathbf{1} \oplus c_0$, $e_1 \Downarrow^{c_1} v_1$ with $dp(c_1) = 1 + dp(c_1')$, $e_2 \Downarrow^{c_2} v_2$ with $dp(c_2) = 1 + dp(c_2')$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. Calculating, we obtain

$$\begin{aligned} dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) \\ &= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) \\ &= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 \\ &= dp(c') + 1, \end{aligned}$$

which completes the proof. □

Corollary 39.7. *If $e \mapsto^{w}_{seq} v$ and $e \mapsto^{d}_{par} v'$, then v is v' and $e \Downarrow^{c} v$ for some c such that $wk(c) = w$ and $dp(c) = d$.*

39.3 Multiple Fork–Join

So far we have confined attention to binary fork–join parallelism induced by the parallel `let` construct. Although technically sufficient for many purposes, a more natural programming model admits an unbounded number of parallel tasks to be spawned simultaneously, rather than forcing them to be created by a cascade of binary forks and corresponding joins. Such a model, often called *data parallelism*, ties the source of parallelism to a data structure of unbounded size. The principal example of such a data structure is a *sequence* of values of a specified type. The primitive operations on sequences provide a natural source of unbounded parallelism. For example, 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 consider here a simple language of sequence operations to illustrate the main ideas:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`seq(`τ`)`	τ `seq`	sequence
Exp	e	::=	`seq(`e_0`,` $\ldots$ `,`e_{n-1}`)`	`[`e_0`,` $\ldots$ `,`e_{n-1}`]`	sequence
			`len(`e`)`	`|`e`|`	size
			`sub(`e_1`;`e_2`)`	e_1`[`e_2`]`	element
			`tab(`x`.`e_1`;`e_2`)`	`tab(`x`.`e_1`;`e_2`)`	tabulate
			`map(`x`.`e_1`;`e_2`)`	`[`e_1 `|` $x \in e_2$`]`	map
			`cat(`e_1`;`e_2`)`	`cat(`e_1`;`e_2`)`	concatenate

The expression `seq(`e_0`,` $\ldots$ `,`e_{n-1}`)` evaluates to an n-sequence whose elements are given by the expressions $e_0, \ldots, e_{n-1}$. The operation `len(`e`)` returns the number of elements in the sequence given by e. The operation `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, `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 `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:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \texttt{seq}(e_0, \ldots, e_{n-1}) : \texttt{seq}(\tau)} \tag{39.8a}$$

$$\frac{\Gamma \vdash e : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{len}(e) : \texttt{nat}} \tag{39.8b}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{sub}(e_1; e_2) : \tau} \tag{39.8c}$$

$$\frac{\Gamma, x : \texttt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{tab}(x.e_1; e_2) : \texttt{seq}(\tau)} \tag{39.8d}$$

$$\frac{\Gamma \vdash e_2 : \texttt{seq}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{map}(x.e_1; e_2) : \texttt{seq}(\tau')} \tag{39.8e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{seq}(\tau)}. \tag{39.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{seq}(e_0, \ldots, e_{n-1}) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{39.9a}$$

$$\frac{e \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1})}{\texttt{len}(e) \Downarrow^{c \oplus \mathbf{1}} \texttt{num}[n]} \tag{39.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{n-1}) \quad e_2 \Downarrow^{c_2} \texttt{num}[i] \quad (0 \leq i < n)}{\texttt{sub}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{39.9c}$$

$$\frac{e_2 \Downarrow^{c} \texttt{num}[n] \quad [\texttt{num}[0]/x]e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad [\texttt{num}[n-1]/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{39.9d}$$

$$\frac{\begin{array}{c} e_2 \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1}) \\ [v_0/x]e_1 \Downarrow^{c_0} v_0' \qquad \ldots \qquad [v_{n-1}/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}' \end{array}}{\texttt{map}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0', \ldots, v_{n-1}')} \tag{39.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{m-1}) \quad e_2 \Downarrow^{c_2} \texttt{seq}(v_0', \ldots, v_{n-1}')}{\texttt{cat}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \bigotimes_{i=0}^{m+n-1} \mathbf{1}} \texttt{seq}(v_0, \ldots, v_{m-1}, v_0', \ldots, v_{n-1}')}. \tag{39.9f}$$

The cost dynamics for sequence operations may be validated by introducing a sequential and parallel cost dynamics and extending the proof of Theorem 39.6 to cover this extension.

39.4 Provably Efficient Implementations

Theorem 39.6 states that the cost dynamics accurately models the dynamics of the parallel `let` construct, whether executed sequentially or in parallel. This validates the cost dynamics from the point of view of the dynamics of the language and permits us to draw conclusions about the asymptotic complexity of a parallel program that abstracts away from the limitations imposed by a concrete implementation. Chief among these is the restriction to a fixed number $p > 0$ of processors on which to schedule the workload. In addition to limiting the available parallelism, this also imposes some synchronization overhead that must be accounted for in order to make accurate predictions of run-time behavior on a concrete parallel platform. A *provably efficient implementation* is one for which we may establish an asymptotic bound on the actual execution time once these overheads are taken into account.

A provably efficient implementation must take account of the limitations and capabilities of the actual hardware on which the program is to be run. Because we are interested in only asymptotic upper bounds, it is convenient to formulate an abstract machine model and to show that the primitives of the language can be implemented on this model with guaranteed time (and space) bounds. One popular model is the *SMP*, or *shared-memory multiprocessor*, which consists of $p > 0$ sequential processors coordinated by an interconnect network that provides constant-time access to shared memory by each of the processors.[1] The multiprocessor is assumed to provide a constant-time synchronization primitive with which to control simultaneous access to a memory cell. There is a variety of such primitives, any of which is sufficient to provide a parallel fetch-and-add instruction that allows each processor to obtain the current contents of a memory cell and update it by adding a fixed constant in a single atomic operation—the interconnect serializes any simultaneous accesses by more than one processor.

Building a provably efficient implementation of parallelism involves two majors tasks. First, we must show that each of the primitives of the language may be implemented efficiently on the abstract machine model. Second, we must show how to schedule the workload across the processors so as to minimize execution time by maximizing parallelism. When working with a low-level machine model such as an SMP, both tasks involve a fair bit of technical detail to show how to use low-level machine instructions, including a synchronization primitive, to implement the language primitives and to schedule the workload. Collecting these together, we may then give an asymptotic bound on the time complexity of the implementation that relates the abstract cost of the computation to cost of implementing the workload on a p-way multiprocessor. The prototypical result of this kind is called *Brent's Theorem*.

Theorem 39.8. *If $e \Downarrow^c v$ with $wk(c) = w$ and $dp(c) = d$, then e may be evaluated on a p-processor SMP in time $O(\max(w/p, d))$.*

[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 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. Observe that if $p = 1$ then the theorem establishes an upper bound of $O(w)$ steps, the sequential complexity of the computation. Moreover, if the work is proportional to the depth, then we are unable to exploit parallelism, and the overall time is proportional to the work alone.

This motivates the definition of a useful figure of merit, called the *parallelizability ratio*, which is the ratio w/d of work to depth. If $w/d \gg p$, then the program is said to be *parallelizable*, because then $w/p \gg d$, and we may therefore reduce running time by using p-processors at each step. If, on the other hand, the parallelizability ratio is a constant, then d will dominate w/p, and we will have little opportunity to exploit parallelism to reduce running time. It is not known, in general, whether a problem admits a parallelizable solution. The best we can say, on present knowledge, is that there are algorithms for some problems that have a high degree of parallelizability, and there are problems for which no such algorithm is known. It is a open problem in complexity theory to characterize which problems are parallelizable and which are not.

To illustrate the essential ingredients of the proof of Brent's Theorem we consider a dynamics that models the scheduling of work onto p parallel processors, each of which implements the dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, as described in Chapter 10. The parallel dynamics is defined on states $\nu\ \Sigma\ \{\ \mu\ \}$ of the form

$$\nu\ a_1 \sim \tau_1, \ldots a_n \sim \tau_n\ \{\ a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n\ \},$$

where $n \geq 1$. Such a state represents a computation that has been broken down into n parallel tasks. Each task is given a *name*. The occurrence of a name within a task represents a dependency of that task on the named task. A task is said to be *blocked* on the tasks on which it depends; a task with no dependencies is said to be *ready*.

There are two forms of transition, the *local* and the *global*. The local transitions represent the steps of the individual processors. These consist of the steps of execution of expressions as defined in Chapter 10, augmented with transitions governing parallelism. The global transitions represent the simultaneous execution of local transitions on up to some fixed number p of processors.

Local transitions have the form

$$\nu\ \Sigma\ a \sim \tau\ \{\ \mu \otimes a \hookrightarrow e\ \} \mapsto_{loc} \nu\ \Sigma'\ a \sim \tau\ \{\ \mu' \otimes a \hookrightarrow e'\ \},$$

where e is ready, and (a) Σ and Σ' are empty and μ and μ' are empty; (b) Σ and μ are empty and Σ' and μ' declare the types and bindings of two distinct names; or (c) Σ' and μ' are both empty, and Σ and μ declare the types and bindings of two distinct names. These conditions correspond to the three possible outcomes of a local step by a task: (a) the task takes a step of computation in the sense of Chapter 10; (b) the task forks two new tasks and waits for their completion; (c) the task joins two parallel tasks that have completed execution:

$$\frac{}{\nu\ a \sim \tau\ \{\ a \hookrightarrow (\lambda\ (x\!:\!\tau_2)\ e)\ (e_2)\ \} \mapsto_{loc} \nu\ a \sim \tau\ \{\ a \hookrightarrow [e_2/x]e\ \}} \tag{39.10a}$$

$$\left\{\begin{array}{c} \nu\, a \sim \tau\, \{\, a \hookrightarrow \mathtt{par}(e_1; e_2; x_1\,.\,x_2\,.\,e)\,\} \\ \mapsto_{loc} \\ \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e)\,\} \end{array}\right\} \tag{39.10b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e)\,\} \\ \mapsto_{loc} \\ \nu\, a \sim \tau\, \{\, a \hookrightarrow [e_1, e_2/x_1, x_2]e\,\} \end{array}\right\}}. \tag{39.10c}$$

Rule (39.10a) illustrates one rule for the dynamics of the nonparallel aspects of the language; additional rules are required to cover the other cases, allowing for nested uses of parallelism. Rule (39.10b) represents the creation of two parallel tasks on which the executing task depends. The expression $\mathtt{par}(a_1; a_2; x_1\,.\,x_2\,.\,e)$ is blocked on tasks a_1 and a_2, so that no local step applies to it. Rule (39.10c) synchronizes a task with the tasks on which it depends once their execution has completed; those tasks are no longer required and are therefore eliminated from the state.

Each global transition represents the simultaneous execution of one step of computation on each of up to $p \geq 1$ processors:

$$\frac{\begin{array}{c} \nu\ \Sigma_1 a_1 \sim \tau_1\, \{\, \mu_1 \otimes a_1 \hookrightarrow e_1\,\} \mapsto_{loc} \nu\ \Sigma_1' a_1 \sim \tau_1\, \{\, \mu_1' \otimes a_1 \hookrightarrow e_1'\,\} \\ \ldots \\ \nu\ \Sigma_n a_n \sim \tau_n\, \{\, \mu_n \otimes a_n \hookrightarrow e_n\,\} \mapsto_{loc} \nu\ \Sigma_n' a_n \sim \tau_n\, \{\, \mu_n' \otimes a_n \hookrightarrow e_n'\,\} \end{array}}{\left\{\begin{array}{c} \nu\ \Sigma_0\ \Sigma_1\, a_1 \sim \tau_1\ \ldots\ \Sigma_n\, a_n \sim \tau_n\, \{\, \mu_0 \otimes \mu_1 \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes \mu_n \otimes a_n \hookrightarrow e_n\,\} \\ \mapsto_{glo} \\ \nu\ \Sigma_0\ \Sigma_1'\, a_1 \sim \tau_1\ \ldots\ \Sigma_n'\, a_n \sim \tau_n\, \{\, \mu_0 \otimes \mu_1' \otimes a_1 \hookrightarrow e_1' \otimes \ldots \otimes \mu_n' \otimes a_n \hookrightarrow e_n'\,\} \end{array}\right\}}. \tag{39.11}$$

At each global step some number $1 \leq n \leq p$ of ready tasks are scheduled for execution.[2] 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 introduces two fresh tasks that are added to the state as a result of the global transition; any local join step eliminates two tasks whose execution has completed.

A subtle point is that it is implicit in our name binding conventions that the names of any created tasks are to be *globally unique*, even though they are *locally created*. In implementation terms, this requires a synchronization step among the processors to ensure that task names are not acccidentally reused among the parallel processors.

The proof of Brent's Theorem for this high-level dynamics is now obvious, provided only that the global scheduling steps are performed greedily so as to maximize the use of

[2] The rule does not require that n be chosen as large as possible. A scheduler that always chooses the largest possible $1 \leq n \leq p$ is said to be *greedy*.

processors at each round. If, at each stage of a computation, there are p ready tasks, then the computation will complete in w/p steps, where w is the work complexity of the program. We may, however, be unable to make full use of all p processors at any given stage. This would only be because the dependencies among computations, which are reflected in the variable occurrences and in the definition of the depth complexity of the computation, inhibits parallelism to the extent that evaluation cannot complete in fewer than d rounds. This limitation is significant only to the extent that d is larger than w/p; otherwise, the overall time is bounded by w/p, making maximal use of all p-processors.

39.5 Notes

Parallelism should not be confused with concurrency. Parallelism is about efficiency, not semantics; the meaning of a program is independent of whether it is executed in parallel or not. Concurrency is about composition, not efficiency; the meaning of a concurrent program is very weakly specified so that we may compose it with other programs without altering its meaning. This distinction, and the formulation of it given here were pioneered by Blelloch (1990). The concept of a cost semantics and the idea of a provably efficient implementation are derived from the work of Blelloch and Greiner (1995, 1996).

40 Futures and Speculations

A *future* is a computation whose evaluation is initiated in advance of any demand for its value. Like a suspension, a future represents a value that is to be determined later. Unlike a suspension, a future is always evaluated, regardless of whether its value is actually required. In a sequential setting futures are of little interest; a future of type τ is just an expression of type τ. In a parallel setting, however, futures are of interest because they provide a means of initiating a parallel computation whose result is not needed until (presumably) much later, by which time it will have been completed.

The prototypical example of the use of futures is to implementing pipelining, a method for overlapping the stages of a multistage computation to the fullest extent possible. This minimizes the latency caused by one stage waiting for the completion of a previous stage by allowing the two stages to proceed in parallel until such time as an explicit dependency is encountered. Ideally, the computation of the result of an earlier stage is completed by the time a later stage requires it. At worst the later stage must be delayed until the earlier stage completes, incurring what is known as a *pipeline stall*.

A *speculation* is a delayed computation whose result may or may not 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 actually required by the main thread. If the value of the speculation is required, 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 required by a sequential execution. Speculations, in contrast, are *work inefficient* in that speculative execution may be in vain—the overall computation may involve more steps than the work required to compute the result. For this reason speculation is a risky strategy for exploiting parallelism. It can make good use of available resources, but perhaps only at the expense of doing more work than necessary.

40.1 Futures

The syntax of futures is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`fut(`τ`)`	τ `fut`	future
Exp	e	::=	`fut(`e`)`	`fut(`e`)`	future
			`fsyn(`e`)`	`fsyn(`e`)`	synchronize

The type τ `fut` is the type of futures of type τ. Futures are introduced by the expression `fut(`e`)`, which schedules e for evaluation and returns a reference to it. Futures are eliminated by the expression `fsyn(`e`)`, which synchronizes with the future referred to by e, returning its value.

40.1.1 Statics

The statics of futures is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{fut}(e) : \texttt{fut}(\tau)} \tag{40.1a}$$

$$\frac{\Gamma \vdash e : \texttt{fut}(\tau)}{\Gamma \vdash \texttt{fsyn}(e) : \tau}\,. \tag{40.1b}$$

These rules are unsurprising, because futures add no new capabilities to the language beyond providing an opportunity for parallel evaluation.

40.1.2 Sequential Dynamics

The sequential dynamics of futures is easily defined. Futures are evaluated eagerly; synchronization returns the value of the future.

$$\frac{e\ \mathsf{val}}{\texttt{fut}(e)\ \mathsf{val}} \tag{40.2a}$$

$$\frac{e \mapsto e'}{\texttt{fut}(e) \mapsto \texttt{fut}(e')} \tag{40.2b}$$

$$\frac{e \mapsto e'}{\texttt{fsyn}(e) \mapsto \texttt{fsyn}(e')} \tag{40.2c}$$

$$\frac{e\ \mathsf{val}}{\texttt{fsyn}(\texttt{fut}(e)) \mapsto e}\,. \tag{40.2d}$$

40.2 Speculations

The syntax of (nonrecursive) speculations is given by the following grammar:[1]

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	`spec(`τ`)`	τ `spec`	speculation
Exp	e	::=	`spec(`e`)`	`spec(`e`)`	speculate
			`ssyn(`e`)`	`ssyn(`e`)`	synchronize

[1] We confine ourselves to the nonrecursive case to facilitate the comparison with futures.

The type τ `spec` is the type of speculations of type τ. The introductory form `spec(`e`)` creates a computation that may be speculatively evaluated, and the eliminatory form `ssyn(`e`)` synchronizes with a speculation.

40.2.1 Statics

The statics of speculations is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{spec}(e) : \texttt{spec}(\tau)} \tag{40.3a}$$

$$\frac{\Gamma \vdash e : \texttt{spec}(\tau)}{\Gamma \vdash \texttt{ssyn}(e) : \tau}\,. \tag{40.3b}$$

Thus the statics for speculations as given by Rules (40.3) is essentially equivalent to the statics for futures given by Rules (40.1).

40.2.2 Sequential Dynamics

The definition of the sequential dynamics of speculations is similar to that of futures, except that speculations are values:

$$\frac{}{\texttt{spec}(e)\ \texttt{val}} \tag{40.4a}$$

$$\frac{e \mapsto e'}{\texttt{spec}(e) \mapsto \texttt{spec}(e')} \tag{40.4b}$$

$$\frac{}{\texttt{ssyn}(\texttt{spec}(e)) \mapsto e}\,. \tag{40.4c}$$

The only difference compared with a future is that synchronization with a speculation may proceed even if the speculated computation is not completed.

40.3 Parallel Dynamics

Futures are interesting only insofar as they admit a parallel dynamics that allows the computation of the future to proceed 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. Interestingly, the parallel dynamics of futures and speculations is *identical*, except for the termination condition. Whereas futures require all tasks to be completed before termination, speculations may be abandoned

before they are completed. For the sake of concision we give the parallel dynamics of futures, remarking only where alterations must be made for the parallel dynamics of speculations.

The parallel dynamics of futures relies on a modest extension to the language given in Section 40.1 to introduce *names* for tasks. Let Σ be a finite mapping assigning types to names. The expression $\mathtt{fut}[a]$ is a value referring to the outcome of task a. The statics of this expression is given by the following rule:[2]

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{fut}[a] : \mathtt{fut}(\tau)}\,. \tag{40.5}$$

Rules (40.1) 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 parallel futures (or speculations) that have been activated thus far in the computation. Formally, μ is a finite mapping assigning expressions to the task names declared in Σ. A state is well-formed according to the following rule:

$$\frac{\vdash_\Sigma e : \tau \quad (\forall a \in dom(\Sigma))\ \vdash_\Sigma \mu(a) : \Sigma(a)}{\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{ok}}\,. \tag{40.6}$$

As discussed in Chapter 36, 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]

$$\frac{}{\mathtt{fut}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{40.7a}$$

$$\frac{}{\nu\ \Sigma\ \{\, \mathtt{fut}(e) \parallel \mu \,\} \mapsto_{loc} \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{fut}[a] \parallel \mu \otimes a \hookrightarrow e \,\}} \tag{40.7b}$$

$$\frac{\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto_{loc} \nu\ \Sigma'\ \{\, e' \parallel \mu' \,\}}{\nu\ \Sigma\ \{\, \mathtt{fsyn}(e) \parallel \mu \,\} \mapsto_{loc} \nu\ \Sigma'\ \{\, \mathtt{fsyn}(e') \parallel \mu' \,\}} \tag{40.7c}$$

$$\frac{e'\ \mathsf{val}_{\Sigma, a \sim \tau}}{\left\{\begin{array}{c} \nu\ \Sigma, a \sim \tau\ \{\, \mathtt{fsyn}(\mathtt{fut}[a]) \parallel \mu \otimes a \hookrightarrow e' \,\} \\ \mapsto_{loc} \\ \nu\ \Sigma, a \sim \tau\ \{\, e' \parallel \mu \otimes a \hookrightarrow e' \,\} \end{array}\right\}}\,. \tag{40.7d}$$

Rule (40.7b) activates a future named a executing the expression e and returns a reference to it. Rule (40.7d) synchronizes with a future whose value has been determined.

[2] A similar rule governs the analogous construct $\mathtt{spec}[a]$ in the case of speculations.

[3] These rules must be augmented by a reformulation of the dynamics of the other constructs of the language phrased in terms of the present notion of state.

Note that a local transition always has the form

$$\nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto_{loc} \nu\ \Sigma\ \ \Sigma'\ \{\, e' \parallel \mu \otimes \mu' \,\}$$

where Σ' is either empty or declares the type of a single symbol, and μ' is either empty or of the form $a \hookrightarrow e'$ 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.

$$\frac{\begin{array}{c}\mu = \mu_0 \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n \\ \mu'' = \mu_0 \otimes a_1 \hookrightarrow e_1' \otimes \ldots \otimes a_n \hookrightarrow e_n' \\ \nu\ \Sigma\ \{\, e \parallel \mu \,\} \mapsto_{loc}^{\mathbf{0,1}} \nu\ \Sigma\ \ \Sigma'\ \{\, e' \parallel \mu \otimes \mu' \,\} \\ (\forall 1 \le i \le n) \quad \nu\ \Sigma\ \{\, e_i \parallel \mu \,\} \mapsto_{loc} \nu\ \Sigma\ \ \Sigma_i'\ \{\, e_i' \parallel \mu \otimes \mu_i' \,\}\end{array}}{\left\{\begin{array}{c}\nu\ \Sigma\ \{\, e \parallel \mu \,\} \\ \mapsto_{glo} \\ \nu\ \Sigma\ \ \Sigma'\ \Sigma_1'\ \ldots\ \Sigma_n'\ \{\, e' \parallel \mu'' \otimes \mu' \otimes \mu_1' \otimes \ldots \otimes \mu_n' \,\}\end{array}\right\}}. \tag{40.8}$$

Rule (40.8) allows the focus expression to take either zero or one step because it may 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, whether for futures or speculations, is defined by the rule

$$\frac{}{\nu\ \emptyset\ \{\, e \parallel \emptyset \,\}\ \mathsf{initial}}\,. \tag{40.9}$$

Final states differ according to whether we are considering futures or speculations. In the case of futures a state is final iff both the focus and all parallel futures have completed evaluation:

$$\frac{e\ \mathsf{val}_\Sigma \quad \mu\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{final}} \tag{40.10a}$$

$$\frac{(\forall a \in dom(\Sigma))\ \mu(a)\ \mathsf{val}_\Sigma}{\mu\ \mathsf{val}_\Sigma}\,. \tag{40.10b}$$

In the case of speculations a state is final iff the focus is a value:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\ \Sigma\ \{\, e \parallel \mu \,\}\ \mathsf{final}}\,. \tag{40.11}$$

This corresponds to the speculative nature of the parallel evaluation of speculations whose outcome may not be needed to determine the final outcome of the program.

40.4 Applications of Futures

Pipelining provides a good example of the use of parallel futures. Consider a situation in which a *producer* builds a list whose elements represent units of work and a *consumer* traverses the work list and acts on each element of that list. The elements of the work list can be thought of as "instructions" to the consumer, which maps a function over that list to carry out those instructions. An obvious sequential implementation first builds the work list, then traverses it to perform the work indicated by the list. This is fine as long as the elements of the list can be produced quickly, but if each element requires a substantial amount of computation, it would be preferable to overlap production of the next list element with execution of the previous unit of work. This can be easily programmed by use of futures.

Let `flist` be the recursive type $\mu t\,.\,\texttt{unit} + (\texttt{nat} \times t\,\texttt{fut})$, whose elements are `nil`, defined to be $\texttt{fold}(\texttt{l} \cdot \langle\rangle)$, and $\texttt{cons}(e_1, e_2)$, defined to be $\texttt{fold}(\texttt{r} \cdot \langle e_1, \texttt{fut}(e_2)\rangle)$. The producer is a recursive function that generates a value of type `flist`:

```
fix produce : (nat → nat opt) → nat → flist is
  λ f. λ i.
    case f(i) {
      null ⇒ nil
    | just x ⇒ cons(x, fut (produce f (i+1)))
    }
```

On each iteration the producer generates a parallel future to produce the tail. This computation proceeds after the producer returns so that it overlaps subsequent computation.

The consumer folds an operation over the work list as follows:

```
fix consume : ((nat×nat)→nat) → nat → flist → nat is
  λ g. λ a. λ xs.
    case xs {
      nil ⇒ a
    | cons (x, xs) ⇒ consume g (g (x, a)) (syn xs)
    }
```

The consumer synchronizes with the tail of the work list just at the point where it makes a recursive call and hence requires the head element of the tail to continue processing. At this point the consumer will block, if necessary, to await computation of the tail before continuing the recursion.

Another application of futures is to provide more control over parallelism in a language with lazy suspensions (as described in Chapter 37). Rather than evaluate suspensions speculatively, which is not work efficient, we may instead add futures to the language in addition to suspensions. One application of futures in such a setting is called a *spark*. A spark is a computation that is executed in parallel with another purely for its effect on suspensions. The spark traverses a data structure, forcing the suspensions within so that their values are computed and stored, but otherwise yielding no useful result. The idea is that the spark forces the suspensions that will be needed by the main computation, but taking advantage of parallelism in the hope that their values will have been computed by the time the main computation requires them.

The sequential dynamics of the spark expression $\texttt{spark}(e_1; e_2)$ is simply to evaluate e_1 before evaluating e_2. This is useful in the context of a by-need dynamics for suspensions, as evaluation of e_1 will record the values of some suspensions in the memo table for subsequent use by the computation e_2. The parallel dynamics specifies, in addition, that e_1 and e_2 are to be evaluated in parallel. The behavior of sparks is captured by the definition of $\texttt{spark}(e_1; e_2)$ in terms of futures:

$$\texttt{let}\ \_\ \texttt{be}\ \texttt{fut}(e_1)\ \texttt{in}\ e_2.$$

Evaluation of e_1 commences immediately, but its value, if any, is abandoned. This encoding does not allow for evaluation of e_1 to be abandoned as soon as e_2 reaches a value, but this scenario is not expected to arise for the intended mode of use of sparks. The expression e_1 should be a quick traversal that does nothing other than force the suspensions in some data structure, exiting as soon as this is complete. Presumably this computation takes less time than it takes for e_2 to perform its work before forcing the suspensions that were forced by e_2, otherwise there is little to be gained from the use of sparks in the first place!

As an example, consider the type `strm` of streams of numbers defined by the recursive type $\mu t.(\texttt{unit} + (\texttt{nat} \times t))\ \texttt{spec}$. 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 make use of a function `mapforce` that forces successive elements of the input stream, but yields no useful output. The computation $\texttt{spark}(\texttt{mapforce}(s); \texttt{map}(f)(s))$ forces the elements of the stream in parallel with the computation of $\texttt{map}(f)(s)$, with the intention that all suspensions in s are forced before their values are required by the main computation.

As another example, we may use futures to encode binary nested parallelism by defining $\texttt{par}(e_1; e_2; x_1.x_2.e)$ to stand for the expression

$$\texttt{let}\ x_1'\ \texttt{be}\ \texttt{fut}(e_1)\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{be}\ e_2\ \texttt{in}\ \texttt{let}\ x_1\ \texttt{be}\ \texttt{fsyn}(x_1')\ \texttt{in}\ e.$$

The order of bindings is important to ensure that evaluation of e_2 proceeds in parallel with evaluation of e_1. Observe that evaluation of e cannot, in any case, proceed until both are complete.

40.5 Notes

Futures were introduced in the MultiLisp language (Halstead, 1985). The same concept was considered by Arvind et al. (1986) under the name "I-structures." The formulation given here is based on that of Greiner and Blelloch (1999).

PART XVI

Concurrency

41 Process Calculus

So far we have mainly studied the statics and dynamics of programs in isolation, without regard to their interaction with the world. But to extend this analysis to even the most rudimentary forms of input and output requires that we consider external agents that interact with the program. After all, the whole purpose of a computer is, ultimately, to interact with a person!

To extend our investigations to interactive systems, we begin with the study of *process calculi*, which are abstract formalisms that capture the essence of interaction among independent agents. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to synchronous and asynchronous communication. The calculus consists of two main syntactic categories, *processes* and *events*. The basic form of process is one that awaits the arrival of an event. Processes are closed under parallel composition (the product of processes), replication, and declaration of a channel. The basic forms of event are signaling on a channel and querying a channel; these are later generalized to sending and receiving data on a channel. Events are closed under a finite choice (sum) of events. When enriched with types of messages and channel references, the process calculus may be seen to be universal in that it is at least as powerful as the untyped λ-caclulus.

41.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an *event*, which specifies the *actions* that a process is prepared to undertake in concert with another process. Two processes interact by undertaking two complementary actions, which may be thought of as a *signal* and a *query* on a *channel*. The processes synchronize when one signals on a channel that the other is querying, after which they both proceed independently to interact with other processes.

To begin with we focus on sequential processes, which simply await the arrival of one of several possible actions, known as an event:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Proc	P	::=	`await(`E`)`	$\$\,E$	synchronize
Evt	E	::=	`null`	$\mathbf{0}$	null
			`or(`E_1`;`E_2`)`	$E_1 + E_2$	choice
			`que[`a`](`P`)`	$?a\,;P$	query
			`sig[`a`](`P`)`	$!a\,;P$	signal

The variable a ranges over symbols serving as *channel names* that mediate communication among the processes.

We do not distinguish between events that differ only up to *structural congruence*, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{E \equiv E'}{\$\,E \equiv \$\,E'} \tag{41.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{41.1b}$$

$$\frac{P \equiv P'}{?a\,;P \equiv ?a\,;P'} \tag{41.1c}$$

$$\frac{P \equiv P'}{!a\,;P \equiv !a\,;P'} \tag{41.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{41.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{41.1f}$$

$$\frac{}{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3}\,. \tag{41.1g}$$

Imposing structural congruence on sequential processes enables us to think of an event as having the form

$$!a\,;P_1 + \ldots + ?a\,;Q_1 + \ldots$$

consisting of a sum of signal and query events, with the sum of no events being the null event $\mathbf{0}$.

An illustrative example of Milner's is a simple vending machine that may take in a `2p` coin, then optionally either permit selection of a cup of tea, or take another `2p` coin, then permit selection of a cup of coffee:

$$V = \texttt{\$(?2p;\$(!tea;}V + \texttt{?2p;\$(!cof;}V\texttt{)))}.$$

As the example indicates, we permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs. (Later how to avoid reliance on recursive definitions is shown.)

Because the computation occurring within a process is suppressed, sequential processes have no dynamics on their own, but only through their interaction with other processes. For the vending machine to operate there must be another process (you) who initiates the events expected by the machine, causing both your state (the coins in your pocket) and its state (as just described) to change as a result.

41.2 Interaction

Processes become interesting when they are allowed to interact with one another to achieve a common goal. To account for interaction we enrich the language of processes with *concurrent composition*:

Sort			Abstract Form	Concrete Form	Description
Proc	P	::=	`await(`E`)`	$\$\, E$	synchronize
			`stop`	$\mathbf{1}$	inert
			`par(`P_1`;` P_2`)`	$P_1 \parallel P_2$	composition

The process $\mathbf{1}$ represents the inert process, and the process $P_1 \parallel P_2$ represents the concurrent composition of P_1 and P_2. We may identify $\mathbf{1}$ with $\$\,\mathbf{0}$, the process that awaits the event that will never occur, but we prefer to treat the inert process as a primitive concept.

