Chapter 2

The Language PCF

2.1 Introduction

This chapter presents a language for *Programming Computable Functions* called PCF, originally formulated by Dana Scott in an influential unpublished manuscript [Sco69]. It is a typed functional language based on lambda calculus. The language is designed to be easily analyzed, rather than as a practical language for writing large programs. However, with certain extensions to the surface syntax, it is possible to write many functional programs in a comfortable style. The presentation of PCF here is informal in the sense that we will discuss the constructs of the language, and the axiomatic, operational and denotational semantics, without going into the proofs of basic theorems. The main topics of the chapter are:

- Introduction to syntax and semantics of typed lambda calculus and related languages by example.
- Treatment of recursive definitions by fixed-point operators.
- Discussion of axiomatic, operational and denotational semantics, with a summary of the relationships between them.
- Demonstration that basic programming methods may be carried out in a simplified functional language.
- Study of expressive power and limitations of the language using operational semantics.

The main goals are to develop a feel for the programming capabilities of lambda-calculus-based languages and summarize general properties and techniques that apply to a variety of languages. Operational semantics are considered in more depth than denotational semantics, since later chapters focus on the denotational and axiomatic semantics. We discuss "programming techniques" at some length, to give some intuition for the way that familiar programming constructs may be represented in lambda calculus and prove both positive and negative results about the expressiveness of PCF. The chapter concludes with a brief overview of extensions and variations of PCF that have either practical or theoretical significance. The technical theorems about PCF that are not proved in this chapter are proved in Chapters 5 and 8, as special cases of more general results.

For those familiar with denotational semantics, we point out several differences between the use of lambda calculus in this book and the traditional use of lambda calculus in denotational semantics. One is typing. The meta-language in standard texts on denotational semantics, such

as [Gor79, MS76, Sch86, Sto77], is not explicitly typed. In contrast, we use only typed lambda calculus. This clarifies the kind of value each expression defines, and simplifies the technical analysis in several ways. Another difference is that we regard PCF itself as a language for writing programs, rather than solely as a meta-language for giving semantics to other languages. One reason for this approach is to develop some intuition for the expressive power of typed lambda calculus. Another is to suggest that lambda calculus may be used not only for denotational semantics, but for studying operational and pragmatic issues in programming language analysis and design.

The evaluation order used in PCF is *lazy* or *left-most.* This is the evaluation order used in the pure functional programming languages Haskell [HF92, H$^+$92] and Miranda [Tur85]. (See [Pey87] for information on the implementation of lazy functional languages.) In contrast, the languages Lisp [McC60, McC78, Ste84] and ML [Mil85b, MTH90, MT91, Ull94] use an *eager* form of evaluation. While most practical languages use eager evaluation, there are some advantages (as well as disadvantages) to developing a basic theory using lazy evaluation. To a first approximation, the choice between evaluation orders is a matter of taste. Since we can simulate eager reduction in a lazy language, and conversely [Plo75], the reduction systems are theoretically equivalent. The denotational semantics are also interdefinable, although there are some simplifications in the eager case. On the other hand, the axiomatic semantics of eager languages seem more complicated. The main reasons we prefer to study lazy PCF in this book are that this gives the simplest correspondence between axiomatic, operational, and denotational semantics, and also the most flexibility in operational semantics since we have equivalent deterministic, nondeterministic and parallel forms of program execution. The theory is also older and more fully developed. Since similar techniques apply in both cases, the ideas described in this book are useful for analyzing both forms of evaluation. A general setting for considering both evaluation orders is the extension of PCF with lifted types, described in Section 2.6.4.

2.2 Syntax of PCF

2.2.1 Overview

Every expression of PCF has a unique type. Therefore, we may summarize the constructs of PCF by listing the types of the language. The basic values are natural numbers and booleans (truth values *true* and *false*), which have types *nat* and *bool*, respectively. PCF also has pairs, which belong to cartesian product types, and functions, belonging to function types. The PCF notation for the cartesian product of types σ and τ is $\sigma \times \tau$. For example, the type of natural number pairs is written $nat \times nat$. The type of a function with domain σ and range τ is written $\sigma \to \tau$. Some notational conventions regarding type expressions are that $\to$ associates to the right, and $\times$ has higher precedence than $\to$. Thus $\sigma \to \tau \to \rho$ is parenthesized as $\sigma \to (\tau \to \rho)$, and $\sigma \times \tau \to \rho$ as $(\sigma \times \tau) \to \rho$.

One property of PCF is that only expressions that satisfy certain typing constraints are actually considered part of the language. For example, although PCF has addition, the expression $true + 1$ is not considered well formed. (It does not make sense to add a truth value to a natural number.) With variables, the typing conditions depend on the context in which an expression is used. For example, $x+5$ only makes sense if the variable x is declared to have type *nat*. A precise description of PCF, using typing rules and assumptions about the types of variables, is given using typing rules in Chapter 4. In the following informal presentation, we assume we have infinitely many variables of each type and that we can tell what type each variable has. When we write $\lambda x{:}\sigma$, it is implicitly assumed that x is a variable of type σ. When we need to refer to the types of a variable that are

not lambda bound, we will simply write typing assumptions $x_1{:}\,\sigma_1, \ldots, x_k{:}\,\sigma_k$ in parentheses or in the text.

A general syntactic issue that many readers will be familiar with is the distinction between object notation and meta-notation. The syntax of PCF, like other languages we will consider, is defined using some set *Var* of variables, some set *Cst* of constant symbols, and other classes of symbols. We do not need to be concerned with what the elements of the set *Var* actually are, as long as we have infinitely many of them (so we don't run out) and we can tell whether two are distinct or the same. We say that the symbols and expressions of PCF belong to the *object language* since PCF is the object of study. In studying PCF, it is convenient to use additional symbols to stand for arbitrary symbols and expressions of the object language. These are said to be symbols of the *meta-language,* the language we use in our study of the object language.

We use letters $x, y, z, \ldots$, possibly with subscripts, primes or superscripts, as meta-variables for arbitrary variables of PCF (elements of *Var*) and letters $M, N, P, \ldots$, again possibly with subscripts, primes or superscripts, as meta-variables for expressions of PCF. It is possible for two metavariables, x and y, to stand for the same object variable. We use the symbol $\equiv$ for syntactic equality of object expressions, writing $x \equiv y$ if x and y stand for the same variable of PCF and $x \not\equiv y$ to indicate that they are distinct variables. If we say, "let x and y be variables" or "let M and N be terms," then these could be distinct variables or expressions, or syntactically identical.

2.2.2 Booleans and natural numbers

The basic boolean expressions are the constants *true* and *false*, and the boolean-valued conditional expressions

$$\texttt{if } \langle \textit{boolean} \rangle \texttt{ then } \langle \textit{boolean} \rangle \texttt{ else } \langle \textit{boolean} \rangle.$$

The basic natural number expressions include *numerals*

$$0, 1, 2, 3, \ldots,$$

the usual symbols for natural numbers, and addition, written $+$. Thus if M and N are natural number expressions, so is $M + N$. We can also compute natural numbers using conditional tests,

$$\texttt{if } \langle \textit{boolean} \rangle \texttt{ then } \langle \textit{natural_number} \rangle \texttt{ else } \langle \textit{natural_number} \rangle$$

and compare natural numbers for equality. The equality test *Eq*? on natural numbers returns a boolean value. For example, *Eq*? 3 0 has the boolean value *false*, since 3 is different from 0, but *Eq*? 5 5 = *true*.

The typing rules of PCF prevent conditional expressions such as

$$\texttt{if } \langle \textit{boolean} \rangle \texttt{ then } 3 \texttt{ else } \textit{true},$$

with one case returning a boolean and the other a natural number. In general, PCF allows any conditional expression $\texttt{if } \langle \textit{boolean} \rangle \texttt{ then } M \texttt{ else } N$, provided alternatives M and N have the same type. This allows us to choose between numeric functions by writing

$$\texttt{if } \langle \textit{boolean} \rangle \texttt{ then } \lambda x{:}\,\textit{nat}.M \texttt{ else } \lambda x{:}\,\textit{nat}.N,$$

for example.

To summarize the expressions we have considered so far, the basic natural number and boolean expressions are given by the following productions. While we have not discussed arbitrary expressions of type σ, we include conditional expressions below since conditional is most naturally considered a boolean operation.

$$\begin{array}{lcl}
\langle \mathit{bool_exp} \rangle & ::= & \langle \mathit{bool_var} \rangle \mid \mathit{true} \mid \mathit{false} \mid \mathit{Eq?}\, \langle \mathit{nat_exp} \rangle\, \langle \mathit{nat_exp} \rangle \mid \\
& & \texttt{if}\ \langle \mathit{bool_exp} \rangle\ \texttt{then}\ \langle \mathit{bool_exp} \rangle\ \texttt{else}\ \langle \mathit{bool_exp} \rangle \\
\\
\langle \mathit{nat_exp} \rangle & ::= & \langle \mathit{nat_var} \rangle \mid 0 \mid 1 \mid 2 \mid \ldots \mid \langle \mathit{nat_exp} \rangle + \langle \mathit{nat_exp} \rangle \mid \\
& & \texttt{if}\ \langle \mathit{bool_exp} \rangle\ \texttt{then}\ \langle \mathit{nat_exp} \rangle\ \texttt{else}\ \langle \mathit{nat_exp} \rangle \\
\\
\langle \sigma\mathit{_exp} \rangle & ::= & \ldots \mid \texttt{if}\ \langle \mathit{bool_exp} \rangle\ \texttt{then}\ \langle \sigma\mathit{_exp} \rangle\ \texttt{else}\ \langle \sigma\mathit{_exp} \rangle
\end{array}$$

Since both natural numbers and booleans may also be computed by function calls, these are not *all* of the natural number and boolean expressions of PCF. The additional forms will presented in subsequent sections discussing product types, function types and recursive definitions.

The equational axioms for natural number and boolean expressions are straightforward. We have an infinite collection of basic axioms

$$0+0=0,\ 0+1=1,\ \ldots,\ 1+0=1,\ 1+1=2,\ \ldots$$

for addition, giving us all true equations of the form $n+m=p$, for numerals n, m and p, and two axiom schemes for conditional expressions of each type

$$\begin{array}{lcl}
\texttt{if}\ \mathit{true}\ \texttt{then}\ M\ \texttt{else}\ N & = & M, \\
\texttt{if}\ \mathit{false}\ \texttt{then}\ M\ \texttt{else}\ N & = & N.
\end{array}$$

There are infinitely many axioms for equality test, determined as follows

$$\begin{array}{lcll}
\mathit{Eq?}\, n\, n & = & \mathit{true}, & \text{each numeral } n, \\
\mathit{Eq?}\, m\, n & = & \mathit{false}, & m, n \text{ distinct numerals.}
\end{array}$$

The operational semantics of natural number and boolean expressions are defined using a set of *reduction axioms,* each obtained by reading one of the equational axioms from left to right. In lambda calculus, these are usually called reduction rules, since they are "rules" telling you how to reduce a term. However, when we axiomatize the reduction relation, the axioms are the basic reduction "rules" obtained by reading an equational axiom from left to right. Some useful terminology is that a term matching the left-hand side of a reduction axiom is called a *redex.*

In the operational semantics (reduction system), we may evaluate an expression by applying reduction axioms to any subexpression. To see how this works, consider the expression

$$\texttt{if}\ \mathit{Eq?}\, (6+5)\, 17\ \texttt{then}\ (1+1)\ \texttt{else}\ 27$$

We cannot simplify the conditional without first producing a boolean constant *true* or *false*. This in turn requires numerals for both arguments to *Eq?*, so we begin by applying the reduction rule $6+5 \to 11$. This gives us the expression

$$\texttt{if}\ \mathit{Eq?}\, 11\, 17\ \texttt{then}\ (1+1)\ \texttt{else}\ 27$$

which is simplified using a reduction rule for *Eq?* to

$$\texttt{if}\ \mathit{false}\ \texttt{then}\ (1+1)\ \texttt{else}\ 27$$

Finally, one of the rules for conditional applies, and we produce the numeral 27. In order to simplify this expression, we needed to evaluate the test before simplifying the conditional. However, it was not necessary to simplify the number expression $1 + 1$ since this is discarded by the conditional. Since we may choose to reduce any subterm at any point, we could have simplified $1+1 \to 2$ between any two of the reduction steps given. With the exception of this "nondeterministic choice," the steps involved mimic the action of any ordinary interpreter fairly closely. We will discuss connections between nondeterministic and deterministic reduction in Section 2.3.4.

The reader may have noticed that we do *not* have the equational axiom $Eq?\,M\,M = true$, for arbitrary natural-number expression M. One reason is that the reduction axiom $Eq?\,M\,M \to true$ would lead to a non-confluent reduction system for PCF (see Section 2.3.4). In reducing an expression $Eq?\,M\,N$, we must first simplify M and N to numerals. This corresponds to actual implementations, where a test for natural number equality involves evaluating both expressions.

The denotational semantics of the basic natural number and boolean expressions falls under the general pattern of algebraic terms and their interpretation in a multi-sorted algebra, which is covered in detail in Chapter 3. To give a hint of things to come, we will summarize some of the main ideas here. We assign a "mathematical meaning," or *denotation* to every natural number and boolean expression by first choosing a set N of natural-number values and a set B of boolean values. In the standard semantics of natural-number and boolean expressions alone, these would be the usual set of natural numbers and the usual set of booleans. However, in the context of PCF, we will also include an extra element in each set to account for the fact that PCF has partial recursive functions on natural numbers, in addition to total functions, and similarly for booleans. The extra element arises from the fact that we may treat a partial function from N to N as a total function from N to $N \cup \{\infty\}$, as is sometimes done in recursive function theory.

Once we have chosen a set of values for each type, we can give meaning to expressions by choosing a value for each free variable. Then, using induction on expressions, we specify a unique mathematical value (element of the appropriate set) for each expression. In the standard interpretation of natural numbers and boolean expressions, $2 + 3$ has its usual value, 5, and so on, so there is nothing very surprising in this case. Since the denotational semantics of basic data types are a special case of first-order semantics (or model theory), the basic idea is probably familiar from logic. The main reason for mentioning the denotational semantics at this point is to contrast this with the axiomatic semantics (equational proof system) and operational semantics (reduction rules), which are entirely syntactic.

Definable functions

Even without lambda abstraction, we can think of basic PCF natural-number expressions as defining numeric functions. To describe this precisely, we need to distinguish between the natural numbers and the symbols (called *numerals)* used in PCF. If $n \in \mathcal{N}$ is a natural number, then one of the symbols $0, 1, 2, \ldots$ of PCF is the numeral for n, *i.e.*, the symbol we usually use to write n on a piece of paper. For any $n \in \mathcal{N}$, let us write $\lceil n \rceil$ for this numeral. Using this notation, we say an expression $M\colon nat$ with natural number variable x *defines a numeric function* $f\colon \mathcal{N} \to \mathcal{N}$ *implicitly* if, for every natural number $n \in \mathcal{N}$, we have $[\lceil n \rceil / x]M \twoheadrightarrow \lceil f(n) \rceil$. In words, we can simplify the expression $[\lceil n \rceil / x]M$, with the numeral for n in place of x, to the numeral for the function value $f(n)$.

Example 2.2.1 The numeric function $f(n) = n + 5$ is defined implicitly by the expression $x + 5$, where x ranges over natural numbers. ∎

The notion of implicit definition may be extended to other types. A boolean expression M: *bool* with boolean variable x *defines a boolean function* f *implicitly* if $[true/x]M \twoheadrightarrow \lceil f(true)\rceil$ and $[false/x]M \twoheadrightarrow \lceil f(false)\rceil$. It is easy to see that negation is defined implicitly by the expression `if` x `then` *false* `else` *true*. Binary functions may also be defined implicitly, as in Exercises 2.2.2 and 2.2.3.

With recursion, we will be able to define all computable numeric functions in PCF. However, using only the basic natural-number and boolean expressions given in this section, there are many functions that cannot be defined. One example is given in Exercise 2.2.3.

Exercise 2.2.2 Write expressions with boolean variables x and y which implicitly define the conjunction $(x \wedge y)$ and disjunction $(x \vee y)$ of x and y.

Exercise 2.2.3 Show that the exponentiation function is not implicitly definable using PCF natural number and boolean expressions. In other words, prove that there is no basic natural number expression M with numeric variables x and y such that $[\lceil n\rceil/x]([\lceil m\rceil/y]M)$ has value $\lceil n^m\rceil$.

2.2.3 Pairing and functions

In PCF, we can form ordered pairs and functions of any type. If M and N are any PCF expressions, then $\langle M, N\rangle$ is the ordered pair whose first component has the same value as M and second component has the same value as N. Every PCF pair has a cartesian product type which is determined by the following simple rule: if M has type σ and N has type τ, then the pair $\langle M, N\rangle$ has type $\sigma \times \tau$. For example, $\langle 3, 5\rangle$ is an ordered pair of natural numbers, with $\langle 3, 5\rangle : nat \times nat$, and the "nested pair" $\langle 3, \langle 5, 7\rangle\rangle$ has type $nat \times (nat \times nat)$.

In addition to forming pairs, we can also "take pairs apart" using projection operations. The projection operations $\mathbf{Proj}_1$ and $\mathbf{Proj}_2$ return the first and second components of a pair. This is formalized in the two equational axioms

$$(proj) \qquad \mathbf{Proj}_1\langle M, N\rangle = M, \quad \mathbf{Proj}_2\langle M, N\rangle = N.$$

The final axiom for pairing is based on the idea that two pairs with the same first and second components must be equal (*i.e.*, the same pair). Since any P: $\sigma \times \tau$ is a pair, we can form a pair $\langle(\mathbf{Proj}_1 P), (\mathbf{Proj}_2 P)\rangle$ from the first and second components of P. This must be the same as P, since it has the same components. This gives us the equational axiom

$$(sp) \qquad \langle(\mathbf{Proj}_1 P), (\mathbf{Proj}_2 P)\rangle = P.$$

One point about the (sp) axiom, called *surjective pairing,* is that this is redundant for explicit pair terms of the form $\langle M, N\rangle$. This is easily demonstrated by substituting $\langle M, N\rangle$ for P and applying equational axioms to subterms:

$$\begin{aligned}&\langle(\mathbf{Proj}_1\langle M, N\rangle), (\mathbf{Proj}_2\langle M, N\rangle)\rangle\\ &= \langle M, (\mathbf{Proj}_2\langle M, N\rangle)\rangle\\ &= \langle M, N\rangle\end{aligned}$$

However, if x: $\sigma \times \tau$ is a variable of product type, then we cannot prove $\langle(\mathbf{Proj}_1 x), (\mathbf{Proj}_2 x)\rangle = x$ without the axiom that gives us this equation directly. This is demonstrated in Example 2.4.3, which appears in a later section since it depends on properties of confluent reduction systems. The reason the axiom is called surjective pairing is that it implies that the pairing function is surjective

onto the set of elements of product type. A consequence of (sp) is given in part (a) of Exercise 2.2.7.

As for natural number and boolean expressions, the reduction rules for pairs are derived by reading the equational axioms from left to right. However, we only include reduction rules corresponding to the $(proj)$ axioms. There is no (sp) reduction rule in the operational semantics of PCF for two reasons. The first is that it is not necessary, as demonstrated by the adequacy of the operational semantics without (sp), discussed in Sections 2.3 and 5.4.1. The second is that the rule causes confluence to fail, when combined with recursive definitions of functions (see Section 4.4.3).

Functions

PCF functions are written using lambda abstraction, and most of the examples given in Section 1.2 are PCF expressions. For example, since we have numerals and addition in PCF, both $\lambda x{:}\, nat.\, x+1$ and $\lambda x{:}\, nat.\, 5$ are acceptable PCF, with type $nat \to nat$. In general, a PCF function may have any PCF type as domain or range. Since we can have functions that take functions as arguments and return functions as results, PCF is more flexible (in this direction) than many programming languages. A simple higher-order function is composition of numeric functions

$$comp \stackrel{def}{=} \lambda f{:}\, nat \to nat.\, \lambda g{:}\, nat \to nat.\, \lambda x{:}\, nat.\, f(gx).$$

To see how this works, suppose f and g are numeric functions. Then $comp\ f\ g$ is the function $\lambda x{:}\, nat.\, f(gx)$; the function which, on argument x, returns $f(gx)$. Thus $comp\ f\ g$ defines $f \circ g$.

As mentioned in Section 1.3, we have the equational axiom

$$(\alpha) \qquad \lambda x{:}\, \sigma.M = \lambda y{:}\, \sigma.[y/x]M, \quad y \text{ not free in } M$$

for renaming bound variables. This axiom is related only to the fact that λ is a binding operator, and does not have anything to do with the way that lambda abstraction defines a function. Since renaming of bound variables is so basic, it is useful to have a special name for it: we may write $M =_\alpha N$ if terms M and N differ only in the names of bound variables. It is often convenient to write $FV(M)$ for the free variables of M, *i.e.*, the variables appearing in M that are not bound by λ.

The second axiom,

$$(\beta) \qquad (\lambda x{:}\, \sigma.M)N = [N/x]M,$$

gives us a way of computing the result of function application by substitution. To avoid any confusion about substitution and bound variables, we define substitution on typed lambda expressions precisely, by the following induction, with three subcases for different forms of lambda abstractions:

$$\begin{array}{rcl}
[N/x]x & = & N \\
[N/x]a & = & a, \text{ for constant of variable } a \neq x \\
[N/x](PQ) & = & ([N/x]P)([N/x]Q) \\
[N/x]\lambda x{:}\, \sigma.M & = & \lambda x{:}\, \sigma.M \\
[N/x]\lambda y{:}\, \sigma.M & = & \lambda y{:}\, \sigma.[N/x]M, \text{ provided } x \neq y \text{ and } y \notin FV(N) \\
[N/x]\lambda y{:}\, \sigma.M & = & (\lambda z{:}\, \sigma.[N/x][z/y]M), \text{ where } z \notin FV(MN) \text{ and } \ y, z \neq x
\end{array}$$

While most of these clauses are easily understood, it may be worth saying a few more words about the last three lines. The fourth line makes sense when we remember that a substitution $[N/x]$

only replaces *free* occurrences of x by N. Since x is bound in $\lambda x{:}\sigma.M$, the substitution $[N/x]$ has no effect on this term. The last two clauses tell how to substitute for a free variable inside the binding operator λ. If y is not free in the term N we are inserting into M, then we may go ahead and perform the substitution. This is the intent of the fifth line of the definition. However, if y is free in N, then we have a conflict and we must rename y to some other variable z to avoid capturing the free y in N. The option is provided by the last line of the definition. Since substitution may involve arbitrary renaming, substitution is only defined up to α-equivalence. In other words, if we compute $[N/x]M$ by this definition, we may end up with any one of an infinite number of possible terms. However, all of the possibilities are $=_\alpha$ to each other. Since we consider α-equivalent terms as essentially the same, and they are provably equal in the simplest possible way, this "nondeterminism" in the definition of substitution is completely harmless. It is easy to see that substitution respects α-equivalence, in the sense that $[N/x]M =_\alpha [N/x]P$ whenever $M =_\alpha P$.

Substitution is extended to PCF by adding cases for addition, conditional, etc. These are all straightforward, and follow the pattern of the application case above: we substitute $[N/x]$ in a compound expression by applying this substitution to each of its parts, *e.g.*, $[N/x](P + Q) = ([N/x]P) + ([N/x]Q)$. Exercise 2.2.6, which may be useful to remember, asks you to prove the following identities involving substitution.

$$\begin{array}{lcl} [M/x]([N/x]P) & = & [([M/x]N)/x]P, \\ [M/x]([N/y]P) & = & [([M/x]N)/y]([M/x]P) \text{ if } y \not\equiv x \text{ and } y \notin FV(M) \end{array}$$

The combination of α-conversion and renaming bound variables in substitution provide *static scope* in PCF, as illustrated in Exercise 2.2.13.

The final equational axiom for functions is based on the idea that two functions are equal if they produce equal values on all arguments. To see how this arises, suppose $M{:}\,\sigma \to \tau$ is a function expression, and $x{:}\,\sigma$ is some variable not appearing in M. The condition on x is to make sure that we do not use x in two different ways. Then since $(Mx){:}\,\tau$, the expression $\lambda x{:}\,\sigma.(Mx)$ also defines a function from σ to τ. For any argument $N{:}\,\sigma$, we can apply either function to N. However, we get the same result in either case, since

$$(\lambda x{:}\,\sigma.Mx)N = MN$$

by (β). (Notice that we get $([N/x]M)N$, which is different, if we drop the assumption that x is not free in M.) Since the two functions return the same result on all arguments, we have the equational axiom

$$(\eta) \qquad \lambda x{:}\,\sigma.Mx = M, \quad x \text{ not free in } M$$

Just as the final axiom for pairing is redundant for explicit pairs, the (η) axiom for functions is redundant for explicit function expressions. Specifically, if M has the form $\lambda y{:}\,\sigma.P$, and x is not free in M, then we can prove $\lambda x{:}\,\sigma.Mx = M$ using (β). However, we cannot prove $\lambda x{:}\,\sigma.Mx = M$ if M is a variable of function type without using (η). A consequence of (η) is given in part (b) of Exercise 2.2.7.

Both (β) and (η) could be used as reduction rules, read left to right. When it is important to distinguish between the equational axioms, we use $(\beta)_{eq}$ and $(\eta)_{eq}$ for the equational axioms and $(\beta)_{red}$ and $(\eta)_{red}$ for the reduction rules. While η-reduction is used in many versions of lambda calculus, we will *not* use $(\eta)_{red}$ in PCF reduction since it is not needed. As discussed in Section 1.3, we keep (α) as the single equational axiom of the reduction system for PCF. In other words, PCF reduction is a relation on α-equivalence classes of expressions.

For easy reference, the syntax of PCF is summarized in Section 2.2.6. The equational proof system appears in Table 2.1 and the reduction system in Table 2.2.

Currying

A traditional topic in lambda calculus is the relationship between multi-argument and higher-order functions. If we have a mathematical function of two arguments, such as the addition function $add(x, y{:}\, nat) = x + y$, then we generally think of this as a function on ordered pairs $add{:}\, (nat \times nat) \to nat$. This type of addition function may be written as the PCF expression

$$add = \lambda p{:}\, nat \times nat.\, (\mathbf{Proj}_1 p) + (\mathbf{Proj}_2 p)$$

In words, this add is the function which takes a pair of natural numbers, and returns the sum of its two components.

A related function is called the "curried" addition function

$$Curry(add) = \lambda x{:}\, nat.\, \lambda y{:}\, nat.\, x + y$$

after lambda calculus pioneer Haskell Curry. The curried addition function yields $x+y$ when applied to x and y, but the form of parameterization is different. Like many other lambda expressions, it is easiest to read $Curry(add)$ from right to left. The final result of applying the function is the expression to the right of all of the λ's, which is $x + y$. The next smallest expression is $\lambda y.\, x + y$. This function adds x, whose value is not determined within this expression, to the function argument. So we might call this the "add x" function. Since this function takes a single natural-number argument and returns a natural number result, we have

$$\lambda y.\, x + y : nat \to nat.$$

Now let us look again at the whole function $Curry(add)$. On argument x, the function $Curry(add)$ returns the "add x" function. So $Curry(add)$ is a function of a single natural-number argument, returning a numeric function. This is reflected in the type of the function

$$Curry(add){:}\, nat \to (nat \to nat).$$

Thus the curried addition function yields $x + y$ when applied to two natural numbers x and y, but as the type indicates, $Curry(add)$ takes two natural number arguments one at a time, rather than all at once.

As suggested by the notation $Curry(add)$, there is a higher-order transformation $Curry$ which maps any binary natural- number function f to its curried form $Curry(f)$. This map can actually be written in PCF, as the lambda expression

$$Curry = \lambda f{:}\, (nat \times nat) \to nat.\, \lambda x{:}\, nat.\, \lambda y{:}\, nat.\, f\langle x, y\rangle.$$

Using the axiom of β-equivalence, it is easy to see that when applied to add, the function $Curry$ produces the lambda expression written above.

$$\begin{aligned}
Curry(add) &= (\lambda f{:}\, (nat \times nat) \to nat.\lambda x{:}\, nat.\lambda y{:}\, nat.\, f\langle x, y\rangle)\ add \\
&= \lambda x{:}\, nat.\lambda y{:}\, nat.\ add\ \langle x, y\rangle \\
&= \lambda x{:}\, nat.\lambda y{:}\, nat.\, (\lambda p{:}\, nat \times nat.\, (\mathbf{Proj}_1 p) + (\mathbf{Proj}_2 p))\ \langle x, y\rangle \\
&= \lambda x{:}\, nat.\lambda y{:}\, nat.\, \mathbf{Proj}_1\langle x, y\rangle + \mathbf{Proj}_2\langle x, y\rangle \\
&= \lambda x{:}\, nat.\lambda y{:}\, nat.\, x + y
\end{aligned}$$

Some properties of currying are given in Exercise 2.2.8.

Exercise 2.2.4 Write a PCF expression to take any numeric function f and return a numeric function g such that for any n, the value of $g(n)$ is twice $f(n)$.

Exercise 2.2.5 Perform the following substitution. (Do not reduce the expression.)

$$[(y+3)/x]\ ((\lambda f\colon nat \to nat.\ \lambda y\colon nat.\ f(x+y))(\lambda x\colon nat.\ x+y))$$

Exercise 2.2.6 Prove the following substitution identities.

(*a*) $$[M/x]([N/x]P) = [([M/x]N)/x]P,$$

(*b*) $$[M/x]([N/y]P) = [([M/x]N)/y]([M/x]P) \text{ if } y \not\equiv x \text{ and } y \notin FV(M).$$

Exercise 2.2.7 Show that the following two inference rules are derivable, *i.e.*, the equation below the line is provable from the equation(s) above the line. For (a), you may assume that from $x = u$ and $y = v$ it is possible to prove $\langle x, y\rangle = \langle u, v\rangle$.

(*a*) $$\frac{\mathbf{Proj}_1 p = \mathbf{Proj}_1 q,\ \mathbf{Proj}_2 p = \mathbf{Proj}_2 q}{p = q}$$

(*b*) $$\frac{Mx = Nx}{M = N}\ x \notin FV(M, N)$$

Exercise 2.2.8 Write a PCF expression *Uncurry* which "uncurries" natural number functions. More specifically, if f is a curried function $f\colon nat \to (nat \to nat)$, then $Uncurry(f)$ should be an ordinary binary function $Uncurry(f)\colon (nat \times nat) \to nat$ taking a pair of numbers and producing a natural number result. Use the axioms for functions and pairing to prove the equations $Uncurry(Curry\, g) = g$ and $Curry(Uncurry\, f) = f$, for any ordinary binary $g\colon (nat \times nat) \to nat$ and any curried $f\colon nat \to (nat \to nat)$.

2.2.4 Declarations and syntactic sugar

One of the appeals of lambda calculus, as a model programming language, is the way that lambda abstraction corresponds to variable binding in common programming languages. This is immediately clear for Lisp and its dialects, which are based on lambda notation. It is also true for Algol-like languages, such as Pascal. For example, consider the following Algol-like program fragment

begin function $f(x\colon nat)\ nat$;
    return x;
    end;
$\langle body\rangle$
end

with a function declaration at the beginning of a block. We can easily write the function f as the lambda expression $\lambda x\colon nat.\ x$. We will also see that the entire block may be translated into lambda notation. The translation may be simplified by adopting a general notational convention.

A reasonable typed syntax for declarations is

$$\texttt{let}\ x\colon \sigma = M\ \texttt{in}\ N,$$

which binds x to M within the declaration body N. In other words, the value of $\texttt{let}\ x{:}\sigma = M\ \texttt{in}\ N$ is the value of N with x set to M. We consider this well-typed only if M has type σ. Using `let`, we can now write the Algol-like block above as

$$\texttt{let}\ f{:}\, nat \to nat = \lambda x{:}\, nat.x\ \texttt{in}\ \langle body \rangle.$$

Instead of adding `let` declarations to PCF directly, we will treat `let` as an abbreviation, or what is often called "syntactic sugar" in computer science. This means that we will feel free to use `let` in writing PCF expressions, but we do not consider `let` part of the actual syntax of PCF. Instead, we will think of `let` as an abbreviation for a lambda expression, according to the rule

$$\texttt{let}\ x{:}\sigma = M\ \texttt{in}\ N \stackrel{def}{=} (\lambda x{:}\sigma.N)M$$

This treatment of `let` allows us to keep the size of PCF relatively small, which will pay off when we come to proving things about the language. In addition, we need not specify any additional equational axioms or reduction rules, as these are inherited directly from λ. To distinguish abbreviations from actual PCF syntax, we will use the symbol $\stackrel{def}{=}$, as above, when defining meta-notational conventions. In general, $M \stackrel{def}{=} N$ means that whenever we write a term of the form M, this is shorthand for a term of form N.

Example 2.2.9 We will translate the following "sugared" PCF expression into pure PCF (without syntactic sugar)

$$\begin{array}{l}\texttt{let}\ compose = \lambda f{:}\, nat \to nat.\lambda g{:}\, nat \to nat.\lambda x{:}\, nat.f(g\,x)\ \texttt{in}\\ \quad \texttt{let}\ h = \lambda x{:}\, nat.\ x + x\ \texttt{in}\\ \quad\quad compose\ h\ h\ 3\end{array}$$

and simplify by applying the appropriate reduction rules. The "de-sugared," pure PCF expression, written using the same names for bound variables, is:

$$\begin{array}{l}(\lambda compose{:}\, (nat \to nat) \to (nat \to nat) \to nat \to nat.\\ \quad (\lambda h{:}\, nat \to nat.\ compose\ h\ h\ 3)\ \ \lambda x{:}\, nat.\ x + x)\\ \quad \lambda f{:}\, nat \to nat.\ \lambda g{:}\, nat \to nat.\ \lambda x{:}\, nat.\ f(g\ x).\end{array}$$

This reduces as follows:

$$\begin{array}{ll}\to & (\lambda h{:}\, nat \to nat.\ (\lambda f{:}\, nat \to nat.\ \lambda g{:}\, nat \to nat.\ \lambda x{:}\, nat.\ f(g\ x))\ h\ h\ 3)(\lambda x{:}\, nat.\ x + x)\\ \to & (\lambda f{:}\, nat \to nat.\ \lambda g{:}\, nat \to nat.\ \lambda x{:}\, nat.\ f(g\ x))\ (\lambda x{:}\, nat.\ x + x)\ (\lambda x{:}\, nat.\ x + x)\ 3\\ \twoheadrightarrow & (\lambda x{:}\, nat.\ x + x)((\lambda x{:}\, nat.\ x + x)\ 3)\\ \to & ((\lambda x{:}\, nat.\ x + x)\ 3) + ((\lambda x{:}\, nat.\ x + x)\ 3)\\ \twoheadrightarrow & (3 + 3) + (3 + 3)\\ \twoheadrightarrow & 12.\end{array}$$

■

The notion of "syntactic sugar" is very useful in programming language analysis. By considering certain constructs as syntactic sugar for others, we can write program examples in a familiar notation, and analyze these programs as if they are written using simpler or less troubling primitives. However, this technique must be used with care. Since computation may be modeled by substitution, as in the reduction rules for PCF, we do not want to make too many transformations under the umbrella of syntactic sugar. Three indications that the definition of `let` does not add any computational power to PCF are:

(i) `let` $x{:}\sigma = M$ `in` N and the lambda expression $(\lambda x{:}\sigma.N)M$ have approximately the same size (counting symbols);

(ii) both have the same typing constraints;

(iii) the two expressions have the same immediate subexpressions (syntactic constituents).

While these conditions are not absolute rules about syntactic sugar, they are a useful guide towards good use of syntactic sugar.

Another syntactic extension allows us to write function declaration in a more familiar form, as follows

$$\texttt{let } f(x{:}\tau){:}\sigma = M \texttt{ in } N \stackrel{def}{=} \texttt{let } f{:}\tau\to\sigma = \lambda x{:}\tau.M \texttt{ in } N$$

We can also add multi-argument function definitions, as in Exercise 2.2.12 below.

Example 2.2.10 The language ISWIM, described in [Lan66], has an expression form M `where` $x{:}\sigma = N$. Intuitively, the value of this expression is the value of M when the variable x is set to N. We can easily add `where` to PCF by the abbreviation

$$M \texttt{ where } x{:}\sigma = N \stackrel{def}{=} \texttt{let } x{:}\sigma = N \texttt{ in } M.$$

■

Exercise 2.2.11 Which PCF reduction rule is applicable to any term of the form `let` $x{:}\sigma = M$ `in` N, and what is the result of a single application of this rule?

Exercise 2.2.12
(a) Show how to add two-argument functions to PCF by defining

$$\lambda\langle x{:}\sigma,\ y{:}\tau\rangle.\, M$$

as syntactic sugar. If $\lambda x{:}\sigma.\,\lambda y{:}\tau.\, M$ has type $\sigma\to\tau\to\rho$, then $\lambda\langle x{:}\sigma,\ y{:}\tau\rangle.\, M$ has type $\sigma\times\tau\to\rho$. You may want to use the results of Exercise 2.2.8.
(b) Introduce a two-argument function form of `let`

$$\texttt{let } f\langle x{:}\sigma,\ y{:}\tau\rangle{:}\rho = M \texttt{ in } N$$

as syntactic sugar, using the result of part (a). Illustrate the induced reduction rule for this definition form by expanding `let` $f\langle x{:}\sigma,\ y{:}\rho\rangle{:}\tau = M$ `in` N into pure PCF and reducing.

Exercise 2.2.13 This exercise is concerned with *static scope* and renaming of bound variables. Intuitively, static scoping of variables means that the binding of a variable is always determined by finding the closest enclosing binding operator for that variable. Moreover, the binding of a variable does not change during evaluation of expressions. With *dynamic scoping,* the way a variable is bound may change during evaluation. This distinction will be intuitively familiar to computer scientists who have studied programming languages.

(a) Under static scope, the free x in the declaration of f is bound by the outer declaration. As a result, the function f always adds 3 to its argument, no matter where it is called.

$$\begin{array}{l}\texttt{let } x\colon nat = 3 \texttt{ in}\\ \quad \texttt{let } f(y\colon nat)\colon nat = x + y \texttt{ in}\\ \quad\quad \texttt{let } x\colon nat = 4 \texttt{ in}\\ \quad\quad\quad f\,5\end{array}$$

Use reduction to simplify this PCF expression to 8. You may save yourself the trouble of rewriting `let`'s to λ's by using the derived rule $\texttt{let } x\colon\sigma = M \texttt{ in } N \to [M/x]N$.

(b) Suppose that in reducing the expression in part (a), you begin by substituting $\lambda y\colon nat.\, x + y$ for f in the inner expression, $\texttt{let } x\colon nat = 4 \texttt{ in } f\,5$. (This actually corresponds to the usual implementations, since we begin evaluation of $f\,5$ by passing 5 to the body of f.) Explain how renaming bound variables provides static scope. In particular, say which variable above (x, f or y) must be renamed, and how a specific occurrence of this variable is therefore resolved statically.

(c) Under dynamic scope, the binding for x in the body of f changes when f is called in the scope of the inner binding of x. Therefore, under dynamic scope, the value of this expression will be $4 + 5 = 9$. Show how you can reduce the expression to 9 by *not* renaming bound variables when you perform substitution.

(d) In the usual statically scoped lambda calculus, α-conversion (renaming bound variables) does not change the value of an expression. Use the example expression from part (a) to explain why α-conversion may change the value of an expression if variables are dynamically scoped.

(e) Show that confluence fails for dynamically-scoped PCF.

2.2.5 Recursion and fixed-point operators

The final construct of PCF provides definition by recursion. Rather than add a new declaration form, we will treat recursive declarations as a combination of `let` and one new basic function, a fixed-point operator fix_σ for each type σ. The reader should be warned that this construct raises more subtle issues than the parts of the language we have already discussed. To begin with, recursion makes it possible to write expressions with no normal form. This changes our basic intuition about the way that an expression defines a mathematical value. In addition, when there are several possible reductions that could be applied to a term, the choice between these becomes important.

We will see how fixed-point operators provide recursion by beginning with the declaration form

$$\texttt{letrec } f\colon\sigma = M \texttt{ in } N.$$

This has the intended meaning that within N, the variable f denotes a solution to the equation $f = M$. In general, f may occur in M. The main typing constraint for this declaration is that M must have type σ, since otherwise the equation $f = M$ does not make sense. We will see that `letrec` may be treated as syntactic sugar for a combination of ordinary `let` and a fixed-point operator.

We can see how `letrec` works by writing the factorial function. To simplify the notation, let us write $x - 1$ for the predecessor of x, and $x * y$ for the product of x and y. (To stay within the natural-numbers, we consider $0 - 1 = 0$.) We will see later how to define predecessor and natural number multiplication in PCF using recursion. With these extra operations, we can define the factorial function and compute 5! by writing

$$\texttt{letrec}\ f{:}\,nat \to nat = \lambda y{:}\,nat.\ (\texttt{if}\ \ Eq?\,y\,0\ \texttt{then}\ 1\ \texttt{else}\ \ y * f(y-1))\ \texttt{in}\ f\,5$$

Since the variable f occurs free in the expression on the right-hand side of the equals sign, we cannot simply take f to refer to the expression on the right. Instead, we want f in the function body to refer "recursively" to the function being defined. We may summarize this by saying that f must be a solution to the equation

$$f = \lambda y{:}\,nat.\ \texttt{if}\ \ Eq?\,y\,0\ \texttt{then}\ 1\ \texttt{else}\ \ y * f(y-1)$$

with two occurrences of the function variable f. From a mathematical point of view, it is not clear that every equation $f{:}\sigma = M$ involving an arbitrary PCF expression M should have a solution, or which solution to choose if several exist. We will consider these questions more carefully when we investigate the denotational semantics of PCF in Chapter 5. However, recursive function declarations have a clear computational interpretation. Therefore, we will assume that every defining equation has *some* solution and add syntax to PCF for expressing this. The associated equational axiom and rewrite rule will allow us to think operationally about recursive definitions and provide a useful guide in discussing denotational semantics in Chapter 5.

Using lambda abstraction, we can represent any recursive definition $f{:}\sigma = M$ by a function $\lambda f{:}\sigma.\,M$. Rather than looking for a solution to a recursive defining equation, we will produce a fixed point of the associated function. In general, if $F{:}\sigma \to \sigma$ is a function from some type to itself, a *fixed point of* F is a value $x{:}\sigma$ such that $F(x) = x$. For example, returning to the factorial function, we can see that factorial is a fixed point of the operator

$$F \stackrel{def}{=} \lambda f{:}\,nat \to nat.\,\lambda y{:}\,nat.\ \texttt{if}\ \ Eq?\,y\,0\ \texttt{then}\ 1\ \texttt{else}\ \ y * f(y-1)$$

on natural-number functions. We show how factorial is defined and computed in PCF by applying a fixed-point operator to this function.

The last basic construct of PCF is a family of functions

$$\mathit{fix}_\sigma : (\sigma \to \sigma) \to \sigma,$$

one for each type σ. The function fix_σ produces a fixed point of any function from σ to σ. Using lambda abstraction and fix_σ, we can regard the recursive `letrec` declaration form as an abbreviation:

$$\texttt{letrec}\ f{:}\sigma = M\ \texttt{in}\ N \stackrel{def}{=} \texttt{let}\ f{:}\sigma = (\mathit{fix}_\sigma\ \lambda f{:}\sigma.M)\ \texttt{in}\ N.$$

Since we often use `letrec` to define functions, we also adopt the syntactic sugar

$$\texttt{letrec}\ f(x{:}\tau){:}\sigma = M\ \texttt{in}\ N \stackrel{def}{=} \texttt{letrec}\ f{:}\tau \to \sigma = \lambda x{:}\tau.M\ \texttt{in}\ N.$$

The equational axiom for $\mathit{fix}_\sigma : (\sigma \to \sigma) \to \sigma$ is that it produces a fixed point

$$(\mathit{fix}) \qquad \mathit{fix}_\sigma = \lambda f{:}\sigma \to \sigma.\, f\,(\mathit{fix}_\sigma\, f).$$

Using (β), it is easy to derive the more intuitive equation

$$\mathit{fix}_\sigma\, M = M\,(\mathit{fix}_\sigma\, M)$$

for any $M : \sigma \to \sigma$. The (fix) reduction rule is obtained by reading the equational axiom from left to right. Some reasonable, non-equational properties of fix_σ are more subtle, and will be discussed in Sections 5.2 and 5.3.

To see how (fix) reduction works, we will continue the factorial example. Using $\mathit{fix}_{nat \to nat}$, the factorial function may be written $\mathit{fact} \stackrel{def}{=} \mathit{fix}_{nat \to nat}\, F$, where F is written out above. As a typographical simplification, we will drop the subscript of fix in the calculation below. To compute $\mathit{fact}\ n$, we may expand the definition, and use reduction to obtain the following.

$$\begin{array}{lcl}
\mathit{fact}\ n & \equiv & (\mathit{fix}\ F)\ n \\
& \twoheadrightarrow & F\ (\mathit{fix}\, F)\ n \\
& \equiv & (\lambda f{:}\mathit{nat} \to \mathit{nat}.\, \lambda y{:}\mathit{nat}.\ \texttt{if}\ \mathit{Eq?}\, y\, 0\ \texttt{then}\ 1\ \texttt{else}\ y * f(y-1))\ (\mathit{fix}\, F)\ n \\
& \twoheadrightarrow & \texttt{if}\ \mathit{Eq?}\, n\, 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\mathit{fix}\, F)(n-1)
\end{array}$$

Note that apart from (fix) reduction, we have used only ordinary β-reduction. When $n = 0$, we can use the axiom for conditional to simplify $\mathit{fact}\ 0$ to 1. For $n > 0$, we can simplify the test to obtain $n * (\mathit{fix}\, F)(n-1)$ and continue as above. For any natural number n, it is clear that we will eventually compute $\mathit{fact}\ n = n!$. Put more formally, we may use ordinary natural- number induction to prove the theorem about PCF that for every natural number n the application of fact to the PCF numeral for n may be reduced to the numeral for $n!$.

As mentioned above, fixed-point operators raise mathematical problems. If we think of the functions from $\mathit{nat} \to \mathit{nat}$ to $\mathit{nat} \to \mathit{nat}$ as ordinary set-theoretic functions, then it does not make sense to postulate that every $f : (\mathit{nat} \to \mathit{nat}) \to (\mathit{nat} \to \mathit{nat})$ has a fixed point x with the property $x = f(x)$. However, as shown in Exercise 2.2.16, we have fixed points of this type iff we allow numeric functions to be defined by recursion. Since recursion is fundamental to computation, it is important to have fixed-point operators in PCF. The way to make mathematical sense of fix is to realize that when we have recursion, we may write algorithms that define partial functions. If $f : (\mathit{nat} \to \mathit{nat}) \to (\mathit{nat} \to \mathit{nat})$, then its fixed point may be a *partial* function, such as the function which is undefined on all integer arguments. Thus, when we add recursion to PCF, we must understand that an expression of type σ defines an algorithm for computing an element of type σ. In the case that this algorithm does not terminate, the expression may not define one of the standard values of that type. (A note for recursion theorists is that if we replace fix by some other construct that does not allow us to write nonterminating functions, there would be total recursive functions we could not write in PCF. This follows from the fact that the set of all total recursive functions is not *r.e.*)

Although fix takes the fixed point of a simple function, we can actually use fix to define any number of mutually recursive functions. This is most easily illustrated for the case of two recursive functions. Suppose we want to define recursive functions f and g satisfying the equations

$$\begin{array}{lcl} f & = & F\, f\, g \\ g & = & G\, f\, g \end{array}$$

where we assume that neither f nor g appears free in F or G. Let us assume that $F{:}\,\sigma \to \tau \to \sigma$ and $G{:}\,\sigma \to \tau \to \tau$ so that both equations are type correct. We may then apply $\mathit{fix}_{\sigma \times \tau}$ to the function on pairs

$$\lambda\langle f{:}\,\sigma, g{:}\,\tau\rangle.\, \langle F\ f\ g,\ G\ f\ g\rangle.$$

This gives us a recursively-defined pair whose first and second components satisfy the original defining equations. The details are left as Exercise 2.2.15. An alternative approach which does not use pairing is given in Exercise 5.3.5.

Exercise 2.2.14 Assuming we have subtraction in PCF, write a `letrec` expression of the form

$$\texttt{letrec}\ \mathit{fib}(x{:}\,\mathit{nat}){:}\,\mathit{nat} = \ \ldots\ \ \texttt{in}\ \mathit{fib}(4)$$

which defines the Fibonacci function and applies it to 4 to compute the fourth Fibonacci number. (Recall that the zero-th Fibonacci number is 1, the first Fibonacci number is 1, and after that each Fibonacci number is the sum of the preceding two.) Show how your `letrec` expression reduces to the 4th Fibonacci number. You should give approximately the same amount of detail as in the factorial example of this section.