We identify processes up to structural congruence, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{}{P \parallel \mathbf{1} \equiv P} \tag{41.2a}$$

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{41.2b}$$

$$\frac{}{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{41.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'}\,. \tag{41.2d}$$

Up to structural congruence every process has the form

$$\$\, E_1 \parallel \ldots \parallel \$\, E_n$$

for some $n \geq 0$, it being understood that when $n = 0$ this stands for the null process, $\mathbf{1}$.

Interaction between processes consists of synchronization of two complementary actions. The dynamics of interaction is defined by two forms of judgment. The transition judgment $P \mapsto P'$ states that the process P evolves to the process P', as a result of a single step of computation. The family of transition judgments, $P \overset{\alpha}{\mapsto} P'$, where α is an *action*, states that the process P may evolve to the process P', provided that the action α is permissible in the context in which the transition occurs (in a sense to be made precise momentarily). As a notational convenience, we often regard the unlabeled transition to be the labeled transition corresponding to the special silent action.

The possible actions are given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Act	α	::=	`que[`a`]`	$a\,?$	query
			`sig[`a`]`	$a\,!$	signal
			`sil`	ε	silent

The *query action* $a\,?$ and the *signal action* $a\,!$ are complementary, and the *silent action* ε is self-complementary. We define the *complementary* action to α to be the action $\overline{\alpha}$ given by the equations $\overline{a\,?} = a\,!$, $\overline{a\,!} = a\,?$, and $\overline{\varepsilon} = \varepsilon$:

$$\frac{}{\$\,(!a\,;P + E) \overset{a\,!}{\mapsto} P} \tag{41.3a}$$

$$\frac{}{\$\,(?a\,;P + E) \overset{a\,?}{\mapsto} P} \tag{41.3b}$$

$$\frac{P_1 \overset{\alpha}{\mapsto} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\mapsto} P_1' \parallel P_2} \tag{41.3c}$$

$$\frac{P_1 \overset{\alpha}{\mapsto} P_1' \quad P_2 \overset{\overline{\alpha}}{\mapsto} P_2' \quad \alpha \neq \varepsilon}{P_1 \parallel P_2 \mapsto P_1' \parallel P_2'}\,. \tag{41.3d}$$

Rules (41.3a) and (41.3b) specify that any of the events on which a process is synchronizing may occur. Rule (41.3d) synchronizes two processes that take complementary actions.

As an example, let us consider the interaction of the vending machine V with the user process U defined as follows:

$$U = \$\,!\mathtt{2p}\,;\$\,!\mathtt{2p}\,;\$\,?\mathtt{cof}\,;\mathbf{1}.$$

Here is a trace of the interaction between V and U:

$$\begin{aligned} V \parallel U &\mapsto \$\,(!\mathtt{tea}\,;V + ?\mathtt{2p}\,;\$\,!\mathtt{cof}\,;V) \parallel \$\,!\mathtt{2p}\,;\$\,?\mathtt{cof}\,;\mathbf{1} \\ &\mapsto \$\,!\mathtt{cof}\,;V \parallel \$\,?\mathtt{cof}\,;\mathbf{1} \\ &\mapsto V. \end{aligned}$$

These steps are justified by the following pairs of labeled transitions:

$$\begin{aligned} U &\overset{\mathtt{2p}\,!}{\longmapsto} U' = \$\,!\mathtt{2p}\,;\$\,?\mathtt{cof}\,;\mathbf{1} \\ V &\overset{\mathtt{2p}\,?}{\longmapsto} V' = \$\,(!\mathtt{tea}\,;V + ?\mathtt{2p}\,;\$\,!\mathtt{cof}\,;V) \end{aligned}$$

$$\begin{aligned} U' &\overset{\mathtt{2p}\,!}{\longmapsto} U'' = \$\,?\mathtt{cof}\,;\mathbf{1} \\ V' &\overset{\mathtt{2p}\,?}{\longmapsto} V'' = \$\,!\mathtt{cof}\,;V \end{aligned}$$

$$\begin{aligned} U'' &\overset{\mathtt{cof}\,?}{\longmapsto} \mathbf{1} \\ V'' &\overset{\mathtt{cof}\,!}{\longmapsto} V. \end{aligned}$$

We have suppressed uses of structural congruence in the preceding derivations to avoid clutter, but it is important to see its role in managing the nondeterministic choice of events by a process.

41.3 Replication

Some presentations of process calculi forego reliance on defining equations for processes in favor of a *replication* construct, which we write as $*P$. This process stands for as many concurrently executing copies of P as we may require, which may be modeled by the structural congruence

$$*P \equiv P \parallel *P. \tag{41.4}$$

Understood as a principle of structural congruence, this rule hides the steps of process creation and gives no hint as to how often it can or should be applied. We could alternatively build replication into the dynamics to model the details of replication more closely:

$$*P \mapsto P \parallel *P. \tag{41.5}$$

Because the application of this rule is unconstrained, it may be applied at any time to effect a new copy of the replicated process P.

So far we have been using recursive process definitions to define processes that interact repeatedly according to some protocol. Rather than take recursive definition as a primitive notion, we may instead use replication to model repetition. This may be achieved by introducing an "activator" process that is contacted to effect the replication. Consider the recursive definition $X = P(X)$, where P is a process expression that may refer to itself as X. Such a self-referential process may be simulated by defining the activator process

$$A = *\,\$\,(?a\,; P(\$\,(!a\,; \mathbf{1}))),$$

in which we have replaced occurrences of X within P by an initiator process that signals the event a to the activator. Observe that the activator A is structurally congruent to the process $A' \parallel A$, where A' is the process

$$\$\,(?a\,; P(\$\,(!a\,; \mathbf{1}))).$$

To start process P we concurrently compose the activator A with an initiator process, $\$\,(!a\,; \mathbf{1})$. Observe that

$$A \parallel \$\,(!a\,; \mathbf{1}) \mapsto A \parallel P(!a\,; \mathbf{1}),$$

which starts the process P while maintaining a running copy of the activator A.

As an example, let us consider Milner's vending machine, written using replication, rather than using recursive process definition:

$$V_0 = \$\,(!v\,; \mathbf{1}) \tag{41.6}$$

$$V_1 = *\,\$\,(?v\,; V_2) \tag{41.7}$$

$$V_2 = \$\,(?\mathtt{2p}\,; \$\,(!\mathtt{tea}\,; V_0 + ?\mathtt{2p}\,; \$\,(!\mathtt{cof}\,; V_0))). \tag{41.8}$$

The process V_1 is a replicated server that awaits a signal on channel v to create another instance of the vending machine. The recursive calls are replaced by signals along v to restart the machine. The original machine V is simulated by the concurrent composition $V_0 \parallel V_1$.

This example motivates a commonly considered restriction on replication that avoids the indeterminacy inherent in choosing when and whether to expand a replication into a parallel composition. To avoid this, we may replace general replication by *replicated synchronization*, which is governed by the following rules:

$$\frac{}{*\$(!a\,;P+E) \overset{a\,!}{\longmapsto} P \parallel *\$(!a\,;P+E)} \tag{41.9a}$$

$$\frac{}{*\$(?a\,;P+E) \overset{a\,?}{\longmapsto} P \parallel *\$(?a\,;P+E)}\,. \tag{41.9b}$$

The process $*\$(E)$ is to be regarded not as a composition of replication and synchronization, but as the inseparable combination of these two constructs. The advantage is that the replication occurs only as needed, precisely when a synchronization with another process is possible. This avoids the need to "guess," either by structural congruence or an explicit step, when to replicate a process.

41.4 Allocating Channels

It is often useful (particularly once we have introduced interprocess communication) to introduce new channels within a process, rather than assume that all channels of interaction are given a priori. To allow for this, the syntax of processes is enriched with a channel declaration primitive:

Sort		**Abstract Form**	**Concrete Form**	**Description**
Proc	$P \quad ::=$	`new`$(a\,.\,P)$	$\nu\, a\,.\,P$	new channel

The channel a is bound within the process P and hence may be renamed at will (avoiding conflicts) within P. To simplify notation we sometimes write $\nu\, a_1, \ldots, a_k\,.\,P$ for the iterated declaration $\nu\, a_1 \ldots \nu\, a_k\,.\,P$.

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{41.10a}$$

$$\frac{P \equiv P'}{\nu\, a\,.\,P \equiv \nu\, a\,.\,P'} \tag{41.10b}$$

$$\frac{a \notin P_2}{(\nu\, a\,.\,P_1) \parallel P_2 \equiv \nu\, a\,.\,(P_1 \parallel P_2)} \tag{41.10c}$$

$$\frac{(a \notin P)}{\nu\, a\,.\,P \equiv P}\,. \tag{41.10d}$$

Rule (41.10c), called *scope extrusion*, will be especially important in Section 41.5. Rule (41.10d) states that channels may be deallocated once they are no longer in use.

To account for the scopes of names (and to prepare for later generalizations) it is useful to introduce a static semantics for processes that ensures that names are properly scoped. A *signature* Σ is, for the time being, a finite set of channels. The judgment $\vdash_\Sigma P \ \mathsf{proc}$ states that a process P is well-formed relative to the channels declared in the signature Σ:

$$\frac{}{\vdash_\Sigma \mathbf{1} \ \mathsf{proc}} \tag{41.11a}$$

$$\frac{\vdash_\Sigma P_1 \ \mathsf{proc} \quad \vdash_\Sigma P_2 \ \mathsf{proc}}{\vdash_\Sigma P_1 \parallel P_2 \ \mathsf{proc}} \tag{41.11b}$$

$$\frac{\vdash_\Sigma E \ \mathsf{event}}{\vdash_\Sigma \$\, E \ \mathsf{proc}} \tag{41.11c}$$

$$\frac{\vdash_{\Sigma,a} P \ \mathsf{proc}}{\vdash_\Sigma \nu\, a\,.\, P \ \mathsf{proc}}\,. \tag{41.11d}$$

The foregoing rules make use of an auxiliary judgment $\vdash_\Sigma E \ \mathsf{event}$, stating that E is a well-formed event relative to Σ:

$$\frac{}{\vdash_\Sigma \mathbf{0} \ \mathsf{event}} \tag{41.12a}$$

$$\frac{\vdash_{\Sigma,a} P \ \mathsf{proc}}{\vdash_{\Sigma,a} ?a\,;P \ \mathsf{event}} \tag{41.12b}$$

$$\frac{\vdash_{\Sigma,a} P \ \mathsf{proc}}{\vdash_{\Sigma,a} !a\,;P \ \mathsf{event}} \tag{41.12c}$$

$$\frac{\vdash_\Sigma E_1 \ \mathsf{event} \quad \vdash_\Sigma E_2 \ \mathsf{event}}{\vdash_\Sigma E_1 + E_2 \ \mathsf{event}}\,. \tag{41.12d}$$

We also have need of the judgment $\vdash_\Sigma \alpha \ \mathsf{action}$ stating that α is a well-formed action relative to Σ:

$$\frac{}{\vdash_{\Sigma,a} a\,? \ \mathsf{action}} \tag{41.13a}$$

$$\frac{}{\vdash_{\Sigma,a} a\,! \ \mathsf{action}} \tag{41.13b}$$

$$\frac{}{\vdash_\Sigma \varepsilon \ \mathsf{action}}\,. \tag{41.13c}$$

The dynamics is correspondingly generalized to keep track of the set of active channels. The judgment $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ states that P transitions to P' with action α relative to channels Σ. The rules defining the dynamics are indexed forms of those previously given, augmented

by an additional rule governing the declaration of a channel. The complete set of rules is given here for the sake of clarity:

$$\frac{}{\$\,(!a\,;P+E)\underset{\Sigma,a}{\overset{a\,!}{\longmapsto}}P} \tag{41.14a}$$

$$\frac{}{\$\,(?a\,;P+E)\underset{\Sigma,a}{\overset{a\,?}{\longmapsto}}P} \tag{41.14b}$$

$$\frac{P_1\underset{\Sigma}{\overset{\alpha}{\longmapsto}}P_1'}{P_1\parallel P_2\underset{\Sigma}{\overset{\alpha}{\longmapsto}}P_1'\parallel P_2} \tag{41.14c}$$

$$\frac{P_1\underset{\Sigma}{\overset{\alpha}{\longmapsto}}P_1'\quad P_2\underset{\Sigma}{\overset{\overline{\alpha}}{\longmapsto}}P_2'\quad \alpha\neq\varepsilon}{P_1\parallel P_2\underset{\Sigma}{\longmapsto}P_1'\parallel P_2'} \tag{41.14d}$$

$$\frac{P\underset{\Sigma,a}{\overset{\alpha}{\longmapsto}}P'\quad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\,a\,.\,P\underset{\Sigma}{\overset{\alpha}{\longmapsto}}\nu\,a\,.\,P'}\,. \tag{41.14e}$$

Rule (41.14e) states that no process may interact with $\nu\,a\,.\,P$ along the locally allocated channel a, because to do so would require that a already be declared in Σ, which is precluded by the freshness convention on binders.

As an example, let us consider again the definition of the vending machine using replication, rather than recursion. The channel v used to initialize the machine should be considered private to the machine itself and not be made available to a user process. This is naturally expressed by the process expression $\nu\,v\,.\,(V_0\parallel V_1)$, where V_0 and V_1 are as previously defined using the designated channel v. This process correctly simulates the original machine V because it precludes interaction with a user process on channel v. If U is a user process, the interaction begins as follows:

$$(\nu\,v\,.\,(V_0\parallel V_1))\parallel U\underset{\Sigma}{\longmapsto}(\nu\,v\,.\,V_2)\parallel U\equiv\nu\,v\,.\,(V_2\parallel U).$$

(The processes V_0, V_1, and V_2 are those defined earlier.) The interaction continues as before, albeit within the scope of the binder, provided that v has been chosen (by structural congruence) to be apart from U, ensuring that it is private to the internal workings of the machine.

41.5 Communication

Synchronization is the coordination of the execution of two processes that are willing to undertake the complementary actions of signaling and querying a common channel. *Synchronous communication* is a natural generalization of synchronization to allow more than

one bit of data to be communicated between two coordinating processes, a *sender* and a *receiver*. In principle any type of data may be communicated from one process to another, and we can give a uniform account of communication that is independent of the type of data communicated between processes. Communication becomes more interesting in the presence of a type of *channel references*, which allow access to a communication channel to be propagated from one process to another, allowing alteration of the interconnection topology among processes during execution. (Channel references are discussed in Section 41.6.)

To account for interprocess communication we must enrich the language of processes to include *variables*, as well as *channels*, in the formalism. Variables range, as always, over types, and are given meaning by substitution. Channels, on the other hand, are assigned types that classify the data carried on that channel and are given meaning by send and receive events that generalize the signal and query events considered earlier. The abstract syntax of communication events is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Evt	E	$::=$	$\mathtt{snd}[a](e; P)$	$!\,a(e; P)$	send
			$\mathtt{rcv}[a](x\,.\,P)$	$?\,a(x\,.\,P)$	receive

The event $\mathtt{rcv}[a](x\,.\,P)$ represents the receipt of a value x on the channel a, passing x to the process P. The variable x is bound within P and hence may be chosen freely, subject to the usual restrictions on the choice of names of bound variables. The event $\mathtt{snd}[a](e; P)$ represents the transmission of (the value of) the expression e on channel a, continuing with the process P only once this value has been received.

To account for the type of data that may be sent on a channel, the syntax of channel declaration is generalized to associate a type with each channel name:

Sort			Abstract Form	Concrete Form	Description
Proc	P	$::=$	$\mathtt{new}[\tau](a\,.\,P)$	$\nu\, a{\sim}\tau\,.\,P$	typed channel

The process $\mathtt{new}[\tau](a\,.\,P)$ introduces a new channel name a with associated type τ for use within the process P. The name a is bound within P, and hence may be chosen at will, subject only to avoidance of confusion of distinct names.

The statics of communication extends that of synchronization by associating types with channels and by considering variables that range over a type. The judgment $\Gamma \vdash_\Sigma P\ \mathsf{proc}$ states that P is a well-formed process involving the channels declared in Σ and the variables declared in Γ. It is inductively defined by the following rules, wherein we assume that the typing judgment $\Gamma \vdash_\Sigma e : \tau$ is given separately:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{41.15a}$$

$$\frac{\Gamma \vdash_\Sigma P_1\ \mathsf{proc} \quad \Gamma \vdash_\Sigma P_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{41.15b}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} P\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a{\sim}\tau\,.\,P\ \mathsf{proc}} \tag{41.15c}$$

$$\frac{\Gamma \vdash_\Sigma E\ \mathsf{event}}{\Gamma \vdash_\Sigma \$\,E\ \mathsf{proc}}\,. \tag{41.15d}$$

Rules (41.15) make use of the auxiliary judgment $\Gamma \vdash_\Sigma E$ event, stating that E is a well-formed event relative to Γ and Σ, which is defined as follows:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0} \text{ event}} \tag{41.16a}$$

$$\frac{\Gamma \vdash_\Sigma E_1 \text{ event} \quad \Gamma \vdash_\Sigma E_2 \text{ event}}{\Gamma \vdash_\Sigma E_1 + E_2 \text{ event}} \tag{41.16b}$$

$$\frac{\Gamma, x : \tau \vdash_{\Sigma, a\sim\tau} P \text{ proc}}{\Gamma \vdash_{\Sigma, a\sim\tau} ?a(x.P) \text{ event}} \tag{41.16c}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} e : \tau \quad \Gamma \vdash_{\Sigma, a\sim\tau} P \text{ proc}}{\Gamma \vdash_{\Sigma, a\sim\tau} !a(e;P) \text{ event}}. \tag{41.16d}$$

Rule (41.16d) makes use of a typing judgment for expressions that ensures that the type of a channel is respected by communication.

The dynamics of synchronous communication is similarly an extension of the dynamics of synchronization. Actions are generalized to include the transmitted value, as well as the channel and its orientation:

Sort			Abstract Form	Concrete Form	Description
Act	α	$::=$	`rcv[`a`](`e`)`	$a\,?\,e$	receive
			`snd[`a`](`e`)`	$a\,!\,e$	send
			`sil`	ε	silent

Complementarity is defined, essentially as before, to switch the orientation of an action: $\overline{a\,?\,e} = a\,!\,e$, $\overline{a\,!\,e} = a\,?\,e$, and $\overline{\varepsilon} = \varepsilon$.

The statics ensures that the expression associated with these actions is a value of a type suitable for the channel:

$$\frac{\vdash_{\Sigma, a\sim\tau} e : \tau \quad e \text{ val}_{\Sigma, a\sim\tau}}{\vdash_{\Sigma, a\sim\tau} a\,!\,e \text{ action}} \tag{41.17a}$$

$$\frac{\vdash_{\Sigma, a\sim\tau} e : \tau \quad e \text{ val}_{\Sigma, a\sim\tau}}{\vdash_{\Sigma, a\sim\tau} a\,?\,e \text{ action}} \tag{41.17b}$$

$$\frac{}{\vdash_\Sigma \varepsilon \text{ action}}. \tag{41.17c}$$

The dynamics of synchronous communication is defined by replacing Rules (41.14a) and (41.14b) with the following rules:

$$\frac{e \xmapsto[\Sigma, a\sim\tau]{} e'}{\$\,(!a(e;P) + E) \xmapsto[\Sigma, a\sim\tau]{} \$\,(!a(e';P) + E)} \tag{41.18a}$$

$$\frac{e \text{ val}_{\Sigma, a\sim\tau}}{\$\,(!a(e;P) + E) \xmapsto[\Sigma, a\sim\tau]{a!e} P} \tag{41.18b}$$

$$\frac{e \text{ val}_{\Sigma, a\sim\tau}}{\$\,(?a(x.P) + E) \xmapsto[\Sigma, a\sim\tau]{a?e} [e/x]P}. \tag{41.18c}$$

Rule (41.18c) is nondeterministic in that it "guesses" the value e to be received along channel a. Rules (41.18) make reference to the dynamics of expressions, which is left unspecified here so as to avoid an a priori commitment as to the nature of values communicated on a channel.

Using synchronous communication, both the sender and the receiver of a message are blocked until the interaction is completed. This means that the sender must be notified whenever a message is received, and hence there must be an implicit reply channel from the receiver to the sender for the notification. This suggests that synchronous communication may be decomposed into a simpler *asynchronous send* operation, which transmits a message on a channel without waiting for its receipt, together with *channel passing* to transmit an acknowledgment channel along with the message data.

Asynchronous communication is defined by removing the synchronous send event from the process calculus and adding a new form of process that simply sends a message on a channel. The syntax of asynchronous send is as follows:

Sort			Abstract Form	Concrete Form	Description
Proc	P	$::=$	$\texttt{asnd}[a](e)$	$!\,a(e)$	send

The process $\texttt{asnd}[a](e)$ sends the message e on channel a and then terminates immediately. Without the synchronous send event, every event is, up to structural congruence, a choice of zero or more read events. The statics of asnychronous send is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau} !\,a(e)\ \mathsf{proc}}\,. \tag{41.19}$$

The dynamics is similarly straightforward:

$$\frac{e\ \mathsf{val}_\Sigma}{!\,a(e) \xmapsto[\Sigma]{a!e} \mathbf{1}}\,. \tag{41.20}$$

The rule for interprocess communication given earlier remains unchanged, because the action associated with the asychronous send is the same as in the synchronous case. We may regard a pending asynchronous send as a "buffer" in which the message is held until a receiver is selected.

41.6 Channel Passing

An interesting case of interprocess communication arises when one process passes one channel to another along a common channel. The channel passed by the sending process need not have been known a priori to the receiving process. This allows for new patterns of communication to be established among processes. For example, two processes P and Q may share a channel a along which they may send and receive messages. If the scope of a is limited to these processes, then no other process R may communicate on that channel; it is, in effect, a *private* channel between P and Q.

It frequently arises, however, that P and Q wish to include the process R in their conversation in a controlled manner. This may be accomplished by first expanding the scope of the channel a to encompass R, then sending (a reference to) the channel a to R along a prearranged channel. On receipt of the channel reference, R may communicate with P and Q using send and receive operations that act on channel references. Bearing in mind that channels are not themselves forms of expression, such a scenario can be enacted by introducing a type τ `chan` whose values are references to channels carrying values of type τ. The elimination forms for the channel type are send and receive operations that act on references, rather than explicitly given channels.[1]

Such a situation may be described schematically by the process expression

$$(\nu\, a{\sim}\tau\,.\,(P \parallel Q)) \parallel R,$$

in which the process R is initially excluded from the scope of the channel a, whose scope encompasses both the processes P and Q. The type τ represents the type of data communicated along channel a; it may be chosen arbitrarily for the sake of this example. The processes P and Q may communicate with each other by sending and receiving along channel a. If these two processes wish to include R in the conversation, then they must communicate the identity of channel a to the process R along some prearranged channel b. If a is a channel carrying values of type τ, then b will be a channel carrying values of type τ `chan`, which are references to τ-carrying channels. The channel b must be known to at least one of P and Q, and also to channel R. This can be described by the following process expression:

$$\nu\, b{\sim}\tau\ \texttt{chan}.((\nu\, a{\sim}\tau\,.\,(P \parallel Q)) \parallel R).$$

Suppose that P wishes to include R in the conversation by sending a reference to the channel a along b. The process R correspondingly receives a *reference* to a channel on the channel b and commences communication with P and Q along that channel. Thus P has the form $\$\,(!\,b(\&\,a\,;P'))$ and R has the form $\$\,(?\,b(x\,.\,R'))$. The overall process has the form

$$\nu\, b{\sim}\tau\ \texttt{chan}.(\nu\, a{\sim}\tau\,.(\$\,(!\,b(\&\,a\,;P')) \parallel Q) \parallel \$\,(?\,b(x\,.\,R'))).$$

Note carefully that the declaration of the channel b specifies that it *carries* a channel reference, not that it *is* a channel reference.

The process P is prepared to send a reference to the channel a along the channel b, where it may be received by the process R. But the scope of a is limited to processes P and Q, so in order for the communication to succeed, we must first expand its scope to encompass R using the concept of scope extrusion introduced in Section 41.4 to obtain the structurally equivalent process

$$\nu\, b{\sim}\tau\ \texttt{chan}.\nu\, a{\sim}\tau\,.(\$\,(!\,b(\&\,a\,;P')) \parallel Q \parallel \$\,(?\,b(x\,.\,R'))).$$

[1] It may be helpful to compare channel types with reference types as described in Chapters 35 and 36. Channels correspond to assignables, and channel types correspond to reference types.

The scope of a has been expanded to encompass R, preparing the ground for communication between P and R, which results in the process

$$\nu\, b{\sim}\tau\ \mathtt{chan}.\nu\, a{\sim}\tau\,.\,(P' \parallel Q \parallel [\&\,a/x]R').$$

The reference to the channel a has been substituted for the variable x within R'.

The process R may now communicate with P and Q by sending and receiving messages along the channel referenced by x. This is accomplished using dynamic forms of send and receive in which the channel on which to communicate is determined by evaluation of an expression, rather than specified statically by an explicit channel name. For example, to send a message e of type τ along the channel referred to by x, the process R' would have the form

$$\$\,(!!\,(x\,;e\,;R'')).$$

Similarly, to receive along the referenced channel, the process R' would have the form

$$\$\,(??\,(x\,;y\,.\,R'')).$$

In both cases the dynamic communication forms evolve to the static communication forms once the referenced channel has been determined.

The syntax of channel types is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	$\mathtt{chan}(\tau)$	τ chan	channel type
Exp	e	::=	$\mathtt{ch}[a]$	$\&\,a$	reference
Evt	E	::=	$\mathtt{sndref}(e_1;e_2;P)$	$!!\,(e_1\,;e_2\,;P)$	send
			$\mathtt{rcvref}(e;x\,.\,P)$	$??\,(e\,;x\,.\,P)$	receive

The events $\mathtt{sndref}(e_1;e_2;P)$ and $\mathtt{rcvref}(e;x\,.\,P)$ are dynamic versions of the events $\mathtt{snd}[a]\,(e;P)$ and $\mathtt{rcv}[a]\,(x\,.\,P)$ in which the channel is determined dynamically by evaluation of an expression, rather than statically as a fixed parameter of the event.

The statics of channel references is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \&\,a : \tau\ \mathtt{chan}} \tag{41.21a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\ \mathtt{chan} \quad \Gamma \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma !!\,(e_1\,;e_2\,;P)\ \mathsf{event}} \tag{41.21b}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau\ \mathtt{chan} \quad \Gamma, x : \tau \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma ??\,(e\,;x\,.\,P)\ \mathsf{event}} \tag{41.21c}$$

The introduction of channel references requires that events be evaluated to determine the referent of a dynamically determined channel. This is accomplished by adding the following rules for evaluation of a synchronizing process:

$$\frac{e\ \mathsf{val}_\Sigma}{\$\,(!!\,(\&\,a\,;e\,;P) + E) \xrightarrow[\Sigma,a\sim\tau]{} \$\,(!\,a(e\,;P) + E)} \tag{41.22a}$$

$$\frac{}{\$\,(??\,(\&\,a\,;x\,.\,P) + E) \xrightarrow[\Sigma,a\sim\tau]{} \$\,(?\,a(x\,.\,P) + E)}\,. \tag{41.22b}$$

In addition we require rules for evaluating each of the constituent expressions of a dynamically determined event; these rules are omitted here for the sake of concision.

41.7 Universality

In the presence of both channel references and recursive types the process calculus with communication is a universal programming language. One way to prove this is to show that it is capable of encoding the untyped λ-calculus with a call-by-name dynamics (see Chapter 17). The main idea of the encoding is to associate with each untyped λ-term, u, a process that represents it. This encoding is defined by induction on the structure of the untyped term u. For the sake of the induction, the representation is defined relative to a channel reference that represents the context in which the term occurs. Because every term in the untyped λ-calculus is a function, the context consists of an *argument* and the *continuation* for the result of the application. Because of the by-name interpretation of application, variables are represented by references to "servers" that listen on a channel for a channel reference representing a call site and activate their bindings with that channel reference.

We write $u\,@\,z$, where u is an untyped λ-term and z is a channel reference representing the continuation of u. The free variables of u are represented by channels on which we may pass an argument and a continuation. Thus the channel reference z is a value of type π, and a free variable x is a value of type $\pi\ \mathtt{chan}$. The type π is chosen to satisfy the isomorphism

$$\pi \cong (\pi\ \mathtt{chan} \times \pi)\ \mathtt{chan}.$$

That is, a continuation is a channel on which is passed an argument and another continuation. An argument, in turn, is a channel on which is passed a continuation.

The encoding of untyped λ-terms as processes is given by the following equations:

$$\begin{aligned}
x\,@\,z &= !!\,(x;z)\\
\lambda\,(x)\,u\,@\,z &= \$\,??\,(\mathtt{unfold}(z);\langle x, z'\rangle.\,u\,@\,z')\\
u_1(u_2)\,@\,z &= \nu\, a_1{\sim}\pi\ \mathtt{chan} \times \pi\,.\,(u_1\,@\,\mathtt{fold}(\&\,a_1)) \parallel \nu\, a{\sim}\pi\,.*\$\,?\,a(z_2.\,u_2\,@\,z_2) \parallel\, !\,a_1(\langle \&\,a, z\rangle).
\end{aligned}$$

Here we have taken a few liberties with the syntax for the sake of readability. We use the asynchronous form of dynamic send operation, because there is no need to be aware of the receipt of the message. Moreover, we use a product pattern, rather than explicit projections, in the dynamic receive to obtain the components of a pair.

The use of static and dynamic communication operations in the translation merits careful examination. The call site of a λ-term is determined dynamically; we cannot predict at translation time the continuation of the term. In particular, the binding of a variable may be used at many different call sites, corresponding to the multiple possible uses of that variable.

However, the channel associated with an argument is determined statically. The server associated with the variable listens on a statically determined channel for a continuation in which to evaluate its binding, which, as just remarked, is determined dynamically.

As a quick check on the correctness of the representation, consider the following derivation:

$$
\begin{aligned}
&(\lambda\,(x)\,x)\,(y)\,@\,z\\
&\quad\mapsto^* \nu\, a_1{\sim}\tau\,.\,(\$\,?\,a_1\,(\langle x, z'\rangle\,.\,!!\,(x\,;z'))) \parallel \nu\, a{\sim}\pi\,.*\$\,?\,a\,(z_2\,.\,!!\,(y\,;z_2)) \parallel\, !\,a_1\,(\langle \&\,a, z\rangle)\\
&\quad\mapsto^* \nu\, a{\sim}\pi\,.*\$\,?\,a\,(z_2\,.\,!!\,(y\,;z_2)) \parallel\, !\,a\,(z)\\
&\quad\mapsto^* \nu\, a{\sim}\pi\,.*\$\,?\,a\,(z_2\,.\,!!\,(y\,;z_2)) \parallel\, !!\,(y\,;z)
\end{aligned}
$$

Apart from the idle server process listening on channel a, this is just the translation $y\,@\,z$. (Using the methods to be developed in detail in Chapter 50, we may show that the result of the computation step is "bisimilar" to the translation of $y\,@\,z$, and hence equivalent to it for all purposes.)

41.8 Notes

Process calculi as models of concurrency and interaction were introduced and extensively developed by Hoare (1978) and Milner (1999). Milner's original formulation, CCS, was introduced to model pure synchronization, whereas Hoare's, formulation, CSP, included value-passing. CCS was subsequently extended to become the π-calculus (Milner, 1999), which includes channel-passing. Dozens upon dozens of variations and extensions of CSP, CCS, and the π-calculus have been considered in the literature, and continue to be a subject of intensive study. (See the work of Engberg and Nielsen, 2000, for an account of some of the critical developments in the area.)

The process calculus given here is derived from the π-calculus as presented by Milner (1999). (In particular, the vending machine example is adapted from Milner's monograph.) Unlike Milner's account (but like related formalisms such as that of Abadi and Fournet, 2001) we enforce a distinction between variables and names. Variables are given meaning by substitution; a type is the range of significance of a variable, the collection of values that may be substituted for it. Names, on the other hand, are given meaning by the operations associated with them; the type associated with a name is the type of data associated with the operations defined on it. The distinction between variables and names is important, because disequality is well-defined for names, but not for variables. We use the general concept of a reference to pass channel names as data; this is sufficient to ensure the universality of the process calculus.