Exercise 2.2.15 Consider the pair of equations

$$\begin{array}{lcl} f & = & F\ f\ g \\ g & = & G\ f\ g \end{array}$$

where $F{:}\,\sigma \to \tau \to \sigma$, $G{:}\,\sigma \to \tau \to \tau$ and neither f nor g appears free in F or G. Show that the first and second components of

$$\mathit{fix}_{\sigma \times \tau}\,(\lambda\langle f{:}\,\sigma, g{:}\,\tau\rangle.\, \langle F\ f\ g,\ G\ f\ g\rangle)$$

satisfy these equations. (An alternative approach to mutually-recursive definitions is given in Exercise 5.3.5.)

Exercise 2.2.16 In this section, we defined `letrec` as syntactic sugar using *fix*. The point of this exercise is to demonstrate that *fix* is also definable from `letrec`.

(a) Write an expression FIX using `letrec` with the property that if we expand $\texttt{letrec}\ f{:}\,\sigma = M\ \texttt{in}\ N$ to $(\lambda f{:}\,\sigma.N)(\mathit{fix}_\sigma\ (\lambda f{:}\,\sigma.M))$, we can reduce FIX to *fix* using only β-reduction and η-reduction.

(b) Write a reduction rule for `letrec` of the form

$$\texttt{letrec}\ f{:}\,\sigma = M\ \texttt{in}\ N\ \to\ P,$$

where P is an expression containing M, N and `letrec`, but not *fix*. As a check that this is plausible, show that P is provably equal to

$$[\,[(\mathit{FIX}\ (\lambda f{:}\,\sigma.M))/f]M\,]\,/\,f]N,$$

which corresponds to the result of doing one *fix* reduction and one β-reduction to the term $(\lambda f{:}\,\sigma.N)(\mathit{fix}_\sigma\ (\lambda f{:}\,\sigma.M))$ mentioned in part *(a)*.

(c) Show that if we take `letrec` as basic, and use the reduction rule given in part *(b)*, the term FIX you gave in answer to part *(a)* is a fixed-point operator. That is, show that for any term M of the correct type, $\mathit{FIX}\ M \twoheadrightarrow MF$, where F is the result of applying one β-reduction to $(\mathit{FIX}\ M)$.

2.2.6 PCF syntax summary and collected examples

Pure PCF

The syntax of pure PCF, without extension by syntactic sugar, is summarized below by a BNF-like grammar. The first set of productions describe the expressions of an arbitrary type σ. These include variables, conditional expressions, and the results of function application, projection functions, and fixed-point application.

$$\begin{array}{lcl}
\langle \sigma_exp \rangle & ::= & \langle \sigma_var \rangle \,|\, \texttt{if}\ \langle bool_exp \rangle\ \texttt{then}\ \langle \sigma_exp \rangle\ \texttt{else}\ \langle \sigma_exp \rangle \,| \\
& & \langle \sigma_application \rangle \,|\, \langle \sigma_projection \rangle \,|\, \langle \sigma_fixed_point \rangle \\
\langle \sigma_application \rangle & ::= & \langle \tau \to \sigma_exp \rangle \langle \tau_exp \rangle \\
\langle \sigma_projection \rangle & ::= & \mathbf{Proj}_1 \langle \sigma \times \tau_exp \rangle \,|\, \mathbf{Proj}_2 \langle \tau \times \sigma_exp \rangle \\
\langle \sigma_fixed_point \rangle & ::= & \mathit{fix}_\sigma \, \langle \sigma \to \sigma_exp \rangle
\end{array}$$

For function and product types, we also have lambda abstraction and explicit pairing.

$$\begin{array}{lcl}
\langle \sigma \to \tau_exp \rangle & ::= & \lambda x{:}\sigma.\langle \tau_exp \rangle \\
\langle \sigma \times \tau_exp \rangle & ::= & \langle\, \langle \sigma_exp \rangle, \langle \tau_exp \rangle \,\rangle
\end{array}$$

The constants and functions for natural numbers and booleans are covered by the following productions.

$$\begin{array}{lcl}
\langle bool_exp \rangle & ::= & \mathit{true} \,|\, \mathit{false} \,|\, \mathit{Eq?}\, \langle nat_exp \rangle \, \langle nat_exp \rangle \\
\langle nat_exp \rangle & ::= & 0 \,|\, 1 \,|\, 2 \,|\, \ldots \,|\, \langle nat_exp \rangle + \langle nat_exp \rangle
\end{array}$$

This concludes the definition of PCF. An alternate definition of the syntax of PCF may be given using the typing rule style of Chapter 4.

Syntactic extensions

The most commonly used extensions of PCF by syntactic sugar are listed below, along with their definitions. To distinguish abbreviations from actual PCF syntax, we use the symbol $\stackrel{def}{=}$ to define meta-notational conventions. The definition $M \stackrel{def}{=} N$ means that whenever we write a term of the form M, this is short-hand for the corresponding term of the form N.

$$
\begin{array}{lcl}
\texttt{let } x{:}\sigma = M \texttt{ in } N & \stackrel{def}{=} & (\lambda x{:}\sigma.\, N)M \\
\texttt{let } f(x{:}\sigma){:}\tau = M \texttt{ in } N & \stackrel{def}{=} & \texttt{let } f{:}\sigma\to\tau = \lambda x{:}\sigma.\, M \texttt{ in } N \\
 & \stackrel{def}{=} & (\lambda f{:}\sigma\to\tau.\, N)(\lambda x{:}\sigma.\, M) \\
\texttt{letrec } f{:}\sigma = M \texttt{ in } N & \stackrel{def}{=} & \texttt{let } f{:}\sigma = (\mathit{fix}_{\sigma}\ (\lambda f{:}\sigma.\, M)) \texttt{ in } N \\
 & \stackrel{def}{=} & (\lambda f{:}\sigma.\, N)(\mathit{fix}_{\sigma}\ (\lambda f{:}\sigma.\, M)) \\
\texttt{letrec } f(x{:}\tau){:}\sigma = M \texttt{ in } N & \stackrel{def}{=} & \texttt{letrec } f{:}\tau\to\sigma = \lambda x{:}\tau.\, M \texttt{ in } N \\
 & \stackrel{def}{=} & \texttt{let } f{:}\tau\to\sigma = (\mathit{fix}_{\tau\to\sigma}\ (\lambda f{:}\tau\to\sigma.\, \lambda x{:}\tau.\, M)) \texttt{ in } N \\
 & \stackrel{def}{=} & (\lambda f{:}\tau\to\sigma.\, N)(\mathit{fix}_{\tau\to\sigma}\ (\lambda f{:}\tau\to\sigma.\, \lambda x{:}\tau.\, M)) \\
\lambda\langle x{:}\sigma, y{:}\tau\rangle.\, M & \stackrel{def}{=} & \lambda p{:}\sigma\times\tau.(\lambda x{:}\sigma.\, \lambda y{:}\tau.\, M)(\mathbf{Proj}_1\, p)(\mathbf{Proj}_2\, p) \\
\texttt{let } \langle x{:}\sigma, y{:}\tau\rangle = M \texttt{ in } N & \stackrel{def}{=} & (\lambda\langle x{:}\sigma, y{:}\tau\rangle.\, N)M
\end{array}
$$

A useful convention is to write $M^n N$ for the term $M\,(M\,\ldots\,(M\,N)\ldots)$ constructed by applying M to N a total of n times. A more precise definition is that $M^0 N \stackrel{def}{=} N$ and $M^{n+1}N \stackrel{def}{=} M(M^n N)$.

Examples

Boolean and natural-number expressions.

$$\texttt{if } \mathit{Eq}?\,(4+5)\,9 \texttt{ then } 27 \texttt{ else } 42$$

$$\texttt{if } (\texttt{if } \mathit{Eq}?\,(4+5)\,9 \texttt{ then } \mathit{false} \texttt{ else } \mathit{true}) \texttt{ then } 42 \texttt{ else } 27$$

$$(\texttt{if } \mathit{Eq}?\,(x+y)\,z \texttt{ then } \lambda x{:}\mathit{nat}.\, x+5 \texttt{ else } \lambda y{:}\mathit{nat}.\, y+7)\ (\texttt{if } \mathit{Eq}?\,(x+y)\,z \texttt{ then } 7 \texttt{ else } 5)$$

Lambda abstraction and application.

$$\lambda x{:}\mathit{nat}.\, x$$

$$\lambda x{:}\mathit{nat}.\, x+1$$

$$(\lambda x{:}\mathit{nat}.\, x+1)((\lambda y{:}\mathit{nat}.\, 5+y)\,3)$$

$$(\lambda f{:}\mathit{nat}\to\mathit{nat}.\, \lambda x{:}\mathit{nat}.\, f(f(f\,x)))\,(\lambda y{:}\mathit{nat}.\, 5+y)\,3$$

Pairing and functions.

$\lambda x{:}\, nat \times nat.\langle \mathbf{Proj}_1\, x + 1, \mathbf{Proj}_2\, x + 1\rangle$	:	$(nat \times nat) \to (nat \times nat)$
$\lambda x{:}\, \sigma \times \tau.\langle \mathbf{Proj}_2\, x, \mathbf{Proj}_1\, x\rangle$	:	$(\sigma \times \tau) \to (\tau \times \sigma)$
$\lambda x{:}\, nat \times nat.$ $\quad$ `if` $Eq?\,(\mathbf{Proj}_1\, x)\, 0$ `then` $\mathbf{Proj}_2\, x$ `else` 0	:	$nat \times nat \to nat$
$\lambda f{:}\, nat \to nat.\ \lambda g{:}\, nat \to nat.\ \lambda x{:}\, nat.\ f\,(g\, x)$	is	composition of numeric functions
$\lambda f{:}\, nat \to nat.\ \lambda g{:}\, nat \to nat.\ \lambda x{:}\, nat.$ $\quad$ `if` $Eq?\,(f\, x)\, 0$ `then` $g\, x$ `else` 0	:	$(nat \to nat) \to (nat \to nat) \to (nat \to nat)$
$\lambda f{:}\, (nat \times nat) \to nat.\ \lambda x{:}\, nat.\ \lambda y{:}\, nat.\ f\langle x, y\rangle$	is	Curry for numeric functions

Declarations and recursion.

```
let comp: (τ → ρ) → (σ → τ) → (σ → ρ)
    = λf: τ → ρ. λg: σ → τ. λx: σ. f (g x)
    in comp f g
```

```
let add: (nat × nat) → nat
    = λx: nat × nat.(Proj_1 x) + (Proj_2 x)
    in let curry: ((nat × nat) → nat) → nat → nat → nat
        = λf: (nat × nat) → nat. λx: nat. λy: nat. f⟨x, y⟩
        in  curry add 5 7
```

Assuming we have a predecessor function *pred* which maps each $n > 0$ to $n - 1$ and zero to zero, we can write multiplication as follows. A definition of *pred* is given in Section 2.5.2, using a general method described there.

```
letrec mult: (nat × nat) → nat
    = λp: nat × nat.
        if Eq? (Proj_1 p) 0 then 0 else (Proj_2 p) + mult⟨pred (Proj_1 p), Proj_2 p⟩
    in mult⟨6, 7⟩
```

We may rewrite this last example using more syntactic sugar to make it clearer:

```
letrec mult⟨x: nat, y: nat⟩: nat = if Eq? x 0 then 0 else y + mult⟨pred x, y⟩
   in mult⟨6, 7⟩
```

For further examples, see Section 2.5 and the exercises.

2.3 PCF Programs and Their Semantics

2.3.1 Programs and results

Now that we have seen all of the constructs of PCF, we will take a step back and discuss the meaning, or *semantics,* of PCF in a general way that is intended to apply to a variety of programming languages. This should put the equational axioms and reduction rules that were presented along with the syntax of PCF in perspective. The three forms of semantics we will consider are axiomatic semantics, given by a proof system, operational semantics arising from a set of reduction rules, and denotational semantics (with details put off to Chapter 4). Each form of semantics has advantages and disadvantages for understanding properties of programs. In addition, there are standard connections between the three forms of semantics that should hold for any programming language. We discuss a few syntactic distinctions in this brief subsection before stating the connections precisely in the next three subsections.

Most programming languages have several different syntactic categories, or "kinds of expressions." These may include expressions as used in assignments or function calls, imperative statements, declarations or modules. However, not all well-formed syntactic entities can be executed or evaluated by themselves. Therefore, in any programming language, we distinguish *programs* from other syntactic forms that may be used only as parts of programs. The two characteristics that distinguish a program from an arbitrary syntactic form are that a program should not refer to any undeclared or unbound variables and a program should have the appropriate type or form to yield a printable value or observable effect.

In PCF, the two syntactic categories are types and terms (expressions that have types). If we think of giving a closed natural number or boolean term to an interpreter, then we could expect the interpreter to print its value. Therefore, we consider closed boolean and numeric expressions programs. But if we give an open natural number term such as $x + 5$ to an interpreter, there is no correct value since the value of x is not given. In addition, we cannot expect the value of a function expression to be printed with the same degree of accuracy as a natural number or boolean. Although an interpreter could certainly print back the function expression we give it, or perhaps print some kind of "optimized" code, no interpreter can print function expressions in a way that would tell us exactly which mathematical function is defined by the function expression typed in. This is because equality for partial recursive functions is undecidable and, as proved in Section 2.5.5, every partial recursive function is definable in PCF. For this reason, we say that natural number and boolean values are *observable,* but $nat \to nat$ values are not.

We give a precise definition of PCF *program* by defining the types of observable values. Although a pair of observable values could be considered observable, since an interpreter can easily print a pair of printable values, we will simplify several technical arguments by choosing a simpler set of observable types. This will not have any significant effect on our theory, as shown in Exercise 2.3.5. We say τ is an *observable type* if τ is either nat or $bool$. A PCF *program* is a well-formed, closed term of observable type. One potentially confusing aspect of this definition is that it does not consider input or output; it might be more accurate to call these "programs that require no additional input and produce one output." The reason we focus on programs which already have whatever input they require is that we use these programs to compare axiomatic, operational and denotational semantics. For this purpose, it is sufficient for each program to yield a single "data point" about the semantics.

Another general term we use in comparing semantics is *result.* Intuitively, a result is an observable effect of evaluating or executing a program. In a functional setting, this generally means a term giving an expected final result of program evaluation. For PCF, we make this precise by

saying that a *result* is a closed normal form of observable type. In other words, the PCF results are the numerals, $0, 1, 2, \ldots$ and boolean constants *true* and *false*.

With this terminology, we can give a general definition of semantics, at least for programming languages: a *semantics of programs* is a relation between programs and results. This is a minimal condition; all of the semantics we consider give more information than the result of evaluating each program. Typically, axiomatic and denotational semantics give more useful information than operational semantics when it comes to expressions that may occur in programs but are not full programs themselves.

2.3.2 Axiomatic semantics

In general, an axiomatic semantics consists of a proof system for deducing properties of programs and their parts. These properties may be equations, as in the PCF proof system (reviewed below), assertions about the output of programs, given certain inputs, or other properties. Since it is difficult to discuss all forms of axiomatic semantics in general, we will focus on equational axiomatic semantics. We may apply this discussion to axiomatic semantics that address other properties of programs by saying that two programs are equivalent in an axiomatic semantics whenever exactly the same assertions are derivable about each of them.

There are three general properties of axiomatic semantics that hold for PCF. The first is that the axiomatic semantics defines program behavior in some way. The other two are relations between the axiomatic and operational or denotational semantics.

- The axiomatic semantics determine the result (or output or observable effect, in general) of any program that has one.
- When two expressions are equivalent, according to the axiomatic semantics, we may safely substitute one for the other in any program without changing the operational semantics of that program. This could be called "soundness of the axiomatic semantics with respect to the operational semantics."
- The axiomatic semantics are sound with respect to the denotational semantics. Specifically, if we can prove any pair of PCF terms equal, then these terms must have the same denotation, regardless of which values we give to their free variables.

The PCF axiomatic semantics satisfies the first requirement since, for any full program that terminates according to the operational semantics, we may prove that this expression is equal to the appropriate numeral ($0, 1, 2, \ldots$) or boolean constant (*true* or *false*). This is a simple consequence of the fact that the PCF reduction axioms are a subset of the equational axioms. It follows from either of the last two conditions that, unless the operational or denotational semantics is degenerate, the axiomatic semantics does not equate all expressions of each type.

The axiomatic semantics of PCF is given by the proof system whose axioms are described in Sections 2.2.2 through 2.2.5 and listed again in Table 2.1. These axioms are combined with inference rules that make provable equality a congruence, also included in Table 2.1. *Congruence* means that provable equality is an equivalence relation (reflexive, symmetric and transitive), and that equality is preserved if we replace any subexpression by an equivalent one. We do not need a proof system for types, since two types are equal iff they are syntactically identical.

All of the congruence rules in Table 2.1, except the two concerned with lambda abstraction and application (the two at the bottom of the table) are in fact redundant. The reason is that we can use lambda abstraction and application to achieve the same effect. For example, suppose we can

Axioms

Equality	
(*ref*)	$\mathrm{M} = \mathrm{M}$
nat and *bool*	
(*add*)	$0+0=0,\ 0+1=1,\ \ldots,\ 3+5=8,\ \ldots$
(*Eq*?)	$Eq?\,n\,n = true$, $Eq?\,n\,m = false$ (n, m distinct numerals)
(*cond*)	$\texttt{if}\ true\ \texttt{then}\ M\ \texttt{else}\ N = M$, $\texttt{if}\ false\ \texttt{then}\ M\ \texttt{else}\ N = N$
Pairs ($\sigma \times \tau$)	
(*proj*)	$\mathbf{Proj}_1\langle M, N\rangle = M \quad \mathbf{Proj}_2\langle M, N\rangle = N$
(*sp*)	$\langle \mathbf{Proj}_1 P, \mathbf{Proj}_2 P\rangle = P$
Binding	
(α)	$\lambda x{:}\,\sigma.M = \lambda y{:}\,\sigma.[y/x]M$, provided y not free in M.
Functions ($\sigma \to \tau$)	
(β)	$(\lambda x{:}\,\sigma.M)N = [N/x]M$
(η)	$\lambda x{:}\,\sigma.Mx = M$, provided x not free in M
Recursion	
(*fix*)	$fix_\sigma = \lambda f{:}\,\sigma \to \sigma.f(fix_\sigma\, f)$

Inference Rules

Equivalence

(sym), ($trans$) $\quad \dfrac{M = N}{N = M} \qquad \dfrac{M = N,\ N = P}{M = P}$

Congruence

nat and *bool*

$$\frac{M = N,\ P = Q}{M + P = N + Q} \qquad \frac{M = N,\ P = Q}{Eq?\,M\,P = Eq?\,N\,Q}$$

$$\frac{M_1 = M_2,\ N_1 = N_2,\ P_1 = P_2}{\texttt{if}\ M_1\ \texttt{then}\ N_1\ \texttt{else}\ P_1 = \texttt{if}\ M_2\ \texttt{then}\ N_2\ \texttt{else}\ P_2}$$

Pairs $\quad \dfrac{M = N}{\mathbf{Proj}_i M = \mathbf{Proj}_i N} \qquad \dfrac{M = N,\ P = Q}{\langle M, P\rangle = \langle N, Q\rangle}$

Functions $\quad \dfrac{M = N}{\lambda x{:}\,\sigma.\ M = \lambda x{:}\,\sigma.\ N} \qquad \dfrac{M = N,\ P = Q}{MP = NQ}$

Table 2.1: Equational proof system for PCF

prove $M = N$ and $P = Q$, for four natural-number expressions M, N, P, and Q. We can derive $M + P = N + Q$ as follows. By the reflexivity axiom, we have the equation

$$\lambda x{:}\, nat.\lambda y{:}\, nat.\, x + y = \lambda x{:}\, nat.\lambda y{:}\, nat.\, x + y.$$

Therefore, by the congruence rule for application, we can prove

$$(\lambda x{:}\, nat.\lambda y{:}\, nat.\, x + y)\, M = (\lambda x{:}\, nat.\lambda y{:}\, nat.\, x + y)\, N.$$

Then, by axiom scheme (β) and transitivity,

$$\lambda y{:}\, nat.\, M + y = \lambda y{:}\, nat.\, N + y.$$

Repeating application, (β) and transitivity we may complete the proof of $M + P = N + Q$. The reason that the congruence rules are listed is that these are important properties of equality. In addition, if we were to develop a first-order theory of natural numbers and booleans, in the absence of lambda abstraction, we would need axioms expressing these properties.

It is worth mentioning that although the axiomatic semantics is powerful enough to determine the meaning of programs, this proof system is not as powerful as one might initially expect. For example, we cannot even prove that addition is commutative. Nor can we prove many interesting equivalences between recursive functions. For these kinds of properties, the most natural approach would be to add induction rules. Induction on natural numbers would let us prove commutativity quite easily, and a form of induction called "fixed-point induction" would allow us to prove a great many more equations between recursively-defined functions. For further information on natural-number induction, the reader may consult Section 1.8 or almost any book on mathematical logic, such as [End72]. Since fixed-point induction is specifically related to programs, and not a traditional topic in mathematical logic, we consider this in Section 5.3. A rudimentary form of fixed-point induction is discussed in Exercise 2.3.3 below.

Exercise 2.3.1 Use the equational proof system to prove that the following two terms are equal, for any M. You do not need any axioms about subtraction to carry out the proof.

$$\texttt{letrec}\ f(x{:}\, nat){:}\, nat = \texttt{if}\ \ Eq?\, x\, 0\ \texttt{then}\ 1\ \texttt{else}\ f(x-1)\ \texttt{in}\ M$$

$$\begin{array}{l}\texttt{let}\ f(x{:}\, nat){:}\, nat = \texttt{if}\ Eq?\, x\, 0\ \texttt{then}\ 1\\ \quad \texttt{else letrec}\ g(x{:}\, nat){:}\, nat = \texttt{if}\ \ Eq?\, x\, 0\ \texttt{then}\ 1\ \texttt{else}\ g(x-1)\ \texttt{in}\ g(x-1)\\ \texttt{in}\ M\end{array}$$

Exercise 2.3.2 A system for proving equations is *inconsistent* if every equation $M = N$ between well-formed expressions M and N of the same type is provable. Show that if *true* = *false* is provable from the axioms and inference rules of PCF, then the proof system is inconsistent. Use the same idea to show that if we can prove $m = n$ for distinct numerals m and n, the proof system is also inconsistent.

Exercise 2.3.3 While we can prove some simple equations involving recursion, such as the one given in Exercise 2.3.1, many similar-looking equations cannot be proved in the equational proof system given here. The reason is subtle but important. Consider two recursive definitions of factorial, for example, $f_1 = \mathit{fix}\, F_1$ and $f_2 = \mathit{fix}\, F_2$. The two functions f_1 and f_2 may give the same result for any natural number argument, even though the functions F_1 and F_2 are quite

different. However, the proof system does not seem to provide any way to deduce $f_1 = f_2$ without essentially showing that F_1 and F_2 are closely related.

A proof rule that helps prove equations involving fixed points is the following.

$$(\mathit{fix}\ \mathit{ind}) \qquad \frac{M \twoheadrightarrow NM}{M = \mathit{fix}\, N}\ .$$

The intuitive explanation of this rule, which is loosely based on McCarthy's rule of *recursion induction* [McC61, McC63], is that when M behaves computationally like a fixed point of N, then we conclude that M and $\mathit{fix}\, N$ have the same value. The computational nature of reduction is important here, since the rule becomes unsound if the hypothesis is replaced by the equation $M = NM$, as may be seen by taking N to be the identity. For further discussion of this rule, see Exercise 5.4.9.

Prove the following equations using $(\mathit{fix}\ \mathit{ind})$ in combination with the other equational proof rules.

(a) $\mathit{fix}(f \circ g) = f(\mathit{fix}(g \circ f))$.

(b) $\mathit{fix}_\sigma = \mathit{fix}_{(\sigma\to\sigma)\to\sigma}\lambda f{:}(\sigma\to\sigma)\to\sigma.\lambda g{:}\sigma\to\sigma.g(fg)$ where $\sigma \equiv \sigma_1 \to \sigma_2$.

2.3.3 Denotational semantics

The PCF denotational semantics assigns a natural number value (or a special additional value corresponding to nontermination) to each expression of type *nat*, a boolean value (or special value for nontermination) to each expression of type *bool*, and a mathematical function or pair of values to any function expression or expression of cartesian product type. The mathematical value of an expression is called its *denotation.* If a term has free variables, its denotation will generally depend on the values assumed for the free variables. While we will not go into details in this chapter, we will give a brief overview of the denotational semantics of PCF so that we can compare the three forms of semantics.

We give denotations to terms by first choosing a set of values for each type. In the standard semantics, the set of mathematical vales for type *nat* will include all of the natural numbers, plus a special element $\bot_{nat}$ representing nontermination. This extra value is needed since we have PCF expressions of type *nat* such as $\texttt{letrec}\ f(x{:}\mathit{nat}){:}\mathit{nat} = f(x+1)\ \texttt{in}\ f(0)$ which do not terminate under the standard interpreter, and do not denote any standard natural number. For similar reasons, the set of denotations of type *bool* includes *true*, *false* and an extra value $\bot_{bool}$. The mathematical values of a product type $\sigma \times \tau$ are ordered pairs, as you would expect. The mathematical values of type $\sigma \to \tau$ are functions from σ to τ with the property that if $\tau = \sigma$, then each function has a fixed point. The precise way of obtaining such a set of functions is discussed in Chapter 5. We interpret addition and other PCF functions in the usual way, for arguments that are ordinary natural numbers, and as the special value $\bot_{nat}$ when one of the arguments is $\bot_{nat}$. This is because we cannot compute an ordinary natural number by adding a number to an expression that does not define a terminating computation.

Once we have chosen a set of values for each type, we assign meanings to terms by choosing an *environment,* which is a mapping from variables to values. If x is a variable of type σ, and η is an environment, then $\eta(x)$ must be a mathematical value from type σ. We then define the meaning $[\![M]\!]\eta$ of term M in environment η by induction, in the manner that may be familiar from first-order logic. Specifically, the meaning $[\![x]\!]\eta$ is the value given to variable x by the environment, namely $\eta(x)$. The meaning of an application $[\![MN]\!]\eta$ is obtained by applying the function $[\![M]\!]\eta$

denoted by M to the argument $[\![N]\!]\eta$ denoted by N. Other cases are handled in a similar way. An important property is that the meaning of a term of type σ will always be one of the mathematical values associated with this type. Therefore, in the case of an application MN, for example, the typing rules of PCF guarantee that the denotation of M will be a function and the denotation of N will be a value in its domain. In this way, the syntactic typing rules of PCF avoid possible complications in the denotational semantics of the language.

There are several properties of denotational semantics that generally hold. The first is an intrinsic property of denotational semantics that distinguishes it from other forms of semantics.

- The denotational semantics is *compositional,* which means that the meaning of any expression is determined from the meanings of its subexpressions. For example,

$$[\![\texttt{if } B \texttt{ then } M \texttt{ else } N]\!]\eta = \begin{cases} [\![M]\!]\eta & \text{if } [\![B]\!]\eta \text{ is true,} \\ [\![N]\!]\eta & \text{if } [\![B]\!]\eta \text{ is false,} \\ \bot & \text{otherwise.} \end{cases}$$

  where $\bot$ is used as the meaning of the expression in the case that evaluation of the test B does not terminate. An immediate consequence of the compositional form of this definition is that if B', M' and N' have the same denotations as B, M and N, respectively, then

$$[\![\texttt{if } B \texttt{ then } M \texttt{ else } N]\!]\eta = [\![\texttt{if } B' \texttt{ then } M' \texttt{ else } N']\!]\eta.$$

  The reason these two must be equivalent is that the meaning of the conditional expression `if` B `then` M `else` N cannot depend on factors such as the syntactic form of B, M and N, only their semantic meaning.

- If we can prove the same assertions about M and N in the axiomatic semantics, then M and N must have the same meaning in the denotational semantics. This is called *soundness,* for an equational proof system. This is a minimal property in the sense that if soundness fails, we would conclude that something was wrong with either the axiomatic or denotational semantics. It follows from soundness, by connections between the axiomatic and operational semantics of PCF, that if $M \twoheadrightarrow N$, then M and N must have the same meaning in the denotational semantics.

Both of these properties hold for the standard denotational semantics of PCF. Further connections between operational, denotational and axiomatic semantics are summarized in Section 2.3.5.

2.3.4 Operational semantics

An operational semantics may be given in several ways. The most common mathematical presentations are proof systems for either deducing the final result of evaluation or for transforming an expression through a sequence of steps. An alternative that may provide more insight into practical implementation is to define an abstract machine, which is a theoretical computing machine that evaluates programs by progressing through a series of machine states. The most common practical presentations of operational semantics are interpreters and compilers. In this book, we concentrate on the first forms of operational semantics, proof systems defining either complete or step-by-step evaluation.

The operational semantics of PCF are given by the reduction axioms mentioned in Sections 2.2.2 through 2.2.5, which are summarized in Table 2.2. The reduction axioms are written with the symbol $\rightarrow$ instead of $=$, to emphasize the direction of reduction. Intuitively, $M \rightarrow N$ means that

nat and *bool*	
(*add*)	$0+0 \to 0,\ 0+1 \to 1,\ \ldots,\ 3+5 \to 8,\ \ldots$
(*Eq*?)	$Eq?\,n\,n \to true,\ Eq?\,n\,m \to false$ (n, m distinct numerals)
(*cond*)	`if` $true$ `then` M `else` $N \to M$, `if` $false$ `then` M `else` $N \to M$
Pairs ($\sigma \times \tau$)	
(*proj*)	$\mathbf{Proj}_1\langle M, N\rangle \to M \quad \mathbf{Proj}_2\langle M, N\rangle \to N$
Rename bound variables	
(α)	$\lambda x{:}\,\sigma.M = \lambda y{:}\,\sigma.[y/x]M$, provided y not free in M.
Functions ($\sigma \to \tau$)	
(β)	$(\lambda x{:}\,\sigma.M)N \to [N/x]M$
Recursion	
(*fix*)	$fix_\sigma \to \lambda f{:}\,\sigma \to \sigma.f(fix_\sigma\, f)$

Table 2.2: Reduction axioms for PCF

with one evaluation step, the expression M may be transformed to the expression N. While N may not be shorter than M, most of the rules have the "feel" of program execution; it seems that we are making progress towards a simpler expression in some way. We may define an *evaluation partial function* from the reduction system by $eval(M) = N$ iff M may be reduced to normal form N in zero or more steps.

This relatively abstract operational semantics, which lacks any specific evaluation order, may be refined in several ways. We will discuss three forms of "symbolic interpreters" in Section 2.4, one deterministic, one nondeterministic, and the third a form of parallel interpreter. While all of the interpreters are defined from reduction axioms, each applies reduction axioms to subexpressions of programs in a different way. The deterministic interpreter is defined by choosing a specific "next reduction step" at each point in program evaluation, while the nondeterministic interpreter may choose to reduce any subexpression, or choose to halt. The parallel interpreter may apply several reductions simultaneously to disjoint subexpressions. All determine the same partial evaluation function.

With three exceptions, the PCF reduction axioms are exactly the left-to-right readings of the equational axioms. The first exception is that we allow renaming of bound variables at any point during reduction, without considering this as a reduction step. From a technical standpoint, the reason is that we cannot always do substitution without renaming. The other exceptions are that we do not have reductions corresponding to the *surjective pairing* axioms (*sp*) for pairs or the *extensionality* axiom (η) for functions. The reason for these two omissions is largely technical and discussed briefly in Section 2.4.1.

- In general, if M is a program (closed expression of observable type) and $eval(M) = N$, then N should be something that cannot be further evaluated. In PCF in particular, if $eval(M) = N$, then N is either a numeral $0, 1, 2, \ldots$ or a boolean constant, depending on the type of M.
- If $eval(M) = N$, then M and N are equivalent according to both the axiomatic and denotational semantics. This is true for PCF, since each term is provably equal to its normal form, and the proof rules are sound for the denotational semantics.

- If M is a program and M has the same denotation as a result N, then the result of executing program M in the operational semantics is N. This is commonly referred to as *computational adequacy* since, if we take the denotational semantics as our guide, it says that we have enough reduction rules to properly determine the value of any program. If computational adequacy fails, we generally look to see if we are missing some reduction rules, or consider whether the denotational semantics gives too many expressions equal meaning. It generally follows from connections between the axiomatic and denotational semantics that if the operational semantics are adequate with respect to the denotational semantics, they are also adequate with respect to the axiomatic semantics.

2.3.5 Equivalence relations defined by each form of semantics

We may summarize the basic connections between axiomatic, operational and denotational semantics by comparing the equivalence relations defined by each one. Since the axiomatic semantics of PCF is a logical system for deriving equations, the obvious equivalence relation is provable equality. The denotational semantics gives us the relation of *denotational equivalence:* two terms are denotationally equivalent if they have the same denotation (for any association of values to free variables). The natural equivalence relation associated with the operational semantics involves substitution of terms into full programs, described below.

We say that two programs of PCF, or any similar language, are operationally equivalent if they have the same value under the operational semantics. In symbolic form, programs M and N are operationally equivalent if $eval(M) \simeq eval(N)$. The "Kleene equation" $eval(M) \simeq eval(N)$ means that either M and N both evaluate to the same term, or neither evaluation is defined. However, this is only an equivalence relation on programs, not arbitrary terms. We extend this relation to terms that may have free variables or non-observable types using an important syntactic notion called a context.

A *context* $\mathcal{C}[\,]$ is a term with a "hole" in it, written as a pair of empty square brackets. An example is the context

$$\mathcal{C}_0[\,] \stackrel{def}{=} \lambda x{:}\, nat.\, x + [\,]$$

If we insert a term into a context, then this is done *without* renaming bound variables. For example, $\mathcal{C}_0[x]$ is $\lambda x{:}\, nat.\, x + x$. We can think of a context as an incomplete program, sitting in the buffer of a text editor. Inserting a program into a context corresponds to using the text editor to fill in the rest of the program. A special case is the empty context $[\,]$, which corresponds to a text editor containing no program at all. In Exercise 2.3.4 below, the empty context is used to show that evaluation respects operational equivalence for programs.

Using contexts, we define *operational equivalence* on arbitrary terms, as follows. Terms M and N of the same type are operationally equivalent if, for every context $\mathcal{C}[\,]$ such that both $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are programs, we have $eval(\mathcal{C}[M]) \simeq eval(\mathcal{C}[N])$. In the literature, operational equivalence is sometimes called "observational equivalence" or "observational congruence."

If we write $M =_{ax} N$ for provable equality in the axiomatic semantics, $M =_{den} N$ for denotational equivalence, and $M =_{op} N$ for operational equivalence, then the minimal requirement on the three semantics, called *adequacy* or *computational adequacy,* is

$$(\forall \textit{ programs } M)\ (\forall \textit{ results } N)\ \ M =_{ax} N \ \text{ iff } \ M =_{den} N \ \text{ iff } \ M =_{op} N.$$

We also expect that for arbitrary terms, the axiomatic semantics are sound for the denotational semantics, and that denotationally equivalent terms are operationally equivalent. These may be

written as the following inclusions between relations on terms:

$$=_{ax} \subseteq =_{den} \subseteq =_{op} .$$

In general, operational equivalence is the coarsest of the three, *i.e.*, more terms are operationally equivalent than denotationally equivalent or provably equivalent. This is not an accidental fact about operational semantics, but a consequence of the way that operational equivalence is defined. For example, if two terms M and N are not operationally equivalent in PCF, then there is some context $\mathcal{C}[\ ]$ with $\mathcal{C}[M]$ and $\mathcal{C}[N]$ different numerals or boolean constants. Consequently, if we equate M and N in the axiomatic semantics, we would have an inconsistent proof system; see Exercise 2.3.2. Similar reasoning applies to the denotational semantics, so in general $=_{ax}$ and $=_{den}$ cannot be any coarser than $=_{op}$. The reason why we usually have $=_{ax} \subseteq =_{den}$ is that we usually justify our axiom system by showing it is sound for denotational semantics (which is just what $=_{ax} \subseteq =_{den}$ means). Since denotational equivalence is usually not recursively enumerable, we do not typically have complete axiom systems (systems where $=_{ax}$ and $=_{den}$ are the same) for languages with recursion (fixed-point operators). A related fact for PCF is given as Corollary 2.5.16 in Section 2.5.5. The relationship between $=_{den}$ and $=_{op}$ is discussed in the next paragraph.

A denotational semantics with $M =_{den} N$ iff $M =_{op} N$, for arbitrary terms, is called *fully abstract.* A fully abstract denotational semantics may be very useful, since reasoning about the denotational semantics therefore allows us to reason about $=_{op}$. This is important since $=_{op}$ is generally difficult to reason about directly, yet it is the most useful form of equivalence for program optimization or transformation. However, it is generally a difficult mathematical problem to construct fully-abstract denotational semantics. The cpo semantics of PCF in Chapter 5, for example, is shown *not* to be fully abstract in Section 5.4.2. On the other hand, it is also shown in Section 5.4.2 that this semantics is fully abstract for an extension of PCF. Some general and historical references on full abstraction are [Cur86, Mil77, Plo77, Sto88].

Exercise 2.3.4 This exercise demonstrates general properties of evaluation and operational equivalence that hold for a variety of languages. For concreteness, however, the problem is stated for PCF. You may assume that the PCF reduction rules are confluent and that *eval* is the partial function from PCF terms to PCF terms such that $eval(M) = N$ iff N is the unique normal form of M. Show that if programs M and N are operationally equivalent, then $eval(M) \simeq eval(N)$. Assuming that all programs with no normal form are operationally equivalent, which is justified for PCF in Exercise 2.5.27, show that if M and N are programs with $eval(M) \simeq eval(N)$, then M and N are operationally equivalent.

Exercise 2.3.5 In Section 2.3.1, we defined programs as closed terms of observable type, and chose *nat* and *bool* as observable. Show that the relation $=_{op}$ remains the same if we change the set of observable types in the following ways.

(a) Only *nat* is considered observable.

(b) We say τ is an observable type if τ is either *nat*, *bool*, or the product $\tau_1 \times \tau_2$ of two observable types τ_1 and τ_2.

Exercise 2.3.6 Show that if *eval* is a partial recursive function on a recursively enumerable language, then operational equivalence is Π^0_2. (This assumes some familiarity with recursive function theory.)

2.4 PCF Reduction and Symbolic Interpreters

2.4.1 Nondeterministic reduction

The operational semantics given by reduction may be regarded as a "nondeterministic symbolic interpreter." However, only the order in which subterms are reduced is nondeterministic, not the final result of reducing all possible subterms. Section 2.4.3 presents a deterministic form of reduction that gives the same final results for terminating PCF programs, and Section 2.4.4 gives a related "parallel" form of reduction. An inequivalent but often used deterministic reduction is considered in Section 2.4.5.

In general, given any set of reduction rules, we say M *reduces to* N *in one step,* and write $M \to N$, if N may be obtained from M by applying one reduction rule to one *subexpression.* For any notion $\to$ of reduction, the corresponding multi-step reduction relation $\twoheadrightarrow$ is defined inductively by

$$M \twoheadrightarrow N \quad \text{if} \quad \begin{array}{l} M =_\alpha N, \text{ or} \\ M \to M' \text{ and } M' \twoheadrightarrow N. \end{array}$$

In other words, $\twoheadrightarrow$ is the least reflexive and transitive relation on α-equivalence classes of expressions that contains one-step reduction.

Example 2.4.1 Some differences between one-step reduction ($\to$), equality ($=$), and multi-step reduction ($\twoheadrightarrow$) are illustrated using PCF below.

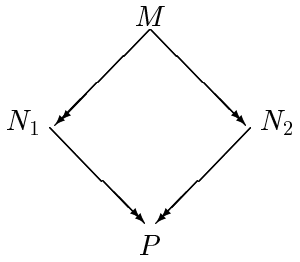
(a)	$(\texttt{if } M \texttt{ then } 3+4 \texttt{ else } 7) = (\texttt{if } M \texttt{ then } 7 \texttt{ else } 3+4)$	since we have $3+4=7$ and equations can be used in either direction
(b)	$(\texttt{if } M \texttt{ then } 3+4 \texttt{ else } 7) \not\to (\texttt{if } M \texttt{ then } 7 \texttt{ else } 3+4)$	since $7 \not\to 3+4$
(c)	$5+2 \twoheadrightarrow 5+2$ but $5+2 \not\to 5+2$	since one-step reduction is not reflexive
(d)	$\texttt{if } (Eq?\,3\,4) \texttt{ then } 5+2 \texttt{ else } 9 \twoheadrightarrow 9$	by first reducing $(Eq?\,3\,4)$

■

It is easy to see from the definition of reduction that if $M \twoheadrightarrow N$, then we can prove the equation $M = N$. It follows that if $M \twoheadrightarrow N$ and $P \twoheadrightarrow N$, we can also prove $M = P$.

For most notions of reduction, a term may be reduced in several ways. The reason is that a term may have many different subterms, each matching the left-hand side of a different reduction rule. We may think of reduction as defining a nondeterministic symbolic interpreter that evaluates an expression M by repeatedly choosing a reduction to apply or choosing to halt. At each step, the "state" of the interpreter is characterized by the expression that is produced. We have a reduction sequence $M \twoheadrightarrow N$ iff this nondeterministic interpreter, executing M, may choose to halt in state N. A necessarily halting state, which is a term N with $N \not\to P$ for any P, is called a *normal form.* Some example PCF normal forms are the truth values *true* and *false*, the numerals $0, 1, 2, \ldots$, a tuple $\langle M_1, \ldots, M_k \rangle$ of distinct normal forms, and any function expression $\lambda x{:}\sigma.M$ with M in normal form. We consider the result of nondeterministic evaluation to be the set of normal forms produced. As we shall see below, there is at most one normal form of any PCF term, in spite of the nondeterministic choices involved.

A useful property of many sets of reduction rules, and PCF reduction in particular, is called *confluence,* or the *Church-Rosser property,* also mentioned in Section 1.3. This property may be sketched graphically as follows.



In this picture, the top two arrows are universally quantified, and the bottom two are existentially quantified. So the picture "says" that whenever $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, there exists a term P with $N_1 \twoheadrightarrow P$ and $N_2 \twoheadrightarrow P$. From Theorem 8.3.24 in Section 8.3.4, we have the following fact about PCF.

Proposition 2.4.2 *PCF reduction is confluent.*

If $\rightarrow$ is determined by directing a set of equational axioms, then confluence implies that an equation is provable iff both terms reduce to a common form. A corollary is that if two distinct terms cannot be reduced at all, they cannot be proved equal. This provides a useful syntactic technique for showing that an equational proof system does not prove all equations. Another simple corollary of confluence is that no term has more than one normal form. These and other general consequences of confluence are proved in Section 3.7.2.

To illustrate the value of confluence in analyzing a proof system, we justify the statement in Section 2.2.3 that (sp) is not derivable from $(proj)$.

Example 2.4.3 Consider the reduction system with only two reduction axioms, $\mathbf{Proj}_1\langle M, N\rangle \rightarrow M$ and $\mathbf{Proj}_2\langle M, N\rangle \rightarrow N$. This system may be proved confluent and therefore an equation is provable from the two equational axioms $\mathbf{Proj}_1\langle M, N\rangle = M$ and $\mathbf{Proj}_2\langle M, N\rangle = N$ using reflexivity and the PCF equational inference rules iff the two terms reduce to a common term. It follows that the equation $x = \langle \mathbf{Proj}_1 x, \mathbf{Proj}_2 x\rangle$ cannot be proved from $\mathbf{Proj}_1\langle M, N\rangle = M$ and $\mathbf{Proj}_2\langle M, N\rangle = N$, since x and $\langle \mathbf{Proj}_1 x, \mathbf{Proj}_2 x\rangle$ are distinct normal forms. ■

While most of us are used to deterministic execution, there are some reasons to consider nondeterministic evaluation. A common reason for the designers of a programming language not to specify the order of evaluation completely is that this allows greater flexibility in optimization. An example of optimization is common subexpression elimination, which effectively reorders the evaluation of parts of an expression. A related issue is parallelism. If we are allowed to evaluate parts of an expression in any order, then it would be possible to evaluate parts of the expression in parallel, without concern for the relative speeds of the processors involved. However, in many languages where the design does not fully determine the order of evaluation, the output of some programs accepted by the compiler may depend on evaluation order. In contrast, confluence of PCF guarantees that the final result does not depend on the order in which subexpressions are evaluated.

The reason we do not use reduction axioms corresponding to equational axioms (η) and (sp) is that these would destroy confluence. It is acceptable to omit these two reductions since these are

not necessary for reducing programs; reduction based on (η) and (sp) only affects the normal forms of terms that either have free variables or are not of observable type. This is an easy consequence of computational adequacy (Corollary 5.4.7). Intuitively, the reason that (η) is unnecessary is that if we have a reduction of the form

$$M \;\twoheadrightarrow\; (\ldots(\lambda x{:}\sigma.\,Mx)\ldots) \to (\ldots(M)\ldots) \;\twoheadrightarrow\; 5$$

for example, then since the result 5 does not contain any function expressions, the function expression M must eventually be applied to an argument or discarded in the reduction $(\ldots(M)\ldots) \to \to 5$. If M is applied, say in a subterm MN, then we would get the same result if we had not used (η) and had stuck with $\lambda x{:}\sigma.\,Mx$ in place of M, since $(\lambda x{:}\sigma.\,Mx)N$ reduces to MN by (β) alone. The situation for surjective pairing is essentially similar.

Example 2.4.4 Consider the expression

$$\texttt{letrec}\ f(x{:}\,nat){:}\,nat = \texttt{if}\ \ Eq?\,x\,0\ \texttt{then}\ 0\ \texttt{else}\ \ x + f(x-1)\ \ \texttt{in}\ \ f\,n$$

which we would expect to produce the natural number $n(n+1)/2$, under any ordinary interpreter. We can see how the nondeterministic PCF interpreter works by eliminating syntactic sugar and reducing. Letting

$$F \;\stackrel{def}{=}\; \lambda f{:}\,nat\to nat.\lambda x{:}\,nat.\texttt{if}\ \ Eq?\,x\,0\ \texttt{then}\ 0\ \texttt{else}\ \ x + f(x-1)$$

and taking $n = 2$, we have the following reduction sequence.

$$\begin{array}{rcl}
(\mathit{fix}\ F)\,2 & \twoheadrightarrow & (\lambda f{:}\,nat\to nat.\lambda x{:}\,nat.\texttt{if}\ \ Eq?\,x\,0\ \texttt{then}\ 0\ \texttt{else}\ \ x + f(x-1))\,(\mathit{fix}\ F)\,2 \\
& \twoheadrightarrow & \texttt{if}\ \ Eq?\,2\,0\ \texttt{then}\ 0\ \texttt{else}\ \ 2 + (\mathit{fix}\ F)(1) \\
& \twoheadrightarrow & 2 + \texttt{if}\ \ Eq?\,1\,0\ \texttt{then}\ 0\ \texttt{else}\ \ 1 + (\mathit{fix}\ F)(0) \\
& \twoheadrightarrow & 2 + 1 + (\mathit{fix}\ F)(0) \\
& \twoheadrightarrow & 2 + 1 + 0 \twoheadrightarrow 3
\end{array}$$

Of course, other reduction sequences may be obtained by choosing *fix*-reduction in place of any of the reduction steps used here. However, by the confluence of PCF reduction, any finite interleaving of *fix*-reductions would still yield an expression that reduces to 3. ∎

Example 2.4.5 Consider the expression $\texttt{letrec}\ f(x{:}\,nat){:}\,nat = f(x+1)\ \texttt{in}\ f\,1$. If we write a similar function declaration in another language like Lisp or Pascal, we would expect $f(n)$ not to terminate (except with stack overflow), regardless of the value of n. This is reflected in the fact that the PCF expression has no normal form. Letting

$$F \;\stackrel{def}{=}\; \lambda f{:}\,nat\to nat.\lambda x{:}\,nat.f(x+1),$$

we can de–sugar the `letrec` expression to $(\lambda f{:}\,nat\to nat.\,f\,1)(\mathit{fix}\ F)$. Then we can reduce to $(\mathit{fix}\ F)1 \to F(\mathit{fix}\ F)1$. At this point, there are two possibilities

$$F(\mathit{fix}\ F)1 \to \begin{cases} F\,(F(\mathit{fix}\ F))\,1 \\ (\lambda x{:}\,nat.(\mathit{fix}\ F)(x+1))\,1 \end{cases}$$

one using *fix*-reduction, and the other β-reduction. By confluence, these two must reduce to some common form. This common form may be found by taking each term and applying the reduction

used to produce the other. Applying β-reduction to the first and *fix*-reduction to the second, we obtain

$$(\lambda x{:}\, nat.(F(\mathit{fix}\ F))(x+1))\,1$$

From here, we could either apply *fix*-reduction or (β). In either case, we obtain a term that reduces to

$$(F^2(\mathit{fix}\ F))2$$

Continuing in this way, we can see that regardless of our choice of reduction, we cannot ever produce an expression that does not contain $(\mathit{fix}\ F)$ as a subterm. Thus $\texttt{letrec}\ f(x{:}\, nat){:}\, nat = f(x+1)\ \texttt{in}\ f\,1$ has no normal form and the nondeterministic PCF interpreter need not halt. Although this brief argument is not a complete proof that the term has no normal form, a rigorous proof can be developed along these lines. A simpler proof method is to use the deterministic evaluator described in Section 2.4.3 (see Exercise 2.4.18). ■

Example 2.4.6 While PCF reduction agrees with the axiomatic semantics of PCF, this form of program execution may be slightly different from what most readers are familiar with. The difference may be illustrated by considering the following program.

$$\begin{array}{l}\texttt{let}\ f(x{:}\, nat){:}\, nat = 3\ \texttt{in}\\ \quad \texttt{letrec}\ g(x{:}\, nat){:}\, nat = g(x+1)\ \texttt{in}\ f(g\,5)\end{array}$$

In most familiar programming languages, this program would not halt, since the call $f(g\,5)$ is compiled or interpreted as code that computes $g\,5$ before calling f. Since the call to g runs forever, the program does not halt. However, in PCF, we can reduce the function call to f without reducing $g\,5$. Since f disregards its argument, we obtain the value 3. Although this explanation of termination uses the ability to reduce any subexpression, this is not essential. In particular, the deterministic evaluator defined in Section 2.4.3 will also reduce the term above to the numeral 3. (See Exercise 2.4.17.) Further comparison of evaluation orders appears in the next section. ■

As mentioned earlier in Section 2.3.4, it is important that the axiomatic semantics respect the operational semantics. Specifically, if we can prove $M = N$, then these two terms should produce the same result when placed inside any PCF program. The reader may wonder about the extensionality axiom (η), which is $\lambda x{:}\,\sigma.Mx = M$ for x not free in M. This equation seems very reasonable when viewed as a statement about mathematical functions, but it is not completely obvious that $\lambda x{:}\,\sigma.Mx$ and M are interchangeable with respect to the nondeterministic PCF interpreter. Since PCF reduction does not include η-reduction, the nondeterministic interpreter is forced to halt on $\lambda x{:}\,\sigma.yx$, for example, which is different from y; clearly $\lambda x{:}\,\sigma.yx$ and y do not have the same normal form if we omit (η). However, when placed inside a context $\mathcal{C}[\ ]$ such that both $\mathcal{C}[\lambda x{:}\,\sigma.yx]$ and $\mathcal{C}[y]$ are programs, we have already observed that $\mathcal{C}[\lambda x{:}\,\sigma.yx]$ and $\mathcal{C}[y]$ will have the same normal form, if any. (The situation for (sp), surjective pairing, is similar.) Thus, the equational proof system of PCF is sound for reasoning about program modification, or interchangeability of program "parts."