The distinction drawn here between static and dynamic events (that is, those that are given syntactically versus those that arise by evaluation) flows naturally from the prior distinction between variables and names. It is possible to formulate the process calculus so that all uses of names are suppressed, but then the dynamics cannot be expressed using

only the machinery of the calculus itself, but instead must be augmented by an internal concept of names. It seems preferable, in the interest of maintaining a structural operational semantics, to work with a formalism that is closed under its own execution rules. The concept of dynamic events is taken one step further in Concurrent ML (Reppy, 1999), wherein events are values of an event type (see also Chapter 42).

42 Concurrent Algol

In this chapter we integrate concurrency into the framework of Modernized Algol described in Chapter 35. The resulting language, called Concurrent Algol, $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$, illustrates the integration of the mechanisms of the process calculus described in Chapter 41 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 41 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 41, may be identified with the concept of a *dynamic class* as described in Chapter 34. 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 may be passed around but not examined. In this way we can model not only the mechanisms described in Chapter 41, but also formulate an abstract account of encryption and decryption in a network using the methods described in Chapter 41.

The formulation of Concurrent Algol is based on a modal separation between commands and expressions, much as in Modernized Algol. It is also possible to consolidate these two levels (so as to allow benign concurrency effects), but this approach is not developed in detail here.

42.1 Concurrent Algol

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$ is obtained by stripping out assignables from $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup\}$, and adding a syntactic level of *processes*:

Sort			Abstract Form	Concrete Form	Description
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
			$\mathtt{proc}(m)$	$\mathtt{proc}(m)$	atomic
			$\mathtt{par}(p_1; p_2)$	$p_1 \parallel p_2$	parallel
			$\mathtt{new}[\tau](a.p)$	$\nu\, a{\sim}\tau\,.\,p$	new channel

The process $\mathtt{proc}(m)$ is an atomic process executing the command m. The other forms of process are adapted from Chapter 41. 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 is given by the judgments $\Gamma \vdash_\Sigma e : \tau$ and $\Gamma \vdash_\Sigma m \sim \tau$ introduced in Chapter 35, augmented by the judgment $\vdash_\Sigma p\ \mathsf{proc}$ stating that p is a well-formed process over the signature Σ. The latter judgment is defined by the following rules:

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{42.1a}$$

$$\frac{\vdash_\Sigma m \sim \tau}{\vdash_\Sigma \mathtt{proc}(m)\ \mathsf{proc}} \tag{42.1b}$$

$$\frac{\vdash_\Sigma p_1\ \mathsf{proc} \quad \vdash_\Sigma p_2\ \mathsf{proc}}{\vdash_\Sigma p_1 \parallel p_2\ \mathsf{proc}} \tag{42.1c}$$

$$\frac{\vdash_{\Sigma, a\sim\tau} p\ \mathsf{proc}}{\vdash_\Sigma \nu\, a{\sim}\tau\,.\,p\ \mathsf{proc}}\,. \tag{42.1d}$$

Processes are tacitly identified up to structural equivalence, as described in Chapter 41.

The transition judgment $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ states that the process p evolves in one step to the process p' with associated action α. The particular actions are specified when specific commands are introduced in Section 42.2. As in Chapter 41 we assume that with each action is associated a complementary action and that the silent action indexes the unlabeled transition judgment:

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma'\{m' \parallel p\}}{\mathtt{proc}(m) \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, \Sigma'\{\mathtt{proc}(m') \parallel p\}} \tag{42.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{proc}(\mathtt{ret}\,e) \underset{\Sigma}{\mapsto} \mathbf{1}} \tag{42.2b}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1'}{p_1 \parallel p_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \parallel p_2} \tag{42.2c}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \quad p_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} p_2'}{p_1 \parallel p_2 \underset{\Sigma}{\mapsto} p_1' \parallel p_2'} \tag{42.2d}$$

$$\frac{p \overset{\alpha}{\underset{\Sigma, a \sim \tau}{\longmapsto}} p' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a \sim \tau \,.\, p \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a \sim \tau \,.\, p'}\,. \tag{42.2e}$$

Rule (42.2a) states that a step of execution of the atomic process $\texttt{proc}(m)$ consists of a step of execution of the command m, which may result in the allocation of some set Σ' of symbols and the creation of a concurrent process p. This rule implements scope extrusion for classes (channels) by expanding the scope of the declaration of a channel to the context in which the command m occurs. Rule (42.2b) states that a completed command evolves to the inert (stopped) process; processes are executed solely for their effect and not for their value. The remaining rules are those of the process calculus that define the interaction between processes and the allocation of symbols within a process.

The auxiliary judgment $m \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ m' \parallel p' \}$ defines the execution behavior of commands. It states that the command m transitions to the command m' while creating new channels Σ' and new processes p'. The action α specifies the interactions of which m is capable when executed. As a notational convenience we drop mention of the new channels or processes when either are trivial. It is important that the right-hand side of this judgment be construed as a triple consisting of Σ', m', and p', rather than as a process expression comprising these parts.

The general rules defining this auxiliary judgment are as follows:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \texttt{ret}\, e'} \tag{42.3a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ m_1' \parallel p' \}}{\texttt{bnd}\, x \leftarrow \texttt{cmd}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ \texttt{bnd}\, x \leftarrow \texttt{cmd}\, m_1' \,;\, m_2 \parallel p' \}} \tag{42.3b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{bnd}\, x \leftarrow \texttt{cmd}\, \texttt{ret}\, e \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} [e/x] m_2} \tag{42.3c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\texttt{bnd}\, x \leftarrow e_1 \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \texttt{bnd}\, x \leftarrow e_1' \,;\, m_2}\,. \tag{42.3d}$$

These generic rules are supplemented by rules governing commands for communication and synchronization among processes.

42.2 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 34. 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 34.4 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:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Cmd	m	::=	`spawn(`e`)`	`spawn(`e`)`	spawn
			`emit(`e`)`	`emit(`e`)`	emit message
			`acc`	`acc`	accept message
			`newch[`τ`]`	`newch`	new channel

The command `spawn(`e`)` spawns a process that executes the encapsulated command given by e. The commands `emit(`e`)` and `acc` emit and accept messages, which are classified values whose class is the channel on which the message is sent. The command `newch[`τ`]` returns a reference to a fresh class carrying values of type τ.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd(unit)}}{\Gamma \vdash_\Sigma \texttt{spawn}(e) \sim \texttt{unit}} \tag{42.4a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma \texttt{emit}(e) \sim \texttt{unit}} \tag{42.4b}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{acc} \sim \texttt{clsfd}} \tag{42.4c}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{newch[}\tau\texttt{]} \sim \texttt{class(}\tau\texttt{)}}\,. \tag{42.4d}$$

The execution of commands for broadcast communication is defined by these rules:

$$\frac{}{\texttt{spawn(cmd(}m\texttt{))} \underset{\Sigma}{\overset{\varepsilon}{\Rightarrow}} \texttt{ret}\,\langle\rangle \parallel \texttt{proc}(m)} \tag{42.5a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{spawn}(e) \underset{\Sigma}{\overset{\varepsilon}{\Rightarrow}} \texttt{spawn}(e')} \tag{42.5b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{emit}(e) \overset{e!}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\,\langle\rangle} \tag{42.5c}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{emit}(e) \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{emit}(e')} \tag{42.5d}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{acc} \overset{e?}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\,e} \tag{42.5e}$$

$$\frac{}{\mathtt{newch}[\tau] \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \nu\, a{\sim}\tau\,.\mathtt{ret}\,(\&a)} \tag{42.5f}$$

Rule (42.5c) specifies that `emit(`e`)` has the effect of emitting the message e. Correspondingly, Rule (42.5e) specifies that `acc` may accept (any) message that is being sent.

As usual, the preservation theorem for $\mathcal{L}\{\mathtt{nat\ cmd} \rightharpoonup \|\}$ ensures that well-typed programs remain well-typed during execution. The proof of preservation requires a lemma governing the execution of commands. First, let us define the judgment $\vdash_\Sigma\ a\ \mathsf{action}$ by the following rules:

$$\frac{}{\vdash_\Sigma\ \varepsilon\ \mathsf{action}} \tag{42.6a}$$

$$\frac{\vdash_\Sigma\ e : \mathtt{clsfd}}{\vdash_\Sigma\ e\,!\ \mathsf{action}} \tag{42.6b}$$

$$\frac{\vdash_\Sigma\ e : \mathtt{clsfd}}{\vdash_\Sigma\ e\,?\ \mathsf{action}}\,. \tag{42.6c}$$

Lemma 42.1. *If* $m \overset{a}{\underset{\Sigma}{\Longrightarrow}} \nu\,\Sigma'\,\{\,m' \parallel p'\,\}$ *and* $\vdash_\Sigma\ m \sim \tau$*, then* $\vdash_\Sigma\ a\ \mathsf{action}$*,* $\vdash_{\Sigma\,\Sigma'}\ m' \sim \tau$*, and* $\vdash_{\Sigma\,\Sigma'}\ p'\ \mathsf{proc}$*.*

Proof By induction on Rules (42.3). □

With this in hand the proof of preservation is straightforward.

Theorem 42.2 (Preservation)**.** *If* $\vdash_\Sigma\ p\ \mathsf{proc}$ *and* $p \underset{\Sigma}{\mapsto} p'$*, then* $\vdash_\Sigma\ p'\ \mathsf{proc}$*.*

Proof By induction on transition, appealing to Lemma 42.1 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 the execution of a process to get stuck.

Theorem 42.3 (Progress)**.** *If* $\vdash_\Sigma p$ *proc, then either* $p \equiv \mathbf{1}$*, or there exists* p' *and* α *such that* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$.

Proof By induction on Rules (42.1) and (42.4). □

The assumption that there exists an action rules out degenerate situations in which there are no channels or all channels carry values of an empty type.

42.3 Selective Communication

Broadcast communication provides no means of restricting acceptance to messages of a particular class (that is, of messages on a particular channel). Using broadcast communication we may restrict attention to a particular channel a of type τ by running the following command:

$$\mathtt{fix}\, \mathit{loop} : \tau\ \mathtt{cmd}\ \mathtt{is}\ \{x \leftarrow \mathtt{acc}\,;\, \mathtt{match}\, x\ \mathtt{as}\, a \cdot y \Rightarrow \mathtt{ret}\, y\ \mathtt{ow} \Rightarrow \mathtt{emit}(x)\,;\, \mathtt{do}\, \mathit{loop}\}$$

This command is always capable of receiving a broadcast message. When one arrives, it is examined to determine whether it is classified by the class a. If so, the underlying value is returned; otherwise the message is rebroadcast to make it available to another process that may be executing a similar command. *Polling* consists of repeatedly executing the preceding command until such time as a message of channel a is successfully accepted, if ever.

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 that are sent on any of several possible channels. This may be accomplished by introducing a type $\mathtt{event}(\tau)$ of *events* consisting of a finite choice of accepts, all of whose associated payload has the type τ:

Sort			Abstract Form	Concrete Form	Description
Typ	τ	::=	$\mathtt{event}(\tau)$	$\tau\ \mathtt{event}$	events
Exp	e	::=	$\mathtt{rcv}[a]$	$?a$	select
			$\mathtt{never}[\tau]$	$\mathtt{never}$	null
			$\mathtt{or}(e_1; e_2)$	$e_1\ \mathtt{or}\ e_2$	choice
Cmd	m	::=	$\mathtt{sync}(e)$	$\mathtt{sync}(e)$	synchronize

Events in $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \|\|\}$ correspond directly to those of the *asynchronous* process calculus described in Chapter 41. One difference is that the select event need not carry with it a continuation, as it does in the process calculus; this is handled by the ambient monadic structure on commands. (However, note that all events in a choice share the same continuation, whereas in process calculus a separate continuation is associated with each event in a choice.) Another difference between the two formalisms is that in $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \|\|\}$ events are values of the type $\tau\ \mathtt{event}$, whereas in the process calculus events are not regarded as a form of expression.

The statics of event expressions is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{rcv}[a] : \mathtt{event}(\tau)} \tag{42.7a}$$

$$\frac{}{\Gamma \vdash_{\Sigma} \mathtt{never}[\tau] : \mathtt{event}(\tau)} \tag{42.7b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \mathtt{event}(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \mathtt{event}(\tau)}{\Gamma \vdash_{\Sigma} \mathtt{or}(e_1; e_2) : \mathtt{event}(\tau)} \tag{42.7c}$$

The corresponding dynamics is defined by these rules:

$$\frac{}{\mathtt{rcv}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{42.8a}$$

$$\frac{}{\mathtt{never}[\tau]\ \mathsf{val}_{\Sigma}} \tag{42.8b}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma} \quad e_2\ \mathsf{val}_{\Sigma}}{\mathtt{or}(e_1; e_2)\ \mathsf{val}_{\Sigma}} \tag{42.8c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1'; e_2)} \tag{42.8d}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma} \quad e_2 \underset{\Sigma}{\mapsto} e_2'}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1; e_2')}\,. \tag{42.8e}$$

Event values are identified up to structural congruence as described in Chapter 41. This ensures that the ordering of events in a choice is immaterial.

Channel references (see Section 34.2) give rise to an additional form of event, $\mathtt{rcvref}(e)$, in which the argument, e, is a reference to the channel on which to accept a message. Its statics is given by the rule

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{class}(\tau)}{\Gamma \vdash_{\Sigma} \mathtt{rcvref}(e) : \mathtt{event}(\tau)}\,. \tag{42.9}$$

Its dynamics is defined to dereference its argument and evaluate to an accept event for the referenced channel:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{rcvref}(e) \underset{\Sigma}{\mapsto} \mathtt{rcvref}(e')} \tag{42.10a}$$

$$\frac{}{\mathtt{rcvref}(\&a) \underset{\Sigma, a \sim \tau}{\longmapsto} \mathtt{rcv}[a]}\,. \tag{42.10b}$$

Turning now to the synchronization command, the statics is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{event}(\tau)}{\Gamma \vdash_{\Sigma} \mathtt{sync}(e) \sim \tau}\,. \tag{42.11}$$

Its execution is defined by these rules:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{sync}(e) \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \texttt{sync}(e')} \tag{42.12a}$$

$$\frac{e \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} m}{\texttt{sync}(e) \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} m}\,. \tag{42.12b}$$

Rule (42.12b) specifies that synchronization may take any action engendered by the event given as argument.

The possible actions engendered by an event value are defined by the judgment $e \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} m$, which states that the event value e engenders action α and activates command m. It is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau} \quad \vdash_{\Sigma,a\sim\tau} e : \tau}{\texttt{rcv}[a] \overset{a\cdot e\,?}{\underset{\Sigma,a\sim\tau}{\Longrightarrow}} \texttt{ret}(e)} \tag{42.13a}$$

$$\frac{e_1 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} m_1}{\texttt{or}(e_1; e_2) \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} m_1}\,. \tag{42.13b}$$

Rule (42.13a) states that an acceptance on a channel a may synchronize only with messages classified by a. In conjunction with the identification of event values up to structural congruence Rule (42.13b) states that any event among a set of choices may be engender an action.

Selective communication and dynamic events may be used together to implement a communication protocol in which a channel reference is passed on a channel in order to establish a communication path with the recipient. Let a be a channel carrying values of type $\texttt{class}(\tau)$, and let b be a channel carrying values of type τ, so that $\&\,b$ may be passed as a message along channel a. A process that wishes to accept a channel reference on a and then accept on that channel has the form

$$\{x \leftarrow \texttt{sync}(?\,a)\ ;\ y \leftarrow \texttt{sync}(??\,x)\ ;\ \ldots\}.$$

The event $?\,a$ specifies a selective receipt on channel a. Once the value x has been accepted, the event $??\,x$ specifies a selective receipt on the channel referenced by x. So, if $\&\,b$ 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.

Selective communication may be seen as a simple form of pattern matching in which patterns are restricted to $a \cdot x$, where a is a channel carrying values of some type τ, and x is a variable of type τ. The idea is that selective communication filters for messages that match a pattern of this form, and proceeds by returning the associated value x. From this point of view it is natural to generalize selective communication to allow arbitrary patterns of type $\texttt{clsfd}$. Because different patterns may bind different variables, it is then natural to associate a separate continuation with each pattern, as in Chapter 13. Basic events are of

the form $p \Rightarrow m$, where p is a pattern of type $\texttt{clsfd}$, $x_1, \ldots, x_k$ are its variables, and m is a command involving these variables. Compound events are compositions of such rules, written $r_1 \mid \ldots \mid r_n$, quotiented by structural congruence to ensure that the order of rules is insignificant.

The statics of pattern-driven events may be readily derived from the statics of pattern matching given in Chapter 13. The dynamics is defined by the following rule defining the action engendered by an event:

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \texttt{clsfd} \quad \theta \Vdash p \lhd e}{p \Rightarrow m \mid rs \overset{e?}{\underset{\Sigma}{\Longrightarrow}} \hat{\theta}(m)}. \tag{42.14}$$

This rule states that we may choose any accept action by a value matching the pattern p, continuing with the corresponding instance of the continuation of the rule.

42.4 Free Assignables as Processes

Scope–free assignables are definable in $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \|\}$ by associating with each assignable a server process that sets and gets the contents of the assignable. With each assignable a of type ρ, 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 ρ. 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 ρ, and b is a channel of type ρ. This message requests that the contents of a be set to e and that the new contents be transmitted on channel b.

In other words, a is a channel of type τ_{srvr} given by

$$[\texttt{get} \hookrightarrow \rho\ \texttt{class}, \texttt{set} \hookrightarrow \rho \times \rho\ \texttt{class}].$$

The server selectively accepts on channel a, then dispatches on the class of the message to satisfy the request.

The server associated with the assignable a of type ρ maintains the contents of a using recursion. When called with the current contents of the assignable, the server selectively accepts on channel a, dispatching on the associated request and calling itself recursively with the (updated, if necessary) contents:

$$\lambda\ (u : \tau_{srvr}\ \texttt{class})\ \texttt{fix}\, srvr : \rho \rightarrow \texttt{void}\ \texttt{cmd}\ \texttt{is}\ \lambda\ (x : \rho)\ \texttt{cmd}\, \{y \leftarrow \texttt{sync}(??\, u)\ ;\ e_{(42.16)}\}. \tag{42.15}$$

The server is a procedure that takes an argument of type ρ, the current contents of the assignable, and yields a command that never terminates, because it restarts the server loop after each request. The server selectively accepts a message on channel a and

dispatches on it as follows:

$$\texttt{case}\ y\ \{\texttt{get}\cdot z \Rightarrow e_{(42.17)} \mid \texttt{set}\cdot\langle x', z\rangle \Rightarrow e_{(42.18)}\}. \tag{42.16}$$

A request to get the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\texttt{mk}(z; x))\ ;\ \texttt{do}\ \mathit{srvr}(x)\}. \tag{42.17}$$

A request to set the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\texttt{mk}(z; x'))\ ;\ \texttt{do}\ \mathit{srvr}(x')\}. \tag{42.18}$$

The type $\tau\ \texttt{ref}$ is defined to be $\tau\ \texttt{class}$, the type of channels (classes) carrying a value of type τ. A new free assignable is created by the command $\texttt{ref}\ e_0$, which is defined to be

$$\{x \leftarrow \texttt{newch}\ ;\ \_ \leftarrow \texttt{spawn}(e_{(42.15)}(x)(e_0))\ ;\ \texttt{ret}\ x\}. \tag{42.19}$$

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.

The commands $*\,e_0$ and $e_0 := e_1$ send a message to the server to get and set the contents of an assignable. The code for $*\,e_0$ is as follows:

$$\{x \leftarrow \texttt{newch}\ ;\ \_ \leftarrow \texttt{emit}(\texttt{mk}(e_0; \texttt{get}\cdot x))\ ;\ \texttt{sync}(??\,(x))\}. \tag{42.20}$$

A channel is allocated for the return value, the server is contacted with a `get` message specifying this channel, and the result of receiving on this channel is returned. Similarly, the code for $e_0 := e_1$ is as follows:

$$\{x \leftarrow \texttt{newch}\ ;\ \_ \leftarrow \texttt{emit}(\texttt{mk}(e_0; \texttt{set}\cdot\langle e_1, x\rangle))\ ;\ \texttt{sync}(??\,(x))\}. \tag{42.21}$$

42.5 Notes

Concurrent Algol is a synthesis of process calculus and Modernized Algol, and may be seen as an "Algol-like" formulation of Concurrent ML (Reppy, 1999) in which interaction is confined to the command modality. The design is influenced by Parallel Algol (Brookes, 2002). The reduction of channels to dynamic classification appears to be new. Most work on concurrent interaction seems to take the notion of communication channel as a central concept (but see Gelernter, 1985 for an alternative viewpoint, albeit in a unityped setting).

43 Distributed Algol

A *distributed* computation is one that takes place at many different *sites*, each of which controls some *resources* located at that site. For example, the sites might be nodes on a network, and a resource might be a device or sensor located at that site or a database controlled by that site. Only programs that execute at a particular site may access the resources situated at that site. Consequently, command execution always takes place at a particular site, called the *locus of execution*. Access to resources at a remote site from a local site is achieved by moving the locus of execution to the remote site, running code to access the local resource, and returning a value to the local site.

In this chapter we consider the language $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$, an extension of Concurrent Algol with a *spatial* type system that mediates access to located resources on a network. The type safety theorem ensures that all accesses to a resource controlled by a site are through a program executing at that site, even though references to local resources may be freely passed around to other sites on the network. The key idea is that channels and events are *located* at a particular site and that synchronization on an event may occur only at the site appropriate to that event. Issues of concurrency, which are to do with nondeterministic composition, are thereby cleanly separated from those of distribution, which are to do with the locality of resources on a network.

The concept of location in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ is sufficiently abstract that it admits another useful interpretation that can be useful in computer security settings. The "location" of a computation may also be thought of as the *principal* on whose behalf the computation is executing. From this point of view, a local resource is one that is accessible to a particular principal, and a mobile computation is one that may be executed by any principal. Movement from one location to another may then be interpreted as executing a piece of code on behalf of another principal, returning its result to the principal that initiated the transfer.

43.1 Statics

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ is inspired by the *possible worlds* interpretation of modal logic. Under that interpretation the truth of a proposition is relative to a *world*, which determines the state of affairs described by that proposition. A proposition may be true in one world and false in another. For example, we may use possible worlds to model counterfactual reasoning, in which we postulate that certain facts that happen to be true in

this, the *actual*, world, might be otherwise in some other, *possible*, world. For instance, in the actual world you, the reader, are reading this book, but in a possible world you may never have taken up the study of programming languages at all. Of course, not everything is possible: There is no possible world in which $2+2$ is other than 4, for example. Moreover, once a commitment has been made to one counterfactual, others are ruled out. We say that one world is *accessible* from another when the first is a sensible counterfactual relative to the first. So, for example, we may consider that relative to a possible world in which one persion is the king, there is no further possible world in which someone else is also the king (there being only one sovereign).

In $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ we interpret possible worlds as sites on a network, with accessibility between worlds expressing network connectivity. We postulate that every site is connected to itself (reflexivity); that if one site is reachable from another, then the second is also reachable from the first (symmetry); and that if a site is reachable from a reachable site, then this site is itself reachable from the first (transitivity). From the point of view of modal logics, the type system of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ is derived from the logic **S5**, for which accessibility is an equivalence relation.

The syntax of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ is a modification of that of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \|\}$. The following grammar summarizes the key changes:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Typ	τ	::=	$\texttt{cmd}[w](\tau)$	τ $\texttt{cmd}[w]$	commands
			$\texttt{chan}[w](\tau)$	τ $\texttt{chan}[w]$	channels
			$\texttt{event}[w](\tau)$	τ $\texttt{event}[w]$	events
Cmd	m	::=	$\texttt{at}[w](m)$	$\texttt{at}\ w\ \{m\}$	change site

The command, channel, and event types are indexed by the site w to which they pertain. There is a new form of command, $\texttt{at}[w](m)$, that changes the locus of execution from one site to another.

A signature Σ in $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ consists of a finite set of declarations of the form $a \sim \rho \ @ \ w$, where ρ is a type and w is a site. Such a declaration specifies that a is a channel carrying a payload of type ρ located at the site w. We may think of a signature Σ as a family of signatures Σ_w, one for each world w, containing the decalarations of the channels located on that world. This partitioning corresponds to the idea that channels are *located* resources in that they are uniquely associated with a site. They may be handled passively at other sites, but their only active role is at the site at which they are declared.

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ is given by the following two judgment forms:

$\Gamma \vdash_\Sigma e : \tau$	expression typing
$\Gamma \vdash_\Sigma m \sim \tau \ @ \ w$	command typing

The expression typing judgment is independent of the site. This corresponds to the idea that the values of a type have a site-independent meaning: The number 3 is the number 3, regardless of where it is used. However, commands can be executed at only a particular site, because they depend on the state located at that site.

A representative selection of the rules defining the statics of $\mathcal{L}\{\mathtt{nat}\ \mathtt{cmd} \rightharpoonup \| \ @\}$ is as follows:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau \ @ \ w}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}[w](\tau)} \tag{43.1a}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \rho @ w} \mathtt{ch}[a] : \mathtt{chan}[w](\rho)} \tag{43.1b}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{never}[\tau] : \mathtt{event}[w](\tau)} \tag{43.1c}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \rho @ w} \mathtt{rcv}[a] : \mathtt{event}[w](\rho)} \tag{43.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{chan}[w](\tau)}{\Gamma \vdash_\Sigma \mathtt{rcvref}(e) : \mathtt{event}[w](\tau)} \tag{43.1e}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{event}[w](\tau) \quad \Gamma \vdash_\Sigma e_2 : \mathtt{event}[w](\tau)}{\Gamma \vdash_\Sigma \mathtt{or}(e_1; e_2) : \mathtt{event}[w](\tau)} \tag{43.1f}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{event}[w](\tau)}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \sim \tau \ @ \ w} \tag{43.1g}$$

$$\frac{\Gamma \vdash_\Sigma m' \sim \tau' \ @ \ w'}{\Gamma \vdash_\Sigma \mathtt{at}[w'](m') \sim \tau' \ @ \ w} \,. \tag{43.1h}$$

Rule (43.1a) states that the type of an encapsulated command records the site at which the command is to be executed. Rules (43.1d) and (43.1e) specify that the type of a (static or dynamic) receive event records the site at which the channel resides. Rules (43.1c) and (43.1f) state that a choice can only be made between events at the same site; there are no cross-site choices. Rule (43.1g) states that the `sync` command returns a value of the same type as that of the event and may be executed only at the site to which the given event pertains. Finally, Rule (43.1h) states that to execute a command at a site w' requires that the command pertain to that site. The returned value is then passed to the original site.

43.2 Dynamics

The dynamics is given by a labeled transition judgment between processes, much as in Chapter 42. The principal difference is that the atomic process consisting of a single command has the form $\mathtt{proc}[w](m)$, which specifies the site w at which the command m is to be executed. The dynamics of processes remains much as in Chapter 42, except for the following rules governing the atomic process:

$$\frac{m \xrightarrow[\Sigma, w]{\alpha} \nu \ \Sigma' \{ m' \| p \}}{\mathtt{proc}[w](m) \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu \ \Sigma' \{ \mathtt{proc}[w](m') \| p \}} \tag{43.2a}$$

$$\frac{}{\mathtt{proc}[w](\mathtt{ret}(\langle\rangle)) \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} \mathtt{stop}} \,. \tag{43.2b}$$

The command execution judgment

$$m \underset{\Sigma\,,\,w}{\overset{\alpha}{\Longrightarrow}} \nu\ \Sigma'\,\{\,m' \parallel p\,\}$$

states that the command m when executed at site w may undertake the action α and in the process create new channels Σ' and a new process p. (The result of the transition is not a process expression, but rather should be construed as a structure having three parts, the newly allocated channels, the newly created processes, and a new command; any part is omitted when it is trivial.) This may be understood as a family of judgments indexed by signatures Σ and sites w. At each site there is an associated labeled transition system defining concurrent interaction of processes *at that site*.

The command execution judgment is defined by the following rules:

$$\frac{}{\texttt{spawn(cmd(}m\texttt{))} \underset{\Sigma\,,\,w}{\overset{\varepsilon}{\Longrightarrow}} \texttt{ret(}\langle\rangle\texttt{)} \parallel \texttt{proc[}w\texttt{](}m\texttt{)}} \tag{43.3a}$$

$$\frac{}{\texttt{newch[}\tau\texttt{]} \underset{\Sigma\,,\,w}{\overset{\varepsilon}{\Longrightarrow}} \nu\ a{\sim}\tau\texttt{@}w\texttt{.ret (\&}a\texttt{)}} \tag{43.3b}$$

$$\frac{m \underset{\Sigma\,,\,w'}{\overset{\alpha}{\Longrightarrow}} \nu\ \Sigma'\,\{\,m' \parallel p'\,\}}{\texttt{at[}w'\texttt{](}m\texttt{)} \underset{\Sigma\,,\,w}{\overset{\alpha}{\Longrightarrow}} \nu\ \Sigma'\,\{\,\texttt{at[}w'\texttt{](}m'\texttt{)} \parallel p'\,\}} \tag{43.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{at[}w'\texttt{](ret(}e\texttt{))} \underset{\Sigma\,,\,w}{\overset{\varepsilon}{\Longrightarrow}} \texttt{ret(}e\texttt{)}} \tag{43.3d}$$

$$\frac{e \underset{\Sigma}{\overset{\alpha}{\Rightarrow}} m}{\texttt{sync(}e\texttt{)} \underset{\Sigma\,,\,w}{\overset{\alpha}{\Longrightarrow}} m}\,. \tag{43.3e}$$

Rule (43.3a) states that new processes created at a site remain at that site—the new process executes the given command at the current site. Rules (43.3c) and (43.3d) state that the command $\texttt{at[}w'\texttt{](}m\texttt{)}$ is executed at site w by executing m at site w' and by returning the result to the site w. Rule (43.3e) states that an action may be undertaken at site w if the given event engenders that action. Notice that no cross-site synchronization is possible. Movement between sites is handled separately from synchronization among the processes at a site.

43.3 Safety

The safety theorem for $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \parallel @\}$ ensures that synchronization on a channel may occur only at the site on which the channel resides, even though channel references may be propagated from one site to another during a computation. By the time the reference is resolved and synchronization is attempted, the computation will, as a consequence of typing, be located at the appropriate site.

The key to the safety proof is the definition of a well-formed process. The judgment $\vdash_\Sigma p$ proc states that the process p is well-formed. Most important, the following rule governs the formation of atomic processes:

$$\frac{\vdash_\Sigma m \sim \texttt{unit} \mathbin{@} w}{\vdash_\Sigma \texttt{proc}[w](m)\ \texttt{proc}}. \tag{43.4}$$

That is, an atomic process is well-formed if and only if the command it is executing is well-formed at the site at which the process is located.

The proof of preservation relies on a lemma stating the typing properties of the execution judgment.

Lemma 43.1 (Execution)**.** *Suppose that* $m \xrightarrow[\Sigma,w]{\alpha} \nu\,\Sigma'\{m' \parallel p\}$. *If* $\vdash_\Sigma m \sim \tau \mathbin{@} w$, *then* $\vdash_\Sigma \alpha$ *action and* $\vdash_\Sigma \nu\,\Sigma\{\texttt{proc}[w](m') \parallel p\}$ *proc.*

Proof By a straightforward induction on Rules (43.3). □

Theorem 43.2 (Preservation)**.** *If* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ *and* $\vdash_\Sigma p$ *proc, then* $\vdash_\Sigma p'$ *proc.*

Proof By induction on Rules (43.1), appealing to Lemma 43.1 for atomic processes. □

The progress theorem states that the only impediment to execution of a well-typed program is the possiblity of synchronizing on an event that will never arise.

Theorem 43.3 (Progress)**.** *If* $\vdash_\Sigma p$ *proc, then either* $p \equiv \mathbf{1}$ *or there exists* α *and* p' *such that* $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$.

43.4 Situated Types

The foregoing formulation of $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \parallel @\}$ relies on indexing command, channel, and event types by the site to which they pertain so that values of these types may be passed around at will without fear of misinterpretation. The price to pay, however, is that the command, channel, and event types are indexed by the site to which they pertain, leading to repetition and redundancy. One way to mitigate this cost is to separate out the *skeleton* ϕ of a type from its *specialization* to a particular site w, which is written as $\phi\langle w\rangle$.