Exercise 2.4.7 Show that (fix) reduction may be applied indefinitely to any expression of the form $\texttt{letrec}\ f{:}\,\sigma = M\ \texttt{in}\ N$. Does this mean that recursive functions never "terminate" in PCF?

Exercise 2.4.8 Compute normal forms for the following terms.

(a) $comp\,(\lambda x{:}\,nat.\,x+1)\,(\lambda x{:}\,nat.\,x+1)\,5$, where the definition of natural number function composition is given in Section 2.2.3.

(b) $\texttt{let}\ not = \lambda x{:}\,bool.\,\texttt{if}\ x\ \texttt{then}\ false\ \texttt{else}\ true\ \texttt{in}$
$\texttt{letrec}\ f = \lambda x{:}\,bool.\,\texttt{if}\ x\ \texttt{then}\ true\ \texttt{else}\ f(not\,x)\ \texttt{in}$
$f\,false$

Exercise 2.4.9 Show that taken together the following three reduction rules for typed lambda calculus with fixed points are not confluent.

$$(\beta) \qquad (\lambda x{:}\,\sigma.M)N \to [N/x]M$$

$$(\eta) \qquad \lambda x{:}\,\sigma.Mx \to M, \quad x \text{ not free in } M$$

$$(fix)_{alt} \qquad fix_\sigma M \to M(fix_\sigma M)$$

More specifically, find a term M that may be reduced in one step to N or P, where N and P cannot be reduced to a common term. *Hint:* It is possible to find such an M which has no (β) redexes. Your counterexample will not work if we use (fix) reduction, $fix_\sigma \to \lambda f{:}\,\sigma\to\sigma.\,f(fix_\sigma f)$, as in the text.

2.4.2 Reduction strategies

We will discuss various deterministic interpreters using the notion of reduction strategy. A *reduction strategy* is a partial function F from terms to terms with the property that if $F(M) = N$, then $M \to N$. This is called a "strategy" since the function may be used to choose one out of many possible reductions. For any reduction strategy F, we may define a *partial evaluation function* $eval_F{:}\,PCF \to PCF$ on PCF expressions by

$$eval_F(M) = \begin{cases} M & \text{if } F(M) \text{ is not defined} \\ N & \text{if } F(M) = M' \text{ and } eval_F(M') = N \end{cases}$$

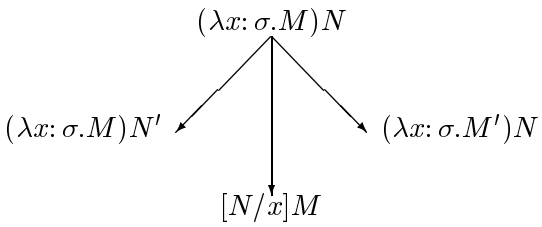
The evaluation function $eval_F$ is the mathematical equivalent of a deterministic interpreter that repeats single reduction steps, following the strategy F, until this strategy cannot be followed any further (*i.e.*, we reach a term where the partial function F is undefined). If the strategy can be followed indefinitely from M, then the function $eval_F$ will be undefined on M, and an actual interpreter following F would run forever. (See Exercise 2.4.10 below for further discussion of the way $eval_F$ is defined from F.) As a notational convenience, we will omit the subscript F if the reduction strategy is clear from context or irrelevant.

In general, we will be interested in reduction strategies that choose a reduction whenever possible, at least if the term is a program. For such reduction strategies, the evaluation $eval(M)$ of a program M will either be the normal form of M, or undefined. In the terminology of [Bar84], the reduction strategies we consider are *one-step* reduction strategies, since they choose a single reduction step based on the form of the term alone. For discussion of multi-step reduction strategies, the reader may consult [Bar84].

It is possible to define a "brute force" reduction strategy F with the property that for any term M, $eval(M)$ is either the normal form of M or undefined. Given M, we can compute $F(M)$ by enumerating (in parallel) all reduction paths from M and seeing if any produces a normal form. If we find one reduction path, say $M \to M_1 \to M_2 \to \ldots \to N$, leading to a normal form, we let $F(M) = M_1$ be the first term along this reduction path. This is not a very satisfying reduction

strategy, however, since it may take a long time to compute $F(M)$; it is not even decidable whether $F(M)$ is defined. However, we will see below that there is an efficiently computable reduction strategy which will always find the normal form of a term.

An important property of PCF is that the idealized nondeterministic interpreter may be implemented deterministically in an efficient manner. More specifically, there is an efficiently computable reduction strategy F such that whenever M reduces to a normal form N, we have $eval_F(M) = N$. This reduction strategy is called *left-most outer-most* or, more simply, *left-most reduction*. It is defined precisely in the next section. Another name that is sometimes used is *call-by-name* reduction, but this is sometimes confusing since call-by-name is also used to refer to a parameter-passing mechanism implemented by "thunks" (see [ASU86], for example). The difference between left-most reduction and other alternatives may be illustrated by considering an expression $(\lambda x{:}\,\sigma.M)N$, where $M \to M'$ and $N \to N'$. There are three possible ways to reduce this expression:



The first alternative is a form of what is called *eager reduction,* or *call-by-value.* (Like call-by-name, call-by-value has other meanings.) The second is called *lazy* or *left-most* reduction, and the third may be regarded as an attempt to "optimize" the function $\lambda x{:}\,\sigma.M$ before supplying the argument N. There is no guarantee that such an "optimization" shortens the reduction to normal form.

An eager strategy, which may choose to reduce a function argument before doing β-reduction, sometimes leads to shorter reductions. The reason is that a function body M may have several occurrences of the formal parameter x. In this case, substituting N for x duplicates N, and so any later reduction of N must be repeated for each of the resulting copies. On the other hand, if x does not occur in M, then there is no point in reducing N. An extreme but important case occurs when N has no normal form. In this situation, an eager interpreter will not terminate on $(\lambda x{:}\,\sigma.M)N$, since it will reduce N indefinitely. However, if N is not used in the final result (*e.g.*, x does not occur in M), a left-most interpreter may terminate without reducing N. Some common optimization techniques, such as "code motion" and "in-line substitution" fall into the category corresponding to the third alternative in the picture above.

Since most of the PCF programming examples we consider are aimed at producing a normal form as a final result, we can think of the PCF interpreter as either a nondeterministic reduction machine, or a deterministic evaluator following a left-most reduction strategy. The latter alternative is particularly useful if we wish to show that a PCF expression has no normal form. In doing hand calculations, however, the nondeterministic execution model lets us choose reduction steps which lead more quickly to normal form.

Exercise 2.4.10 The definition of $eval_F$ at the beginning of this section appears to be a definition by induction. However, it is not clear that the definition is well-founded. Give an equivalent inductive definition of the following form. For each natural number n, define a partial function $eval^{[n]}$. Intuitively, $eval^{[n]}(M)$ will be defined if M reduces to normal form in n (or fewer) steps, and undefined otherwise. Your definition of the $eval^{[n]}$'s should use induction on n. Then define

$eval_F$ as the union of the $eval^{[n]}$'s. (Recall that a partial function is a set of ordered pairs, so it makes sense to take the union of a set of partial functions.)

2.4.3 The left-most and lazy reduction strategies

We will define a deterministic interpreter by defining a specialized reduction relation $\overset{left}{\to}$ that is a subset of nondeterministic PCF reduction. An optimization of left-most reduction, called *lazy reduction* is also described in this section. While lazy reduction does not find a normal form for every term that has one, lazy reduction has the same effect as left-most reduction for programs (closed terms of observable type) that reduce to results.

Intuitively, left-most reduction works by applying the reduction step that involves the left-most symbol in the term. Another way to describe this is that we compare two possible reductions by looking at the set of symbols in the term that match the left-hand side of the appropriate reduction axiom. In left-most reduction, we apply the reduction that involves the left-most symbol possible. The relation $\overset{left}{\to}$ is defined in Table 2.3 by a set of inference rules, following a style that is commonly called *structured operational semantics.* The definition is structured in the sense that each rule applies to exactly one form of term, so that we can see which rule applies by examining the structure of a term. The first rule in the table says that if $M \to N$ is a reduction axiom (these are collected in Table 2.2), then this is a left-most reduction step. In other words, if M is a redex, then the left-most reduction of M is to reduce M. The remaining rules describe reductions on subterms. An intuitive way to read a rule such as

$$\frac{M \overset{left}{\to} M'}{MN \overset{left}{\to} M'N}$$

is in a counter-clockwise direction, starting from the bottom left. If we want to reduce an application MN, then we perform the left-most reduction step in M. If this results in a term M', then the left-most reduction step from MN is the one taking us from MN to $M'N$. Multi-step left-most reduction, written $\overset{left}{\to\!\!\to}$, is the reflexive and transitive closure of $\overset{left}{\to}$, as usual.

In order to simplify the table, some side conditions are omitted. The convention to be used in reading the rules in Table 2.3 is that the reductions on subterms are to be applied only if a reduction axiom does not apply to the entire term. Thus the left-most reduction of $(\lambda x{:}\,\sigma.M)N$ yields $[N/x]M$, as opposed to some reduction of $\lambda x{:}\,\sigma.M$. An alternate presentation of left-most reduction may be given using what are called "evaluation contexts," which are discussed in Section 2.5.6.

For any M, there is at most one term N with $M \overset{left}{\to} N$. This means that $\overset{left}{\to}$ is a reduction strategy, and we may write $\overset{left}{\to}(M) = N$ in place of $M \overset{left}{\to} N$ when desired. However, the notation $M \overset{left}{\to} N$ generally seems more suggestive.

Example 2.4.11 The left-most reduction of the term $((\lambda x{:}\,nat.\,\lambda y{:}\,nat.\,x + y)\,7)\,5 + (\lambda x{:}\,nat.\,x)\,3$

Axioms

$$\frac{M \to N}{M \overset{left}{\to} N} \qquad M \to N \ \textit{is a reduction axiom}$$

Subterm Rules

nat and *bool*

$$\frac{M \overset{left}{\to} M'}{M + N \overset{left}{\to} M' + N} \qquad \frac{M \overset{left}{\to} M'}{N + M \overset{left}{\to} N + M'} \qquad N \ \textit{a normal form}$$

$$\frac{M \overset{left}{\to} M'}{Eq?MN \overset{left}{\to} Eq?M'N} \qquad \frac{M \overset{left}{\to} M'}{Eq?NM \overset{left}{\to} Eq?NM'} \qquad N \ \textit{a normal form}$$

$$\frac{M \overset{left}{\to} M'}{\texttt{if } M \texttt{ then } N \texttt{ else } P \overset{left}{\to} \texttt{if } M' \texttt{ then } N \texttt{ else } P}$$

$$\frac{N \overset{left}{\to} N'}{\texttt{if } M \texttt{ then } N \texttt{ else } P \overset{left}{\to} \texttt{if } M \texttt{ then } N' \texttt{ else } P} \qquad M \ \textit{a normal form}$$

$$\frac{P \overset{left}{\to} P'}{\texttt{if } M \texttt{ then } N \texttt{ else } P \overset{left}{\to} \texttt{if } M \texttt{ then } N \texttt{ else } P'} \qquad M,N \ \textit{normal forms}$$

Pairs

$$\frac{M \overset{left}{\to} M'}{\langle M, N\rangle \overset{left}{\to} \langle M', N\rangle} \qquad \frac{N \overset{left}{\to} N'}{\langle M, N\rangle \overset{left}{\to} \langle M, N\rangle} \qquad M \ \textit{a normal form}$$

$$\frac{M \overset{left}{\to} M'}{\mathbf{Proj}_i M \overset{left}{\to} \mathbf{Proj}_i M'}$$

Functions

$$\frac{M \overset{left}{\to} M'}{MN \overset{left}{\to} M'N} \qquad \frac{N \overset{left}{\to} N'}{MN \overset{left}{\to} MN'} \qquad M \ \textit{a normal form}$$

$$\frac{M \overset{left}{\to} M'}{\lambda x{:}\sigma.\, M \overset{left}{\to} \lambda x{:}\sigma.\, M'}$$

Subterm rules apply only when no axiom applies to the entire term. For example, $\mathbf{Proj}_i M \overset{left}{\to} \mathbf{Proj}_i M'$ only when M is not of the form $\langle M_1, M_2\rangle$ and MN reduces to $M'N$ only when M is not of the form $\lambda x{:}\sigma.\, M_1$.

Table 2.3: Left-most reduction for PCF.

is written below, with the left-most redex underlined in each step.

$$\begin{array}{ll} & \underline{((\lambda x\colon nat.\ \lambda y\colon nat.\ x+y)\ 7)}\ 5 + (\lambda x\colon nat.\ x)\ 3 \\ \overset{left}{\rightarrow} & \underline{(\lambda y\colon nat.\ 7+y)\ 5} + (\lambda x\colon nat.\ x)\ 3 \\ \overset{left}{\rightarrow} & (\underline{7+5}) + (\lambda x\colon nat.\ x)\ 3 \\ \overset{left}{\rightarrow} & 12 + \underline{(\lambda x\colon nat.\ x)\ 3} \\ \overset{left}{\rightarrow} & \underline{12+3} \\ \overset{left}{\rightarrow} & 15 \end{array}$$

A second example, not producing a numeral, follows.

$$\begin{array}{ll} & \underline{(\lambda x\colon nat.\ \lambda y\colon nat.\ x + (x+y))\ ((\lambda z\colon nat.\ z)\ 12)} \\ \overset{left}{\rightarrow} & \lambda y\colon nat.(\underline{(\lambda z\colon nat.\ z)\ 12}) + (((\lambda z\colon nat.\ z)\ 12) + y) \\ \overset{left}{\rightarrow} & \lambda y\colon nat.\ 12 + ((\underline{(\lambda z\colon nat.\ z)\ 12}) + y) \\ \overset{left}{\rightarrow} & \lambda y\colon nat.\ 12 + (12 + y) \end{array}$$

The last term is a normal form. Neither sum can be reduced since the parentheses associate the sums to the right. If we had $(12+12)+y$ instead, the body of this lambda term could be reduced to $24+y$. ■

As stated in the following proposition, left-most reduction will find the normal form of any term that has one.

Proposition 2.4.12 *Let M be a PCF term of any type. Then for any normal form N, we have $M \overset{left}{\rightarrow\!\!\!\!\rightarrow} N$ iff $M \twoheadrightarrow N$.*

This follows from Theorem 8.3.26 in Section 8.3.4, which gives a general condition guaranteeing completeness of left-most reduction for PCF-like languages.

Since $\overset{left}{\rightarrow}$ is a reduction strategy, we may apply the general definition of evaluation function from Section 2.4.2 to obtain an evaluation (partial) function $eval_{\overset{left}{\rightarrow}}$. For typographical reasons, we will write $eval_{left}$ instead of $eval_{\overset{left}{\rightarrow}}$. A corollary of Proposition 2.4.12 is that this evaluator finds the normal form of any term that has one.

Proposition 2.4.13 *Let M be a PCF term of any type. Then $eval_{left}(M) = N$ iff $M \twoheadrightarrow N$ and N is a normal form.*

If we only want Propositions 2.4.12 and 2.4.13 to hold for PCF programs (closed terms of observable type), then we may omit many of the reductions on subterms. The result is *lazy reduction,* which is far more commonly implemented in practice than left-most reduction. This is given by axioms and inference rules in Table 2.4. As usual, we write $\overset{lazy}{\rightarrow\!\!\!\!\rightarrow}$ for the reflexive and transitive closure of $\overset{lazy}{\rightarrow}$. We say a term M is in *lazy normal form* if there is no N with $M \overset{lazy}{\rightarrow} N$. Note that the lazy normal forms of types *nat* and *bool* are exactly the normal forms of these types. However, the lazy normal forms of function or product types may have subterms that are not in normal form (lazy or otherwise), as illustrated in the following example.

Axioms

$$\frac{M \to N}{M \overset{lazy}{\to} N} \qquad M \to N \ \textit{is a reduction axiom}$$

Subterm Rules

nat and *bool*

$$\frac{M \overset{lazy}{\to} M'}{M + N \overset{lazy}{\to} M' + N} \qquad \frac{M \overset{lazy}{\to} M'}{n + M \overset{lazy}{\to} n + M'} \quad n \ \textit{a numeral}$$

$$\frac{M \overset{lazy}{\to} M'}{Eq?MN \overset{lazy}{\to} Eq?M'N} \qquad \frac{M \overset{lazy}{\to} M'}{Eq?nM \overset{lazy}{\to} Eq?nM'} \quad n \ \textit{a numeral}$$

$$\frac{M \overset{lazy}{\to} M'}{\texttt{if } M \texttt{ then } N \texttt{ else } P \overset{lazy}{\to} \texttt{if } M' \texttt{ then } N \texttt{ else } P}$$

Pairs

$$\frac{M \overset{lazy}{\to} M'}{\mathbf{Proj}_i M \overset{lazy}{\to} \mathbf{Proj}_i M'}$$

Functions

$$\frac{M \overset{lazy}{\to} M'}{MN \overset{lazy}{\to} M'N}$$

Table 2.4: Lazy reduction for PCF.

Example 2.4.14 The lazy reduction of the term $((\lambda x\colon nat.\, \lambda y\colon nat.\, x + y)\, 7)\, 5 + (\lambda x\colon nat.\, x)\, 3$ is exactly the left-most reduction given in Example 2.4.11. For the second term given in Example 2.4.11, lazy reduction terminates sooner, since lazy reduction does not change any subexpression inside the scope of a λ or inside a pair of the form $\langle \cdot, \cdot \rangle$.

$$\begin{array}{l}(\lambda x\colon nat.\, \lambda y\colon nat.\, x + (x + y))\, ((\lambda z\colon nat.\, z)\, 12) \\ \overset{lazy}{\to} \quad \lambda y\colon nat.((\lambda z\colon nat.\, z)\, 12) + (((\lambda z\colon nat.\, z)\, 12) + y)\end{array}$$

The last term is a lazy normal form, since there is no lazy reduction of this term. ∎

The main property of lazy reduction, in comparison with left-most reduction, follows from the following proposition.

Proposition 2.4.15 *If M is a closed PCF term that does not have the form $\lambda x\colon \sigma.\, M_1$ or $\langle M_1, M_2 \rangle$, then for any term N, we have $M \overset{lazy}{\to} N$ iff $M \overset{left}{\to} N$.*

Proof It is easy to see that if $M \overset{lazy}{\to} N$ then $M \overset{left}{\to} N$. We prove the converse by induction on the proof (using the rules that appear in Table 2.3) that $M \overset{left}{\to} N$. The base case is that $M \to N$ is a reduction axiom, in which case $M \overset{lazy}{\to} N$.

The induction step for each inference rule that appears in both in Table 2.3 and Table 2.4 is straightforward. The only observation that is required, in the case $N + M \overset{left}{\to} N + M'$ with N a normal form, is that since N is closed, it must be a numeral. Therefore, $N + M \overset{lazy}{\to} N + M'$, and similarly for $Eq?\, N\, M \overset{left}{\to} Eq?\, N\, M'$.

If the left-most reduction is by

$$\texttt{if } M \texttt{ then } N \texttt{ else } P \overset{left}{\to} \texttt{if } M \texttt{ then } N' \texttt{ else } P$$

with M a normal form, then the assumption that the entire term is closed means that M must be *true* or *false*. Therefore, the left-most reduction should have been to eliminate the conditional and proceed with N or P.

The hypothesis that M is neither a λ-abstraction or pair rules out the cases for left-most reduction inside a λ-abstraction or pair.

The final case is the reduction of a closed term $MN \stackrel{left}{\to} MN'$ where M is a normal form. But since M must be closed, it must have the form $M \equiv \lambda x{:}\sigma.\, M_1$ or $M \equiv \mathit{fix}\, M_1$. In either case, this contradicts the assumption that $MN \stackrel{left}{\to} MN'$, concluding the proof. ∎

Corollary 2.4.16 *If P is a PCF program and R a result (closed normal form of the same type), then $P \stackrel{lazy}{\to\!\!\!\!\to} R$ iff $P \stackrel{left}{\to\!\!\!\!\to} R$.*

Exercise 2.4.17 Show, by performing left-most reduction, that the deterministic evaluator halts with value 3 on the program given in Example 2.4.6.

Exercise 2.4.18 Show that the PCF expression $\texttt{letrec}\ f(x{:}\,nat){:}\,nat\ =\ f(x+1)\ \texttt{in}\ f\,1$, described in Example 2.4.5, has no normal form.

2.4.4 Parallel reduction

The general idea behind parallel evaluation of PCF is that whenever we can reduce either of two subterms independently, we may reduce both simultaneously. We might reach a normal form faster this way, although of course this is not guaranteed. However, there is no harm in doing so: since PCF reduction is confluent, we will not reach a different normal form by doing extra reduction in parallel.

We will define a parallel reduction relation $\Rightarrow$ from nondeterministic reduction using two inference rules. The first rule below says that a single reduction may be considered a special case of parallel reduction.

$$\frac{M \to N}{M \Rightarrow N}$$

where we intend $M \to N$ to indicate that N results from a single reduction to M itself or one of its subterms. If $\mathcal{C}[\ \ldots\ ]$ is a context with places for k terms to be inserted, then a term of the form $\mathcal{C}[M_1, \ldots, M_k]$ will have nonoverlapping subterms $M_1, \ldots, M_k$. In this case, it makes sense to reduce these subterms simultaneously, possibly in parallel. This is expressed by the following rule.

$$\frac{M_1 \Rightarrow N_1, \ldots, M_k \Rightarrow N_k}{\mathcal{C}[M_1, \ldots, M_k] \ \Rightarrow\ \mathcal{C}[N_1, \ldots, N_k]}$$

As usual, multi-step parallel reduction $\Rightarrow\!\!\!\!\to$ is the reflexive and transitive closure of single-step parallel reduction.