We now reformulate the statics of $\mathcal{L}\{\texttt{nat}\ \texttt{cmd} \rightharpoonup \parallel @\}$ using judgments of the form $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$ and $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, where Φ consists of hypotheses of the form $x_i : \phi_i \mathbin{@} w_i$. The type of an expression or command is factored into two parts, the skeleton ϕ and a site w at which to specialize it. The meaning of the factored judgments is captured by the following conditions:

1. If $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.
2. If $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w\rangle \mathbin{@} w$.

If Φ is a context of the form $x_1 : \phi_1 \ @ \ w_1, \ldots, x_n : \phi_n \ @ \ w_n$, then $\widehat{\Phi}$ is the context $x_1 : \phi_1\langle w_1\rangle, \ldots, x_n : \phi_n\langle w_n\rangle$.

The syntax of skeletons is similar to that for types, with the addition of a means of specializing a skeleton to a particular site:

Sort			**Abstract Form**	**Concrete Form**	**Description**
Fam	ϕ	$::=$	`nat`	`nat`	numbers
			`arr(`ϕ_1`;`ϕ_2`)`	$\phi_1 \to \phi_2$	functions
			`cmd(`ϕ`)`	ϕ `cmd`	computations
			`chan(`ϕ`)`	ϕ `chan`	channels
			`event(`ϕ`)`	ϕ `event`	events
			`at[`w`](`ϕ`)`	ϕ `at` w	situated

The situated type ϕ `at` w fixes the interpretation of ϕ at the site w.

The instantiation of a family ϕ at a site w is written as $\phi\langle w\rangle$ and is inductively defined by the following rules:

$$\frac{}{\texttt{nat}\langle w\rangle = \texttt{nat}} \tag{43.5a}$$

$$\frac{\phi_1\langle w\rangle = \tau_1 \quad \phi_2\langle w\rangle = \tau_2}{(\phi_1 \to \phi_2)\langle w\rangle = \tau_1 \to \tau_2} \tag{43.5b}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\ \texttt{cmd}\langle w\rangle = \tau\ \texttt{cmd[}w\texttt{]}} \tag{43.5c}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\ \texttt{chan}\langle w\rangle = \tau\ \texttt{chan[}w\texttt{]}} \tag{43.5d}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\ \texttt{event}\langle w\rangle = \tau\ \texttt{event[}w\texttt{]}} \tag{43.5e}$$

$$\frac{\phi\langle w'\rangle = \tau'}{(\phi\ \texttt{at}\ w')\langle w\rangle = \tau'}\,. \tag{43.5f}$$

Crucially, Rule (43.5f) states that the situated family ϕ `at` w' is to be interpreted at w by the interpretation of ϕ at w'. Otherwise instantiation serves merely to decorate the constituent command, channel, and event skeletons with the site at which they are being interpreted.

Any type τ of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ may be embedded as a constant family, `exactly(`τ`)`, such that `exactly(`τ`)`$\langle w\rangle = \tau$ for any site w. The constant family is inductively defined by the following rules:

$$\frac{}{\texttt{exactly(nat)} = \texttt{nat}} \tag{43.6a}$$

$$\frac{\texttt{exactly(}\tau_1\texttt{)} = \phi_1 \quad \texttt{exactly(}\tau_2\texttt{)} = \phi_2}{\texttt{exactly(}\tau_1 \to \tau_2\texttt{)} = \phi_1 \to \phi_2} \tag{43.6b}$$

$$\frac{\texttt{exactly(}\tau\texttt{)} = \phi}{\texttt{exactly(}\tau\ \texttt{cmd[}w\texttt{])} = \phi\ \texttt{cmd at}\ w} \tag{43.6c}$$

$$\frac{\texttt{exactly}(\tau) = \phi}{\texttt{exactly}(\tau\ \texttt{chan}[w]) = \phi\ \texttt{chan at}\ w} \tag{43.6d}$$

$$\frac{\texttt{exactly}(\tau) = \phi}{\texttt{exactly}(\tau\ \texttt{event}[w]) = \phi\ \texttt{event at}\ w}. \tag{43.6e}$$

It is easy to check that $\texttt{exactly}(\tau)$ is a constant family:

Lemma 43.4. *For any site w, $\texttt{exactly}(\tau)\langle w\rangle = \tau$.*

The statics of $\mathcal{L}\{\texttt{nat cmd} \rightharpoonup \| \ @\}$ may be given in factored form, as is illustrated by the following selection of typing rules:

$$\frac{\Phi \vdash_\Sigma e : \phi\ @\ w}{\Phi \vdash_\Sigma \texttt{ret}\, e \sim \phi\ @\ w} \tag{43.7a}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi_1\ @\ w \quad \Phi, x : \phi_1\ @\ w \vdash_\Sigma m_2 \sim \phi_2\ @\ w}{\Phi \vdash_\Sigma \texttt{bnd}\, x \leftarrow e_1\, ; m_2 \sim \phi_2\ @\ w} \tag{43.7b}$$

$$\frac{\Phi \vdash_\Sigma m \sim \phi\ @\ w}{\Phi \vdash_\Sigma \texttt{cmd}\, m : \phi\ \texttt{cmd}\ @\ w} \tag{43.7c}$$

$$\frac{\texttt{exactly}(\rho) = \phi}{\Phi \vdash_{\Sigma, a \sim \rho @ w} \texttt{\&}\, a : \phi\ \texttt{chan}\ @\ w} \tag{43.7d}$$

$$\frac{}{\Phi \vdash_\Sigma \texttt{never} : \phi\ \texttt{event}\ @\ w} \tag{43.7e}$$

$$\frac{\texttt{exactly}(\rho) = \phi}{\Phi \vdash_{\Sigma, a \sim \rho @ w} \texttt{?}\, a : \phi\ \texttt{event}\ @\ w} \tag{43.7f}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \texttt{chan}\ @\ w}{\Phi \vdash_\Sigma \texttt{??}\, e : \phi\ \texttt{event}\ @\ w} \tag{43.7g}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi\ \texttt{event}\ @\ w \quad \Phi \vdash_\Sigma e_2 : \phi\ \texttt{event}\ @\ w}{\Phi \vdash_\Sigma e_1\ \texttt{or}\ e_2 : \phi\ \texttt{event}\ @\ w} \tag{43.7h}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \texttt{event}\ @\ w}{\Phi \vdash_\Sigma \texttt{sync}(e) \sim \phi\ @\ w} \tag{43.7i}$$

$$\frac{\Phi \vdash_\Sigma m' \sim \phi'\ @\ w' \quad \phi'\ \mathsf{mobile}}{\Phi \vdash_\Sigma \texttt{at}\ w'\ \{m'\} \sim \phi'\ @\ w}. \tag{43.7j}$$

Rule (43.7d) specifies that a reference to a channel carrying a value of type ρ is classified by the *constant* family yielding the type ρ at each site. Rule (43.7j) is the most interesting rule, because it includes a restriction on the family ϕ'. To see how this arises, inductively we have that $\widehat{\Phi} \vdash_\Sigma m' \sim \phi'\langle w'\rangle\ @\ w'$, which is enough to ensure that $\widehat{\Phi} \vdash_\Sigma \texttt{at}\ w'\ \{m'\} \sim \phi'\langle w'\rangle\ @\ w$. But we are required to show that $\widehat{\Phi} \vdash_\Sigma \texttt{at}\ w'\ \{m'\} \sim \phi'\langle w\rangle\ @\ w$! This

will be the case only if $\phi'\langle w\rangle = \phi'\langle w'\rangle$, which is to say that ϕ' is a constant family whose meaning does not depend on the site at which it is instantiated.

The judgment ϕ mobile states that ϕ is a mobile family. It is inductively defined by the following rules:

$$\frac{}{\texttt{nat mobile}} \tag{43.8a}$$

$$\frac{\phi_1 \text{ mobile} \quad \phi_2 \text{ mobile}}{\phi_1 \to \phi_2 \text{ mobile}} \tag{43.8b}$$

$$\frac{}{\phi\ \texttt{at}\ w \text{ mobile}}\,. \tag{43.8c}$$

The remaining families are not mobile, precisely because their instantiation specifies the site of their instances; these do not determine constant families.

Lemma 43.5.

1. *If ϕ mobile, then for every w and w', $\phi\langle w\rangle = \phi\langle w'\rangle$.*
2. *For any type τ, `exactly(`τ`)` mobile.*

We may then verify that the intended interpretation is valid:

Theorem 43.6.

1. *If $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.*
2. *If $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w\rangle \mathbin{@} w$.*

Proof By induction on Rules (43.7). □

43.5 Notes

The use of a spatial modality to express locality and mobility constraints in a distributed program was inspired by ML5 (Murphy et al., 2004). The separation of locality concerns from concurrency concerns is expressed here by supporting communication and synchronization within a site and treating movement between sites separately. The formulation of situated types is based on Licata and Harper (2010).

PART XVII

Modularity

44 Components and Linking

Modularity is the most important technique for controlling the complexity of programs. Programs are decomposed into separate *components* with precisely specified, and tightly controlled, interactions. The pathways for interaction among components determine dependencies that constrain the process by which the components are integrated, or *linked*, to form a complete system. Different systems may use the same components, and 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 broken down into a collection of definitions and lemmas. Cross references among lemmas determine a dependency structure that constrains their integration to form a complete proof of the main theorem. Of course, one 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, as, for example, a Lie group is a group structure on a manifold.

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 consists of substitution and discharge of the hypothesis.

44.1 Simple Units and Linking

Breaking down a program into units amounts to exploiting the transitivity of the hypothetical judgment (see Chapter 3). The decomposition may be expressed as an interaction between two parties, the *client* and the *implementor*, that is mediated by an agreed-on contract, called an *interface*. The client *assumes* that the implementor upholds the contract, and the implementor *guarantees* that the contract will be upheld. The assumption made by the client amounts to a declaration of its dependence on the implementor that is discharged by *linking* the two parties in accordance with their agreed-on contract.

The interface that mediates the interaction between a client and an implementor is a *type*. Linking is nothing other than the implementation of the composite structural

rules of substitution and transitivity:

$$\frac{\Gamma \vdash e_{impl} : \tau_{intf} \quad \Gamma, x : \tau_{intf} \vdash e_{client} : \tau_{client}}{\Gamma \vdash [e_{impl}/x]e_{client} : \tau_{client}}. \tag{44.1}$$

The type τ_{intf} is the interface type. It defines the capabilities to be provided by the implementor e_{impl} that are relied upon by the client e_{client}. The free variable x expresses the dependency of e_{client} on e_{impl}. 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 commonly called an *application program interface*, or *API*, because it determines the operations that the client (application) may expect from the implementor. A more sophisticated form of interface is one that defines an abstract type 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 a variety of ways. One method is called *separate compilation*. The expressions e_{client} and e_{impl}, called in this context *source modules*, are translated (compiled) into another, lower-level, language, resulting in *object modules*. Linking consists of performing the required substitution at the level of the object language in such a way that the result corresponds to the translation of $[e_{impl}/x]e_{client}$.[2] Another method, called *separate checking*, shifts the requirement for translation to the linker. The client and implementor units are ensured to be type correct with respect to the interface requirements, but are not translated into lower-level form. Linking then consists of translating the composite program as a whole, often resulting in a more efficient outcome than would be possible when compiling separately.

A more sophisticated, and widely used, implementation of substitution is called *dynamic linking*. Informally, this means that execution of the client commences before the implementation of the components on which it depends are provided. Rather than link prior to execution, we instead execute and link "on the fly." At first blush this might seem to be a radical departure from the methodology developed in this book, because we have consistently required that execution be defined only on expressions with no free variables. But looks can be deceiving. What is really going on with dynamic linking is that the client is implemented by a *stub* that forwards accesses to a stored implementation (typically, in a "file system" or similar data structure). The actual implementation code is not accessed until the client requests it, which may not happen at all. This tends to reduce latency and makes it possible to replace the implementation without recompiling the client.

[1] See Chapters 21 and 49 for a discussion of type abstraction.

[2] The correspondence need not be exact, but must be equivalent for all practical purposes, in the sense discussed in Chapter 48.

What is important is not how linking is implemented, but rather that the linking principle enables *separate development*. Once the common interface has been agreed on, the client and implementor are free to proceed with their work independently of one another. All that is required is that both parties complete their work before the system as a whole can be built.

44.2 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 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 managed by paying attention to the modal distinction between expressions and commands introduced in Chapter 35. Specifically, a component that may have an effect when executed does not have type τ_{intf} of implementations of the interface type, but rather the type $\tau_{intf}\,\texttt{cmd}$ of encapsulated commands that, when executed, have effects and yield such an implementation. Being encapsulated, a value of this type is itself free of effects, but it may have effects when evaluated.

The distinction between the types τ_{intf} and $\tau_{intf}\,\texttt{cmd}$ is mediated by the sequentialization command introduced in Chapter 35. For the sake of generality, let us assume that the client is itself an encapsulated command of type $\tau_{client}\,\texttt{cmd}$, so that it may itself have effects when executed, and may serve as a component of a yet larger system. Assuming that the client refers to the encapsulated implementation by the variable x, the command

$$\texttt{bnd}\,x \leftarrow x\,;\,\texttt{do}\,e_{client}$$

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 sequentialization command, which ensures that the implementor's effects occur prior to those of the client, precisely because the client depends on the implementor for its execution.

More generally, to manage such interactions in a large program it is common to isolate an *initialization procedure* whose role is to stage the effects engendered by the various components according to some policy or convention. Rather than attempt to survey all possible policies, which are numerous and complex, let us simply observe that the upshot

of such conventions is that the initialization procedure is a command of the form

$$\{x_1 \leftarrow x_1 \,;\, \ldots x_n \leftarrow x_n \,;\, m_{main}\},$$

where $x_1, \ldots, x_n$ represent the components of the system and m_{main} is the main (startup) routine. After linking the initialization procedure has the form

$$\{x_1 \leftarrow e_1 \,;\, \ldots x_n \leftarrow e_n \,;\, m_{main}\},$$

where $e_1, \ldots, e_n$ are the encapsulated implementations of the linked components. When the initialization procedure is executed, it 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.

44.3 Notes

The relationship between the structural properties of entailment and the practical problem of separate development was implicit in much early work on programming languages, but became explicit once the correspondence between propositions and types was developed. There are many indications of this correspondence, for example in *Proofs and Types* (Girard, 1989) and *Intuitionistic Type Theory* (Martin-Löf, 1984), but it was first made explicit by Cardelli (1997).

45 Type Abstractions and Type Classes

An interface is a contract that specifies the rights of a client and the responsibilities of an implementor. Being a specification of behavior, an interface is a type. In principle any type may serve as an interface, but in practice it is usual to structure code into *modules* consisting of separable and reusable components. An interface specifies the behavior of a module expected by a client and imposed on the implementor. It is the fulcrum on which is balanced the tension between separability and integration. As a rule, a module should have a well-defined behavior that can be understood separately, but it is equally important that it be easy to combine modules 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 for an interface to be a labeled tuple of functions and procedures with specified types. The types of the fields of the tuple are traditionally 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 an abstract machine. The client makes use of these instructions in its code, and the implementor agrees to provide their implementations.

The problem with procedural abstraction is that it does not provide as much insulation as we might like. For example, a module that implements a dictionary must expose in the types of its operations the exact representation of the tree as, say, a recursive type (or, in more rudimentary languages, a pointer to a structure that itself may contain such pointers). Yet the client really should not depend on this representation: The whole point of abstraction is to eliminate such dependencies. The solution, as discussed in Chapter 21, is to extend the abstract machine metaphor to allow the internal state of the machine to be hidden from the client. In the case of a dictionary the representation of the dictionary as a binary search tree is hidden by existential quantification. This is called *type abstraction*, because the type of the underlying data (state of the abstract machine) is hidden.

Type abstraction is a powerful method for limiting the dependencies among the modules that constitute a program. It is very useful in many circumstances, but is not universally applicable. It is not always appropriate to use abstract types; often it is useful to expose, rather than obscure, type information across a module boundary. A typical example is the implementation of a dictionary, which is a mapping from keys to values. To use, say, a binary search tree to implement a dictionary, we require that the key type admit a total ordering with which keys can be compared. The dictionary abstraction does not depend on

the exact type of the keys, but only requires that the key type be constrained to provide a comparison operation. A *type class* is a specification of such a requirement. The class of comparable types, for example, specifies a type t together with an operation `leq` of type $(t \times t) \to \texttt{bool}$ 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 rather as a constraint on its behavior expressed by demanding that certain operations, such as comparison, be available, without limiting the other operations that might be defined on it.

Type abstractions and type classes are extremal cases of a general concept of module type that we discuss in detail in this chapter. The crucial idea is the *controlled revelation* of type information across module boundaries. Type abstractions are opaque; type classes are transparent. These are both instances of *translucency*, which arises from the combination of existential types (Chapter 21), subtyping (Chapter 23), and singleton kinds and subkinding (Chapter 24). Unlike in Chapter 21, however, we distinguish the types of modules, which we called *signatures*, from the types of ordinary values. The distinction is not essential, but it is helpful to keep the two concepts separate at the outset, deferring discussion of how to relax the segregation once the basic concepts are in place.

45.1 Type Abstraction

Type abstraction is captured by a form of existential type quantification similar to that described in Chapter 21. For example, a dictionary with keys of type τ_{key} and values of type τ_{val} implements the signature σ_{dict}, defined as follows:

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \tau_{\mathsf{key}} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow \tau_{\mathsf{key}} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt} \rangle].$$

The type variable t of kind `T` is the abstract type of dictionaries on which are defined three operations, `emp`, `ins`, and `fnd`, with the specified types. It is not essential to fix the type τ_{val}, because the dictionary operations impose no restrictions on it; here it is done only for the sake of simplicity. However, it is essential, at this stage, that the key type τ_{key} be fixed, for reasons that will become clearer as we proceed.

An implementation of the signature σ_{dict} is a *structure* M_{dict} of the form

$$[\tau_{\mathsf{dict}}; \langle \mathtt{emp} \hookrightarrow \ldots, \mathtt{ins} \hookrightarrow \ldots, \mathtt{fnd} \hookrightarrow \ldots \rangle],$$

where the elided parts implement the dictionary operations in terms of the chosen representation type τ_{dict}. For example, τ_{dict} might be a recursive type defining a balanced binary search tree, such as a red–black tree. The dictionary operations work on the underlying representation of the dictionary as such a tree, just as would a package of existential type discussed in Chapter 21.

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

$$M_{\mathsf{dict}} \mid \sigma_{\mathsf{dict}}.$$

The effect of sealing is to ensure that the *only* information about M_{dict} that is propagated to the client is given by σ_{dict}. In particular, because σ_{dict} specifies only that the type t have kind $\mathtt{T}$, no information about the choice of t as τ_{dict} in M_{dict} is made available to the client.

A module is a *two-phase* object consisting of a *static part* and a *dynamic part*. The static part is a constructor of a specified kind; the dynamic part is a value of a specified type. There are two elimination forms that extract the static and dynamic parts of a module. These are a form of constructor and a form of expression, respectively. 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 introductory 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 τ_{dict}.

But consider the static part of a sealed module, which has the form $(M_{\mathsf{dict}} \mid \sigma_{\mathsf{dict}}) \cdot \mathtt{s}$. Because sealing hides the representation of an abstract type, this constructor should not be equivalent to τ_{dict}. If M'_{dict} is another implementation of σ_{dict}, should $(M_{\mathsf{dict}} \mid \sigma_{\mathsf{dict}}) \cdot \mathtt{s}$ be equivalent to $(M'_{\mathsf{dict}} \mid \sigma_{\mathsf{dict}}) \cdot \mathtt{s}$? To ensure reflexivity of type equivalence, this equation should hold whenever 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 fundamental 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 $M \mid \sigma \cdot \mathtt{s}$ is deemed ill-formed. More generally, we disallow formation of $M \cdot \mathtt{s}$ 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 35 engenders the effects induced by an encapsulated command. A consequence of this is that two distinct bindings of the same sealed module result in two distinct 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 be prepared for them to be different).

45.2 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 mechanism is called a *type class*, or a *view*, in which case an implementation is called an *instance* of the type class or view. Because the signature of a type class serves only as a constraint specifying the minimum capabilities of an unknown module, some other means of working with values of that type must be available. The key to achieving this is to expose, rather than hide, the identity of the static part of a module. In this sense, type classes are the "opposite" of type abstractions, but we subsequently see 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, for example, the only requirement is that keys come equipped with a total ordering given by a comparison operation. This can be expressed by a signature σ_{ord} given by

$$[t :: \mathtt{T}; \langle \mathtt{leq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Because a given type may be ordered in many ways, it is essential that the ordering be packaged with the type to determine a type of keys.

The implementation of dictionaries as binary search trees takes the form

$$X : \sigma_{\mathsf{ord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}},$$

where σ^X_{dict} is the signature

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt} \rangle]$$

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. This implies that 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 on only these specified capabilities, and no others. When linking with a module defining X, the implementation need not be sealed with this signature, but must rather 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 amounts to a categorization of a preexisting type, 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. This may be achieved using singleton kinds (Chapter 24). 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

$$[t :: \mathtt{S(nat)};\mathtt{leq} \hookrightarrow (t \times t) \to \mathtt{bool}],$$

which, by the rules of equivalence (defined formally in Section 45.3), is equivalent to the signature

$$[\_ :: \mathtt{S(nat)};\mathtt{leq} \hookrightarrow (\mathtt{nat} \times \mathtt{nat}) \to \mathtt{bool}].$$

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 24, together with the evident 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 whenever a module of signature σ_{ord} is required. Therefore M_{natord} may be linked to X in M^X_{bstdict}.

The combination of subtyping and 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

$$[t :: \mathtt{S}(\tau^X_{\mathsf{bst}});\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\,\mathtt{opt}\rangle],$$

where τ^X_{bst} is the type of binary search trees with keys given by the module X of signature σ_{ord}. This 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}. This, too, is achieved by 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)`, and hence the following typing is also valid:

$$X : \sigma_{\mathsf{natord}} \vdash M^{X}_{\mathsf{bstdict}} : \sigma_{\mathsf{natdict}}.$$

Here $\sigma_{\mathsf{natdict}}$ is the closed signature

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow \mathtt{nat} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt} \rangle]$$

in which the representation of dictionaries is held abstract, but the representation of keys as natural numbers is publicized. The dependency on X has been eliminated by replacing all occurrences of $X \cdot \mathtt{s}$ within $\sigma^{X}_{\mathsf{dict}}$ by the type `nat`. Having derived this typing we may link X with M_{natord} as described in Chapter 44 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 23, 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 works only 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.

45.3 A Module Language

The language $\mathcal{L}\{\mathtt{mod}\}$ is a codification of the ideas 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 22

and 24, with singleton and dependent kinds. The syntax of $\mathcal{L}\{\mathtt{mod}\}$ is summarized by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Sig	σ	$::=$	$\mathtt{sig}[\kappa](t.\tau)$	$[t :: \kappa\,;\tau]$	signature
Mod	M	$::=$	X	X	variable
			$\mathtt{str}(c\,;e)$	$[c\,;e]$	structure
			$\mathtt{seal}[\sigma](M)$	$M \upharpoonright \sigma$	seal
			$\mathtt{let}[\sigma](M_1; X.M_2)$	$(\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2) : \sigma$	definition
Con	c	$::=$	$\mathtt{stat}(M)$	$M \cdot \mathtt{s}$	static part
Exp	e	$::=$	$\mathtt{dyn}(M)$	$M \cdot \mathtt{d}$	dynamic part

The statics of $\mathcal{L}\{\mathtt{mod}\}$ consists of the following forms of judgment, in addition to those governing the kind and type levels:

$\Gamma \vdash \sigma\ \mathsf{sig}$	well-formed signature
$\Gamma \vdash \sigma_1 \equiv \sigma_2$	equivalent signatures
$\Gamma \vdash \sigma_1 <: \sigma_2$	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 do not explicitly state the value hypotheses associated with module and expression variables, under the convention that all such variables implicitly come paired with such an assumption.

The formation, equivalence, and subsignature judgments are defined by the following rules:

$$\frac{\Gamma \vdash \kappa\ \mathsf{kind} \quad \Gamma, u :: \kappa \vdash \tau\ \mathsf{type}}{\Gamma \vdash [u :: \kappa\,;\tau]\ \mathsf{sig}} \tag{45.1a}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [u :: \kappa_1\,;\tau_1] \equiv [u :: \kappa_2\,;\tau_2]} \tag{45.1b}$$

$$\frac{\Gamma \vdash \kappa_1 :<: \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Gamma \vdash [u :: \kappa_1\,;\tau_1] <: [u :: \kappa_2\,;\tau_2]} \tag{45.1c}$$

Most important, signatures are covariant in both the kind and type positions: Subkinding and subtyping are preserved by the formation of a signature. It is a consequence of Rule (45.1b) that

$$[u :: \mathtt{S}(c)\,;\tau] \equiv [\_ :: \mathtt{S}(c)\,;[c/u]\tau]$$

and, further, it is a consequence of Rule (45.1c) that

$$[\_ :: \mathtt{S}(c)\,;[c/u]\tau] \mathrel{<:} [\_ :: \mathtt{T};[c/u]\tau]$$

and therefore

$$[u :: \mathtt{S}(c)\,;\tau] \mathrel{<:} [\_ :: \mathtt{T};[c/u]\tau].$$

It is also the case that

$$[u :: \mathtt{S}(c)\,;\tau] \mathrel{<:} [u :: \mathtt{T};\tau].$$

But the two supersignatures of $[u :: \mathtt{S}(c)\,;\tau]$ are *incomparable* with respect to the subsignature judgment. This fact is important in the statics of module definitions, as detailed shortly.

The statics of module expressions is given by the following rules:

$$\frac{}{\Gamma, X : \sigma \vdash X : \sigma} \tag{45.2a}$$

$$\frac{\Gamma \vdash c :: \kappa \quad \Gamma \vdash e :: [c/u]\tau}{\Gamma \vdash [c\,;e] : [u :: \kappa\,;\tau]} \tag{45.2b}$$

$$\frac{\Gamma \vdash \sigma \ \mathsf{sig} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash M \upharpoonright \sigma : \sigma} \tag{45.2c}$$

$$\frac{\Gamma \vdash \sigma \ \mathsf{sig} \quad \Gamma \vdash M_1 : \sigma_1 \quad \Gamma, X : \sigma_1 \vdash M_2 : \sigma}{\Gamma \vdash (\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2) : \sigma : \sigma} \tag{45.2d}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma \mathrel{<:} \sigma'}{\Gamma \vdash M : \sigma'}\,. \tag{45.2e}$$

In Rule (45.2b) 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 (45.2c) is to be used in conjunction with subsumption [Rule (45.2e)] 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 (45.2d) would be perfectly sensible were the signature σ omitted from the syntax of the definition. However, omitting this information greatly complicates type checking. If σ were omitted from the syntax of the definition, the type checker would be required to find a signature σ for the body of the definition that *avoids* the module variable X. Inductively, we may suppose that we have found a signature σ_1 for the module M_1 and a signature σ_2 for the module M_2, under the assumption that X has signature σ_1. To find a signature for an unadorned definition, we must find a supersignature σ of σ_2 that avoids X. To ensure that all possible choices of σ are accounted for, we seek to find the least (most precise) such signature with respect to the subsignature relation; this is called the *principal signature* of a module. The problem is that there may be no least supersignature of a given signature that avoids a specified variable. (Consider the preceding example of a

signature with two incomparable supersignatures. The example may 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.

In the presence of modules we have a new form of constructor expression, $M \cdot \mathsf{s}$, and a new form of value expression, $M \cdot \mathsf{d}$. These operations extract the static and dynamic parts of the module M, respectively. Their formation rules are as follows:

$$\frac{\Gamma \vdash M \ \mathsf{val} \quad \Gamma \vdash M : [u :: \kappa\,;\tau]}{\Gamma \vdash M \cdot \mathsf{s} :: \kappa} \tag{45.3a}$$

$$\frac{\Gamma \vdash M : [\_ :: \kappa\,;\tau]}{\Gamma \vdash M \cdot \mathsf{d} : \tau}\,. \tag{45.3b}$$

Rule (45.3a) requires that the module expression M be a value in accordance with the following rules:

$$\frac{}{\Gamma, X : \sigma, X \ \mathsf{val} \vdash X \ \mathsf{val}} \tag{45.4a}$$

$$\frac{\Gamma \vdash e \ \mathsf{val}}{\Gamma \vdash [c\,;e] \ \mathsf{val}}\,. \tag{45.4b}$$

(It is not strictly necessary to insist that the dynamic part of a structure be a value in order for the structure itself to be a value, but we impose this requirement to be consistent with the general policy to employ eager evaluation and to obtain laziness through types, as described in Chapter 37.)

Rule (45.3a) 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 ensures representation independence for abstract types, as discussed in Section 45.1. For if $M \cdot \mathsf{s}$ were admissible when M is a sealed module, it would be a type whose identity depends on the underlying implementation, in violation of the abstraction principle. Module variables are, however, values, so that if $X : [t :: \mathtt{T}\,;\tau]$ is a module variable, then $X \cdot \mathsf{s}$ is a well-formed type. What this means in practice is that sealed modules must be bound to variables before they can be used. It is for this reason that we include definitions among module expressions.

Rule (45.3b) requires that the signature of the module M be nondependent, so that the result type τ does not depend on the static part of the module. This may not always be the case. For example, if M is a sealed module, say $N \upharpoonright [t :: \mathtt{T}\,;t]$ for some module N, then projection $M \cdot \mathsf{d}$ is ill-formed. For if it were to be well-formed, its type would be $M \cdot \mathsf{s}$, which would violate representation independence for abstract types. But if M is a module value, then it is always possible to derive a nondependent signature for it, provided that we include the following rule of *self-recognition*:

$$\frac{\Gamma \vdash M : [u :: \kappa\,;\tau] \quad \Gamma \vdash M \ \mathsf{val}}{\Gamma \vdash M : [u :: \mathtt{S}(M \cdot \mathsf{s} :: \kappa)\,;\tau]}\,. \tag{45.5}$$

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:

$$\frac{\Gamma \vdash [c\,;e] : [t :: \kappa\,;\tau] \quad \Gamma \vdash [c\,;e]\ \mathsf{val}}{\Gamma \vdash [c\,;e] \cdot \mathsf{s} \equiv c :: \kappa}\,. \tag{45.6}$$

The requirement that the expression e be a value, which is implicit in the second premise of the rule, is not strictly necessary, but does no harm. A consequence of this rule is that apparent dependencies of closed constructors (or kinds) on modules may always be eliminated. In particular the identity of the constructor $[c\,;e] \cdot \mathsf{s}$ is independent of e, as would be expected if representation independence is to be ensured.

The dynamics of modules is entirely straightforward:

$$\frac{e \mapsto e'}{[c\,;e] \mapsto [c\,;e']} \tag{45.7a}$$

$$\frac{e\ \mathsf{val}}{[c\,;e] \cdot \mathsf{d} \mapsto e} \tag{45.7b}$$

There is no need to evaluate constructors at run time, because the dynamics of expressions does not depend on their types. It is straightforward to prove type safety for this dynamics relative to the foregoing statics.

45.4 First and Second Class

It is common to draw a distinction between *first-class* and *second-class* modules in programming languages. The purported distinction has little force, because it is not precisely defined, but the terminology is chosen to suggest that the former is somehow superior to the latter. As is often the case with such informal concepts, careful analysis reveals that the situation is exactly opposite to what is suggested. To make this precise, we must first give a definition of what is meant by the terms. Simply put, a module system is first class if signatures are forms of type, and is otherwise not. Here we are using "type" in a precise technical sense as a classifier of expressions, rather than in a loose sense as any form of classifier. If signatures are types in the narrow sense, then modules may be bound to (substituted for) variables and hence may be passed as arguments to functions and returned as results from them. Moreover, they may be stored in mutable cells, if there are such, and in general may be handled like any other value, precisely because they are classified by types. If, however, signatures are not types in the narrow sense, then there are limitations to how they may be used that would seem to limit their expressive power, rendering them less useful than they would otherwise be.

However, this superficial impression is misleading, for two related reasons. First, the apparent restriction to second-class modules allows for more precise distinctions to be

drawn than are possible in the purely first-class case. If a module is a value of a certain type, then a module expression may be an arbitrary computation that, in full generality, depends on run-time state as well as on the form of the expression itself. For example, we may form a module expression that conditionally branches on the phase of the moon at the time of evaluation, yielding modules with different static components in each case. Because of such a possibility, it is not sensible to track the identity of the static component of a module in the type system, because it quite literally does not have a single static component to track. Consequently, first-class module systems are *incompatible* with extensions that rely on tracking the identity of the static part of a module. (One example is the concept of an applicative functor discussed in Chapter 46.)

Moreover, a second-class module system is compatible with extensions that *permit* modules to be handled as first-class values, without *requiring* that all modules be first-class values. In this important sense it is the second-class modules that are the more expressive, because they allow consideration of first-class modules while retaining the advantages of second-class modules.[2] Specifically, we may account for first-class modules within the language considered in the preceding section by taking the following steps. First, we admit existential types, described in Chapter 21, as types. A "first-class module" is nothing other than a package of existential type, which may be handled just like any other value of any other type. Observe that a module value M of signature $[t :: \kappa\,;\tau]$ may be turned into a value of type $\exists\, t :: \kappa\,.\,\tau$ by simply forming the package $\mathtt{pack}\, M \cdot \mathtt{s}\,\mathtt{with}\, M \cdot \mathtt{d}\,\mathtt{as}\, \exists(t\,.\,\tau)$ consisting of the static and dynamic parts of M. Second, to allow packages to be treated as modules, we introduce the module $\mathtt{open}\, e$ that "opens" a package as a module according to the following rule:

$$\frac{\Gamma \vdash e : \exists\, t :: \kappa\,.\,\tau}{\Gamma \vdash \mathtt{open}\, e : [t :: \kappa\,;\tau]}\,. \tag{45.8}$$

Such a module cannot be considered to be a value, because e is an arbitrary computation, and hence must generally be bound to a module variable before it is used. This mimics exactly the elimination form for existential types, which similarly binds the components of a package to variables before they are used. In this manner we may support both first- and second-class modules in a single framework, without having to make a priori commitments to one or the other.

45.5 Notes

The use of dependent types to express modularity was first proposed by MacQueen (1986). Subsequent studies extended this proposal to model the *phase distinction* between compile time 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

[2] The situation is analogous to that between static and dynamic type systems discussed in Chapter 18. At first glance it sounds as though dynamic typing would be more expressive, but on careful analysis the situation is revealed to be the other way around.

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 that of Lillibridge (1997) and Dreyer (2005). The self-recognition rule was introduced by Harper and Lillibridge (1994) and by Leroy (1994). It was subsequently identified as a manifestation of higher-order singletons (Stone and Harper, 2006). A consolidation of these ideas was used as the foundation for a mechanization of the metatheory 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 is based on *elaboration*, a translation of modularity constructs into more primitive notions such as polymorphism and higher-order functions. *The Definition of Standard ML* (Milner et al., 1997) pioneered the elaboration approach. Building on the 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. It seems clear that some form of elaboration is required (to handle identifier scope resolution and type inference, for example), but it is as yet unclear where best to draw the line.

46 Hierarchy and Parameterization

To be adequately expressive it is essential that a module system support the construction of module *hierarchies*. Hierarchical structure arises naturally in programming, both as an organizational device for partitioning of a large program into manageable pieces and as a localization device that allows one type abstraction or type class to be layered on top of another. In such a scenario the lower layer plays an auxiliary role relative to the upper layer, and we may think of the upper layer as being parameterized by the lower in the sense that any implementation of the lower layer induces an instance of the upper layer corresponding to that instance. The pattern of dependency of one abstraction on another may be captured by an *abstraction*, or *parameterization*, mechanism that allows the implementation of one abstraction to be considered a function of the implementation of another. Hierarchies and parameterization work in tandem to provide an expressive language for organizing programs.

46.1 Hierarchy

It is common in modular programming to layer a type class or a type abstraction on top of a type class. For example, the class of *equality types*, which are those that admit a Boolean equivalence test, is described by the signature σ_{eq}, defined as follows:

$$[t :: \mathtt{T}; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Instances of this class consist of a type together with a binary equality operation defined on it. Such instances are modules with a subsignature of σ_{eq}, for example σ_{nateq} given by

$$[t :: \mathtt{S(nat)}; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

A module value of this signature has the form

$$[\mathtt{nat}; \langle \mathtt{eq} \hookrightarrow \ldots \rangle],$$

where the elided expression implements an equivalence relation on the natural numbers. All other instance values of the class σ_{eq} have a similar form, differing in the choice of type, the choice of comparison operation, or both.

The class of *ordered types* may be considered to be an extension of the class of equality types with an additional binary operation for the (strict) comparison of two elements of

that type. One way to formulate this is as the signature

$$[t :: \mathtt{T}; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool}, \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle],$$

which is a subsignature of σ_{eq} according to the rules of subtyping given in Chapter 23. This relationship amounts to the requirement that every ordered type is a fortiori an equality type.

This is well and good, but it would be even better if there were a way to incrementally extend the equality type class to the ordered type class without having to rewrite the signature as we have done in the foregoing example. Instead, we would like to *layer* the comparison aspect on top of an equality type class to obtain the ordered type class. This is achieved using a *hierarchical signature* σ_{eqord} of the form

$$\sum X : \sigma_{\mathsf{eq}} \cdot \sigma_{\mathsf{ord}}^{X}.$$

In this signature we write $\sigma_{\mathsf{ord}}^{X}$ for the signature

$$[t :: \mathtt{S}(X \cdot \mathtt{s}); \langle \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle],$$

which refers to the static part of X, namely the type on which the equality relation is defined. The notation $\sigma_{\mathsf{ord}}^{X}$ emphasizes that this signature has a free module variable X occurring within it, and hence is meaningful only in a context in which X has been declared.

A value of the signature σ_{eqord} is a pair of modules, $\langle M_{\mathsf{eq}} \; ; \; M_{\mathsf{ord}} \rangle$, in which M_{eq} comprises a type equipped with an equality relation on it, and the second comprises a type equipped with an ordering relation on it. Crucially, the second type is constrained by the singleton kind in $\sigma_{\mathsf{ord}}^{X}$ to be the *same* as the first type. Such a constraint is called a *sharing specification*. The process of drawing out of the consequences of a sharing specification is called *sharing propagation*.

Sharing propagation is achieved by a combination of subkinding (as described in Chapter 24) and subtyping for signatures. For example, a particular ordering M_{natord} of the natural numbers is a module with signature

$$\sum X : \sigma_{\mathsf{nateq}} \cdot \sigma_{\mathsf{ord}}^{X}.$$

By covariance of the hierarchical signature, this signature is a subsignature of σ_{eqord}, so that by subsumption we may regard M_{natord} as a module of the latter signature. The static part of the subsignature is a singleton, so we may apply the rules of sharing propagation given in Chapter 24 to show that the subsignature is equivalent to the signature

$$\sum X : \sigma_{\mathsf{nateq}} \cdot \sigma_{\mathsf{natord}},$$

where σ_{natord} is the closed signature

$$[t :: \mathtt{S}(\mathtt{nat}); \langle \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Notice that sharing propagation has replaced the type $X \cdot \mathtt{s}$ in the signature with `nat`, eliminating the dependency on the module variable X. After another round of sharing propagation, this signature may be shown to be equivalent to the signature ρ_{natord} given by

$$[\_ :: \mathtt{S}(\mathtt{nat}); \langle \mathtt{lt} \hookrightarrow (\mathtt{nat} \times \mathtt{nat}) \to \mathtt{bool} \rangle].$$

Here we have replaced both occurrences of t in the type of the comparison operation with `nat` as a consequence of the kind of t. The net effect is to propagate the identity of the static part of M_{natord} to the signature of the second component of M_{natord}.

Although its value is a pair, which seems symmetric, a module of signature σ_{eqord} is asymmetric in that the signature of the second component is dependent on the first component itself. This dependence is made manifest by the occurrence of the module variable X in the signature σ_{ord}. Thus, for $\langle M_{\mathsf{eq}} \,;\, M_{\mathsf{ord}} \rangle$ to be a well-formed module of signature σ_{eqord}, the first component M_{eq} must have signature σ_{eq}, which is meaningful independently of the other component of the pair. On the other hand, the second component M_{ord} must have signature σ_{eq}^{X}, with the understanding that X stands for the module M_{eq}. In general this signature is not meaningful independently of M_{eq} itself, and hence it may not be possible to handle M_{ord} independently of M_{eq}.

Turning this the other way around, if M is *any* module of signature σ_{eqord}, then it is always sensible to project it onto its first coordinate to obtain a module $M \cdot 1$ of signature σ_{eq}. But it is not always sensible to project it onto its second coordinate, because it may not be possible to give a signature to $M \cdot 2$ in the case that the dependency on the first component cannot be resolved statically. This can happen if, for example, the $M \cdot 1$ is a sealed module, whose static part cannot be formed in order to ensure representation independence. In such a situation the dependence of the signature $\sigma_{\mathsf{ord}}^{X}$ on the module variable X cannot be eliminated, and so no signature may be given to the second projection. For this reason the first component of a module hierarchy is called a *submodule* of the hierarchy, whereas the second component may or may not be a submodule of it. Put in other terms, the second component of a hierarchy is "projectible" exactly when the dependence of its signature on the first component is eliminable by sharing propagation. That is, we may know enough about the first component statically to ensure that an independent type for the second component may be given. In that case the second component may be considered to be a submodule of the pair; otherwise, the second is inseparable from the first, and therefore cannot be projected from the pair.

Consider, for example, a module M_{natord} of signature σ_{natord}, which, we noted earlier, is a subsignature of σ_{eqord}. The first projection $M_{\mathsf{natord}} \cdot 1$ is a well-formed module of closed signature σ_{eq} and hence is a submodule of M_{natord}. The situation is less clear for the second projection $M_{\mathsf{natord}} \cdot 2$, because its signature $\sigma_{\mathsf{ord}}^{X}$ depends on the first component via the variable X. However, we previously noted that the signature σ_{natord} is equivalent to the signature

$$\sum\nolimits_{\_ : \sigma_{\mathsf{nateq}}} \rho_{\mathsf{natord}}$$

in which the dependency on X has been eliminated by sharing propagation. This, too, is a valid signature for M_{natord}, and hence the second projection $M_{\mathsf{natord}} \cdot 2$ is a well-formed module of closed signature ρ_{natord}. If, on the other hand, the *only* signature available for M_{natord} were σ_{eqord}, then the second projection would be ill-formed—the second component would not be separable from the first, and hence could not be considered a submodule of the pair.

The hierarchical dependency of the signature of the second component of a pair on the first component gives rise to a useful alternative interpretation of a hierarchical module

signature as describing a *family of modules* given by the second component, thought of as being indexed by the first component. In the case at hand, the totality of modules of the signature σ_{eqord} gives rise to a family of modules of signature $\sigma_{\mathsf{ord}}^{X}$, where X ranges over σ_{eq}. That is, with each choice M_{eq} of signature σ_{eq}, we associate the collection of choices M_{ord}, coherent with the first choice in accordance with the sharing constraint in $\sigma_{\mathsf{ord}}^{X}$, taking X to be M_{ord}. This collection is called the *fiber over* M_{eq}, and the totality of modules of signature σ_{eqord} is said to be *fibered over* σ_{eq} (by the first projection).

The preceding example illustrates the layering of one type class on top of another. It is also useful to layer a type abstraction over a type class. A good example is provided by a dictionary abstraction in which the type of keys is required to be of the class of ordered types, but is otherwise unspecified. The signature $\sigma_{\mathsf{keydict}}$ of such a dictionary is given as follows:

$$\sum X : \sigma_{\mathsf{eqord}} \cdot \sigma_{\mathsf{dict}}^{X},$$

where σ_{eqord} is the signature of ordered equality types (in either of the two forms just discussed), and $\sigma_{\mathsf{dict}}^{X}$ is the signature of dictionaries of some type τ, given as follows:

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau\ \mathtt{opt} \rangle].$$

The `ins` and `fnd` operations make use of the type $X \cdot \mathtt{s}$ of keys given by the submodule of the dictionary module. We may think of $\sigma_{\mathsf{keydict}}$ as specifying a family of dictionary modules, one for each choice of the ordered type of keys. Regardless of the interpretation, an implementation of the signature $\sigma_{\mathsf{keydict}}$ consists of a two-level hierarchy of the form $\langle M_1 ; M_2 \rangle$, where M_1 specifies the key type and its ordering, and M_2 implements the dictionary for keys of this type in terms of this ordering.

46.2 Parameterization

The signature $\sigma_{\mathsf{keydict}}$ may be understood as describing a family of dictionary modules indexed by a module of ordered keys. The totality of such modules evaluates to pairs consisting of the ordered type of keys together with the dictionary per se, specialized to that choice of keys. Although it is possible that the code of the dictionary operations differs for each choice of keys, it is more often the case that the *same* implementation may be used for *all* choices of keys, the only difference being that references to, say, $X \cdot \mathtt{lt}$ refers to a different function for each choice of key module X.

Such a uniform implementation of dictionaries is provided by the concept of a *parameterized module*, or *functor*. A functor is a module expressed as a function of an unknown module of specified signature. The uniform dictionary module would be expressed as a functor parameterized over the module implemeting keys, which is to say as a λ-abstraction of the form

$$\lambda\, Z : \sigma_{\mathsf{eqord}} \cdot M_{\mathsf{keydict}}.$$

Here M_{keydict} is the generic implementation of dictionaries in terms of an unspecified module Z of signature σ_{eqord}. The signature of Z expresses the requirement that the dictionary implementation relies on the keys being an ordered type, but makes no other requirement on it.

A functor is a form of module, and hence has a signature as well, called (oddly enough) a *functor signature*. The signature $\sigma_{\mathsf{dictfun}}$ of the functor M_{keydict} has the form

$$\prod Z : \sigma_{\mathsf{eqord}} \cdot \rho^{Z}_{\mathsf{keydict}},$$

which specifies that its *domain* is the signature σ_{eqord} of ordered types and whose *range* is a signature $\rho^{Z}_{\mathsf{keydict}}$ that depends on the module Z.

The range $\rho^{Z}_{\mathsf{keydict}}$ of the dictionary functor is defined to be a subsignature of $\sigma_{\mathsf{keydict}}$ in which the key type of the result is constrained to be the same as the key type given as argument. This constraint may be expressed by defining $\rho^{Z}_{\mathsf{keydict}}$ to be the hierarchical signature

$$\sum X : \rho^{Z}_{\mathsf{eqord}} \cdot \sigma^{X}_{\mathsf{dict}},$$

where $\rho^{Z}_{\mathsf{eqord}}$ is a subsignature of σ_{eqord} imposing the desired sharing constraint. This is itself a hierarchical signature of the form

$$\sum X : \rho^{Z}_{\mathsf{eq}} \cdot \sigma^{X}_{\mathsf{ord}},$$

where ρ^{Z}_{eq} is the subsignature of σ_{eq} given by

$$[t :: \mathtt{S}(Z \cdot \mathtt{1} \cdot \mathtt{s}) ; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

The singleton kind in ρ^{Z}_{eq} expresses the required sharing constraint between the key type in the result of the functor and the key type given as its argument.

It is evidently rather tedious to write out $\rho^{Z}_{\mathsf{eqord}}$ separately from σ_{eqord}, because it requires repetition of so much that is already present in the latter signature. To lighten the notation it is preferable to express more directly the close relationship between the two signatures using *signature modification* to impose a type sharing constraint on a given signature. In the present case we may define $\rho^{Z}_{\mathsf{eqord}}$ to be the modification of σ_{eqord} given by

$$Y : \sigma_{\mathsf{eqord}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{s} = Z \cdot \mathtt{1} \cdot \mathtt{s}.$$

The modification states that the signature σ_{eqord} is to be altered by imposing the constraint that the static part of its equality part is to share with the static part of the equality part of Z, which, recall, also has signature σ^{eqord}. We may similarly define the range signature $\rho^{Z}_{\mathsf{keydict}}$ of the dictionary functor using signature modification as follows:

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{1} \cdot \mathtt{s} = Z \cdot \mathtt{1} \cdot \mathtt{s}.$$

The left-hand side of the sharing constraint refers to the type of keys in the submodule of the dictionary, and the right-hand side refers to the type of keys given as argument to the functor.

The dictionary functor M_{dictfun} defines a generic implementation of dictionaries in terms of an ordered type of keys. An instance of the dictionary for a specific choice of keys is obtained by *applying*, or *instantiating*, it with a module of its domain signature σ_{eqord}.

For example, because M_{natord}, the type of natural numbers ordered in the usual way, is such a module, we may form the instance $M_{\mathsf{keydict}}\,(M_{\mathsf{natord}})$ to obtain a dictionary with numeric keys. By choosing other modules of signature σ_{eqord} we may obtain corresponding instances of the dictionary functor. More generally, if M is any module of signature $\sigma_{\mathsf{dictfun}}$, then it is a functor that we may apply to any module M_{key} of signature σ_{eqord} to obtain the instance $M\,(M_{\mathsf{key}})$.

But what is the signature of such an instance, and how may it be deduced? Recall that the result signature of $\sigma_{\mathsf{dictfun}}$ is dependent on the argument itself, and not just its signature. It is therefore not immediately clear what signature to assign to the instance; the dependency on the argument must be resolved in order to obtain a signature that makes sense independently of the argument. The situation is broadly similar to the problem of computing the signature of the second component of a hiearchical module, and similar methods are used to resolve the dependencies, namely to exploit subtyping for signatures to obtain a specialization of the result signature appropriate to the argument.

This is best illustrated by example. First, we note that by contravariance of subtyping for functor signatures, we may *weaken* a functor signature by *strengthening* its domain signature. In the case of the signature $\sigma_{\mathsf{dictfun}}$ of the dictionary functor, we may obtain a supersignature $\sigma_{\mathsf{natdictfun}}$ by strengthening its domain to require that the key type be the type of natural numbers:

$$\prod Z : \sigma_{\mathsf{natord}} \cdot \rho^{Z}_{\mathsf{keydict}}.$$

Fixing Z to be a module variable of the specialized signature σ_{natord}, the range signature $\rho^{Z}_{\mathsf{keydict}}$ is given by the modification

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot 1 \cdot 1 \cdot \mathsf{s} = Z \cdot 1 \cdot \mathsf{s}.$$

By sharing propagation this is equivalent to the closed signature ρ_{natdict}, given by

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot 1 \cdot 1 \cdot \mathsf{s} = \mathtt{nat},$$

because we may derive the equivalence of $Z \cdot 1 \cdot \mathsf{s}$ and `nat` once the signature of Z is specialized to σ_{natord}.

Now by subsumption if M is a module of signature $\sigma_{\mathsf{dictfun}}$, then M is also a module of the supersignature

$$\prod Z : \sigma_{\mathsf{natord}} \cdot \rho^{Z}_{\mathsf{keydict}}.$$

We have just shown that the latter signature is equivalent to the nondependent functor signature

$$\prod \_ : \sigma_{\mathsf{natord}} \cdot \rho_{\mathsf{natdict}}.$$

The range is now given independently of the argument, so we may deduce that if M_{natkey} has signature σ_{natord}, then the application $M\,(M_{\mathsf{natkey}})$ has the signature ρ_{natdict}.

The crucial point is that the dependence of the range signature on the domain signature is eliminated by propagating knowledge about the type components of the argument itself. Absent this knowledge, the functor application cannot be regarded as well-formed, much as the second projection from a hierarchy cannot be admitted if the dependency of its signature

on the first component cannot be eliminated. If the argument to the functor is a value, then it is always possible to find a signature for it that maximizes the propagation of type sharing information so that the dependency of the range on the argument can always be eliminated.

46.3 Extending Modules With Hierarchies and Parameterization

This section sketches the extension of the module language introduced in Chapter 45 to account for module hierarchies and module parameterization.

The syntax of $\mathcal{L}\{\texttt{mod}\}$ is enriched with the following clauses:

Sort			Abstract Form	Concrete Form	Description
Sig	σ	$::=$	$\texttt{hier}(\sigma_1;X.\sigma_2)$	$\sum X:\sigma_1\cdot\sigma_2$	hierarchy
			$\texttt{fun}(\sigma_1;X.\sigma_2)$	$\prod X:\sigma_1\cdot\sigma_2$	functor
Mod	M	$::=$	$\texttt{hier}(M_1;M_2)$	$\langle M_1\ ;\ M_2\rangle$	hierarchy
			$\texttt{fst}(M)$	$M\cdot 1$	first component
			$\texttt{snd}(M)$	$M\cdot 2$	second component
			$\texttt{fun}[\sigma](X.M)$	$\lambda\, X{:}\sigma\cdot M$	functor
			$\texttt{app}(M_1;M_2)$	$M_1\,(M_2)$	instance

The syntax of signatures is extended to include hierarchies and functors, and the syntax of modules is correspondingly extended with introduction and elimination forms for these signatures.

The judgment M projectible states that the module M is *projectible* in the sense that its constituent types may be referenced by compositions of projections, including the static part of a structure. This judgment is inductively defined by the following rules:

$$\frac{}{\Gamma, X:\sigma \vdash x \text{ projectible}} \tag{46.1a}$$

$$\frac{\Gamma \vdash M_1 \text{ projectible} \quad \Gamma \vdash M_2 \text{ projectible}}{\Gamma \vdash \langle M_1\ ;\ M_2\rangle \text{ projectible}} \tag{46.1b}$$

$$\frac{\Gamma \vdash M \text{ projectible}}{\Gamma \vdash M\cdot 1 \text{ projectible}} \tag{46.1c}$$

$$\frac{\Gamma \vdash M \text{ projectible}}{\Gamma \vdash M\cdot 2 \text{ projectible}}\,. \tag{46.1d}$$

All module variables are deemed projectible, even though this condition is relevant only for hierarchies of basic structures. Because the purpose of sealing is to hide the representation of an abstract type, no sealed module is deemed projectible. Furthermore, no functor is projectible, because there is no concept of projection for a functor. More important, no functor instance is projectible either. This ensures that any two instances of the same functor define distinct abstract types; functors are therefore said to be *generative*. (See Section 46.4 for a discussion of an alternative treatment of functors.)

The signature formation judgment is extended to include these rules:

$$\frac{\Gamma \vdash \sigma_1 \ \mathsf{sig} \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \ \mathsf{sig}}{\Gamma \vdash \textstyle\sum X : \sigma_1 \cdot \sigma_2 \ \mathsf{sig}} \tag{46.2a}$$

$$\frac{\Gamma \vdash \sigma_1 \ \mathsf{sig} \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \ \mathsf{sig}}{\Gamma \vdash \textstyle\prod X : \sigma_1 \cdot \sigma_2 \ \mathsf{sig}}\,. \tag{46.2b}$$

Signature equivalence is defined to be compatible with the two new forms of signature:

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \equiv \sigma_2'}{\Gamma \vdash \textstyle\sum X : \sigma_1 \cdot \sigma_2 \equiv \sum X : \sigma_1' \cdot \sigma_2'} \tag{46.3a}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \equiv \sigma_2'}{\Gamma \vdash \textstyle\prod X : \sigma_1 \cdot \sigma_2 \equiv \prod X : \sigma_1' \cdot \sigma_2'} \tag{46.3b}$$

The subsignature judgment is augmented with the following rules:

$$\frac{\Gamma \vdash \sigma_1 <: \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 <: \sigma_2'}{\Gamma \vdash \textstyle\sum X : \sigma_1 \cdot \sigma_2 <: \sum X : \sigma_1' \cdot \sigma_2'} \tag{46.4a}$$

$$\frac{\Gamma \vdash \sigma_1' <: \sigma_1 \quad \Gamma, X : \sigma_1' \vdash \sigma_2 <: \sigma_2'}{\Gamma \vdash \textstyle\prod X : \sigma_1 \cdot \sigma_2 <: \prod X : \sigma_1' \cdot \sigma_2'}\,. \tag{46.4b}$$

Rule (46.4a) specifies that the hierarchical signature is covariant in both positions, whereas Rule (46.4b) specifies that the functor signature is contravariant in its domain and covariant in its range.

The statics of module expressions is extended by the following rules:

$$\frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma \vdash M_2 : \sigma_2}{\Gamma \vdash \langle M_1 \ ; \ M_2 \rangle : \textstyle\sum \_ : \sigma_1 \cdot \sigma_2} \tag{46.5a}$$

$$\frac{\Gamma \vdash M : \textstyle\sum X : \sigma_1 \cdot \sigma_2}{\Gamma \vdash M \cdot 1 : \sigma_1} \tag{46.5b}$$

$$\frac{\Gamma \vdash M : \textstyle\sum \_ : \sigma_1 \cdot \sigma_2}{\Gamma \vdash M \cdot 2 : \sigma_2} \tag{46.5c}$$

$$\frac{\Gamma, X : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \lambda\, X : \sigma_1 \cdot M_2 : \textstyle\prod X : \sigma_1 \cdot \sigma_2} \tag{46.5d}$$

$$\frac{\Gamma \vdash M_1 : \textstyle\prod \_ : \sigma_2 \cdot \sigma \quad \Gamma \vdash M_2 : \sigma_2}{\Gamma \vdash M_1\,(M_2) : \sigma}\,. \tag{46.5e}$$

Rule (46.5a) states that an explicit module hierarchy is given a signature in which there is no dependency of the signature of the second component on the first component (indicated here by the underscore in place of the module variable). A dependent signature may be given to a hierarchy by sealing, which makes it into a nonvalue, even if the components are values. Rule (46.5b) states that the first projection is defined for general hierarchical

signatures. However, Rule (46.5c) restricts the second projection to nondependent hierarchies, as discussed in the preceding section. Similarly, Rule (46.5e) restricts instantiation to functors whose types are nondependent, forcing any dependencies to be resolved using the subsignature relation and sharing propagation prior to application.

The self-recognition rules given in Chapter 45 are extended to account for the formation of hierarchical module value by the following rules:

$$\frac{\Gamma \vdash M \text{ projectible} \quad \Gamma \vdash M : \sum X : \sigma_1 \cdot \sigma_2 \quad \Gamma \vdash M \cdot 1 : \sigma_1'}{\Gamma \vdash M : \sum X : \sigma_1' \cdot \sigma_2} \tag{46.6a}$$

$$\frac{\Gamma \vdash M \text{ projectible} \quad \Gamma \vdash M : \sum\nolimits_{\_} : \sigma_1 \cdot \sigma_2 \quad \Gamma \vdash M \cdot 2 : \sigma_2'}{\Gamma \vdash M : \sum\nolimits_{\_} : \sigma_1 \cdot \sigma_2'}. \tag{46.6b}$$

Rules (46.6a) and (46.6b) permit the specialization of the signature of a hierarchical module value to express that its constructor components are equivalent to their projections from the module itself.

46.4 Applicative Functors

In the module language just described, functors are regarded as generative in the sense that any two instances, even with arguments, are considered to "generate" distinct abstract types. This is ensured by treating a functor application $M\,(M_1)$ to be nonprojectible, so that if it defines an abstract type in the result, that type cannot be referenced without first binding the application to a variable. Any two such bindings are necessarily to distinct variables X and Y and so the abstract types $X \cdot \mathsf{s}$ and $Y \cdot \mathsf{s}$ are distinct, regardless of their bindings.

The justification for this design decision merits careful consideration. By treating functors as generative, we are ensuring that a client of the functor cannot in any way rely on the implementation of that functor. That is, we are extending the principle of representation independence for abstract types to functors in a natural way. One consequence of this policy is that the module language is compatible with extensions such as a *conditional module* that branches on an arbitrary dynamic condition that might even depend on external conditions such as the phase of the moon! A functor with such an implementation *must* be considered generative, because the abstract types arising from any instance cannot be regarded as well-defined until the moment when the application is evaluated, which amounts to the point at which it is bound to a variable. By regarding all functors as generative, we are, in effect, maximizing opportunities to exploit changes of representation without disrupting the behavior of clients of the functor, a bedrock principle of modular decomposition.

But because the module language considered in the preceding section does not include anything so powerful as a conditional module, we might consider that the restriction to generative functors is too severe and may be usefully relaxed. One such alternative is the

concept of an *applicative* functor. An applicative functor is one for which instances by values are regarded as projectible:[1]

$$\frac{M \text{ projectible} \quad M_1 \text{ val}}{M\ (M_1)\ \text{projectible}}\,. \tag{46.7}$$

It is important to bear in mind that, because of this rule, applicative functors are *not compatible* with conditional modules. Thus a module language based on applicative functors is inherently restricted as compared with one based on generative functors.

The benefit of regarding a functor instance as projectible is that we may form types such as $(M\ (M_1)) \cdot \mathsf{s}$, which projects the static part of the instance. But this raises the question: When are two such type expressions to be deemed equivalent? The difficulty is that the answer to this question depends on the functor argument. For suppose that F is an applicative functor variable; under what conditions should $(F\ (M_1)) \cdot \mathsf{s}$ and $(F\ (M_2)) \cdot \mathsf{s}$ be regarded as the same type? In the case of generative functors we did not have face this question, because the instances are not projectible, but for applicative functors the question cannot be dodged, but must be addressed. We will return to this point in a moment, after considering one further complication that raises a similar issue.

The difficulty is that the body of an applicative functor cannot be sealed to impose abstraction, and, according to the rules given in the preceding section, no sealed module is projectible. Because sealing is the only means of imposing abstraction, we must relax this condition and allow sealed projectible modules to be projectible:

$$\frac{M \text{ projectible}}{M \upharpoonright \sigma \text{ projectible}}\,. \tag{46.8}$$

Thus, we may form type expressions of the form $(M \upharpoonright \sigma) \cdot \mathsf{s}$, which project the static part of a sealed module. And once again we are faced with the issue that the equivalence of two such types must involve the equivalence of the sealed modules themselves, in apparent violation of representation independence.

Summarizing, if we are to deem functors to be applicative, then some compromise of the principle of representation independence for abstract types is required. We must define equivalence for the static parts of sealed modules, and doing so requires at least checking whether the underlying modules are identical. This has two consequences. Because the underlying modules have both static and dynamic parts, this means comparing their executable code for equivalence during type checking. More significantly, the formation of a client may depend on the equivalence of two modules; we cannot change the representation of a sealed module without fear of disrupting the typing or behavior of the client. This undermines the very purpose of having a module system in the first place!

[1] We may, in addition, regard functor abstractions as projectible, but because all variables are projectible, there is no harm in omitting this and instead insisting that functors be bound to variables before being used.

46.5 Notes

Module hierarchies and functors in the form discussed here were introduced by Milner et al. (1997), which also employed the reading of a module hierarchy as an indexed family of modules. The theory of hierarchies and functors was first studied by Harper and Lillibridge (1994) and Leroy (1994), building on earlier work by Mitchell and Plotkin (1988) on existential types. The concept of an applicative functor was introduced by Leroy (1995) and is central to the module system of O'Caml (OCaml).

PART XVIII

Equational Reasoning

47 Equational Reasoning for T

The beauty of functional programming is that equality of expressions in a functional language corresponds very closely to familiar patterns of mathematical reasoning. For example, in the language $\mathcal{L}\{\texttt{nat}\to\}$ of Chapter 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* $\mathcal{L}\{\mathit{nat}\to\}$ is commutative.

This may seem to be obviously true, but *why*, precisely, is it so? More important, what do we even *mean* for two expressions to be equal in this sense? It is intuitively obvious that these two expressions are not *definitionally* equivalent, because they cannot be shown equivalent by symbolic execution. We may say that these two expressions are definitionally inequivalent because they describe different algorithms: one proceeds by recursion on x, the other by recursion on y. On the other hand, the two expressions are interchangeable in any complete computation of a natural number, because the only use we can make of them is to apply them to arguments and compute the result. Two functions are *logically equivalent* if they give equal results for equal arguments—in particular, they agree on all possible arguments. Because their behavior on arguments is all that matters for calculating observable results, we may expect that logically equivalent functions are equal in the sense of being interchangeable in all complete programs. Thinking of the programs in which these functions occur as *observations* of their behavior, these functions are said to be *observationally equivalent*. The main result of this chapter is that observational and logical equivalences coincide for a variant of $\mathcal{L}\{\texttt{nat}\to\}$ in which the successor is evaluated eagerly, so that a value of type `nat` is a numeral.