This parallel reduction relation gives us a nondeterministic form of reduction with the property that $M \to\!\!\!\!\to N$ by ordinary PCF reduction iff $M \Rightarrow\!\!\!\!\to N$ by parallel reduction steps. It is also possible to define parallel reduction strategies which give the same normal form as sequential reduction. This may be done in a way that maximizes parallelism, optionally up to some bound which we may think of as the maximum feasible number of parallel processes for some parallel architecture. However, we will not go into the details.

Exercise 2.4.19 This exercise asks you to compare parallel and sequential reduction.

(a) Prove that $M \Rightarrow\!\!\!\!\!\Rightarrow N$ iff $M \twoheadrightarrow N$. You may use the fact that for any context $\mathcal{C}[\,]$, the term $\mathcal{C}[M_1, \ldots, M_k]$ has independent subterms $M_1, \ldots, M_k$ which may be replaced separately by $N_1, \ldots, N_k$ to yield $\mathcal{C}[N_1, \ldots, N_k]$.

(b) Suppose $M \twoheadrightarrow N$ and N is a normal form. Use confluence of $\twoheadrightarrow$ and part (a) to show that if $M \Rightarrow\!\!\!\!\!\Rightarrow P$, then $P \Rightarrow\!\!\!\!\!\Rightarrow N$.

(c) Show by counterexample that part (b) fails if $\twoheadrightarrow$ is not confluent. In other words, give a single-step reduction relation $\rightarrow$ on some set of terms such that there exist M, N and P with $M \twoheadrightarrow N$, term N not reducible, and $M \Rightarrow\!\!\!\!\!\Rightarrow P$ with P not reducible to N by either parallel or sequential reduction.

2.4.5 Eager PCF

Left-most reduction matches the axiomatic semantics of PCF and allows nondeterministic or parallel implementations. However, most implementations of existing programming languages do not follow this evaluation order. (An exception is Algol 60, whose "copy rule" procedure-call semantics match PCF reduction; more modern examples are Haskell and Miranda, as noted in Section 2.1.) The more common order is eager evaluation, which does *not* match the PCF axiomatic semantics, as illustrated in Example 2.4.6 (see also Example 2.4.21 below). In this section, we give a precise definition of eager evaluation for PCF and consider a few properties. Eager PCF is also discussed in Section 2.6.4 in connection with "explicit lifting."

An important idea in eager reduction is the notion of *value,* which is a term that is not further reduced by eager reduction. The terms that are not values are function applications and pairs of non-values. The main difference between eager reduction and other reduction strategies considered in this chapter is that under eager reduction, we only apply β-reduction and $\mathbf{Proj}_i$-reduction when a function argument is a value. There is also a change in *fix* reduction to halt reduction of the argument to *fix*, as explained below. The name "value" comes from the fact that values are considered "fully evaluated." One-step *eager* or *call-by-value reduction* is defined in Table 2.5. Like lazy reduction, eager reduction is only intended to produce a numeral or boolean constant from a full (closed) program; it is not intended to produce normal forms or fully reduce open terms that may have free variables. This can be seen in the rules for addition: there is no eager reduction from a term of the form $x + M$, even though M could be further reduced.

A significant restriction on eager PCF is that *we only have fixed-point operator fix_σ for function types $\sigma = \sigma_1 \to \sigma_2$.* The intuitive reason is that since a recursively-defined natural number, boolean or pair would be fully evaluated before any function could be applied to it, any such expression would cause the program containing it to diverge. In more detail, reduction of closed terms only halts on lambda abstractions, pairs of values and constants. While lambda abstractions and pairs of values could contain occurrences of *fix*, analysis of a pair $\langle V_1, V_2 \rangle$ with V_1 and V_2 values shows that any occurrence of *fix* must occur inside a lambda abstraction. Therefore, the only values that could involve recursion are functions. If we changed the system so that a pair $\langle M, N \rangle$ were a value, for any terms M and N, then it would make sense to have a fixed-point operator fix_σ for each product type $\sigma = \sigma_1 \times \sigma_2$.

It is easy to verify, by examination of Table 2.5, that for any M, there is at most one N with $M \stackrel{eager}{\rightarrow} N$. Since no values are reduced, $\stackrel{eager}{\rightarrow}$ is a partial function on terms whose domain contains only non-values.

Values

V is a *value* if V is a constant, variable,
lambda abstraction or pair of values.

$delay_{\sigma\to\tau}[M] \stackrel{def}{=} \lambda x{:}\sigma.\, Mx \quad x$ not free in $M{:}\sigma\to\tau$

Axioms

$(\lambda x{:}\sigma.M)V \stackrel{eager}{\to} [V/x]M$ $\quad V$ a value

$\mathbf{Proj}_i\langle V_1, V_2\rangle \stackrel{eager}{\to} V_i$ $\quad V_1,\ V_2$ values

$fix_{\sigma\to\tau}\, V \stackrel{eager}{\to} V(delay_{\sigma\to\tau}[fix_{\sigma\to\tau}\, V])$ $\quad V$ a value

$$0+0 \stackrel{eager}{\to} 0,\ 0+1 \stackrel{eager}{\to} 1,\ \ldots,\ 3+5 \stackrel{eager}{\to} 8,\ \ldots$$

$$Eq?\, n\, n \stackrel{eager}{\to} true,\ Eq?\, n\, m \stackrel{eager}{\to} false \qquad n,m \text{ distinct numerals}$$

$$\texttt{if } true \texttt{ then } M \texttt{ else } N \stackrel{eager}{\to} M,\ \texttt{if } false \texttt{ then } M \texttt{ else } N \stackrel{eager}{\to} N$$

Subterm Rules

nat

$$\frac{M \stackrel{eager}{\to} M'}{M+N \stackrel{eager}{\to} M'+N} \qquad \frac{M \stackrel{eager}{\to} M'}{n+M \stackrel{eager}{\to} n+M'} \quad n \text{ a numeral}$$

bool

$$\frac{M \stackrel{eager}{\to} M'}{Eq?MN \stackrel{eager}{\to} Eq?M'N} \qquad \frac{M \stackrel{eager}{\to} M'}{Eq?nM \stackrel{eager}{\to} Eq?nM'} \quad n \text{ a numeral}$$

$$\frac{M \stackrel{eager}{\to} M'}{\texttt{if } M \texttt{ then } N \texttt{ else } P \stackrel{eager}{\to} \texttt{if } M' \texttt{ then } N \texttt{ else } P}$$

Pairs

$$\frac{M \stackrel{eager}{\to} M'}{\langle M, N\rangle \stackrel{eager}{\to} \langle M', N\rangle} \qquad \frac{N \stackrel{eager}{\to} N'}{\langle V, N\rangle \stackrel{eager}{\to} \langle V, N\rangle} \quad V \text{ a value}$$

$$\frac{M \stackrel{eager}{\to} M'}{\mathbf{Proj}_i M \stackrel{eager}{\to} \mathbf{Proj}_i M'}$$

Functions

$$\frac{M \stackrel{eager}{\to} M'}{MN \stackrel{eager}{\to} M'N} \qquad \frac{N \stackrel{eager}{\to} N'}{VN \stackrel{eager}{\to} VN'} \quad V \text{ a value}$$

Table 2.5: Eager PCF reduction.

The use of *delay* and *fix* reduction requires some explanation. Since eager reduction does not reduce under a lambda abstraction, a term of the form $delay_{\sigma\to\tau}[M] \equiv \lambda x{:}\sigma.\, Mx$ will not be reduced. This explains why we call the mapping $M \mapsto \lambda x{:}\sigma.\, Mx$ "delay." The reason that *delay* is used in *fix* reduction is to halt reduction of a recursive function definition until an argument is supplied. An example is given in Exercise 2.4.24; see also Exercise 2.4.25.

Example 2.4.20 Some of the characteristics of eager reduction are illustrated by reducing the term

$$(\mathit{fix}\ (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y))\ ((\lambda z{:}\,nat.\ z+1)\,2)$$

to a value. This is only a trivial fixed point, but the term does give us a chance to see the order of evaluation. The first step in determining which reduction to apply is to check the entire term. This has the form of an application MN, but neither the function M nor the argument N is a value. According to the rule at the bottom left of Table 2.5, we must eager-reduce the function $M \equiv \mathit{fix}\ (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y)$. Since the argument to *fix* is a value, we apply the reduction axiom for *fix*. Followed by β-reduction, since $delay[\ldots]$ is a value, this gives us a function value (lambda abstraction) as follows:

$$\begin{array}{ll} & \mathit{fix}\ (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y) \\ \overset{eager}{\to} & (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y)\ (delay[\mathit{fix}\ (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y)]) \\ \overset{eager}{\to} & \lambda y{:}\,nat.\ y \end{array}$$

Note that without $delay[\,]$, the eager strategy would have continued *fix* reduction indefinitely.

We have now reduced the original term MN to a term VN with function $V \equiv \lambda y{:}\,nat.\ y$ a value but argument $N \equiv (\lambda z{:}\,nat.\ z+1)\,2$ not a value. According to the rule at the bottom right of Table 2.5, we must eager-reduce the function argument. Since $\lambda z{:}\,nat.\ z+1$ and 2 are both values, we can apply β-reduction, followed by reduction of the sum of two numerals:

$$(\lambda z{:}\,nat.\ z+1)\,2 \quad \overset{eager}{\to} \quad 2+1 \quad \overset{eager}{\to} \quad 3$$

This now gives us the application $(\lambda y{:}\,nat.\ y)\,3$ of one value to another, which can be reduced to 3. ∎

Example 2.4.21 Divergence of eager evaluation can be seen in the term

$$\begin{array}{l} \texttt{let}\ f(x{:}\,nat){:}\,nat = 3\ \texttt{in} \\ \quad \texttt{letrec}\ g(x{:}\,nat){:}\,nat = g(x+1)\ \texttt{in}\ f(g\,5) \end{array}$$

from Example 2.4.6. Exercise 2.4.22 asks you to reduce this term. Since we can easily prove that this term is equal to 3, this term shows that the equational proof system of PCF is not sound for proving equivalence under eager evaluation. It is possible to develop an alternative proof system for eager equivalence, restricting β-conversion to the case where the function argument is a value, for example. However, due to restrictions on replacing subexpressions, the resulting system is more complicated than the equational system for PCF. ∎

The deterministic call-by-value evaluator $eval_V$ is defined from $\overset{eager}{\to}$ in the usual way. Since the partial function $\overset{eager}{\to}$ selects a reduction step iff the term is not a value, $eval_V$ may also be defined as follows.

$$eval_V(M) = \begin{cases} M & \text{if } M \text{ is a value} \\ N & \text{if } M \overset{eager}{\to} M' \text{ and } eval_V(M') = N \end{cases}$$

There seem to be two main reasons for implementing eager rather than left-most (or lazy) evaluation in practice. The first is that even for a purely functional language such as PCF (*i.e.*, a language without assignment or other operations with side effects), the usual implementation of left-most reduction is less efficient. The reason for the inefficiency appears to be that when an argument such as $f\,x$ is passed to a function g, it is necessary to pass a pointer to the code for f and to keep a record of the appropriate lexical environment. As a result, there is significantly more overhead to implementing function calls. It would be simpler just to call f with argument x immediately and then pass the resulting integer, for example, to g. The second reason that left-most reduction is not usually implemented has to do with side effects. As illustrated by the many tricks used in Algol 60, the combination of left-most evaluation and assignment is often confusing. In addition, in the presence of side effects, left-most evaluation does not coincide with nondeterministic or parallel evaluation. The reason is that the order in which assignments are made to a variable will generally affect the program output. We cannot expect different orders of evaluation to produce the same result. Since most languages in common use include assignment, many of the advantages of left-most or lazy evaluation are lost.

Exercise 2.4.22 Show, by performing eager reduction, that the eager interpreter does not halt on the program given in Example 2.4.21.

Exercise 2.4.23 Assuming appropriate reduction rules for the function symbols used in the factorial function *fact* of Section 2.2.5, show that $fact\,3 \stackrel{eager}{\rightarrow\!\!\!\!\rightarrow} 6$.

Exercise 2.4.24 An alternate eager reduction for *fix* might be

$$fix_{\,\sigma}\,(\lambda x{:}\,\sigma.\,M) \stackrel{eager}{\rightarrow} [fix_{\,\sigma}\,(\lambda x{:}\,\sigma.\,M)/x]M$$

Find a term $\lambda x{:}\,\sigma.\,M$ where eager reduction as defined in Table 2.5 would terminate, but the alternate form given by this rule would not. Explain why, for programs of the form `letrec` $f(x{:}\,\sigma) = M$ `in` N with *fix* not occurring in M or N, it does not seem possible to distinguish between the two possible eager reductions for *fix*.

Exercise 2.4.25 Eager or call-by-value reduction may also be applied to untyped lambda calculus. With ordinary (left-most) reduction, an untyped fixed-point operator, Y, may be written $Y \stackrel{def}{=} \lambda f.(\lambda x.\,f(xx))(\lambda x.\,f(xx))$. Under eager, or call-by-value reduction, the standard untyped fixed-point operator is written $Z \stackrel{def}{=} \lambda f.(\lambda x.\,f(delay[xx]))(\lambda x.\,f(delay[xx]))$ where $delay[U] \stackrel{def}{=} \lambda z.\,Ux$ for x not free in untyped term U. Show that for $M \stackrel{def}{=} \lambda x.\,\lambda y.\,y$, the application YM lazy or left-most reduces to a term beginning with a lambda, and similarly for ZM using untyped eager reduction. What happens when we apply eager reduction to YM?

2.5 PCF Programming Examples, Expressive Power and Limitations

2.5.1 Records and n-tuples.

Records are a useful data type in Pascal and many other languages. We will see that record expressions may be translated into PCF using pairing and projection. Essentially, a record is an aggregate of one or more *components,* each with a different label. We can combine any expressions $M_1{:}\,\sigma_1, \ldots, M_k{:}\,\sigma_k$ and form a record $\{\ell_1 = M_1, \ldots, \ell_k = M_k\}$ whose ℓ_i component has the value of M_i. The type of this record may be written $\mathbf{record}\{\ell_1{:}\,\sigma_1, \ldots, \ell_k{:}\,\sigma_k\}$. We select the ℓ_i component of any record r of this type $(0 \leq i \leq k)$ using the "dot" notation $r.\ell_i$. For example, the record $\{A = 3, B = \mathit{true}\}$ with components labeled A and B has type $\mathbf{record}\{A{:}\,\mathit{nat}, B{:}\,\mathit{bool}\}$. The A component is selected by writing $\{A = 3, B = \mathit{true}\}.A$. In general, we have the equational axiom

$$(\mathit{record}) \qquad \{\ell_1 = M_1, \ldots, \ell_k = M_k\}.\ell_i = M_i$$

for component selection, which may be used as a reduction rule from left to right.

One convenient aspect of records is that component order does not matter (in the syntax just introduced and in most languages). We can access the A component of a record r without having to remember which component we listed first in defining r. In addition, we may choose mnemonic names for labels, as in

$$\texttt{let}\ \mathit{person} = \mathbf{record}\{\mathit{name}{:}\,\mathit{string}, \mathit{age}{:}\,\mathit{nat}, \mathit{married}{:}\,\mathit{bool}, \ldots\}$$

However, these are the only substantive differences between records and cartesian products; by choosing an ordering of components, we can translate records and record types into pairs and product types of PCF. For records with two components, such as $\mathbf{record}\{A{:}\,\mathit{nat}, B{:}\,\mathit{bool}\}$, we can translate directly into pairs by choosing one component to be first. However, for records with more than two components, we must translate into "nested" pairs.

To simplify the translation of n-component records into PCF, we will define n-tuples as syntactic sugar. For any sequence of types $\sigma_1, \ldots, \sigma_n$, we introduce the n-ary product notation

$$\sigma_1 \times \ldots \times \sigma_n \stackrel{def}{=} \sigma_1 \times (\sigma_2 \times \ldots (\sigma_{n-1} \times \sigma_n) \ldots)$$

by associating to the right. (This is an arbitrary decision. We could just as easily associate n-ary products to the left.) To define elements of an n-ary product, we use the tupling notation

$$\langle M_1, \ldots, M_n \rangle \stackrel{def}{=} \langle M_1, \langle M_2, \ldots \langle M_{n-1}, M_n \rangle \ldots \rangle \rangle$$

as syntactic sugar for a nested pair. It is easy to check that if $M_i{:}\,\sigma_i$, then

$$\langle M_1, \ldots, M_n \rangle : \sigma_1 \times \ldots \times \sigma_n$$

To retrieve the components of an n-tuple, we use the following notation for combinations of binary projection functions.

$$\begin{aligned} \mathbf{Proj}_i^{\sigma_1 \times \ldots \times \sigma_n} &\stackrel{def}{=} \lambda x{:}\,\sigma_1 \times \ldots \times \sigma_n.\mathbf{Proj}_1(\mathbf{Proj}_2^{i-1} x) \qquad (i < n) \\ \mathbf{Proj}_n^{\sigma_1 \times \ldots \times \sigma_n} &\stackrel{def}{=} \lambda x{:}\,\sigma_1 \times \ldots \times \sigma_n.(\mathbf{Proj}_2^{n-1} x) \end{aligned}$$

We leave it as an exercise to check that

$$\mathbf{Proj}_i^{\sigma_1 \times \ldots \times \sigma_n} \langle M_1, \ldots, M_n \rangle \twoheadrightarrow M_i,$$

justifying the use of this notation. A useful piece of meta-notation is to write σ^k for the product $\sigma \times \ldots \times \sigma$ of k σ's.

Using n-tuples, we can now translate records with more than two components into PCF quite easily. If we want to eliminate records of some type $\mathbf{record}\{\ell_1 \colon \sigma_1, \ldots, \ell_k \colon \sigma_k\}$, we choose some ordering of the labels $\ell_1, \ldots, \ell_k$ and write each type expression and record expression using this order. (Any ordering will do, as long as we are consistent. For concreteness, we could use alpha-numeric lexicographical order.) We then translate expressions with records into PCF as follows.

$$\begin{array}{rcl} \mathbf{record}\{\ell_1 \colon \sigma_1, \ldots, \ell_k \colon \sigma_k\} & \stackrel{def}{=} & \sigma_1 \times \ldots \times \sigma_k \\ \{\ell_1 = M_1, \ldots, \ell_k = M_k\} & \stackrel{def}{=} & \langle M_1, \ldots, M_k \rangle \\ M.\ell_i & \stackrel{def}{=} & \mathbf{Proj}_i^{\sigma_1 \times \ldots \times \sigma_k} M \end{array}$$

If an expression contains more than one type of records, we apply the same process to each type independently.

Example 2.5.1 We will translate the expression

$$\texttt{let } r \colon \{A \colon \mathit{int},\ B \colon \mathit{bool}\} = \{A = 3,\ B = \mathit{true}\} \texttt{ in if } r.B \texttt{ then } r.A \texttt{ else } r.A + 1$$

into PCF. The first step is to number the record labels. Using the number 1 for A and 2 for B, the type $\{A \colon \mathit{int},\ B \colon \mathit{bool}\}$ becomes $\mathit{int} \times \mathit{bool}$ and a record $\{A = x,\ B = y\} : \{A \colon \mathit{int},\ B \colon \mathit{bool}\}$ becomes a pair $\langle x, y \rangle : \mathit{int} \times \mathit{bool}$. Following the general procedure outlined above, we desugar the expression with records to the following expression with product types and pairing:

$$\texttt{let } r \colon \mathit{int} \times \mathit{bool} = \langle 3,\ \mathit{true} \rangle \texttt{ in if } \mathbf{Proj}_2 r \texttt{ then } \mathbf{Proj}_1 r \texttt{ else } (\mathbf{Proj}_1 r) + 1$$

Some slightly more complicated examples are given in the exercise. ∎

Exercise 2.5.2 Desugar the following expressions with records to expressions with product types and pairing.

(a)

$$\begin{array}{l} \texttt{let } r \colon \{A \colon \mathit{int}, B \colon \mathit{bool}, C \colon \mathit{int} \to \mathit{int}\} = \{A = 5, B = \mathit{false}, C = \lambda x \colon \mathit{int}.\ x\} \\ \texttt{in if } r.B \texttt{ then } r.A \texttt{ else } (r.C)(r.A) \end{array}$$

(b)

$$\begin{array}{l} \texttt{let } f(r \colon \{A \colon \mathit{int}, C \colon \mathit{bool}\}) \colon \{A \colon \mathit{int}, B \colon \mathit{bool}\} = \{A = r.A, B = r.C\} \\ \texttt{in } f\{A = 3, C = \mathit{true}\} \end{array}$$

You may wish to eliminate one type of records first and then the other. If you do this, the intermediate term should be a well-typed expression with only one type of records.

2.5.2 Searching the natural numbers.

One useful programming technique in PCF is to "search" the natural numbers, starting from 0. In recursive function theory, this method is called *minimization,* and written using the operator μ. Specifically, if p is a computable predicate on the natural numbers (which means a computable function from *nat* to *bool*), then $\mu x[p\,x]$ is the least natural number n such that $p\,n = \mathit{true}$ and $p\,n' = \mathit{false}$ for all $n' < n$. If there is no such n, then $\mu x[p\,x]$ is "undefined," which means that there is no natural number n with $\mu x[p\,x] = n$.

In PCF, given a natural number predicate $p\colon \mathit{nat} \to \mathit{bool}$, we can compute $\mu x[p\,x]$ using the expression

$$\texttt{letrec}\ f(x\colon \mathit{nat})\colon \mathit{nat} = \texttt{if}\ p\,x\ \texttt{then}\ x\ \texttt{else}\ f(x+1)\ \ \texttt{in}\ \ f\,0$$

Intuitively, the recursive function f tests to see if its argument x is has the property p. If so, then this is the result of the function call. Otherwise, the function f is called recursively on $x+1$. Since the first call is to $f\,0$, we start with 0 and test all natural numbers in succession. However, if $p\,n$ may be reduced indefinitely without producing either *true* or *false*, then it is possible to reduce this expression indefinitely without testing $p(n+1)$.

Since we will use minimization several times, we will adopt the abbreviation

$$\mathit{search} \stackrel{\mathit{def}}{=} \lambda p\colon \mathit{nat} \to \mathit{bool}.\ \texttt{letrec}\ f(x\colon \mathit{nat})\colon \mathit{nat} = \texttt{if}\ \ (p\,x)\ \texttt{then}\ x\ \texttt{else}\ f(x+1)\ \ \texttt{in}\ \ f\,0$$

The main property of *search* is described by the following proposition, which relies on the fact that no expression reduces to both *true* and *false*.

Proposition 2.5.3 *Let $M\colon \mathit{nat} \to \mathit{bool}$ be any PCF predicate on the natural numbers. If $M\,n \twoheadrightarrow \mathit{true}$ and $M\,n' \twoheadrightarrow \mathit{false}$ for all $n' < n$, then search $M \twoheadrightarrow n$.*

The astute reader may notice a convenient pun in the statement of this proposition. The statement includes a condition $n' < n$ on n and n' of PCF, not natural numbers. To be precise, we should distinguish between numerals, symbols used in expressions of PCF, and the "mathematical objects" we usually refer to by these symbols. However, it is convenient to order the numerals by the usual ordering on natural numbers, use induction on numerals, and transfer other properties of the natural numbers to numerals. This makes many statements about PCF, such as the inductive argument in the proof below, much easier to phrase.

Proof We will prove the following statement by induction on n:

$$\begin{array}{ll} & \text{If } M\,n' \twoheadrightarrow \mathit{false} \text{ for all } n' < n, \text{ then} \\ (*) & \quad \mathit{search}\ M \twoheadrightarrow \texttt{if}\ \ (Mn)\ \texttt{then}\ n\ \texttt{else}\ (\mathit{fix}\ F)(n+1), \\ & \text{where } F \text{ is } \lambda f\colon \mathit{nat} \to \mathit{nat}.\lambda x\colon \mathit{nat}.\ \texttt{if}\ \ (Mx)\ \texttt{then}\ x\ \texttt{else}\ f(x+1). \end{array}$$

The proposition follows from $(*)$ by noticing that if $Mn \twoheadrightarrow \mathit{true}$, the conditional expression reduces to n.