47.1 Observational Equivalence

When are two expressions equal? Whenever we cannot tell them apart! This may seem tautological, but it is not, because it depends on what we consider to be a means of telling expressions apart. What "experiment" are we permitted to perform on expressions in order

to distinguish them? What counts as an observation that, if different for two expressions, is a sure sign that they are different?

If we permit ourselves to consider the syntactic details of the expressions, then very few expressions could be considered equal. For example, if it is deemed significant that an expression contains, say, more than one function application, or that it has an occurrence of λ-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation and not in the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamics of $\mathcal{L}\{\mathtt{nat}\to\}$ is given only for expressions without free variables, so a complete program should clearly be a *closed* expression. Second, the outcome of a computation should be *observable*, so that it is evident whether the outcome of two computations differs or not. We define a *complete program* to be a closed expression of type `nat`, and define the *observable behavior* of the program to be the numeral to which it evaluates.

An *experiment* on, or *observation* about, an expression is any means of using that expression within a complete program. We define an *expression context* to be an expression with a "hole" in it serving as a placeholder for another expression. The hole is permitted to occur anywhere, including within the scope of a binder. The bound variables within whose scope the hole lies are said to be *exposed to capture* by the expression context. These variables may be assumed, without loss of generality, to be distinct from one another. A *program context* is a closed expression context of type `nat`—that is, it is a complete program with a hole in it. The metavariable $\mathcal{C}$ 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 as $\mathcal{C}\{e\}$. 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 as $\circ$, for which $\circ\{e\} = e$.

The statics of expressions of $\mathcal{L}\{\mathtt{nat}\to\}$ is extended to expression contexts by defining the typing 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 may be inductively defined by a collection of rules derived from the statics of $\mathcal{L}\{\mathtt{nat}\to\}$ [see Rules (9.1)]. Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \tag{47.1a}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})}{\mathtt{s}(\mathcal{C}) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})} \tag{47.1b}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,\mathcal{C}\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{47.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau') \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,e\,\{\mathtt{z} \Rightarrow \mathcal{C}_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{47.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \rhd \tau')}{\mathtt{rec}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow \mathcal{C}_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{47.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \tau_1 \rhd \tau_2)}{\lambda\,(x:\tau_1)\,\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2)} \tag{47.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{47.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')}\,. \tag{47.1h}$$

Lemma 47.1. *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 47.2. *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 47.3. *If* $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ *and* $x \notin \mathit{dom}(\Gamma)$, *then* $\mathcal{C} : (\Gamma, x : \rho \rhd \tau) \rightsquigarrow (\Gamma', x : \rho \rhd \tau')$.

Proof By induction on Rules (47.1). □

A *complete program* is a closed expression of type `nat`.

Definition 47.4. *Two complete programs, e and e', are* Kleene equal*, written as $e \simeq e'$, iff there exists $n \geq 0$ such that $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$.*

Kleene equality is evidently reflexive and symmetric; transitivity follows from determinacy of evaluation. Closure under converse evaluation also follows directly from determinacy. It is immediate from the definition that $\overline{0} \not\simeq \overline{1}$.

Definition 47.5. *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 as $\Gamma \vdash e \cong e' : \tau$, iff $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ for every program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$.*

In other words, for all possible experiments, the outcome of an experiment on e is the same as the outcome on e'. This is obviously an equivalence relation. For the sake of brevity, we often write $e \cong_\tau e'$ for $\emptyset \vdash e \cong e' : \tau$.

A family of equivalence 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 $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is *consistent* iff $\emptyset \vdash e \ \mathcal{E} \ e' : \mathtt{nat}$ implies $e \simeq e'$.

Theorem 47.6. *Observational equivalence is the coarsest consistent congruence on expressions.*

Proof Consistency follows directly from the definition by noting that the trivial context is a program context. Observational equivalence is obviously an equivalence relation. To show that it is a congruence, we need only observe that type-correct composition of a program context with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $\Gamma \vdash e \ \mathcal{E} \ e' : \tau$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, then by congruence $\emptyset \vdash \mathcal{C}\{e\} \ \mathcal{E} \ \mathcal{C}\{e'\} : \mathtt{nat}$, 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 47.7. *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 $\mathcal{C} : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$ be a program context; we are to show that $\mathcal{C}\{\hat{\gamma}(e)\} \simeq \mathcal{C}\{\hat{\gamma}(e')\}$. 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 47.3 we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows directly that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$. 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 $\mathcal{D}'$ similarly to $\mathcal{D}$, but based on γ' rather than on 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 47.6 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 $\Gamma \vdash e\ \mathcal{E}\ e' : \tau$. It can be difficult to construct such a relation. The next section provides a general method for doing so that exploits types.

47.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of expressions of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that merely manipulate expressions without actually inspecting them. For example, we may pass an expression of type τ to a function that merely returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate expressions at arm's length, affording no opportunities to distinguish one from another.

This leads to the definition of logical equivalence alluded to in the introduction.

Definition 47.8. 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 is, of course, justified by mathematical induction on $n \geq 0$, where $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$ by the definition of Kleene equivalence.

Lemma 47.9. *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 that $e' \sim_\tau e$. 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 as $\Gamma \vdash e \sim e' : \tau$, is defined to mean that $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

Lemma 47.10. *Open logical equivalence is symmetric and transitive.*

Proof Follows immediately from Lemma 47.9 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.

47.3 Logical and Observational Equivalences Coincide

In this section we prove the coincidence of observational and logical equivalences.

Lemma 47.11 (Converse Evaluation)**.** *Suppose that $e \sim_\tau e'$. If $d \mapsto e$, then $d \sim_\tau e'$, and if $d' \mapsto e'$, then $e \sim_\tau d'$.*

Proof By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows from the closure of Kleene equivalence under converse evaluation. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \sim_\tau e'$, and $d \mapsto e$. To show that $d \sim_\tau e'$, we assume that $e_1 \sim_{\tau_1} e_1'$ and show that $d(e_1) \sim_{\tau_2} e'(e_1')$. It follows from the assumption that $e(e_1) \sim_{\tau_2} e'(e_1')$. Noting that $d(e_1) \mapsto e(e_1)$, the result follows by induction. □

Lemma 47.12 (Consistency). *If $e \sim_{\mathtt{nat}} e'$, then $e \simeq e'$.*

Proof Immediate, from Definition 47.8. □

Theorem 47.13 (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 47.11, 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 directly.

Now consider the case of Rule (9.1d), for which we are to show that

$$\mathtt{rec}(\hat{\gamma}(e); \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{rec}(\widehat{\gamma'}(e); \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)).$$

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 `nat`-induction. It suffices to show that

$$\mathtt{rec}(\mathtt{z}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{rec}(\mathtt{z}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)) \tag{47.2}$$

and that

$$\mathtt{rec}(\mathtt{s}(\overline{n}); \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{rec}(\mathtt{s}(\overline{n}); \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)), \tag{47.3}$$

assuming

$$\mathtt{rec}(\overline{n}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1)) \sim_\tau \mathtt{rec}(\overline{n}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)). \tag{47.4}$$

To show (47.2), by Lemma 47.11 it is enough to show that $\hat{\gamma}(e_0) \sim_\tau \widehat{\gamma'}(e_0)$. This is ensured by the outer inductive hypothesis applied to the second premise of Rule (9.1d).

To show (47.3), define

$$\delta = \gamma \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}(\overline{n}; \hat{\gamma}(e_0); x\,.\,y\,.\,\hat{\gamma}(e_1))$$

and

$$\delta' = \gamma' \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}(\overline{n}; \widehat{\gamma'}(e_0); x\,.\,y\,.\,\widehat{\gamma'}(e_1)).$$

By (47.4) 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 47.11, the required follows. □

Corollary 47.14 (Equivalence). *Open logical equivalence is an equivalence relation.*

Corollary 47.15 (Termination). *If $e : \mathtt{nat}$, then $e \mapsto^* e'$ for some e' val.*

Lemma 47.16 (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 that $\Gamma \vdash e \sim e' : \tau$, 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 47.8 it is enough to show that

$$\hat{\gamma_1}(\mathcal{C}_2\{e\}) \sim_{\tau_2} \hat{\gamma_1'}(\mathcal{C}_2\{e'\}),$$

which follows immediately from the inductive hypothesis. □

Theorem 47.17. *If* $\Gamma \vdash e \sim e' : \tau$*, then* $\Gamma \vdash e \cong e' : \tau$.

Proof By Lemmas 47.12 and 47.16 and Theorem 47.6. □

Corollary 47.18. *If* $e : \mathtt{nat}$*, then* $e \cong_{\mathtt{nat}} \overline{n}$*, for some* $n \geq 0$.

Proof By Theorem 47.13 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 47.17, $e \cong_{\mathtt{nat}} \overline{n}$. □

Lemma 47.19. *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 47.17 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 47.20. *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 47.17 we have $\gamma \cong_\Gamma \gamma'$, so by Lemma 47.7 $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. Therefore, by Lemma 47.19, $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. □

Corollary 47.21. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \sim e' : \tau$.

The principle of *symbolic evaluation* states that definitional equivalence is sufficient for observational equivalence:

Theorem 47.22. *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 47.13 and Lemma 47.16, then appealing to Theorem 47.17. □

Corollary 47.23. *If* $e \equiv e' : \mathtt{nat}$, *then there exists* $n \geq 0$ *such that* $e \mapsto^* \overline{n}$ *and* $e' \mapsto^* \overline{n}$.

Proof By Theorem 47.22 we have $e \sim_{\mathtt{nat}} e'$ and hence $e \simeq e'$. □

47.4 Some Laws of Equality

In this section we summarize some useful principles of observational equivalence for $\mathcal{L}\{\mathtt{nat} \to\}$. For the most part these may be proved as laws of logical equivalence and then transferred to observational equivalence by appeal to Corollary 47.21. 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.

47.4.1 General Laws

Logical equivalence is indeed an equivalence relation: It is reflexive, symmetric, and transitive:

$$\frac{}{\Gamma \vdash e \cong e : \tau} \tag{47.5a}$$

$$\frac{\Gamma \vdash e' \cong e : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{47.5b}$$

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau}. \tag{47.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences:

$$\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e \cong e' : \tau}. \tag{47.6}$$

This is called the *principle of symbolic evaluation*.

Observational equivalence is a congruence: We may replace equals by equals anywhere in an expression:

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\Gamma' \vdash \mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'}. \tag{47.7}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e/x]e_2 \cong [e/x]e_2' : \tau'} \tag{47.8a}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'}. \tag{47.8b}$$

47.4.2 Equality Laws

Two functions are equal if they are equal on all arguments:

$$\frac{\Gamma, x : \tau_1 \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2}. \tag{47.9}$$

Consequently, every expression of function type is equal to a λ-abstraction:

$$\frac{}{\Gamma \vdash e \cong \lambda\,(x:\tau_1)\,e(x) : \tau_1 \to \tau_2}. \tag{47.10}$$

47.4.3 Induction Law

An equation involving a free variable x of type `nat` can be proved by induction on x.

$$\frac{\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \text{ (for every } n \in \mathbb{N})}{\Gamma, x : \mathtt{nat} \vdash e \cong e' : \tau}. \tag{47.11}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing that

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 $\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau$.

47.5 Notes

The technique of *logical relations* interprets types as relations (here, equivalence relations) by associating with each type constructor a relational action that transforms the relation

interpreting its arguments to the relation interpreting the constructed type. Logical relations (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 essentially an adaptation of the NuPRL semantics to the simpler setting of Gödel's System **T**.

48 Equational Reasoning for PCF

In this chapter we develop the theory of observational equivalence for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, with an eager interpretation of the type of natural numbers. The development proceeds along lines similar to those in Chapter 47, 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*.

48.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 47, but modified to account for the possibility of nontermination.

The collection of well-formed $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ contexts is inductively defined in a manner directly analogous to that in Chapter 47. Specifically, we define the judgment $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ by rules similar to Rules (47.1), modified for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. (The precise definition is left 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 $\mathtt{nat}$.

Definition 48.1. *We say that two complete programs e and e' are* Kleene equal*, written as $e \simeq e'$, iff for every $n \geq 0$, $e \mapsto^* \overline{n}$ iff $e' \mapsto^* \overline{n}$.*

Kleene equality is easily seen to be an equivalence relation and to be closed under converse evaluation. Moreover, $\overline{0} \not\simeq \overline{1}$, and, if e and e' are both divergent, then $e \simeq e'$.

Observational equivalence is defined just as it is in Chapter 47.

Definition 48.2. *We say that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are* observationally*, or* contextually*,* equivalent *iff for every program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.*

Theorem 48.3. *Observational equivalence is the coarsest consistent congruence.*

Proof See the proof of Theorem 47.6. □

Lemma 48.4 (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 47.7. □

48.2 Logical Equivalence

Definition 48.5. 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 47, except that the definition of Kleene equivalence is altered to account for nontermination. 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 47.9, logical equivalence is symmetric and transitive, as is its open extension.

Lemma 48.6 (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 48.7 (Converse Evaluation)**.** *Suppose that* $e \sim_\tau e'$*. If* $d \mapsto e$*, then* $d \sim_\tau e'$*, and if* $d' \mapsto e'$*, then* $e \sim_\tau d'$*.*

48.3 Logical and Observational Equivalences Coincide

The proof of coincidence of logical and observational equivalences relies on the concept of *bounded recursion*, which we define by induction on $m \geq 0$ as follows:

$$\mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e \triangleq \mathtt{fix}\, x : \tau \,\mathtt{is}\, x$$
$$\mathtt{fix}^{m+1}\, x : \tau \,\mathtt{is}\, e \triangleq [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e/x]e.$$

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 $m \geq 0$:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}^m\,[\tau]\,(x\,.\,e) : \tau}\ . \tag{48.1}$$

The proof is by induction on $m \geq 0$ and amounts to an iteration of the substitution lemma for the statics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

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 is stated and proved in Section 48.4.

Theorem 48.8 (Fixed Point Induction)**.** *Suppose that* $x : \tau \vdash e : \tau$. *If*

$$(\forall m \geq 0)\ \mathit{fix}^m\, x : \tau\ \mathtt{is}\ e \sim_\tau \mathit{fix}^m\, x : \tau\ \mathtt{is}\ e',$$

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 $\mathcal{A} : \rho \rightsquigarrow \tau$ is a special case of the general typing judgment for contexts.) Define logical equivalence of applicative contexts, written as $\mathcal{A} \sim \mathcal{A}' : \rho \rightsquigarrow \tau$, by induction on the structure of $\mathcal{A}$ as follows:

1. $\circ \sim \circ : \rho \rightsquigarrow \rho$;
2. if $\mathcal{A} \sim \mathcal{A}' : \rho \rightsquigarrow \tau_2 \to \tau$ and $e_2 \sim_{\tau_2} e_2'$, then $\mathcal{A}(e_2) \sim \mathcal{A}'(e_2') : \rho \rightsquigarrow \tau$.

We prove by induction on the structure of τ, if $\mathcal{A} \sim \mathcal{A}' : \rho \rightsquigarrow \tau$ and

$$\text{for every } m \geq 0,\ \mathcal{A}\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e'\}, \tag{48.2}$$

then

$$\mathcal{A}\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e'\}. \tag{48.3}$$

Choosing $\mathcal{A} = \mathcal{A}' = \circ$ (so that $\rho = \tau$) completes the proof.

If $\tau = \mathtt{nat}$, then assume that $\mathcal{A} \sim \mathcal{A}' : \rho \rightsquigarrow \mathtt{nat}$ and (48.2). By Definition 48.5, we are to show that

$$\mathcal{A}\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e'\}.$$

By Corollary 48.17 there exists $m \geq 0$ such that

$$\mathcal{A}\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e\} \simeq \mathcal{A}\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e\}.$$

By (48.2) we have

$$\mathcal{A}\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e\} \simeq \mathcal{A}'\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e'\}.$$

By Corollary 48.17

$$\mathcal{A}'\{\mathtt{fix}^m\, x : \rho\ \mathtt{is}\ e'\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \rho\ \mathtt{is}\ e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 48.5, it is enough to show that

$$\mathcal{A}\{\mathtt{fix}\, x : \rho\, \mathtt{is}\, e\}(e_1) \sim_{\tau_2} \mathcal{A}'\{\mathtt{fix}\, x : \rho\, \mathtt{is}\, e'\}(e_1')$$

whenever $e_1 \sim_{\tau_1} e_1'$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (48.2) that for every $m \geq 0$

$$\mathcal{A}_2\{\mathtt{fix}^m\, x : \rho\, \mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}^m\, x : \rho\, \mathtt{is}\, e'\}.$$

Noting that $\mathcal{A}_2 \sim \mathcal{A}_2' : \rho \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\, x : \rho\, \mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}\, x : \rho\, \mathtt{is}\, e'\},$$

as required. □

Lemma 48.9 (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 47.13. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider Rule (10.1g). 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 48.8 it is enough to show that, for every $m \geq 0$,

$$\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \widehat{\gamma'}(e).$$

We proceed by an inner induction on m. When $m = 0$ the result is immediate, because both sides of the desired equivalence diverge. Assuming the result for m and applying Lemma 48.7, it is enough to show that $\hat{\gamma}(e_1) \sim_\tau \widehat{\gamma'}(e_1)$, where

$$e_1 = [\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \tag{48.4}$$

$$e_1' = [\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e). \tag{48.5}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \widehat{\gamma'}(e),$$

then

$$[\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e) \sim_\tau [\mathtt{fix}^m\, x : \tau\, \mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e).$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows. □

Symmetry and transitivity of eager logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager logical equivalence is therefore an equivalence relation.

Lemma 48.10 (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 48.9. □

Logical equivalence is consistent, by definition. Consequently, it is contained in observational equivalence.

Theorem 48.11. *If* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma \vdash e \cong e' : \tau$.

Proof By consistency and congruence of logical equivalence. □

Lemma 48.12. *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 $\tau = \tau_1 \to \tau_2$, then suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 48.11 $e_1 \cong_{\tau_1} e_1'$, and hence by Lemma 48.4 $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

Theorem 48.13. *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 48.11 we have $\gamma \cong_\Gamma \gamma'$, and so by Lemma 48.4 we have

$$\hat{\gamma}(e) \cong_\tau \hat{\gamma'}(e').$$

Therefore by Lemma 48.12 we have

$$\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e').$$

□

Corollary 48.14. $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \sim e' : \tau$.

48.4 Compactness

The principle of fixed point induction is derived from a critical property of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called *compactness*. This property states that only finitely many unwindings of a fixed point expression are needed in a complete evaluation of a program. 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 48.16) makes use of the stack machine for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ defined in Chapter 27, augmented with the following transitions for bounded recursive expressions:

$$\frac{}{k \triangleright \mathtt{fix}^0\, x{:}\tau\ \mathtt{is}\ e \mapsto k \triangleright \mathtt{fix}^0\, x{:}\tau\ \mathtt{is}\ e} \tag{48.6a}$$

$$\frac{}{k \triangleright \mathtt{fix}^{m+1}\, x{:}\tau\ \mathtt{is}\ e \mapsto k \triangleright [\mathtt{fix}^m\, x{:}\tau\ \mathtt{is}\ e/x]e}\,. \tag{48.6b}$$

It is straightforward to extend the proof of correctness of the stack machine (Corollary 27.4) to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function f in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$:

$$\mathtt{fix}\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda\, (x\mathtt{:nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}.$$

Obviously evaluation of $f(\overline{n})$ requires n recursive calls to the function itself. This means that, for a given input n we may place a *bound* m on the recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the m-bounded form of general recursion,

$$\mathtt{fix}^m\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda\, (x\mathtt{:nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}.$$

Call this expression $f^{(m)}$. It follows from the definition of f that if $f(\overline{n}) \mapsto^* \overline{p}$, then $f^{(m)}(\overline{n}) \mapsto^* \overline{p}$ for some $m \geq 0$ (in fact, $m = n$ suffices).

When considering expressions of higher type, we cannot expect to get the *same* result from the bounded recursion as from the unbounded. For example, consider the addition function a of type $\tau = \mathtt{nat} \rightharpoonup (\mathtt{nat} \rightharpoonup \mathtt{nat})$, given by the expression

$$\mathtt{fix}\, p\mathtt{:}\tau\, \mathtt{is}\, \lambda\, (x\mathtt{:nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow id \mid \mathtt{s}(x') \Rightarrow s \circ (p(x'))\},$$

where $id = \lambda\, (y\mathtt{:nat})\, y$ is the identity, $e' \circ e = \lambda\, (x\mathtt{:}\tau)\, e'(e(x))$ is composition, and $s = \lambda\, (x\mathtt{:nat})\, \mathtt{s}(x)$ is the successor function. The application $a(\overline{n})$ terminates after three transitions, regardless of the value of n, resulting in a λ-abstraction. When n is positive, the result contains a *residual* copy of a itself, which is applied to $n-1$ as a recursive call. The m-bounded version of a, written as $a^{(m)}$, 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 $f^{(\omega)} = \mathtt{fix}\, x\mathtt{:}\tau\, \mathtt{is}\, e_x$, and let $f^{(m)} = \mathtt{fix}^m\, x\mathtt{:}\tau\, \mathtt{is}\, e_x$. Observe that $f^{(\omega)} : \tau$ and $f^{(m)} : \tau$ for any $m \geq 0$.

The following technical lemma governing the stack machine permits the bound on occurrences of a recursive expression to be raised without affecting the outcome of evaluation.

Lemma 48.15. *For every $m \geq 0$, if $[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$, then $[f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$.*

Proof By induction on $m \geq 0$ and then induction on transition. □

Theorem 48.16 (Compactness)**.** *Suppose that $y : \tau \vdash e : \mathtt{nat}$, where $y \notin f^{(\omega)}$. If $[f^{(\omega)}/y]e \mapsto^* \overline{n}$, then there exists $m \geq 0$ such that $[f^{(m)}/y]e \mapsto^* \overline{n}$.*

Proof We prove simultaneously the stronger statements that if

$$[f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

and if

$$[f^{(\omega)}/y]k \triangleleft [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleleft [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}.$$

(Note that if $[f^{(\omega)}/y]e$ val, then $[f^{(m)}/y]e$ val for all $m \geq 0$.) The result then follows by the correctness of the stack machine (Corollary 27.4).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)},$$

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)} \mapsto [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

Noting that $[f^{(\omega)}/x]e_x = [f^{(\omega)}/y][y/x]e_x$, we have by induction that there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

By Lemma 48.15

$$[f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}$$

and we need recall only that

$$[f^{(m+1)}/y]k \triangleright f^{(m+1)} = [f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x$$

to complete the proof. If, however, the initial step is an unrolling, but $e \neq y$, then we have for some $z \notin f^{(\omega)}$ and $z \neq y$

$$[f^{(\omega)}/y]k \triangleright \mathtt{fix}\, z\mathtt{:}\tau\ \mathtt{is}\, d_\omega \mapsto [f^{(\omega)}/y]k \triangleright [\mathtt{fix}\, z\mathtt{:}\tau\ \mathtt{is}\, d_\omega/z]d_\omega \mapsto^* \epsilon \triangleleft \overline{n}$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [\mathtt{fix}\, z\mathtt{:}\tau\ \mathtt{is}\, d_m/z]d_m \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 48.15 we have

$$[f^{(m+1)}/y]k \triangleright [\mathtt{fix}\, z\mathtt{:}\tau\ \mathtt{is}\, d_{m+1}/z]d_{m+1} \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

Corollary 48.17. *There exists $m \geq 0$ such that $[f^{(\omega)}/y]e \simeq [f^{(m)}/y]e$.*

Proof If $[f^{(\omega)}/y]e$ diverges, then taking m to be zero suffices. Otherwise, apply Theorem 48.16 to obtain m and note that the required Kleene equivalence follows. □

48.5 Conatural Numbers

If we change the dynamics of the successor operator in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ so that $\mathtt{s}(e)$ is a value regardless of whether e is a value, then the type $\mathtt{nat}$ admits an infinite "number" $\omega = \mathtt{fix}\, x\!:\!\mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x)$. We may think of ω as an infinite stack of successors, hence larger than any finite natural number. Obviously the principle of mathematical induction is not valid for this type, because we may prove by induction that every natural number is finite, whereas ω is infinite. When the successor is evaluated lazily, it is preferable to rename $\mathtt{nat}$ to $\mathtt{conat}$, the type of *conatural numbers*, which includes ω in addition to all the finite natural numbers.

The definition of logical equivalence must be correspondingly altered to account for the conatural numbers. 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 $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and vice versa; and (b) if $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$, and vice versa.

Corollary 48.17 can be proved for the conatural numbers by essentially the same argument as before.

The definition of logical equivalence at type $\mathtt{conat}$ is defined to be the *weakest* equivalence relation $\mathcal{E}$ between closed terms of type $\mathtt{conat}$ satisfying the following *consistency conditions*: If $e\ \mathcal{E}\ e' : \mathtt{conat}$, and

1. if $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and vice versa;
2. if $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$ with $e_1\ \mathcal{E}\ e_1' : \mathtt{conat}$, and vice versa.

It is immediate that if $e \sim_{\mathtt{conat}} e'$, then $e \simeq e'$, and so logical equivalence is consistent. It is also strict in that if e and e' are both divergent expressions of type $\mathtt{conat}$, then $e \sim_{\mathtt{conat}} e'$.

The principle of *proof by coinduction* states that to show $e \sim_{\mathtt{conat}} e'$, it suffices to exhibit a relation $\mathcal{E}$, such that

1. $e\ \mathcal{E}\ e' : \mathtt{conat}$, and
2. $\mathcal{E}$ satisfies the preceding consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in logical equivalence at type $\mathtt{conat}$, and hence $e \sim_{\mathtt{conat}} e'$, as required.

As an application of coinduction, let us consider the proof of Theorem 48.8. The overall argument remains as before, but the proof for the type $\mathtt{conat}$ must be altered as follows. Suppose that $\mathcal{A} \sim \mathcal{A}' : \rho \rightsquigarrow \mathtt{conat}$, and let $a = \mathcal{A}\{\mathtt{fix}\, x\!:\!\rho\, \mathtt{is}\, e\}$ and $a' = \mathcal{A}'\{\mathtt{fix}\, x\!:\!\rho\, \mathtt{is}\, e'\}$.

Writing $a^{(m)} = \mathcal{A}\{\mathtt{fix}^m\, x\!:\!\rho\ \mathtt{is}\ e\}$ and $a'^{(m)} = \mathcal{A}'\{\mathtt{fix}^m\, x\!:\!\rho\ \mathtt{is}\ e'\}$, assume that

$$\text{for every } m \geq 0,\ a^{(m)} \sim_{\mathtt{conat}} a'^{(m)}.$$

We are to show that

$$a \sim_{\mathtt{conat}} a'.$$

Define the functions p_n for $n \geq 0$ on closed terms of type `conat` by the following equations:

$$p_0(d) = d$$

$$p_{(n+1)}(d) = \begin{cases} d' & \text{if } p_n(d) \mapsto^* \mathtt{s}(d') \\ \textit{undefined} & \text{otherwise} \end{cases}.$$

For $n \geq 0$, let $a_n = p_n(a)$ and $a'_n = p_n(a')$. Correspondingly, let $a_n^{(m)} = p_n(a^{(m)})$ and $a_n'^{(m)} = p_n(a'^{(m)})$. Define $\mathcal{E}$ to be the strongest relation such that $a_n\ \mathcal{E}\ a'_n : \mathtt{conat}$ 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 $a\ \mathcal{E}\ a' : \mathtt{conat}$ (by construction), the result follows immediately.

To show that $\mathcal{E}$ is consistent, suppose that $a_n\ \mathcal{E}\ a'_n : \mathtt{conat}$ for some $n \geq 0$. We have by Corollary 48.17 that $a_n \simeq a_n^{(m)}$ for some $m \geq 0$, and hence, by the assumption, $a_n \simeq a_n'^{(m)}$, and so by Corollary 48.17 again, $a_n'^{(m)} \simeq a'_n$. Now if $a_n \mapsto^* \mathtt{s}(b_n)$, then $a_n^{(m)} \mapsto^* \mathtt{s}(b_n^{(m)})$ for some $b_n^{(m)}$, and hence there exists $b_n'^{(m)}$ such that $a_n'^{(m)} \mapsto^* b_n'^{(m)}$, and so there exists b'_n such that $a'_n \mapsto^* \mathtt{s}(b'_n)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n\ \mathcal{E}\ b'_n : \mathtt{conat}$ by construction, as required.

48.6 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.

49 Parametricity

The motivation for introducing polymorphism was to enable more programs to be written—those that are "generic" in one or more types, such as the composition function given in Chapter 20. Then if a program *does not* depend on the choice of types, we can code it by using polymorphism. Moreover, if we wish to insist that a program *cannot* depend on a choice of types, we demand that it be polymorphic. Thus polymorphism can be used both to expand the collection of programs we may write and also to limit the collection of programs that are permissible in a given context.

The restrictions imposed by polymorphic typing give rise to the experience that in a polymorphic functional language, if the types are correct, then the program is correct. Roughly speaking, if a function has a polymorphic type, then the strictures of type genericity vastly cut down the set of programs with that type. Thus if you have written a program with this type, it is quite likely to be the one you intended!

The technical foundation for these remarks is called *parametricity*. The goal of this chapter is to give an account of parametricity for $\mathcal{L}\{\to\forall\}$ under a call-by-name interpretation.

49.1 Overview

We begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value f has the type $\forall(t.t \to t)$. What function could it be? When instantiated at a type τ it should evaluate to a function g of type $\tau \to \tau$ that, when further applied to a value v of type τ returns a value v' of type τ. Because f is polymorphic, g cannot depend on v, so v' must be v. In other words, g must be the identity function at type τ and f must therefore be the *polymorphic identity*.

Suppose that f is a function of type $\forall(t.t)$. What function could it be? A moment's thought reveals that it cannot exist at all. For it must, when instantiated at a type τ, return a value of that type. But not every type has a value (including this one), so this is an impossible assignment. The only conclusion is that $\forall(t.t)$ is an *empty* type.

Let N be the type of polymorphic Church numerals introduced in Chapter 20, namely $\forall(t.t \to (t \to t) \to t)$. What are the values of this type? Given any type τ and values $z : \tau$ and $s : \tau \to \tau$, the expression

$$f[\tau](z)(s)$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ is by using the element z and the function s passed to it. A moment's thought reveals that the application must amount to the n-fold composition

$$s(s(\ldots s(z)\ldots)).$$

That is, the elements of N are in one-to-one correspondence with the natural numbers.

49.2 Observational Equivalence

The definition of observational equivalence given in Chapters 47 and 48 is based on identifying a type of *answers* that are observable outcomes of complete programs. Values of function type are not regarded as answers, but are treated as "black boxes" with no internal structure, only input–output behavior. In $\mathcal{L}\{\to\forall\}$, however, there are no (closed) base types. Every type is either a function type or a polymorphic type, and hence no types suitable to serve as observable answers.

One way to manage this difficulty is to augment $\mathcal{L}\{\to\forall\}$ with a base type of answers to serve as the observable outcomes of a computation. The only requirement is that this type have two elements that can be immediately distinguished from each other by evaluation. We may achieve this by enriching $\mathcal{L}\{\to\forall\}$ with a base type $\mathbf{2}$ containing two constants, $\mathbf{tt}$ and $\mathbf{ff}$, that serve as possible answers for a complete computation. A complete program is a closed expression of type $\mathbf{2}$.

Kleene equality is defined for complete programs by requiring that $e \simeq e'$ iff either (a) $e \mapsto^* \mathbf{tt}$ and $e' \mapsto^* \mathbf{tt}$ or (b) $e \mapsto^* \mathbf{ff}$ and $e' \mapsto^* \mathbf{ff}$. This is obviously an equivalence relation, and it is immediate that $\mathbf{tt} \not\simeq \mathbf{ff}$, because these are two distinct constants. As before, we say that a type-indexed family of equivalence relations between closed expressions of the same type is *consistent* if it implies Kleene equality at the answer type, $\mathbf{2}$.

To define observational equivalence, we must first define the concept of an expression context for $\mathcal{L}\{\to\forall\}$ as an expression with a "hole" in it. More precisely, we may give an inductive definition of the judgment

$$\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau'),$$

which states that $\mathcal{C}$ is an expression context that, when filled with an expression $\Delta; \Gamma \vdash e : \tau$ yields an expression $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} : \tau'$. (The precise definition of this judgment and the verification of its properties are left as an exercise for the reader.)

Definition 49.1. *Two expressions of the same type are* observationally equivalent*,* $\Delta; \Gamma \vdash e \cong e' : \tau$*, iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset; \emptyset \triangleright \mathbf{2})$.

Lemma 49.2. *Observational equivalence is the coarsest consistent congruence.*

Proof Essentially the same as the the proof of Theorem 47.6. □

Lemma 49.3.

1. *If* $\Delta, t; \Gamma \vdash e \cong e' : \tau$ *and* ρ *type, then* $\Delta; [\rho/t]\Gamma \vdash [\rho/t]e \cong [\rho/t]e' : [\rho/t]\tau$.
2. *If* $\emptyset; \Gamma, x : \rho \vdash e \cong e' : \tau$ *and* $d : \rho$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d/x]e' : \tau$. *Moreover, if* $d \cong_\rho d'$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d'/x]e : \tau$ *and* $\emptyset; \Gamma \vdash [d/x]e' \cong [d'/x]e' : \tau$.

Proof

1. Let $\mathcal{C} : (\Delta; [\rho/t]\Gamma \triangleright [\rho/t]\tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\rho/t]e\} \simeq \mathcal{C}\{[\rho/t]e'\}.$$

   Because $\mathcal{C}$ is closed, this is equivalent to

$$[\rho/t]\mathcal{C}\{e\} \simeq [\rho/t]\mathcal{C}\{e'\}.$$

   Let $\mathcal{C}'$ be the context $\Lambda\,(t\,.\,\mathcal{C}\{\circ\})\,[\rho]$ and observe that

$$\mathcal{C}' : (\Delta, t; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2}).$$

   Therefore, from the assumption,

$$\mathcal{C}'\{e\} \simeq \mathcal{C}'\{e'\}.$$

   But $\mathcal{C}'\{e\} \simeq [\rho/t]\mathcal{C}\{e\}$ and $\mathcal{C}'\{e'\} \simeq [\rho/t]\mathcal{C}\{e'\}$, from which the result follows.
2. By an argument essentially similar to that for Lemma 47.7. □

49.3 Logical Equivalence

This section introduces a form of logical equivalence that captures the informal concept of parametricity and also provides a characterization of observational equivalence. This will permit us to derive properties of observational equivalence of polymorphic programs of the kind suggested earlier.

The definition of logical equivalence for $\mathcal{L}\{\rightarrow\forall\}$ is somewhat more complex than for $\mathcal{L}\{\mathtt{nat}\rightarrow\}$. The main idea is to define logical equivalence for a polymorphic type, $\forall(t\,.\,\tau)$ to satisfy a very strong condition that captures the essence of parametricity. As a first approximation, we might say that two expressions e and e' of this type should be logically equivalent if they are logically equivalent for "all possible" interpretations of the type t. More precisely, we might require that $e\,[\rho]$ be related to $e'\,[\rho]$ at type $[\rho/t]\tau$, for any choice of type ρ. But this runs into two problems, one technical, the other conceptual. The same device used to solve both problems.

The technical problem stems from impredicativity. In Chapter 47 logical equivalence is defined by induction on the structure of types. But when polymorphism is impredicative,

the type $[\rho/t]\tau$ might well be larger than $\forall(t.\tau)$. At the very least we would have to justify the definition of logical equivalence on some other grounds, but no criterion appears to be available. The conceptual problem is that, even if we could make sense of the definition of logical equivalence, it would be too restrictive. For such a definition amounts to saying that the unknown type t is to be interpreted as logical equivalence at whatever type it turns out to be when instantiated. To obtain useful parametricity results, we ask for much more than this. What we do is to consider *separately* instances of e and e' by types ρ and ρ' and treat the type variable t as standing for *any relation* (of some form) between ρ and ρ'. We may suspect that this is asking too much: perhaps logical equivalence is the *empty* relation. Surprisingly, this is not the case, and indeed it is this very feature of the definition that we exploit to derive parametricity results about the language.

To manage both of these problems we consider a generalization of logical equivalence that is parameterized by a relational interpretation of the free type variables of its classifier. The parameters determine a separate binding for each free type variable in the classifier for each side of the equation, with the discrepancy being mediated by a specified relation between them. Thus related expressions need not have the same type, with the differences between them mediated by the given relation.

We restrict attention to a certain collection of "admissible" binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

Definition 49.4 (Admissibility)**.** *A relation R between expressions of types ρ and ρ' is* admissible*, written as $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:*

1. *Respect for observational equivalence: If $R(e, e')$ and $d \cong_\rho e$ and $d' \cong_{\rho'} e'$, then $R(d, d')$.*
2. *Closure under converse evaluation: If $R(e, e')$, then if $d \mapsto e$, then $R(d, e')$ and if $d' \mapsto e'$, then $R(e, d')$.*

Closure under converse evaluation will turn out to be a consequence of respect for observational equivalence, but we are not yet in a position to establish this fact.

The judgment $\delta : \Delta$ states that δ is a *type substitution* that assigns a closed type to each type variable $t \in \Delta$. A type substitution δ induces a substitution function $\hat{\delta}$ on types given by the equation

$$\hat{\delta}(\tau) = [\delta(t_1), \ldots, \delta(t_n)/t_1, \ldots, t_n]\tau,$$

and similarly for expressions. Substitution is extended to contexts pointwise by defining $\hat{\delta}(\Gamma)(x) = \hat{\delta}(\Gamma(x))$ for each $x \in dom(\Gamma)$.

Let δ and δ' be two type substitutions of closed types to the type variables in Δ. An *admissible relation assignment* η between δ and δ' is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ to each $t \in \Delta$. The judgment $\eta : \delta \leftrightarrow \delta'$ states that η is an admissible relation assignment between δ and δ'.

Logical equivalence is defined in terms of its generalization, called *parametric logical equivalence*, written as $e \sim_\tau e'$ $[\eta : \delta \leftrightarrow \delta']$ and defined as follows.

Definition 49.5 (Parametric Logical Equivalence)**.** *The relation* $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$ *is defined by induction on the structure of* τ *by the following conditions:*

$$
\begin{array}{lll}
e \sim_t e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e, e') \\
e \sim_{\mathbf{2}} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e \simeq e' \\
e \sim_{\tau_1 \to \tau_2} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim_{\tau_1} e_1' \ [\eta : \delta \leftrightarrow \delta'] \textit{ implies that} \\
 & & e\,(e_1) \sim_{\tau_2} e'\,(e_1') \ [\eta : \delta \leftrightarrow \delta'] \\
e \sim_{\forall\,(t\,.\,\tau)} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every } \rho,\ \rho',\ \textit{and every admissible } R : \rho \leftrightarrow \rho', \\
 & & e\,[\rho] \sim_\tau e'\,[\rho'] \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'].
\end{array}
$$

Logical equivalence is defined in terms of parametric logical equivalence by considering all possible interpretations of its free type and expression variables. An *expression substitution* γ for a context Γ, written as $\gamma : \Gamma$, is a substitution of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in dom(\Gamma)$. An expression substitution $\gamma : \Gamma$ induces a substitution function $\hat{\gamma}$, defined by the equation

$$\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]e,$$

where the domain of Γ consists of the variables $x_1, \ldots, x_n$. The relation $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$ is defined to hold iff $dom(\gamma) = dom(\gamma') = dom(\Gamma)$, and $\gamma(x) \sim_{\Gamma(x)} \gamma'(x) \ [\eta : \delta \leftrightarrow \delta']$ for every variable, x, in their common domain.

Definition 49.6 (Logical Equivalence)**.** *The expressions* $\Delta; \Gamma \vdash e : \tau$ *and* $\Delta; \Gamma \vdash e' : \tau$ *are* logically equivalent*, written as* $\Delta; \Gamma \vdash e \sim e' : \tau$ *iff, for every assignment* δ *and* δ' *of closed types to type variables in* Δ*, and every admissible relation assignment* $\eta : \delta \leftrightarrow \delta'$*, if* $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$*, then* $\hat{\gamma}(\hat{\delta}(e)) \sim_\tau \widehat{\gamma'}(\widehat{\delta'}(e')) \ [\eta : \delta \leftrightarrow \delta']$.

When e, e', and τ are closed, this definition states that $e \sim_\tau e'$ iff $e \sim_\tau e' \ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

Lemma 49.7 (Closure Under Converse Evaluation)**.** *Suppose that* $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$*. If* $d \mapsto e$*, then* $d \sim_\tau e'$*, and if* $d' \mapsto e'$*, then* $e \sim_\tau d'$.

Proof By induction on the structure of τ. When $\tau = t$, the result holds by the definition of admissibility. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications. □

Lemma 49.8 (Respect for Observational Equivalence)**.** *Suppose that* $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$*. If* $d \cong_{\hat{\delta}(\tau)} e$ *and* $d' \cong_{\widehat{\delta'}(\tau)} e'$*, then* $d \sim_\tau d' \ [\eta : \delta \leftrightarrow \delta']$.

Proof By induction on the structure of τ, relying on the definition of admissibility and the congruence property of observational equivalence. For example, if $\tau = \forall(t\,.\,\tau_2)$, then we are to show that for every admissible $R : \rho \leftrightarrow \rho'$,

$$d\,[\rho] \sim_{\tau_2} d'\,[\rho'] \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'].$$

Because observational equivalence is a congruence, we have that $d\,[\rho] \cong_{[\rho/t]\hat{\delta}(\tau_2)} e\,[\rho]$ and $d'\,[\rho'] \cong_{[\rho'/t]\widehat{\delta'}(\tau_2)} e'\,[\rho']$. It follows that

$$e\,[\rho] \sim_{\tau_2} e'\,[\rho']\ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'],$$

from which the result follows by induction. □

Corollary 49.9. *The relation $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\widehat{\delta'}(\tau)$.*

Proof By Lemmas 49.7 and 49.8. □

Corollary 49.10. *If $\Delta; \Gamma \vdash e \sim e' : \tau$, $\Delta; \Gamma \vdash d \cong e : \tau$, and $\Delta; \Gamma \vdash d' \cong e' : \tau$, then $\Delta; \Gamma \vdash d \sim d' : \tau$.*

Proof By Lemma 49.3 and Corollary 49.9. □

Lemma 49.11 (Compositionality)**.** *Let $R : \hat{\delta}(\rho) \leftrightarrow \widehat{\delta'}(\rho)$ be the relational interpretation of some type ρ, which is to say $R(d, d')$ holds iff $d \sim_\rho d'\ [\eta : \delta \leftrightarrow \delta']$. Then $e \sim_{[\rho/t]\tau} e'\ [\eta : \delta \leftrightarrow \delta']$ iff*

$$e \sim_\tau e'\ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \hat{\delta}(\rho) \leftrightarrow \delta' \otimes t \hookrightarrow \widehat{\delta'}(\rho)].$$

Proof By induction on the structure of τ. When $\tau = t$, the result is immediate from the choice of the relation R. When $\tau = t' \neq t$, the result follows directly from Definition 49.5. When $\tau = \tau_1 \to \tau_2$, the result follows by induction, using Definition 49.5. Similarly, when or $\tau = \forall(u\,.\,\tau_1)$, the result follows by induction, noting that we may assume, without loss of generality, that $u \neq t$ and $u \notin \rho$. □

Despite the strong conditions on polymorphic types, logical equivalence is not overly restrictive—every expression satisfies its constraints. This result is sometimes called the *parametricity theorem*.

Theorem 49.12 (Parametricity)**.** *If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma \vdash e \sim e : \tau$.*

Proof By rule induction on the statics of $\mathcal{L}\{\to\forall\}$ given by Rules (20.2).

We consider two representative cases here.

Rule (20.2d). Suppose $\delta : \Delta, \delta' : \Delta, \eta : \delta \leftrightarrow \delta'$, and $\gamma \sim_\Gamma \gamma'\ [\eta : \delta \leftrightarrow \delta']$. By induction we have that for all ρ, ρ' and admissible $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}\,(\hat{\delta}(e)) \sim_\tau [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e))\ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

where $\eta_* = \eta \otimes t \hookrightarrow R$, $\delta_* = \delta \otimes t \hookrightarrow \rho$, and $\delta'_* = \delta' \otimes t \hookrightarrow \rho'$. Because

$$\Lambda\,(t\,.\,\hat{\gamma}\,(\hat{\delta}(e)))\ [\rho] \mapsto^* [\rho/t]\hat{\gamma}\,(\hat{\delta}(e))$$

and

$$\Lambda\,(t\,.\,\widehat{\gamma}'(\widehat{\delta}'(e)))\,[\rho']\mapsto^*[\rho'/t]\widehat{\gamma}'(\widehat{\delta}'(e)),$$

the result follows by Lemma 49.7.

Rule (20.2e). Suppose $\delta:\Delta,\delta':\Delta,\eta:\delta\leftrightarrow\delta'$, and $\gamma\sim_\Gamma\gamma'\,[\eta:\delta\leftrightarrow\delta']$. By induction we have

$$\hat{\gamma}\,(\hat{\delta}(e))\sim_{\forall(t\,.\,\tau)}\widehat{\gamma}'(\widehat{\delta}'(e))\,[\eta:\delta\leftrightarrow\delta'].$$

Let $\hat{\rho}=\hat{\delta}(\rho)$ and $\hat{\rho}'=\widehat{\delta}'(\rho)$. Define the relation $R:\hat{\rho}\leftrightarrow\hat{\rho}'$ by $R(d,d')$ iff $d\sim_\rho d'\,[\eta:\delta\leftrightarrow\delta']$. By Corollary 49.9, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}\,(\hat{\delta}(e))\,[\hat{\rho}]\sim_\tau\widehat{\gamma}'(\widehat{\delta}'(e))\,[\hat{\rho}']\,[\eta\otimes t\hookrightarrow R:\delta\otimes t\hookrightarrow\hat{\rho}\leftrightarrow\delta'\otimes t\hookrightarrow\hat{\rho}'].$$

By Lemma 49.11

$$\hat{\gamma}\,(\hat{\delta}(e))\,[\hat{\rho}]\sim_{[\rho/t]\tau}\widehat{\gamma}'(\widehat{\delta}'(e))\,[\hat{\rho}']\,[\eta:\delta\leftrightarrow\delta'].$$

But

$$\begin{aligned}\hat{\gamma}\,(\hat{\delta}(e))\,[\hat{\rho}]&=\hat{\gamma}\,(\hat{\delta}(e))\,[\hat{\delta}(\rho)] &(49.1)\\ &=\hat{\gamma}\,(\hat{\delta}(e\,[\rho])), &(49.2)\end{aligned}$$

and similarly

$$\begin{aligned}\widehat{\gamma}'(\widehat{\delta}'(e))\,[\hat{\rho}']&=\widehat{\gamma}'(\widehat{\delta}'(e))\,[\widehat{\delta}'(\rho)] &(49.3)\\ &=\widehat{\gamma}'(\widehat{\delta}'(e\,[\rho])), &(49.4)\end{aligned}$$

from which the result follows. □

Corollary 49.13. *If* $\Delta;\Gamma\vdash e\cong e':\tau$, *then* $\Delta;\Gamma\vdash e\sim e':\tau$.

Proof By Theorem 49.12 $\Delta;\Gamma\vdash e\sim e:\tau$, and hence by Corollary 49.10, $\Delta;\Gamma\vdash e\sim e':\tau$. □

Lemma 49.14 (Congruence)**.** *If* $\Delta;\Gamma\vdash e\sim e':\tau$ *and* $\mathcal{C}:(\Delta;\Gamma\triangleright\tau)\rightsquigarrow(\Delta';\Gamma'\triangleright\tau')$, *then* $\Delta';\Gamma'\vdash\mathcal{C}\{e\}\sim\mathcal{C}\{e'\}:\tau'$.

Proof By induction on the structure of $\mathcal{C}$, following along very similar lines to the proof of Theorem 49.12. □

Lemma 49.15 (Consistency)**.** *logical equivalence is consistent.*

Proof Follows immediately from the definition of logical equivalence. □

Corollary 49.16. *If* $\Delta;\Gamma\vdash e\sim e':\tau$, *then* $\Delta;\Gamma\vdash e\cong e':\tau$.

Proof By Lemma 49.15 logical equivalence is consistent, and by Lemma 49.14, it is a congruence, and hence is contained in observational equivalence. □

Corollary 49.17. *Logical and observational equivalence coincide.*

Proof By Corollaries 49.13 and 49.16. □

If $d : \tau$ and $d \mapsto e$, then $d \sim_\tau e$, and hence by Corollary 49.16, $d \cong_\tau e$. Therefore if a relation respects observational equivalence, it must also be closed under converse evaluation. This shows that the second condition on admissibility is redundant, now that we have established the coincidence of logical and observational equivalence.

Corollary 49.18 (Extensionality)**.**

1. $e \cong_{\tau_1 \to \tau_2} e'$ *iff for all* $e_1 : \tau_1$, $e(e_1) \cong_{\tau_2} e'(e_1)$.
2. $e \cong_{\forall(t.\tau)} e'$ *iff for all* ρ, $e[\rho] \cong_{[\rho/t]\tau} e'[\rho]$.

Proof The forward direction is immediate in both cases, because observational equivalence is a congruence, by definition. The backward direction is proved similarly in both cases, by appeal to Theorem 49.12. In the first case, by Corollary 49.17 it suffices to show that $e \sim_{\tau_1 \to \tau_2} e'$. To this end suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By the assumption we have $e(e_1') \cong_{\tau_2} e'(e_1')$. By parametricity we have $e \sim_{\tau_1 \to \tau_2} e$, and hence $e(e_1) \sim_{\tau_2} e(e_1')$. The result then follows by Lemma 49.8. In the second case, by Corollary 49.17 it is sufficient to show that $e \sim_{\forall(t.\tau)} e'$. Suppose that $R : \rho \leftrightarrow \rho'$ for some closed types ρ and ρ'. It suffices to show that $e[\rho] \sim_\tau e'[\rho']\ [\eta : \delta \leftrightarrow \delta']$, where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$. By the assumption we have $e[\rho'] \cong_{[\rho'/t]\tau} e'[\rho']$. By parametricity $e \sim_{\forall(t.\tau)} e$, and hence $e[\rho] \sim_\tau e'[\rho']\ [\eta : \delta \leftrightarrow \delta']$. The result then follows by Lemma 49.8. □

Lemma 49.19 (Identity Extension)**.** *Let* $\eta : \delta \leftrightarrow \delta$ *be such that* $\eta(t)$ *is observational equivalence at type* $\delta(t)$ *for each* $t \in dom(\delta)$*. Then* $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta]$ *iff* $e \cong_{\hat{\delta}(\tau)} e'$.

Proof The backward direction follows immediately from Theorem 49.12 and respect for observational equivalence. The forward direction is proved by induction on the structure of τ, appealing to Corollary 49.18 to establish observational equivalence at function and polymorphic types. □

49.4 Parametricity Properties

The parametricity theorem enables us to deduce properties of expressions of $\mathcal{L}\{\to\forall\}$ that hold solely because of their type. The stringencies of parametricity ensure that a

polymorphic type has very few inhabitants. For example, we may prove that *every* expression of type $\forall(t.t \to t)$ behaves like the identity function.

Theorem 49.20. *Let $e : \forall(t.t \to t)$ be arbitrary, and let id be $\Lambda(t.\lambda\,(x:t)\,x)$. Then $e \cong_{\forall(t.t\to t)} id$.*

Proof By Corollary 49.17 it is sufficient to show that $e \sim_{\forall(t.t\to t)} id$. Let ρ and ρ' be arbitrary closed types, let $R : \rho \leftrightarrow \rho'$ be an admissible relation, and suppose that $e_0\ R\ e_0'$. We are to show that

$$e[\rho](e_0)\ R\ id[\rho](e_0'),$$

which, given the definition of id and closure under converse evaluation, is to say

$$e[\rho](e_0)\ R\ e_0'.$$

It suffices to show that $e[\rho](e_0) \cong_\rho e_0$, for then the result follows by the admissibility of R and the assumption $e_0\ R\ e_0'$.

By Theorem 49.12 we have $e \sim_{\forall(t.t\to t)} e$. Let the relation $S : \rho \leftrightarrow \rho$ be defined by $d\ S\ d'$ iff $d \cong_\rho e_0$ and $d' \cong_\rho e_0$. This is clearly admissible, and we have $e_0\ S\ e_0$. It follows that

$$e[\rho](e_0)\ S\ e[\rho](e_0),$$

and so, by the definition of the relation S, $e[\rho](e_0) \cong_\rho e_0$. □

In Chapter 20 we showed that product, sum, and natural numbers types are all definable in $\mathcal{L}\{\to\forall\}$. The proof of definability in each case consisted of showing that the type and its associated introduction and elimination forms are encodable in $\mathcal{L}\{\to\forall\}$. The encodings are correct in the (weak) sense that the dynamics of these constructs as given in the earlier chapters is derivable from the dynamics of $\mathcal{L}\{\to\forall\}$ via these definitions. By taking advantage of parametricity we may extend these results to obtain a strong correspondence between these types and their encodings.

As a first example, let us consider the representation of the unit type `unit` in $\mathcal{L}\{\to\forall\}$, as defined in Chapter 20 by the following equations:

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

It is easy to see that $\langle\rangle$: `unit` according to these definitions. But this merely says that the type `unit` is inhabited (has an element). What we would like to know is that, up to observational equivalence, the expression $\langle\rangle$ is the *only* element of that type. But this is precisely the content of Theorem 49.20. We say that the type `unit` is *strongly definable* within $\mathcal{L}\{\to\forall\}$.

Continuing in this vein, let us examine the definition of the binary product type in $\mathcal{L}\{\to\forall\}$, also given in Chapter 20:

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

It is easy to check that $\langle e_1, e_2\rangle \cdot \mathtt{l} \cong_{\tau_1} e_1$ and $\langle e_1, e_2\rangle \cdot \mathtt{r} \cong_{\tau_2} e_2$ by a direct calculation.

We wish to show that the ordered pair, as previously defined, is the unique such expression and hence that Cartesian products are strongly definable in $\mathcal{L}\{\to\forall\}$. We make use of a lemma governing the behavior of the elements of the product type whose proof relies on Theorem 49.12.

Lemma 49.21. *If $e : \tau_1 \times \tau_2$, then $e \cong_{\tau_1\times\tau_2} \langle e_1, e_2\rangle$ for some $e_1 : \tau_1$ and $e_2 : \tau_2$.*

Proof Expanding the definitions of pairing and the product type and applying Corollary 49.17, we let ρ and ρ' be arbitrary closed types and let $R : \rho \leftrightarrow \rho'$ be an admissible relation between them. Suppose further that

$$h \sim_{\tau_1\to\tau_2\to t} h'\ [\eta : \delta \leftrightarrow \delta'],$$

where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$ (and each is undefined on $t' \neq t$). We are to show that for some $e_1 : \tau_1$ and $e_2 : \tau_2$,

$$e\,[\rho]\,(h) \sim_t h'\,(e_1)\,(e_2)\ [\eta : \delta \leftrightarrow \delta'],$$

which is to say

$$e\,[\rho]\,(h)\ R\ h'\,(e_1)\,(e_2).$$

Now by Theorem 49.12 we have $e \sim_{\tau_1\times\tau_2} e$. Define the relation $S : \rho \leftrightarrow \rho'$ by $d\ S\ d'$ iff the following conditions are satisfied:

1. $d \cong_\rho h\,(d_1)\,(d_2)$ for some $d_1 : \tau_1$ and $d_2 : \tau_2$;
2. $d' \cong_{\rho'} h'\,(d_1')\,(d_2')$ for some $d_1' : \tau_1$ and $d_2' : \tau_2$;
3. $d\ R\ d'$.

This is clearly an admissible relation. Noting that

$$h \sim_{\tau_1\to\tau_2\to t} h'\ [\eta' : \delta \leftrightarrow \delta'],$$

where $\eta'(t) = S$ and $\eta'(t')$ is undefined for $t' \neq t$, we conclude that $e\,[\rho]\,(h)\ S\ e\,[\rho']\,(h')$, and hence

$$e\,[\rho]\,(h)\ R\ h'\,(d_1')\,(d_2'),$$

as required. □

Now suppose that $e : \tau_1 \times \tau_2$ is such that $e \cdot \mathtt{l} \cong_{\tau_1} e_1$ and $e \cdot \mathtt{r} \cong_{\tau_2} e_2$. We wish to show that $e \cong_{\tau_1 \times \tau_2} \langle e_1, e_2 \rangle$. From Lemma 49.21 it follows that $e \cong_{\tau_1 \times \tau_2} \langle e \cdot \mathtt{l}, e \cdot \mathtt{r} \rangle$ by congruence and direct calculation. Hence, by congruence we have $e \cong_{\tau_1 \times \tau_2} \langle e_1, e_2 \rangle$.

By a similar line of reasoning we may show that the Church encoding of the natural numbers given in Chapter 20 strongly defines the natural numbers in that the following properties hold:

1. $\mathtt{iter}\,\mathtt{z}\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\} \cong_\rho e_0$.
2. $\mathtt{iter}\,\mathtt{s}(e)\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\} \cong_\rho [\mathtt{iter}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}/x]e_1$.
3. Suppose that $x : \mathtt{nat} \vdash r(x) : \rho$. If
   a. $r(\mathtt{z}) \cong_\rho e_0$, and
   b. $r(\mathtt{s}(e)) \cong_\rho [r(e)/x]e_1$,
   then for every $e : \mathtt{nat}$, $r(e) \cong_\rho \mathtt{iter}\,e\,\{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$.

The first two equations, which constitute weak definability, are easily established by calculation, using the definitions given in Chapter 20. The third property, the unicity of the iterator, is proved using parametricity by showing that every closed expression of type $\mathtt{nat}$ is observationally equivalent to a numeral $\overline{n}$. We then argue for unicity of the iterator by mathematical induction on $n \geq 0$.

Lemma 49.22. *If $e : \mathtt{nat}$, then either $e \cong_{\mathtt{nat}} \mathtt{z}$, or there exists $e' : \mathtt{nat}$ such that $e \cong_{\mathtt{nat}} \mathtt{s}(e')$. Consequently, there exists $n \geq 0$ such that $e \cong_{\mathtt{nat}} \overline{n}$.*

Proof By Theorem 49.12 we have $e \sim_{\mathtt{nat}} e$. Define the relation $R : \mathtt{nat} \leftrightarrow \mathtt{nat}$ to be the strongest relation such that $d\ R\ d'$ iff either $d \cong_{\mathtt{nat}} \mathtt{z}$ and $d' \cong_{\mathtt{nat}} \mathtt{z}$, or $d \cong_{\mathtt{nat}} \mathtt{s}(d_1)$, $d' \cong_{\mathtt{nat}} \mathtt{s}(d_1')$, and $d_1\ R\ d_1'$. It is easy to see that $\mathtt{z}\ R\ \mathtt{z}$, and if $e\ R\ e'$, then $\mathtt{s}(e)\ R\ \mathtt{s}(e')$. Letting $\mathtt{zero} = \mathtt{z}$ and $\mathtt{succ} = \lambda\,(x\!:\!\mathtt{nat})\,\mathtt{s}(x)$, we have

$$e[\mathtt{nat}](\mathtt{zero})(\mathtt{succ})\ R\ e[\mathtt{nat}](\mathtt{zero})(\mathtt{succ}).$$

The result follows by the induction principle arising from the definition of R as the strongest relation satisfying its defining conditions. □

49.5 Representation Independence, Revisited

In Section 21.4 we discussed the property of *representation independence* for abstract types. This property states that if two implementations of an abstract type are "similar," then the client behavior is not affected by replacing one for the other. The crux of the matter is the definition of similarity of two implementations. Informally, two implementations of an abstract type are similar if there is a relation R between their representation types that is *preserved* by the operations of the type. The relation R may be thought of as expressing the "equivalence" of the two representations; checking that each operation preserves R amounts

to checking that the result of performing that operation on equivalent representations yields equivalent results.

As an example, we argued informally in Section 21.4 that two implementations of a queue abstraction are similar. The two representations of queues are related by a relation R such that $q\ R\ (b, f)$ iff q is b followed by the reversal of f. We then argued that the operations preserve this relationship and then claimed, without proof, that the behavior of the client would not be disrupted by changing one implementation to the other.

The proof of this claim relies on parametricity, as may be seen by considering the definability of existential types in $\mathcal{L}\{\rightarrow\forall\}$ given in Section 21.3. According to that definition, the client e of an abstract type $\exists(t.\tau)$ is a polymorphic function of type $\forall(t.\tau \rightarrow \tau_2)$, where τ_2, the result type of the computation, does not involve the type variable t. Being polymorphic, the client enjoys the parametricity property given by Theorem 49.12. Specifically, suppose that ρ_1 and ρ_2 are two closed representation types and that $R : \rho_1 \leftrightarrow \rho_2$ is an admissible relation between them. For example, in the case of the queue abstraction, ρ_1 is the type of lists of elements of the queue, ρ_2 is the type of a pair of lists of elements, and R is the relation previously given. Suppose further that $e_1 : [\rho_1/t]\tau$ and $e_2 : [\rho_2/t]\tau$ are two implementations of the operations such that

$$e_1 \sim_\tau e_2\ [\eta : \delta_1 \leftrightarrow \delta_2], \tag{49.5}$$

where $\eta(t) = R$, $\delta_1(t) = \rho_1$ and $\delta_2(t) = \rho_2$. In the case of the queues example, the expression e_1 is the implementation of the queue operations in terms of lists, and the e_2 is the implementation in terms of pairs of lists described earlier. Condition (49.5) states that the two implementations are similar in that they preserve the relation R between the representation types. By Theorem 49.12 it follows that the client e satisfies

$$e \sim_{\tau_2} e\ [\eta : \delta_1 \leftrightarrow \delta_2].$$

But because τ_2 is a closed type (in particular, does not involve t), this is equivalent to

$$e \sim_{\tau_2} e\ [\emptyset : \emptyset \leftrightarrow \emptyset].$$

But then by Lemma 49.19 we have

$$e[\rho_1](e_1) \cong_{\tau_2} e[\rho_2](e_2).$$

That is, the client behavior is not affected by the change of representation.

49.6 Notes

The concept of parametricity is latent in the proof of normalization for System **F** (Girard, 1972). The work of Reynolds (1983), though technically flawed because of its reliance on a (nonexistent) set-theoretic model of polymorphism, emphasizes the centrality of

logical equivalence for characterizing equality of polymorphic programs. The application of parametricity to representation independence was suggested by Reynolds and developed for existential types by Mitchell (1986) and Pitts (1998). The extension of System **F** with a positive (in the sense of Chapter 38) observable type appears to be needed to even define observational equivalence, but this point seems not to have been made elsewhere in the literature.

50 Process Equivalence

As the name implies, a process is an ongoing computation that may interact with other processes by sending and receiving messages. From this point of view a concurrent computation has no definite "final outcome" but rather affords an opportunity for interaction that may well continue indefinitely. The notion of equivalence of processes must therefore be based on their potential for interaction, rather than on the "answer" that they may compute. Let P and Q be such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc. We say that P and Q are *equivalent*, written as $P \approx_\Sigma Q$, iff there is a bisimulation $\mathcal{R}$ such that $P\,\mathcal{R}_\Sigma\,Q$. A family of relations $\mathcal{R} = \{\mathcal{R}_\Sigma\}_\Sigma$ is a *bisimulation* iff whenever P may evolve to P' taking the action α, then Q may also evolve to some process Q' taking the same action such that $P'\,\mathcal{R}_\Sigma\,Q'$, and, conversely, if Q may evolve to Q' taking action α, then P may evolve to P' taking the same action, and $P'\,\mathcal{R}_\Sigma\,Q'$. This captures the idea that the two processes afford the same opportunities for interaction in that they each simulate each other's behavior with respect to their ability to interact with their environment.

50.1 Process Calculus

We consider a process calculus that consolidates the main ideas explored in Chapters 41 and 42. We assume as given an ambient language of expressions that includes the type `clsfd` of classified values (see Chapter 34). Channels are treated as dynamically generated classes with which to build messages, as described in Chapter 42.