The inductive proof begins with the base case $n = 0$. By expanding definitions and reducing, we have

$$\begin{array}{rcl} \mathit{search}\ M & \twoheadrightarrow & (\lambda f\colon \mathit{nat} \to \mathit{nat}.\, f\,0)(\mathit{fix}\ F) \\ & \twoheadrightarrow & (\mathit{fix}\ F)\,0 \\ & \twoheadrightarrow & F\,(\mathit{fix}\ F)\,0 \\ & \twoheadrightarrow & \texttt{if}\ \ (M0)\ \texttt{then}\ 0\ \texttt{else}\ (\mathit{fix}\ F)(1) \end{array}$$

To prove the inductive step, suppose the claim holds for n, and that $M\,n' \twoheadrightarrow \mathit{false}$ for all $n' < n+1$. Starting with the inductive hypothesis and continuing with the assumption that $M\,n \twoheadrightarrow \mathit{false}$, we have

$$\begin{array}{rcl} \mathit{search}\ M & \twoheadrightarrow & \mathtt{if}\ (Mn)\ \mathtt{then}\ n\ \mathtt{else}\ (\mathit{fix}\ F)(n+1) \\ & \twoheadrightarrow & (\mathit{fix}\ F)(n+1) \\ & \twoheadrightarrow & F\,(\mathit{fix}\ F)(n+1) \\ & \twoheadrightarrow & \mathtt{if}\ (M\,(n+1))\ \mathtt{then}\ n+1\ \mathtt{else}\ (\mathit{fix}\ F)((n+1)+1). \end{array}$$

Since n is a meta-variable for some numeral $0, 1, 2, \ldots$ of PCF, we can simplify $n+1$ to a numeral, obtaining an expression of the desired form. This proves $(*)$, and hence the proposition. ∎

Example 2.5.4 The definition of factorial uses a *predecessor* function $\lambda x{:}\,\mathit{nat}.\,x-1$, which is not a basic operation of PCF. However, using *search,* it is easy to compute the predecessor of any natural number x. We simply search for the first y satisfying $y+1=x$. Accounting for the special case $x=0$, whose "predecessor" is 0 by convention, we define the predecessor function on the natural numbers by the PCF expression

$$\mathit{pred} \stackrel{\mathit{def}}{=} \lambda x{:}\,\mathit{nat}.\,\mathtt{if}\ \ \mathit{Eq}?\,x\,0\ \mathtt{then}\ 0\ \mathtt{else}\ (\mathit{search}\ \lambda y{:}\,\mathit{nat}.\,\mathit{Eq}?\,(y+1)\,x)$$

Using Proposition 2.5.3 and properties of $\mathit{Eq}?$, it is straightforward to show that *pred* computes the predecessor function. ∎

Exercise 2.5.5 Write a PCF function $\mathit{half}{:}\,\mathit{nat}\to\mathit{nat}$ mapping any numeral n to the numeral for $\lfloor n/2 \rfloor$, the greatest natural number not exceeding $n/2$.

Exercise 2.5.6 Write a function *comp* that maps any natural number n to the n-fold composition function $\lambda f{:}\,\mathit{nat}\to\mathit{nat}.\lambda x{:}\,\mathit{nat}.\,f^n x$, where $f^n x$ is an abbreviation for the application $f(f(\ldots(fx)\ldots))$ of f to x a total of n times. Use this to define multiplication $\mathit{mult}{:}\,\mathit{nat}\times\mathit{nat}\to\mathit{nat}$ in PCF. (Hint: multiply $m\cdot n$ by repeatedly adding n to itself.)

Exercise 2.5.7 This question is about a restricted form of recursion called *primitive recursion.* One property of primitive recursion is that if f is defined from g and h by primitive recursion, and g and h are both total functions (*i.e.*, yield natural-number results for all natural number arguments), then f is also total.

(a) A function $f{:}\,\mathit{nat}\to\sigma$ is *defined from* $g{:}\,\sigma$ *and* $h{:}\,(\sigma\times\mathit{nat})\to\sigma$ *by primitive recursion* if

$$\begin{array}{rcl} f\,0 & = & g \\ f\,(n+1) & = & h\,\langle (f\,n), n\rangle \end{array}$$

Write a PCF function $\mathit{prim}_\sigma{:}\,\sigma\to(\sigma\times\mathit{nat}\to\sigma)\to(\mathit{nat}\to\sigma)$ such that for any g and h of the appropriate types, the function $(\mathit{prim}_\sigma\,g\,h)$ satisfies the equations for f above. You will need to use the fact that PCF allows conditional expressions of any type.

(b) In PCF, both addition and equality test are basic operations. However, if we add primitive recursion to PCF, we may remove $+$ and $\mathit{Eq}?$ from PCF and replace them with the simpler functions of successor and zero test, without changing the set of definable functions. This may be proved by showing that addition and equality test are definable in a version of PCF with addition and equality replaced by primitive recursion, successor and zero test. This

problem asks you to prove slightly more. Using functions $succ\colon nat \to nat$, $zero?\colon nat \to bool$, and $prim\colon \sigma \to (\sigma \times nat \to \sigma) \to nat \to \sigma$, satisfying the infinite collection of equations

$$\begin{aligned}
&succ\, 0 = 1\\
&succ\, 1 = 2\\
&\vdots\\
&zero?\, 0 = true\\
&zero?\, 1 = false\\
&\vdots\\
&prim\, g\, h\, 0 = g\\
&prim\, g\, h\, (succ\, n) = h\, \langle prim\, g\, h\, n, n\rangle,
\end{aligned}$$

along with the other basic operations of PCF (except $+$, $Eq?$ and *fix*), show how to define predecessor, addition and equality test. Since `letrec` is defined using *fix*, do not use `letrec`. (*Hint:* Try predecessor first.)

2.5.3 Iteration and tail recursion.

Many algorithms, when written in an imperative language like Pascal, use the following pattern of initialization and iteration.

Initialize;
while $\neg$*Done* **do** *Stmt* **end**

If the sections *Initialize* and *Stmt* only change the values of a fixed, finite set of variables, and *Done* is a side-effect-free test depending on the values of these variables, then we may easily transform a program segment of this form into a functional program. Let us assume for simplicity that there is only one variable involved; there is no loss of generality in this since we can replace a finite set of variables by a record or n-tuple containing their values. To put this in the context of PCF, suppose we have PCF expressions

$$\begin{aligned}
&init : \sigma\\
&next : \sigma \to \sigma\\
&done : \sigma \to bool
\end{aligned}$$

so that the iteration could be written

$x := init$;
while $\neg(done\ x)$ **do** $x := (next\ x)$ **end**

using an assignable variable x of type σ. We may compute the final value of x produced by this loop, using essentially the same sequence of operations, with the expression *loop init next done* defined by.

$$loop\ init\ next\ done \stackrel{def}{=} \texttt{letrec}\ f(x\colon\sigma) = \texttt{if}\ (done\ x)\ \texttt{then}\ x\ \texttt{else}\ f\,(next\, x)\ \texttt{in}\ (f\ init)$$

Since *loop* is a straightforward generalization of the function *search,* the analysis given in Proposition 2.5.3 applies. To be more specific, writing $next^n\, x$ for the result $next(next(\ldots(next\, x)\ldots))$ of applying *next* to x a total of n times, we we have the following proposition.

Proposition 2.5.8 *If* $done\ (next^i\ init) \twoheadrightarrow false$ *for all* $i < n$ *and* $done\ (next^n\ init) \twoheadrightarrow true$, *then*

$$loop\ init\ next\ done \twoheadrightarrow next^n\ init$$

The translation of iterative loops into recursive functions always produces a recursive function with a particular form. A recursive function with a definition of the form

$$f(x) \;=\; \texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ f(N),$$

where neither B, M nor N contains f, is called a *tail-recursive function.* In languages such as Lisp and Scheme, where recursive functions are used extensively, tail-recursive functions are often recognized by the compiler. The reason for treating tail-recursive functions separately is that these may be compiled into efficient iterative code that does not require a new activation record (stack frame) for each recursive call in the source program.

Example 2.5.9 Using pairing, we may translate the following iterative algorithm into PCF.

$x := 100;$
$y := 0;$
while $\neg(Eq?\, x\, y)$ **do** $x := x - 1; y := y + 1$ **end**

The initial value of the pair $\langle x, y\rangle$ is $\langle 100, 0\rangle$. Using the syntactic sugar introduced in Exercise 2.2.12, we may write the body of the **while** loop as the following "next" function

$$next \stackrel{def}{=} \lambda\langle x{:}\, nat, y{:}\, nat\rangle.\, \langle x - 1,\, x + 1\rangle.$$

The test for loop termination may be written similarly as $done \stackrel{def}{=} \lambda\langle x{:}\, nat, y{:}\, nat\rangle.\, (Eq?\, x\, y)$. Putting these together, the **while** loop may be written as

$$L \stackrel{def}{=} loop\, \langle 100, 0\rangle\, next\, done$$

If we let F be the term

$$F \stackrel{def}{=} \lambda f{:}\, nat \times nat \to nat \times nat.\, \lambda p{:}\, nat \times nat.\, \texttt{if}\ \ (done\, p)\ \texttt{then}\ p\ \texttt{else}\ f(next\, p).$$

then we may reduce the PCF expression representing the `while` loop as follows:

$$\begin{array}{rcl}
L & \equiv & (\lambda f{:}\, nat \times nat \to nat \times nat.\, f\langle 100, 0\rangle)(\mathit{fix}\ F) \\
 & \to & (\mathit{fix}\ F)\langle 100, 0\rangle \\
 & \to & F(\mathit{fix}\ F)\langle 100, 0\rangle \\
 & \to & (\lambda p{:}\, nat \times nat.\, \texttt{if}\ \ Eq?(Proj_1 p)(Proj_2 p)\ \texttt{then}\ p \\
 & & \quad \texttt{else}\ \ (\mathit{fix}\ F)\langle (Proj_1 p) - 1, (Proj_2 p) + 1\rangle)\langle 100, 0\rangle \\
 & \to & \texttt{if}\ \ Eq?(Proj_1\langle 100, 0\rangle)(Proj_2\langle 100, 0\rangle)\ \texttt{then}\ \langle 100, 0\rangle \\
 & & \quad \texttt{else}\ \ (\mathit{fix}\ F)\langle (Proj_1\langle 100, 0\rangle) - 1, (Proj_2\langle 100, 0\rangle) + 1\rangle
\end{array}$$

At this point, we depart from left-most reduction order and evaluate all the projections, additions and subtractions before continuing.

$$\begin{array}{rcl}
 & \twoheadrightarrow & \texttt{if}\ \ Eq?\, 100\, 0\ \texttt{then}\ \langle 100, 0\rangle\ \texttt{else}\ (\mathit{fix}\ F)\langle 99, 1\rangle \\
 & \to & \texttt{if}\ \mathit{false}\ \texttt{then}\ \langle 100, 0\rangle\ \texttt{else}\ (\mathit{fix}\ F)\langle 99, 1\rangle \\
 & \to & (\mathit{fix}\ F)\langle 99, 1\rangle
\end{array}$$

At this point, we have completed "execution" of one iteration of the `while` loop. Continuing in this manner, we can see that the functional expression computes the final values of x and y by approximately the same sequence of additions, subtractions and tests as the iterative **while** loop.
■

We may compare the sequence of operations involved in a **while** loop and its PCF translation by giving a reduction rule for **while** loops. A natural rule which corresponds to the way that iteration is actually implemented is

$$\textbf{while } \neg B \textbf{ do } S \textbf{ end} \quad \rightarrow \quad \textbf{if } B \textbf{ then } \mathit{skip} \textbf{ else}(S; \textbf{while } \neg B \textbf{ do } S \textbf{ end})$$

where *skip* is a statement that does nothing. We may use this reduction rule to argue informally that an iterative algorithm

$$\begin{array}{l} x := \mathit{init}; \\ \textbf{while } \neg(\mathit{done}\ x) \textbf{ do } x := (\mathit{next}\ x) \textbf{ end} \end{array}$$

and its translation into PCF perform the same sequence of evaluations of *init,next* and *done.* The reader is encouraged to work this out in Exercise 2.5.10.

Exercise 2.5.10 Translate the imperative algorithm

$q := 0;$
$r := m;$
`while` $r \geq n$ `do`
    $q := q + 1;$
    $r := r - n;$
    `od`

into PCF by defining functions *init next done.* (You will have to use recursion to implement the test $r \geq n$.) Use the reduction rule for **while** loops to argue informally that the loop and its PCF translation involve essentially the same arithmetic operations. You may assume left-most order of PCF evaluation.

Exercise 2.5.11 Translate the following imperative algorithm to compute the greatest common divisor into PCF by defining functions *init next done.*

`while` $n \neq 0$ `do`
    $r := m;$
    `while` $r \geq n$ `do`
        $r := r - n;$
        `od`
    $m := n;$
    $n := r;$
    `od`

Think of this program as using m and n as input, and producing a new value of m as output.

2.5.4 Total recursive functions

In this section and the next, we compare the natural-number functions definable in PCF with the classes of total and partial recursive functions (defined below and in the next section). The main results are that all recursive total and partial functions on the natural numbers are definable in PCF. Similar proofs using Turing machines are given in the exercises.

A commonly accepted belief, called *Church's thesis,* is that every numeric function that is computable by any practical computer is recursive. This thesis was formulated in the 1930's, before electronic computers, during a period when mathematicians were actively investigating the possibility and impossibility of solving problems by systematic algorithms. One reason for believing Church's thesis is that all of the formalisms for defining computable functions that were proposed in the 1930's, and since, give rise to the same set of functions on the natural numbers. The early formalisms include the recursive functions, Turing machines, and lambda calculus. It follows from Church's thesis (although this is no way to prove something rigorously), that every partial or total function definable in PCF is recursive. The reason is that we have an algorithm for computing any definable function, namely reduction. It is not hard to give a rigorous proof that every PCF-definable function is recursive, although we will not do so since this involves tricks with recursive functions and does not shed much light on PCF itself. One reason for proving that every recursive total or partial function is definable in PCF is that, via Church's thesis, this gives evidence that every computable function on the natural numbers is definable in PCF. In other words, this is as close as we can get to showing that, at least for natural-number functions, PCF is a "universal" programming language. The second reason is that, by appealing to undecidable properties of recursive functions, we obtain interesting undecidability properties of PCF.

A subtle issue that is not often discussed in basic courses on computability or complexity theory is the limitation of Church's thesis for functions on types other than the natural numbers. For basic data such as booleans, strings or arrays of such data, it makes sense to think of the computable functions as precisely those functions we can compute when we code each boolean, string or array by a natural number. By this reasoning, we can see that any programming language that lets us associate natural numbers with its basic data types, and compute all recursive functions on the natural numbers, is "universal" for defining computable functions on all the basic data types. However, for "infinite" values, such as functions, the issue is not as clear cut. In particular, mathematical logicians have identified several distinct classes of "computable functions on the natural-number functions" and have not been able to prove that these are identical. A related phenomenon is discussed in Section 2.5.6, where we show that certain "parallel" operations are not definable in PCF, in spite of the ability to define all recursive functions on the natural numbers.

Since there are some subtle points about termination and function composition for partial functions, we begin with the simple case of total functions. Since we have pairing in PCF, we will consider total recursive functions of more than one argument. A function f is *numeric* if $f{:}\mathcal{N}^k \to \mathcal{N}$ for some $k > 0$. If $\mathcal{C}$ is a class of numeric functions, we say $\mathcal{C}$ is

- *Closed under composition* if, for every $f_1, \ldots, f_\ell{:}\mathcal{N}^k \to \mathcal{N}$ and $g{:}\mathcal{N}^\ell \to \mathcal{N}$ from $\mathcal{C}$, the class $\mathcal{C}$ also contains the function h defined by
$$h(n_1, \ldots, n_k) = g(f_1(n_1, \ldots, n_k), \ldots, f_\ell(n_1, \ldots, n_k)),$$
- *Closed under primitive recursion* if, for every $f{:}\mathcal{N}^k \rightharpoonup \mathcal{N}$ and $g{:}\mathcal{N}^{k+2} \rightharpoonup \mathcal{N}$ from $\mathcal{C}$, the class $\mathcal{C}$ contains the function h defined by
$$\begin{array}{lcl} h(0, n_1, \ldots, n_k) & = & f(n_1, \ldots, n_k) \\ h(m+1, n_1, \ldots, n_k) & = & g(h(m, n_1, \ldots, n_k), m, n_1, \ldots, n_k), \end{array}$$

- *Closed under minimization* if, for every $f{:}\mathcal{N}^{k+1} \to \mathcal{N}$ from $\mathcal{C}$ such that $\forall n_1, \ldots, n_k.\, \exists m.\, f(m, n_1, \ldots, n_k) = 0$, the class $\mathcal{C}$ contains the function g defined by

$$g(n_1, \ldots, n_k) = \text{the least } m \text{ such that } f(m, n_1, \ldots, n_k) = 0.$$

The class $\mathcal{R}$ of *total recursive functions* is the least class of numeric functions that contains the projection functions $\mathbf{Proj}_i^k(n_1, \ldots, n_k) = n_i$, the successor function $\lambda x{:}\, nat.\, x{+}1$, the constant zero function $\lambda x{:}\, nat.\, 0$, and that is closed under composition, primitive recursion, and minimization.

We can show that every total recursive function is definable in PCF, in the following precise sense. For any natural number $n \in \mathcal{N}$, let us write $\lceil n \rceil$ for the corresponding numeral of PCF. We say a numeric function $f{:}\mathcal{N}^k \to \mathcal{N}$ is *PCF-definable,* or simply *definable,* if there is a closed PCF expression $M{:}\, nat^k \to nat$, where nat^k is the k-ary product type $nat \times \ldots \times nat$, such that

$$\forall n_1, \ldots, n_k \in \mathcal{N}.\ M\langle \lceil n_1 \rceil, \ldots, \lceil n_k \rceil \rangle = \lceil f(n_1, \ldots, n_k) \rceil.$$

Theorem 2.5.12 *Every total recursive function is definable in PCF.*

Proof To show that every total recursive function is definable in PCF, we must give the projection functions $\mathbf{Proj}_i^k$, the successor function, and the constant zero function, and show closure under composition, primitive recursion, and minimization. Some of the work has already been done in Exercise 2.5.7 and Proposition 2.5.3. Following the definitions in Section 2.5.1, we let

$$\begin{array}{lcll} \mathbf{Proj}_1^1 & \stackrel{def}{=} & \lambda x{:}\, nat.\, x & \\ \mathbf{Proj}_1^k & \stackrel{def}{=} & \lambda x{:}\, nat^k.\, \mathbf{Proj}_1\, x & (1 < k) \\ \mathbf{Proj}_i^k & \stackrel{def}{=} & \lambda x{:}\, nat^k.\, \mathbf{Proj}_{i-1}^{k-1}(\mathbf{Proj}_2\, x) & (1 < i \leq k) \end{array}$$

As in the definition of the class of total recursive functions, we may define successor and the constant function returning zero by

$$\begin{array}{lcl} succ & \stackrel{def}{=} & \lambda x{:}\, nat.\, x + 1, \\ zero & \stackrel{def}{=} & \lambda x{:}\, nat.\, 0. \end{array}$$

If $f_1 \ldots f_\ell{:}\mathcal{N}^k \to \mathcal{N}$ and $g{:}\mathcal{N}^\ell \to \mathcal{N}$ are represented by the PCF terms $M_1 \ldots M_\ell$ and N, respectively, then the function h defined by composition is represented by the term

$$\lambda x{:}\, nat^k.\, N\langle M_1 x,\ \ldots\ , M_\ell x \rangle.$$

If $f{:}\mathcal{N}^k \to \mathcal{N}$ and $g{:}\mathcal{N}^{k+2} \to \mathcal{N}$ are represented by M and N, then the function h defined by primitive recursion is represented by

$$\lambda\langle x{:}\, nat, y{:}\, nat^k \rangle.\, prim\ (M\ y)\ (\lambda\langle m{:}\, nat, n{:}\, nat \rangle.\, N\ \langle m, \langle n, y \rangle\rangle)\ x,$$

where *prim* is as in Exercise 2.5.7. Finally, to establish closure under minimization, suppose $f{:}\mathcal{N}^{k+1} \to \mathcal{N}$ is represented by M, and for every $n_1 \ldots n_k$ there exists an m such that $f(m, n_1 \ldots n_k) = 0$. By Proposition 2.5.3, the function g defined by minimization is represented by

$$\lambda x{:}\, nat^k.\, search(\lambda n{:}\, nat.\, M\ \langle n, x \rangle).$$

■

Exercise 2.5.13 Although a Turing machine may compute a partial function, there is a natural total function associated with every Turing machine, namely, the function giving the contents of the tape after n steps. This exercise asks you to show directly that this function is definable in PCF and conclude that every function computed by a Turing machine that halts on all input is definable in PCF.

There are a number of equivalent definitions of Turing machines. As a reminder, and to set notation, we briefly review one standard definition (from [HU79]) before stating the problem. A *Turing machine* is given by a tuple

$$\langle Q, \Sigma, \Gamma, \delta, q_1, \not b, F \rangle$$

where $Q = \{q_1, ..., q_n\}$ is a finite set of *states;* Σ is the (finite) input alphabet, not including $\not b$, the "blank" symbol; Γ is the (finite) tape alphabet with $\Gamma \supseteq \Sigma$;

$$\delta : Q \times \Gamma \rightharpoonup Q \times \Gamma \times \{L, R\}$$

is a partial function telling the next state, symbol to write on the tape, and direction to move the tape head; q_1 is the start state; $\not b$ is the "blank" symbol; and $F \subseteq Q$ is the set of final *accepting states.* Since δ may be a partial function, $\delta(q, g)$ may be undefined for some state $q \in Q$ and tape symbol $g \in \Gamma$. In this case, the machine halts and does not move. (The machine also halts if it moves off the left end of the tape.) The machine accepts its input if it halts in a final state $q \in F$. Since Q, Γ and $\{L, R\}$ are all finite, the transition function δ is a finite set. There is no loss of generality in assuming that there is exactly one final state $q_f \in F$ and that $\delta(q, g)$ is defined iff $q \neq q_f$. This makes δ a total function from $(Q - \{q_f\}) \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$. We may assume that $\Sigma = \{0, 1\}$ and $\Gamma = \Sigma \cup \{\not b\}$, since any symbol may be coded by a sequence of bits (using ASCII, for example). In this problem, you may represent $\not b$ by the number 2.

(a) We will code a Turing machine tape using a coding of pairs as natural numbers. Specifically, a pair of natural numbers $\langle n, m \rangle$ may be coded by the natural number

$$pr(n, m) = (n + m)(n + m + 1)/2 + m.$$

This may be understood by arranging all the pairs of natural numbers in an infinite table, with $\langle 0, 0 \rangle$ in the upper left corner and first and second coordinates increasing as we move down and right in the table (respectively). If we "walk" through this table along the northeast diagonals, moving to the next diagonal when we reach the top border, then the pair $\langle n, m \rangle$ is reached after $pr(n, m)$ steps. Show that the pairing function pr and the corresponding projection functions p_1 and p_2 mapping $pr(n, m)$ to n and m are definable in PCF.

(b) We may code a finite sequence of natural numbers as follows:

$$\begin{array}{lcl} seq() & = & pr(0, 0) \\ seq(ns) & = & pr((k + 1),\ pr(n,\ seq(s))) \end{array}$$

where in the $seq(ns)$ case we may follow the convention that k is the length of sequence s. Show that if $m = seq(ns)$, we may compute n from m in PCF. Show that if $m = seq(s)$, we may compute $seq(ns)$ from m and n in PCF.

(c) Let $\mathcal{M}$ be a Turing machine, as described above, with states numbered 1 through q_M. Suppose $q \leq q_M$ is a natural number representing a state of $\mathcal{M}$, natural number n_ℓ codes

the sequence of tape symbols to the left of the tape head, as in (b) above, and n_r codes the sequence of tape symbols under and to the right of the tape head. Show that the next state and two numbers representing the contents of the Turing machine tape are computable from the q, n_ℓ and n_r in PCF.

(d) Show that for any Turing machine $\mathcal{M}$, the function

$$f_{\mathcal{M}}(j, s, n) = \text{ the } j\text{th tape symbol after } n \text{ computation steps of } \mathcal{M} \text{ on initial tape } s$$

is representable in PCF, where the initial tape contents are coded as a natural number.

(e) A Turing machine $\mathcal{M}$ computes a total numeric function $f{:}\mathcal{N} \to \mathcal{N}$ if, when started with the binary representation of n on the tape, the machine halts with the binary representation of $f(n)$ on the tape. Show that every total function computed by a Turing machine is definable in PCF.

2.5.5 Partial recursive functions

In this section, we show that every partial recursive function is definable in PCF. Since it is generally believed that all mechanically computable functions are partial recursive functions, as discussed in Section 2.5.4, the main theorem of this section suggests that every function on the natural numbers that could be computed by any ordinary computer is definable in PCF. Two corollaries are that there is no algorithm to determine whether a PCF expression has a normal form and no algorithm to determine whether two PCF expressions are provably equal.