The syntax of the process calculus is given by the following grammar:

Sort			Abstract Form	Concrete Form	Description
Proc	P	::=	`stop`	$\mathbf{1}$	inert
			`par`$(P_1; P_2)$	$P_1 \parallel P_2$	composition
			`await`(E)	$\$\,E$	synchronize
			`new[`τ`]`$(a\,.\,P)$	$\nu\, a{\sim}\tau\,.\,P$	allocation
			`emit`(e)	$!\,e$	broadcast
Evt	E	::=	`null`	$\mathbf{0}$	null
			`or`$(E_1; E_2)$	$E_1 + E_2$	choice
			`acc`$(x\,.\,P)$	$?\,(x\,.\,P)$	acceptance

The statics is given by the judgments $\Gamma \vdash_\Sigma P$ proc and $\Gamma \vdash_\Sigma E$ event defined by the following rules. We assume as given a judgment $\Gamma \vdash_\Sigma e : \tau$ for τ a type including

the type `clsfd` of classified values:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{50.1a}$$

$$\frac{\Gamma \vdash_\Sigma P_1\ \mathsf{proc} \quad \Gamma \vdash_\Sigma P_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{50.1b}$$

$$\frac{\Gamma \vdash_\Sigma E\ \mathsf{event}}{\Gamma \vdash_\Sigma \$E\ \mathsf{proc}} \tag{50.1c}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} P\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a\sim\tau\,.\,P\ \mathsf{proc}} \tag{50.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma !e\ \mathsf{proc}} \tag{50.1e}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{50.1f}$$

$$\frac{\Gamma \vdash_\Sigma E_1\ \mathsf{event} \quad \Gamma \vdash_\Sigma E_2\ \mathsf{event}}{\Gamma \vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{50.1g}$$

$$\frac{\Gamma, x : \mathtt{clsfd} \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma ?(x\,.\,P)\ \mathsf{event}}\,. \tag{50.1h}$$

The dynamics is given by the judgments $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ and $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, defined as in Chapter 41. We assume as given the judgments $e \underset{\Sigma}{\mapsto} e'$ and $e\ \mathsf{val}_\Sigma$ for expressions. Processes and events are identified up to structural congruence, as described in Chapter 41:

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2} \tag{50.2a}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \quad P_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} P_2'}{P_1 \parallel P_2 \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2'} \tag{50.2b}$$

$$\frac{E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P}{\$E \overset{\alpha}{\underset{\Sigma}{\mapsto}} P} \tag{50.2c}$$

$$\frac{P \overset{\alpha}{\underset{\Sigma, a\sim\tau}{\longmapsto}} P' \quad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\, a\sim\tau\,.\,P \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a\sim\tau\,.\,P'} \tag{50.2d}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \mathtt{clsfd}}{!e \overset{e!}{\underset{\Sigma}{\mapsto}} \mathbf{1}} \tag{50.2e}$$

$$\frac{E_1 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P}{E_1 + E_2 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P} \tag{50.2f}$$

$$\frac{e\ \mathsf{val}_\Sigma}{?\,(x\,.\,P) \overset{e\,?}{\underset{\Sigma}{\Rightarrow}} [e/x]P}\,. \tag{50.2g}$$

Assuming that substitution is valid for expressions, it is also valid for processes and events.

Lemma 50.1.

1. *If* $\Gamma, x : \tau \vdash_\Sigma P$ *proc and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]P$ *proc.*
2. *If* $\Gamma, x : \tau \vdash_\Sigma E$ *event and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]E$ *event.*

Transitions preserve well-formedness of processes and events.

Lemma 50.2.

1. *If* $\vdash_\Sigma P$ *proc and* $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$, *then* $\vdash_\Sigma P'$ *proc.*
2. *If* $\vdash_\Sigma E$ *event and* $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, *then* $\vdash_\Sigma P$ *proc.*

50.2 Strong Equivalence

Bisimilarity makes precise the informal idea that two processes are equivalent if they each can take the same actions and, in doing so, evolve into equivalent processes. A *process relation* $\mathcal{P}$ is a family $\{\mathcal{P}_\Sigma\}$ of binary relations between processes P and Q such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc, and an *event relation* $\mathcal{E}$ is a family $\{\mathcal{E}_\Sigma\}$ of binary relations between events E and F such that $\vdash_\Sigma E$ event and $\vdash_\Sigma F$ event. A *(strong) bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation $\mathcal{P}$ and an event relation $\mathcal{E}$ satisfying the following conditions:

1. If $P\ \mathcal{P}_\Sigma\ Q$, then
   a. if $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$, there exists Q' such that $Q \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and
   b. if $Q \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$, there exists P' such that $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$.
2. If $E\ \mathcal{E}_\Sigma\ F$, then
   a. if $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, then there exists Q such that $F \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$ with $P\ \mathcal{P}_\Sigma\ Q$ and
   b. if $F \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$, then there exists P such that $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$ with $P\ \mathcal{P}_\Sigma\ Q$.

The qualifier "strong" refers to the fact that the action, α, in the conditions on being a bisimulation include the silent action, ε. (In Section 50.3 we discuss another notion of bisimulation in which the silent actions are treated specially.)

(Strong) equivalence is the pair $(\approx, \approx)$ of process and event relations such that $P \approx_\Sigma Q$ and $E \approx_\Sigma F$ iff there exists a strong bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P\ \mathcal{P}_\Sigma\ Q$, and $E\ \mathcal{E}_\Sigma\ F$.

Lemma 50.3. *Strong equivalence is a strong bisimulation.*

Proof Follows immediately from the definition. □

The definition of strong equivalence gives rise to the principle of *proof by coinduction*. To show that $P \approx_\Sigma Q$, it is enough to give a bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P\ \mathcal{P}_\Sigma\ Q$ (and similarly for events). An instance of coinduction that arises fairly frequently is to choose $(\mathcal{P}, \mathcal{E})$ to be $(\approx \cup \mathcal{P}_0, \approx \cup \mathcal{E}_0)$ for some $\mathcal{P}_0$ and $\mathcal{E}_0$ such that $P\ \mathcal{P}_0\ Q$ and show that this expansion is a bisimulation. Because strong equivalence is itself a bisimulation, this reduces to show that if $P'\ \mathcal{P}_0\ Q'$ and $P' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P''$, then $Q' \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q''$ for some Q'' such that either $P'' \approx_\Sigma Q''$ or $P''\ \mathcal{P}_0\ Q''$ (and analogously for transitions from Q' and similarly for event transitions). This proof method amounts to *assuming what we are trying to prove* and showing that this assumption is tenable. The proof that the expanded relation is a bisimulation may make use of the assumptions $\mathcal{P}_0$ and $\mathcal{E}_0$; in this sense "circular reasoning" is a perfectly valid method of proof.

Lemma 50.4. *Strong equivalence is an equivalence relation.*

Proof For reflexivity and symmetry, it suffices to note that the identity relation is a bisimulation, as is the converse of a bisimulation. For transitivity we need that the composition of two bisimulations is again a bisimulation, which follows directly from the definition. □

It remains to verify that strong equivalence is a congruence, which means that each of the process- and event-forming constructs respects strong equivalence. To show this we require the *open extension* of strong equivalence to processes and events with free variables. The relation $\Gamma \vdash_\Sigma P \approx Q$ is defined for processes P and Q such that $\Gamma \vdash_\Sigma P\ \mathsf{proc}$ and $\Gamma \vdash_\Sigma Q\ \mathsf{proc}$ to mean that $\hat{\gamma}(P) \approx_\Sigma \hat{\gamma}(Q)$ for every substitution γ of closed values of appropriate type for the variables Γ.

Lemma 50.5. *If* $\Gamma, x : \mathtt{clsfd} \vdash_\Sigma P \approx Q$, *then* $\Gamma \vdash_\Sigma ?\,(x\,.\,P) \approx ?\,(x\,.\,Q)$.

Proof Fix a closing substitution γ for Γ, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. By assumption we have $x : \mathtt{clsfd} \vdash_\Sigma \hat{P} \approx \hat{Q}$. We are to show that $?\,(x\,.\,\hat{P}) \approx_\Sigma ?\,(x\,.\,\hat{Q})$. The proof is by coinduction, taking $\mathcal{P} = \approx$ and $\mathcal{E} = \approx \cup\, \mathcal{E}_0$, where

$$\mathcal{E}_0 = \{(?\,(x\,.\,P'), ?\,(x\,.\,Q')) \mid x : \mathtt{clsfd} \vdash_\Sigma P' \approx Q'\}.$$

Clearly $?\,(x\,.\,\hat{P})\ \mathcal{E}_0\ ?\,(x\,.\,\hat{Q})$. Suppose that $?\,(x\,.\,P')\ \mathcal{E}_0\ ?\,(x\,.\,Q')$. By inspection of Rules (50.2), if $?\,(x\,.\,P') \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P''$, then $\alpha = v\,?$ and $P'' = [v/x]P'$ for some v val_Σ such

that $\vdash_\Sigma v : \mathtt{clsfd}$. But $?(x\,.\,Q') \overset{v?}{\underset{\Sigma}{\Rightarrow}} [v/x]Q'$, and we have that $[v/x]P' \approx_\Sigma [v/x]Q'$ by the definition of $\mathcal{E}_0$, and hence $[v/x]P'\ \mathcal{E}_0\ [v/x]Q'$, as required. The symmetric case follows symmetrically, completing the proof. □

Lemma 50.6. *If* $\Gamma \vdash_{\Sigma,a\sim\tau} P \approx Q$, *then* $\Gamma \vdash_\Sigma \nu\, a\sim\tau\,.\,P \approx \nu\, a\sim\tau\,.\,Q$.

Proof Fix a closing value substitution γ for Γ, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. Assuming that $\hat{P} \approx_{\Sigma,a\sim\tau} \hat{Q}$, we are to show that $\nu\, a\sim\tau\,.\,\hat{P} \approx_\Sigma \nu\, a\sim\tau\,.\,\hat{Q}$. The proof is by coinduction, taking $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\,(\nu\, a\sim\tau\,.\,P', \nu\, a\sim\tau\,.\,Q') \mid P' \approx_{\Sigma,a\sim\tau} Q'\,\}.$$

Clearly $\nu\, a\sim\tau\,.\,\hat{P}\ \ \mathcal{P}_0\ \ \nu\, a\sim\tau\,.\,\hat{Q}$. Suppose that $\nu\, a\sim\tau\,.\,P'\ \ \mathcal{P}_0\ \ \nu\, a\sim\tau\,.\,Q'$, and that $\nu\, a\sim\tau\,.\,P' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P''$. By inspection of Rules (50.2), we see that $\vdash_\Sigma \alpha$ action and that $P'' = \nu\, a\sim\tau\,.\,P'''$ for some P''' such that $P' \xrightarrow[\Sigma,a\sim\tau]{\alpha} P'''$. But by definition of $\mathcal{P}_0$ we have $P' \approx_{\Sigma,a\sim\tau} Q'$, and hence $Q' \xrightarrow[\Sigma,a\sim\tau]{\alpha} Q'''$ with $P''' \approx_{\Sigma,a\sim\tau} Q'''$. Letting $Q'' = \nu\, a\sim\tau\,.\,Q'$, we have that $\nu\, a\sim\tau\,.\,Q' \xrightarrow[\Sigma,a\sim\tau]{\alpha} Q''$ and by definition of $\mathcal{P}_0$ we have $P''\ \mathcal{P}_0\ Q''$, as required. The symmetric case is proved symmetrically, completing the proof. □

Lemmas 50.5 and 50.6 capture two different cases of binding, the former of variables and the latter of classes. The hypothesis of Lemma 50.5 relates all substitutions for the variable x in the recipient processes, whereas the hypothesis of Lemma 50.6 relates the constituent processes schematically in the class name a. This makes all the difference, for if we were to consider all substitution instances of a class name by another class name, then a class would no longer be "new" within its scope, because we could identify it with an "old" class by substitution. On the other hand we *must* consider substitution instances for variables, because the meaning of a variable is given in such terms. This shows that classes and variables must be distinct concepts. (See Chapter 34 for an example of what goes wrong when the two concepts are confused.)

Lemma 50.7. *If* $\Gamma \vdash_\Sigma P_1 \approx Q_1$ *and* $\Gamma \vdash_\Sigma P_2 \approx Q_2$, *then* $\Gamma \vdash_\Sigma P_1 \parallel P_2 \approx Q_1 \parallel Q_2$.

Proof Let γ be a closing value substitution for Γ, and let $\hat{P}_i = \hat{\gamma}(P_i)$ and $\hat{Q}_i = \hat{\gamma}(Q_i)$ for $i = 1, 2$. The proof is by coinduction, considering the relation $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\,(P_1' \parallel P_2', Q_1' \parallel Q_2') \mid P_1' \approx_\Sigma Q_1' \text{ and } P_2' \approx_\Sigma Q_2'\,\}.$$

Suppose that $P_1' \parallel P_2'\ \mathcal{P}_0\ Q_1' \parallel Q_2'$, and that $P_1' \parallel P_2' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P''$. There are two cases to consider, the interesting one being Rule (50.2b). In this case we have $P'' = P_1'' \parallel P_2''$ with $P_1' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1''$ and $P_2' \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} P_2''$. By definition of $\mathcal{P}_0$ we have that $Q_1' \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q_1''$ and $Q_2' \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} Q_2''$ with $P_1'' \approx_\Sigma Q_1''$ and $P_2'' \approx_\Sigma Q_2''$. Letting $Q'' = Q_1'' \parallel Q_2''$, we have that $P''\ \mathcal{P}_0\ Q''$, as required. The symmetric case is handled symmetrically, and Rule (50.2a) is handled similarly. □

Lemma 50.8. *If* $\Gamma \vdash_\Sigma E_1 \approx F_1$ *and* $\Gamma \vdash_\Sigma E_2 \approx F_2$, *then* $\Gamma \vdash_\Sigma E_1 + E_2 \approx F_1 + F_2$.

Proof Follows immediately from Rules (50.2) and the definition of bisimulation. □

Lemma 50.9. *If* $\Gamma \vdash_\Sigma E \approx F$*, then* $\Gamma \vdash_\Sigma \$\,E \approx \$\,F$.

Proof Follows immediately from Rules (50.2) and the definition of bisimulation. □

Lemma 50.10. *If* $\Gamma \vdash_\Sigma d \cong e : \texttt{clsfd}$*, then* $\Gamma \vdash_\Sigma !\,d \approx !\,e$.

Proof The process calculus introduces no new observations on expressions, so that d and e remain indistinguishable as actions. □

Theorem 50.11. *Strong equivalence is a congruence.*

Proof Follows immediately from the preceding lemmas, which cover each case separately. □

50.3 Weak Equivalence

Strong equivalence expresses the idea that two processes are equivalent if they simulate each other step-by-step. Every action taken by one process is matched by a corresponding action taken by the other. This seems natural for the nontrivial actions $e\,!$ and $e\,?$, but is arguably overly restrictive for the silent action, ε. Silent actions correspond to the actual steps of computation, whereas the send and receive actions express the potential to interact with another process. Silent steps are therefore of a very different flavor than the other forms of action and therefore might usefully be treated differently from them. Weak equivalence seeks to do just that.

Silent actions arise within the process calculus itself (when two processes communicate), but they play an even more important role when the dynamics of expressions is considered explicitly (as in Chapter 42). For then each step $e \underset{\Sigma}{\mapsto} e'$ of evaluation of an expression corresponds to a silent transition for any process in which it is embedded. In particular, $!\,e \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} !\,e'$ whenever $e \underset{\Sigma}{\mapsto} e'$. We may also consider atomic processes of the form $\texttt{proc}(m)$ consisting of a command to be executed in accordance with the rules of some underlying dynamics. Here again we would expect that each step of command execution induces a silent transition from one atomic process to another.

From the point of view of equivalence, it therefore seems sensible to allow that a silent action by one process may be mimicked by one or more silent actions by another. For example, there appears to be little to be gained by distinguishing, say, $\texttt{proc(ret 3+4)}$ from $\texttt{proc(ret (1+2)+(2+2))}$ merely because the latter takes more steps to compute the same value than the former! The purpose of weak equivalence is precisely to disregard such trivial distinctions by allowing a transition to be matched by a matching transition, possibly preceded by any number of silent transitions.

A *weak bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation $\mathcal{P}$ and an event relation $\mathcal{E}$ satisfying the following conditions:

1. If $P\ \mathcal{P}_\Sigma\ Q$, and
   a. if $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$, where $\alpha \neq \varepsilon$, then there exists Q'' and Q' such that $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}{}^{*}\ Q'' \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} P'$, then $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}{}^{*}\ Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;
   b. if $Q \overset{\alpha}{\underset{\Sigma}{\mapsto}} Q'$, where $\alpha \neq \varepsilon$, then there exists P'' and P' such that $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}{}^{*}\ P'' \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $Q \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} Q'$, then $P \overset{\varepsilon}{\underset{\Sigma}{\mapsto}}{}^{*}\ P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;
2. If $E\ \mathcal{E}_\Sigma\ F$, and
   a. if $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, then there exists Q such that $F \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$ with $P\ \mathcal{P}_\Sigma\ Q$, and
   b. if $F \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$, then there exists P such that $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$ with $P\ \mathcal{P}_\Sigma\ Q$.

(The conditions on the event relation are the same as for strong bisimilarity because there are, in this calculus, no silent actions on events.)

Weak equivalence is the pair $(\sim, \sim)$ of process and event relations defined by $P \sim_\Sigma Q$ and $E \sim_\Sigma F$ iff there exists a weak bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P\ \mathcal{P}_\Sigma\ Q$, and $E\ \mathcal{E}_\Sigma\ F$. The open extension of weak equivalence, written as $\Gamma \vdash_\Sigma P \sim Q$ and $\Gamma \vdash_\Sigma E \sim F$, is defined exactly as is the open extension of strong equivalence.

Theorem 50.12. *Weak equivalence is an equivalence relation and a congruence.*

Proof The proof proceeds along similar lines to that of Theorem 50.11. □

50.4 Notes

The literature on process equivalence is extensive. Numerous variations have been considered for an equally numerous array of formalisms. Milner (1999) recounts the history and development of the concept of bisimilarity in his monograph on the π-calculus, crediting David Park with its original conception (Park, 1981). The development in this chapter is inspired by the work of Milner and by a proof of congruence of strong bisimilarity given by Bernardo Toninho for the process calculus considered in Chapter 41.

PART XIX

Appendix

Appendix: Finite Sets and Finite Functions

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 ω of natural numbers. A set X is *countable* iff there is a bijection, $f : X \cong \omega$, 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 $n \in \omega$, 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 $u \sim v$ 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 $u_1 \sim v_1, \ldots, u_n \sim v_n$, for the finite function

$$u_1 \hookrightarrow v_1 \otimes \ldots \otimes u_n \hookrightarrow v_n.$$

Bibliography

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag.

Abadi, M. and C. Fournet (2001). Mobile values, new names, and secure communication. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 104–15, Association for Computing Machinery.

Aczel, P. (1977). An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pp. 783–818. North-Holland.

Allen. J. (1978). *The Anatomy of LISP*. Computer Science Series. McGraw-Hill.

Allen, S. (1987). A non-type-theoretic definition of Martin-Löf's types. In *LICS*, pp. 215–21. IEEE Computer Society.

Allen, S. F., M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran (2005). Innovations in computational type theory using NuPRL. *J. Appl. Logic*, 4: 428–69.

Andreoli, J.-M. (1992). Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2: 297–347.

Ariola, Z. M. and M. Felleisen (1997). The call-by-need lambda calculus. *J. Funct. Prog.*, 7: 265–301.

Arvind, R., S. Nikhil, and K. Pingali (1986). I-structures: Data structures for parallel computing. In J. H. Fasel and R. M. Keller, editors, *Graph Reduction*, Volume 279 of Lecture Notes in Computer Science Series, pp. 336–69. Springer.

Avron, A. (1991). Simple consequence relations. *Inf. Comput.*, 92:105–39.

Barendregt, H. (1984). *The Lambda Calculus, Its Syntax and Semantics*. Volume 103 of Studies in Logic and the Foundations of Mathematics. North-Holland.

Barendregt, H. (1992). Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E Maibaum, editors, *Handbook of Logic in Computer Science*, Volume 2 of Computational Structures Series. Oxford University Press.

Bertot, Y., G. Huet, J.-J. Lévy, and G. Plotkin (editors) (2009). *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*. Cambridge University Press.

Blelloch, G. E. (1990). *Vector Models for Data-Parallel Computing*. MIT Pres.

Blelloch, G. E. and J. Greiner (1995). Parallelism in sequential functional languages. In *FPCA*, pp. 226–37.

Blelloch, G. E. and J. Greiner (1996). A provable time and space efficient implementation of NESL. In *ICFP*, pp. 213–25.

Brookes, S. D. (2002). The essence of parallel Algol. *Inf. Comput.*, 179(1): 118–49.

Burstall, R. M., D. B. MacQueen, and D. Sannella (1980). Hope: An experimental applicative language. In *LISP Conference*, pp. 136–43.

Buss, S. R. (editor). (1998). *Handbook of Proof Theory*. Elsevier.

Cardelli, L. (1988). Structural subtyping and the notion of power type. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 70–9. Association for Computing Machinery.

Cardelli, L. (1997). Program fragments, linking, and modularization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 266–77. Association for Computing Machinery.

Castagna, G. and B. C. Pierce (1994). Decidable bounded quantification. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 151–62. Association for Computing Machinery.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press.

Colmerauer, A. and P. Roussel (1993). The birth of Prolog. In *Proceedings of the Second ACM SIGPLAN Conference on the History of Programming Languages, HOPL-II*, pp. 37–52. Association for Computing Machinery.

Constable, R. L. (1986). *Implementing Mathematics with the Nupri Proof Development System*. Prentice-Hall.

Constable, R. L. (1998). Types in logic, mathematics, and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X. Elsevier.

Constable, R. L. and S. F. Smith (1987). Partial objects in constructive type theory. In *LICS*, pp. 183–93. IEEE Computer Society.

Cook, W. R. (2009). On understanding data abstraction, revisited. In *OOPSLA*, pp. 557–72.

Cousineau, G. and M. Mauny (1998). *The Functional Approach to Programming*. Cambridge University Press.

Crary, K. (2009). A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '09*, pp. 21–29. Association for Computing Machinary.

Dreyer, D. (2005). *Understanding and Evolving the ML Module System*. Ph.D. dissertation. Carnegie Mellon University.

Engberg, U. and M. Nielsen. (2000). A calculus of communicating systems with label passing – ten years after. In G. D. Plotkin, C. Stirling, and M. Tofte (editors), *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pp. 599–622. MIT Press.

Felleisen, M. and R. Hieb (1992). The revised report on the syntactic theories of sequential control and state. *TCS: Theoretical Computer Science*, 103.

Gelernter, D. (1985). Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7: 80–112.

Gentzen, G. (1969). Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pp. 68–213. North-Holland.

Girard, J.-Y. (1972). *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*. These d'etat, Universite Paris VII.

Girard, J.-Y. (1989). *Proofs and Types*. Translated by P. Taylor and Y. Lafont. Cambridge University Press.

Gödel, K. (1980). On a hitherto unexploited extension of the finitary standpoint. Translated by W. Hodges and B. Watson. *Philos. Logic*, 9: 133–42.

Gordon, M. J., A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, Volume 78 of Lecture Notes in Computer Science Series. Springer-Verlag.

Greiner, J. and G. E. Blelloch (1999). A provably time-efficient parallel implementation of full speculation. *ACM Trans. Prog. Lang. Syst.*, 21: 240–85.

Griffin, T. (1990). A formulae-as-types notion of control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 47–58. Association for Computing Machinery.

Halstead, R. H. Jr. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7: 501–38.

Harper, R. (1992). Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14: 71–84.

Harper, R. (1994). A simplified account of polymorphic references. *Inf. Process. Lett.*, 51: 201–6.

Harper, R. and M. Lillibridge (1994). A type-theoretic approach to higher order modules with sharing. In *Proceecings of the ACM Symposium on Principles of Programming Languages*, pp. 123–37. Association for Computing Machinery.

Harper, R., J. C. Mitchell, and E. Moggi (1990). Higher-order modules and the phase distinction. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 341–54. Association for Computing Machinery.

Harper, R., F. Honsell, and G. Plotkin (1993). A framework for defining logics. *J. Assoc. Comput. Mach.*, 40: 194–204.

Hinze, R. and J. Jeuring (2003). Generic Haskell: practice and theory. In R. C. Backhouse and J. Gibbons, editors, *Generic Programming*, Volume 2793 of *Lecture Notes in Computer Science*, pp. 1–56. Springer.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21: 666–77.

Hoare, T. (2009). Null references: the billion dollar mistake. Paper presented at QCon.

Jones, S. L. P. (2003). Haskell 98: Introduction. *J. Funct. Program.*, 13: 0–6.

Kleene, S. C. (1952). *Introduction to Metamathematics*. van Nostrand.

Kowalski, R. A. (1988). The early years of logic programming. *Commun. ACM*, 31: 38–43.

Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

Landin, P. J. (1965). A correspondence between Algol 60 and Church's lambda notation. *Commun. ACM*, 8: 89–101; 158–65.

Lee, D. K., K. Crary, and R. Harper (2007). Towards a mechanized metatheory of Standard ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 173–84. Association for Computing Machinery.

Leroy, X. (1994). Manifest types, modules, and separate compilation. In *Pro. ACM Symposium on Principles of Programming Languages*, pp. 109–22, 1994.

Leroy, X. (1995). Applicative functors and fully transparent higher-order modules. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 142–53. Association for Computing Machinery.

Licata, D. R. and R. Harper (2010). A monadic formalization of ML5. In K. Crary and M. Miculan, editors, *Logical Frameworks and Metalanguages: Theory and Practice*, Volume 34 of *EPTCS* Series, pp. 69–83. Association for Computing Mechinery.

Lillibridge, M. (1997). *Translucent Sums: A Foundation for Higher-Order Module Systems*. Ph.D. dissertation, Carnegie Mellon University School of Computer Science.

Liskov, B. and J. M. Wing (1994). A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16: 1811–41.

MacLane, S. (1998). *Categories for the Working Mathematician* (2nd ed). Graduate Texts in Mathematics Series. Springer-Verlag.

MacQueen, D. B. (1986). Using dependent types to express modular structure. In *Proceedings of the ACM Symposium on Principles of Programming Languges*, pp. 277–86. Association for Computing Machinery.

MacQueen, D. B. (2009). Kahn networks at the dawn of functional programming. In Bertot et al., editors, *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*, chapter 5. Cambridge University Press.

Martin-Löf, P. (1980). Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science IV*, pp. 153–75. North-Holland.

Martin-Löf, P. (1983). On the meanings of the logical constants and the justifications of the logical laws. Unpublished lecture notes.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Studies in Proof Theory Series. Bibliopolis.

Martin-Löf, P. (1987). Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73: 407–20.

McCarthy, J. (1965). *LISP1.5 Programmer's Manual*. MIT Press.

Mendler, N. P. (1987). Recursive types and type constraints in second-order lambda calculus. In *LICS*, pp. 30–6. IEEE Computer Society.

Milner, R. (1978). A theory of type polymorphism in programming. *JCSS*, 17: 348–75.

Milner, R. (1999). *Communicating and Mobile Systems – the Pi-Calculus*. Cambridge University Press.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.

Mitchell, J. C. (1984). Coercion and type inference. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 175–85. Association for Computing Machinery.

Mitchell, J. C. (1986). Representation independence and data abstraction. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 263–76. Association for Computing Machinery.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. MIT Press.

Mitchell, J. C. and G. D. Plotkin (1988). Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10: 470–502.

Murphy, T. VII, K. Crary, R. Harper, and F. Pfenning (2004). A symmetric modal lambda calculus for distributed computing. In *LICS*, pp. 286–95. IEEE Computer Society.

Murthy, C. R. (1991). An evaluation semantics for classical proofs. In *LICS*, pp. 96–107. IEEE Computer Society.

Nanevski, A. (2003). From dynamic binding to state via modal possibility. In *PPDP*, pp. 207–18. Association for Computing Machinery.

Nederpelt, R. P., J. H. Geuvers, and R. C. deVrijer (editors) (1994). *Selected Papers on Automath*, Volume 133 of *Studies in Logic and the Foundations of Mathematics Series*. North-Holland.

Ocaml (2012). Available at http://caml.inria.fr/ocaml/.

Park, D. M. R. (1981). Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, Volume 104 of Lecture Notes in Computer Science Series, pp. 167–83. Springer.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pierce, B. C. (2004). *Advanced Topics in Types and Programming Languages*. MIT Press.

Pitts, A. M. (1998). Existential types: Logical relations and operational equivalence. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP*, Volume 1443 of Lecture Notes in Computer Science Series, pp. 309–26. Springer.

Pitts, A. M. (2000). Operational semantics and program equivalence. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM*, Volume 2395 of Lecture Notes in Computer Science Series, pp. 378–412. Springer.

Pitts, A. M. and I. D. B. Stark (1993). Observable properties of higher order functions that dynamically create local names, or what's new? In A. M. Borzyszkowski and S. Sokolowski, editors, *MFCS*, Volume 711 of Lecture Notes in Computer Science Series, pp. 122–141. Springer.

Plotkin, G. D. (1977). LCF considered as a programming language. *Theor. Comput. Sci.*, 5: 223–55.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMIFN-19, Aarhus University, Computer Science Department.

Plotkin, G. D. (2004). The origins of structural operational semantics. *J. Logic Algebraic Program.*, 60: 3–15.

Reppy, J. H. (1999). *Concurrent Programming in ML*. Cambridge University Press.

Reynolds, J. C. (1974). Towards a theory of type structure. In B. Robinet, editor, *Symposium on Programming*, Volume 19 of Lecture Notes in Computer Science Series, pp. 408–23. Springer.

Reynolds, J. C. (1980). Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, Volume 94 of Lecture Notes in Computer Science Series, pp. 211–58. Springer.

Reynolds, J. C. (1981). The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pp. 345–72. North-Holland.

Reynolds, J. C. (1983). Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pp. 513–23. North-Holland.

Reynolds, J. C. (1993). The discoveries of continuations. *Lisp Symbol. Comput.*, 6: 233–48.

Reynolds, J. C. (1998). *Theories of Programming Languages*. Cambridge University Press, Cambridge, England.

Rossberg, A., C. V. Russo, and D. Dreyer (2010). F-ing modules. In A. Kennedy and N. Benton, editors, *TLDI*, pp. 89–102. ACM.

Scott, D. (1980). Lambda calculus: Some models, some philosophy. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pp. 223–65. North-Holland, Amsterdam.

Scott, D. S. (1976). Data types as lattices. *SIAM J. Comput.*, 5(3): 522–87.

Scott, D. S. (1982). Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *ICALP*, volume 140 of Lecture Notes in Computer Science, pp. 577–613. Springer.

Smyth, M. B. and G. D. Plotkin (1982). The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4): 761–83.

Statman, R. (1985). Logical relations and the typed lambda-calculus. *Information and Control*, 65(2/3): 85–97.

Steele, G. L. (1990). *Common Lisp: The Language* (2nd ed.). Digital Press.

Stone, C. A. and R. Harper (2006). Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7: 676–722.

Taylor, P. (1999). *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press.

Turner, D. (1987). An overview of Miranda. *Bull. EATCS*, 33: 103–14.

Wadler, P. (1989). Theorems for free! In *FPCA*, pp. 347–59.

Wadler, P. (2003). Call-by-value is dual to call-by-name. In C. Runciman, and O. Shivers, editors, *ICFP*, pp. 189–201. Association for Computing Machinery.

Wand, M. (1979). Fixed-point constructions in order-enriched categories. *Theor. Comput. Sci.*, 8: 13–30.

Watkins, K., I. Cervesato, F. Pfenning, and D. Walker (2008). Specifying properties of concurrent computations in clf. *Electr. Notes Theor. Comput. Sci.*, 199: 67–87.

Wright, A. K. and M. Felleisen (1994). A syntactic approach to type soundness. *Inf. Comput.*, 115: 38–94.

Zeilberger, N. (2008). On the unity of duality. *Ann. Pure Appl. Logic*, 153: 66–96.

Index