Although we may prove the main theorem of this section without repeating the inductive argument of Theorem 2.5.12, there are two reasons for giving a direct inductive proof. The first is to emphasize the difference between representing a total function and representing a partial one. The second is to gain further intuition for the evaluation mechanism of PCF and its relation to standard mathematical conventions about partial functions. This intuition may be useful in considering "parallel" functions in the next section.

We discuss the representation of partial functions before defining the class of partial recursive functions.

Suppose we have some partial function $f{:}\mathcal{N} \rightharpoonup \mathcal{N}$ which we would like to represent by a PCF expression. We clearly want a closed term $M{:}\, nat \to nat$ such that $M\lceil n \rceil$ gives the value of $f(n)$ when $f(n)$ is defined. However, it is not as clear what property M should have if $f(n)$ is *not* defined. One possibility, of course, is to not to require any property of $M\lceil n \rceil$. However, an accurate representation of a partial function is a term M with $M\lceil n \rceil$ "defined" iff $f(n)$ is defined. This requires some notion of "undefined term." A convenient representation of undefinedness, or nontermination, is that $M\lceil n \rceil$ should have no normal form when $f(n)$ is undefined. We will adopt this below for terms that represent numeric functions.

We say a partial function $f{:}\mathcal{N}^k \rightharpoonup \mathcal{N}$ is *PCF-definable,* or simply *definable,* if there is a closed PCF expression $M{:}\, nat^k \to nat$ such that for all $n_1, \ldots, n_k \in \mathcal{N}$, we have

$$M\langle \lceil n_1 \rceil, \ldots, \lceil n_k \rceil \rangle = \begin{cases} \lceil f(n_1, \ldots, n_k) \rceil & \text{if } f(n_1, \ldots, n_k) \text{ is defined,} \\ \text{has no normal form} & \text{otherwise} \end{cases}$$

The reader familiar with untyped lambda calculus may know that the convention for partial functions is to represent undefinedness by lack of a head normal form. For PCF, a *head normal form*, is either a numeral, a boolean constant (*true* or *false*), or term of the form $\lambda x{:}\sigma.M$ or $\langle M, N \rangle$,

according to its type. Since a term of type *nat* or *bool* has a normal form iff it has a head normal form, the type constraints of PCF make the two possible definitions equivalent. Those interested in untyped lambda calculus may wish to consult [Bar84, Section 8.4] for a discussion of partial recursive functions in the pure, untyped lambda calculus.

To show that every partial recursive function is definable in PCF, we give a precise definition of the class of all partial recursive functions. A partial function f is *numeric* if $f{:}\mathcal{N}^k \rightharpoonup \mathcal{N}$ for some $k > 0$. If $\mathcal{C}$ is a class of partial numeric functions, we say $\mathcal{C}$ is

- *Closed under composition* if, for all partial functions $f_1, \ldots, f_\ell{:}\mathcal{N}^k \rightharpoonup \mathcal{N}$ and $g{:}\mathcal{N}^\ell \rightharpoonup \mathcal{N}$ from $\mathcal{C}$, the class $\mathcal{C}$ also contains the partial function h defined by

$$h(n_1, \ldots, n_k) = \begin{cases} g(m_1, \ldots, m_\ell) & \text{if } m_i = f_i(n_1, \ldots, n_k) \text{ defined } 1 \le i \le \ell \\ & \text{and } g(m_1, \ldots, m_\ell) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

- *Closed under primitive recursion* if, for every $f{:}\mathcal{N}^k \rightharpoonup \mathcal{N}$ and $g{:}\mathcal{N}^{k+2} \rightharpoonup \mathcal{N}$ from $\mathcal{C}$, the class $\mathcal{C}$ contains the partial function h defined by

$$h(0, n_1, \ldots, n_k) = \begin{cases} f(n_1, \ldots, n_k) & \text{if } f(n_1, \ldots, n_k) \text{ defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$h(m+1, n_1, \ldots, n_k) = \begin{cases} g(p, m, n_1, \ldots, n_k), & \text{if } p = h(m, n_1, \ldots, n_k) \text{ and} \\ & g(p, m, n_1, \ldots, n_k) \text{ are defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

- *Closed under minimization* if, for every partial $f{:}\mathcal{N}^{k+1} \rightharpoonup \mathcal{N}$ from $\mathcal{C}$, the class $\mathcal{C}$ contains the partial function g defined by

$$g(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ with } f(m, n_1, \ldots, n_k) = 0, \\ \text{when such an } m \text{ exists,} \\ \\ \text{undefined otherwise.} \end{cases}$$

The class $\mathcal{PR}$ of *partial recursive functions* is the least class of partial numeric functions containing the projection functions $\mathbf{Proj}_i^k(n_1, \ldots, n_k) = n_i$, the successor function $\lambda x{:}\, nat.\, x+1$, the constant zero function $\lambda x{:}\, nat.\, 0$, and closed under composition, primitive recursion, and minimization.

Theorem 2.5.14 *Every partial recursive function is definable in PCF*

Proof The proof is similar to the proof of Theorem 2.5.12, except that the term representing a partial recursive function must not have a normal form when the partial recursive function is undefined. We can see that this requires some changes by considering the composition case. If $f, g{:}\mathcal{N} \rightharpoonup \mathcal{N}$ are unary partial recursive functions, represented by PCF terms M and N, then in the proof of Theorem 2.5.12, we represent the composition $h(n) = g(f(n))$ by the term $\lambda x{:}\, nat.\, N\,(Mx)$. However, this does not have the correct termination behavior when f may be partial. For example, let g be the constant function $g(x) = 3$ and f a partial function with $f(5)$ undefined. The ordinary mathematical convention is to consider $h(5) = g(f(5))$ undefined since $f(5)$ is undefined. However, the application $(\lambda x{:}\, nat.\, N\,(Mx))\ 5$ reduces to 3 in PCF.

A simple trick is to define a function *cnvg* that may be used to force nontermination in PCF when required. Due to typing restrictions, we will define a separate function $cnvg_k$ for each natural number $k > 0$. If f and g are numeric functions of k arguments, then $cnvg_k\, f\, g$ will behave like g, except that an application to k arguments will only have a normal form (or "halt") when the application of f has a normal form. For any $k > 0$, the function $cnvg_k$ may be written as follows:

$$cnvg_k\, f\, g \stackrel{def}{=} \lambda x{:}\, nat^k.\, \texttt{if}\ \ Eq?\,(f\,x)\,0\ \texttt{then}\ g\,x\ \texttt{else}\ g\,x.$$

For any x, if $f\,x$ can be reduced to normal form then (whether this is equal to zero or not) the result is $g\,x$. However, if $f\,x$ cannot be reduced to a normal form, then the conditional can never be eliminated and $cnvg_k\, f\, g$ will not have a normal form. The main reason why this works is that we cannot reduce $Eq?\,(f\,x)\,0$ to *true* or *false* without reducing $f\,x$ to normal form.

Using *cnvg,* we can define a composition f and g that is defined only when f is defined. For example, if $f, g{:}\,\mathcal{N} \rightharpoonup \mathcal{N}$ are unary partial recursive functions represented by PCF terms M and N, we represent their composition by the term $cnvg\, M\, (\lambda x{:}\, nat.(N(Mx)))$. The generalization to the composition of partial functions $f_1, \ldots, f_\ell{:}\,\mathcal{N}^k \rightharpoonup \mathcal{N}$ and $g{:}\,\mathcal{N}^\ell \rightharpoonup \mathcal{N}$ is straightforward and left to the reader.

The remaining cases of the proof are treated similarly, using $cnvg_k$ for appropriate k. The reader is encouraged to write out the argument for primitive recursion, and convince him- or herself that no modification is needed for minimization. ∎

This theorem has some important corollaries. Both of the corollaries below follow from a well-known undecidability property of recursive functions. Specifically, there is no recursive, or algorithmic, method for deciding whether a recursive partial function is defined on some argument. This fact is often referred to as the recursive unsolvability of the *Halting Problem.* Exercise 2.5.18 describes a direct proof that the PCF halting problem is not solvable in PCF.

Corollary 2.5.15 *There is no algorithm for deciding whether a given PCF expression has a normal form.*

Proof If there were such an algorithm, then by Theorem 2.5.14, we could decide whether a given partial recursive function was defined on some argument, simply by writing the function applied to its argument in PCF. ∎

Corollary 2.5.16 *There is no algorithm for deciding equality between PCF expressions.*

Proof If there were such an algorithm, then by Theorem 2.5.14, we could decide whether a given partial recursive function was defined on some argument. Specifically, given a description of a partial function f, we may produce PCF term M representing the application $f(n)$ by the straightforward algorithm outlined in the proof of Theorem 2.5.14, for any n. Then, using the supposed algorithm to determine whether $\texttt{if}\ \ Eq?\, M\, M\ \ \texttt{then}\ 0\ \texttt{else}\ 0 = 0$, we could decide whether $f(n)$ is defined. ∎

Exercise 2.5.17 A *primitive recursive function* is a total recursive function that is defined without using minimization. The Kleene normal form theorem states that every partial recursive function $f{:}\,\mathcal{N}^k \rightharpoonup \mathcal{N}$ may be written in the form

$$f(n_1, \ldots, n_k) = p(\text{the least } n \text{ with } t_k(i, n_1, \ldots, n_k, n) = 1)$$

where p and t_k are primitive recursive functions that are independent of f, and i depends on f. This is called the *Kleene normal form of* f, after Stephen C. Kleene, and the number i is called the

index of partial recursive function f. The Kleene normal form theorem may be found in standard books on computability or recursion theory, such as [Rog67].

(a) Use the Kleene normal form theorem and Theorem 2.5.12 to show that every partial recursive function is definable in PCF.

(b) Using the results of Exercise 2.5.13, show that every partial function computed by a Turing machine is definable in PCF.

Exercise 2.5.18 It is easy to show the halting problem for PCF is not solvable in PCF. Specifically, the *halting function on type* σ, H_σ, is a function which, when applied to any PCF term M_σ, returns *true* if M has a normal form ("halts"), and *false* otherwise. In this exercise, you are asked to show that there is no PCF term H_{bool} defining the halting function on type *bool*, using a variant of the diagonalization argument from recursive function theory. Since the proof proceeds by contradiction, assume there is a PCF term $H_{bool}: bool \to bool$ defining the halting function.

(a) Show that using H_{bool}, we may write a PCF term $G: bool \to bool$ with the property that for any PCF term $M: bool$, the application GM has a normal form iff M does not.

(b) Derive a contradiction by considering whether or not *fix* G has a normal form.

2.5.6 Non-definability of parallel operations

We have seen in Sections 2.5.4 and 2.5.5 that the PCF-definable functions on numerals correspond exactly to the recursive functions and (in the exercises) the numeric functions computable by Turing machines. By Church's thesis, as discussed in Section 2.5.4, this suggests that all mechanically computable functions on the natural numbers are definable in PCF. However, as we shall see in this section, there *are* operations that are computable in a reasonable way, but *not* definable in PCF. The main theorem and the basic idea of Lemma 2.5.24 are due to Plotkin [Plo77]. A semantic proof of Theorem 2.5.19 appears as Example 8.6.3, using logical relations and the denotational semantics of PCF.

The main example we consider in this section is called *parallel-or.* Before discussing this function, it is worth remembering that for a PCF term to represent a k-ary function on the natural numbers, we only consider the result of applying the PCF term to a tuple of numerals. It is not important how the PCF term behaves when applied to expressions that are not in normal form. The difference between the positive results on representing numeric functions in Sections 2.5.4 and 2.5.5 and the negative result in this section is that we now take into account the way a function evaluates its arguments.

It is easiest to describe the parallel-or function algorithmically. Suppose we are given closed terms $M, N\colon \mathit{bool}$ and wish to compute the logical disjunction, $M \vee N$. We know that either $M \twoheadrightarrow \mathit{true}$, $M \twoheadrightarrow \mathit{false}$, or M has no normal form, and similarly for N. One way to compute $M \vee N$ is to first reduce both M and N to normal form, then return *true* if either is *true* and *false* otherwise. This algorithm computes what is called the *sequential-or* of M and N; it will only terminate if both M and N have a normal form. For *parallel-or,* we wish to return *true* if either M or N reduces to *true,* regardless of whether the other term has a normal form. It is easy to see how to compute parallel-or in a computational setting with explicit parallelism. We begin reducing M and N in parallel. If either terminates with value *true*, we abort the other computation and return *true.* Otherwise we continue to reduce both, hoping to produce two normal forms. If both reduce to *false*, then we return *false*. To summarize, the parallel-or of M and N is *true* if either M or N reduces to *true*, *false* if both reduce to *false*, and nonterminating otherwise.

The rest of this section will be devoted to proving that parallel-or is not definable in PCF, as stated in the following theorem.

Theorem 2.5.19 *There is no PCF expression POR with the following behavior*

$$\mathit{POR}\, M\, N \twoheadrightarrow \begin{cases} \mathit{true} & \text{if } M \twoheadrightarrow \mathit{true} \text{ or } N \twoheadrightarrow \mathit{true} \\ \mathit{false} & \text{if } M \twoheadrightarrow \mathit{false} \text{ and } N \twoheadrightarrow \mathit{false} \\ \text{no normal form} & \text{otherwise} \end{cases}$$

for all closed boolean expressions M and N.

We will prove Theorem 2.5.19 by analyzing the operational semantics of PCF. This gives us an opportunity to develop some general tools for operational reasoning. The first important decision is to choose a deterministic reduction strategy instead of reasoning about arbitrary reduction order. Since we will only be interested in reduction on terms of type *nat* or *bool*, we can use lazy reduction, defined in Section 2.4.3, instead of the more complicated left-most reduction strategy. A convenient way to analyze the reduction of subterms is to work with contexts. To identify a critical subterm, we define a special form of context called an *evaluation context.* While the analysis of parallel-or only requires evaluation contexts for boolean terms, we give the general definition for use in the exercises and in Section 5.4.2. The evaluation contexts of PCF are defined in Table 2.6.

$$\begin{array}{rcl}
\textsc{ev}[\,] & ::= & [\,] \;\mid\; \textsc{ev}[\,] + M \;\mid\; n + \textsc{ev}[\,] \text{ for numeral } n \\
& & \mid \mathit{Eq}?\,\textsc{ev}[\,]\,M \;\mid\; \mathit{Eq}?\,n\,\textsc{ev}[\,] \text{ for numeral } n \\
& & \mid \texttt{if } \textsc{ev}[\,] \texttt{ then } N \texttt{ else } P \\
& & \mid \mathbf{Proj}_i\textsc{ev}[\,] \;\mid\; \textsc{ev}[\,]\,M
\end{array}$$

Table 2.6: Evaluation contexts for lazy PCF reduction.

An example appears below, after two basic lemmas. To make the outline of the proof of Theorem 2.5.19 as clear as possible, we postpone the proofs of the lemmas to the end of the section.

There are two main properties of evaluation contexts. The first is that if $M \overset{lazy}{\to} N$, then M matches the form of some evaluation context and N is obtained by reducing the indicated term.

Lemma 2.5.20 *If $M \overset{lazy}{\to} N$, then there is a unique evaluation context $\textsc{ev}[\,]$ such that $M \equiv \textsc{ev}[M']$, the reduction $M' \to N'$ is an instance of one of the PCF reduction axioms, and $N \equiv \textsc{ev}[N']$.*

Since we can determine the unique evaluation context mentioned in the lemma by pattern matching, evaluation contexts provide a complete characterization of lazy reduction.

The second basic property of evaluation contexts is that the lazy reduction of $\textsc{ev}[M]$ is the lazy reduction of M, when M has one. This is stated more precisely in the following lemma.

Lemma 2.5.21 *If $M \overset{lazy}{\to} M'$ then, for any evaluation context $\textsc{ev}[\,]$, we have $\textsc{ev}[M] \overset{lazy}{\to} \textsc{ev}[M']$.*

A subtle point is that when M is a lazy normal form, the lazy reduction of $\textsc{ev}[M]$ may be a reduction that does not involve M.

Example 2.5.22 The left-most reduction of the term $((\lambda x{:}\,nat.\,\lambda y{:}\,nat.\,x+y)\,7)\,5 + (\lambda x{:}\,nat.\,x)\,3$ is given in Example 2.4.11. As observed in Example 2.4.14 this is also the lazy reduction of this term. We illustrate the use of evaluation contexts by writing the evaluation context for each term in the reduction sequence. Since we underline the active redex, the evaluation context is exactly the part of the term that is not underlined.

$$\begin{array}{lll}
\underline{((\lambda x{:}\,nat.\,\lambda y{:}\,nat.\,x+y)\,7)}\,5 + (\lambda x{:}\,nat.\,x)\,3 & \textsc{ev}[\,] \equiv ([\,]\,5) + (\lambda x{:}\,nat.\,x)\,3 \\
\overset{left}{\to}\quad \underline{(\lambda y{:}\,nat.\,7+y)\,5} + (\lambda x{:}\,nat.\,x)\,3 & \textsc{ev}[\,] \equiv [\,] + (\lambda x{:}\,nat.\,x)\,3 \\
\overset{left}{\to}\quad (\underline{7+5}) + (\lambda x{:}\,nat.\,x)\,3 & \textsc{ev}[\,] \equiv [\,] + (\lambda x{:}\,nat.\,x)\,3 \\
\overset{left}{\to}\quad 12 + \underline{(\lambda x{:}\,nat.\,x)\,3} & \textsc{ev}[\,] \equiv 12 + [\,] \\
\overset{left}{\to}\quad \underline{12+3} & \textsc{ev}[\,] \equiv [\,] \\
\overset{left}{\to}\quad 15 &
\end{array}$$

If we look at the second evaluation context, $\textsc{ev}[\,] \equiv [\,] + (\lambda x{:}\,nat.\,x)\,3$, then it is easy to see that when $M \overset{lazy}{\to} M'$ the lazy reduction of $\textsc{ev}[M]$ will be $\textsc{ev}[M] \overset{lazy}{\to} \textsc{ev}[M']$, as guaranteed by Lemma 2.5.21. However, if we insert a lazy normal form such as 2 into the context, then the resulting term, $\textsc{ev}[2] \equiv 2 + (\lambda x{:}\,nat.\,x)\,3$, will have its lazy reduction completely to the right of the position marked by the placeholder $[\,]$ in the context. Another case arises with the first context, $\textsc{ev}[\,] \equiv ([\,]\,5) + (\lambda x{:}\,nat.\,x)\,3$. If we insert the lazy normal form $M \equiv \lambda y{:}\,nat.\,7+y$, we obtain a term $\textsc{ev}[M]$ whose lazy reduction involves a larger subterm than M. ∎

We note here that left-most reduction can also be characterized using evaluation contexts. To do so, we add six more forms to the definition of evaluation context, corresponding to the six rules that appear in Table 2.3 but not in Table 2.4. Lemma 2.5.20 extends easily to left-most reduction, but Lemma 2.5.21 requires the additional hypothesis that M does not have the form $\lambda x{:}\sigma.\, M_1$ or $\langle M_1, M_2\rangle$.

We use Lemma 2.5.21 in the analysis of parallel-or by showing that if the normal form of a compound term depends on the subterms $M_1, \ldots, M_k$, as we expect the parallel-or of M_1 and M_2 to depend on M_1 and M_2, then we may reduce the term to the form $\textsc{ev}[M_i]$, for some i independent of the form of $M_1, \ldots, M_k$. The intuition behind this is that some number of reduction steps may be independent of the chosen subterms. But if any of these subterms has any effect on the result, then eventually we must come to some term $\textsc{ev}[M_i]$ where the next reduction depends on the form of M_i. The "sequential" nature of PCF is that when we reach a term of the form $\textsc{ev}[M_i]$, it follows from Lemma 2.5.21 that if M_i is not a lazy normal form, we continue to reduce M_i until this subterm term does not lazy-reduce further.

Since we are interested in analyzing the reduction steps applied to a function of several arguments, we will use contexts with more than one "hole" (place for inserting a term). A *context* $\mathcal{C}[\,\cdot, \ldots, \cdot\,]$ *with* k *holes* is a syntactic expression with exactly the same form as a term, but containing zero or more occurrences of the placeholders $[\,]_1, \ldots, [\,]_k$, each assumed to have a fixed type within this expression. As for contexts with a single hole, we write $\mathcal{C}[M_1, \ldots, M_k]$ for the result of replacing the ith placeholder $[\,]_i$ by M_i, without renaming bound variables.

Our first lemma about reduction in arbitrary contexts is that the lazy reduction of a term of the form $\mathcal{C}[M_1, \ldots, M_k]$ is either a reduction on $\mathcal{C}$, independent of the terms $M_1, \ldots, M_k$ placed in the context, or a reduction that depends on the form of one of the terms $M_1, \ldots, M_k$.

Lemma 2.5.23 *Let $\mathcal{C}[\,\cdot, \ldots, \cdot\,]$ be a context with k holes. Suppose that there exist closed terms $N_1, \ldots, N_k$ of the appropriate types such that $\mathcal{C}[N_1, \ldots, N_k]$ is not in lazy normal form. Then $\mathcal{C}$ must have one of the following two properties:*

(i) There is a context $\mathcal{C}'[\,\cdot, \ldots, \cdot\,]$ such that for all closed terms $M_1, \ldots, M_k$ of the appropriate types $\mathcal{C}[M_1, \ldots, M_k] \stackrel{lazy}{\to} \mathcal{C}'[M_1, \ldots, M_k]$.

(ii) There is some i such that for all closed terms $M_1, \ldots, M_k$ of the appropriate types there is an evaluation context $\textsc{ev}[\,]$ with $\mathcal{C}[M_1, \ldots, M_k] \equiv \textsc{ev}[M_i]$.

We use Lemma 2.5.23 to prove the following statement about sequences of lazy reductions.

Lemma 2.5.24 *Let $\mathcal{C}[\,\cdot, \ldots, \cdot\,]$ be a context with k holes, let $M_1, \ldots, M_k$ be closed terms of the appropriate types for $\mathcal{C}$ and suppose $\mathcal{C}[M_1, \ldots, M_k]$ is a program with normal form N. Then either $\mathcal{C}[M'_1, \ldots, M'_k]$ reduces to N for all closed $M'_1, \ldots, M'_k$ of the appropriate types or there is some integer i such that for all closed $M'_1, \ldots, M'_k$ of the appropriate types there is an evaluation context $\textsc{ev}[\,]$ with $\mathcal{C}[M'_1, \ldots, M'_k] \stackrel{lazy}{\twoheadrightarrow} \textsc{ev}[M'_i]$.*

An intuitive reading of this lemma is that some number of reduction steps of a term $\mathcal{C}[M_1, \ldots, M_k]$ may be independent of all the M_i. If this does not produce a normal form, then eventually the reduction of $\mathcal{C}[M_1, \ldots, M_k]$ will reach a step that depends on the form of some M_i, with the number i depending only on $\mathcal{C}$.

If we replace lazy reduction with left-most reduction, then we may change lazy normal form to normal form in Lemma 2.5.23 and drop the assumption that $\mathcal{C}[M_1, \ldots, M_k]$ is a program in Lemma 2.5.24.

Using Lemmas 2.5.21 and 2.5.24, we now prove Theorem 2.5.19.

Proof of Theorem 2.5.19 Suppose we have a PCF expression POR defining parallel-or and consider the context $\mathcal{C}[\cdot\,,\cdot] \stackrel{def}{=} POR\,[\,]_1\,[\,]_2$. Since POR defines parallel-or, $\mathcal{C}[true, true] \twoheadrightarrow true$ and $\mathcal{C}[false, false] \twoheadrightarrow false$. By Lemma 2.5.24, there are two possibilities for the lazy reduction of $\mathcal{C}[true, true]$ to $true$. The first implies that $\mathcal{C}[M_1, M_2] \twoheadrightarrow true$ for all closed boolean terms M_1 and M_2. But this contradicts $\mathcal{C}[false, false] \twoheadrightarrow false$. Therefore, there is an integer $i \in \{1, 2\}$ such that for all closed boolean terms M_1, M_2, there is an evaluation context $\textsc{ev}[\,]$ with $\mathcal{C}[M_1, M_2] \stackrel{lazy}{\twoheadrightarrow} \textsc{ev}[M_i]$. But, by Lemma 2.5.21, this implies that if we apply POR to $true$ and the term $fix_{bool}\,(\lambda x\colon bool.\, x)$ with no normal form, making $fix_{bool}\,(\lambda x\colon bool.\, x)$ the ith argument, we cannot reduce the resulting term to $true$. This contradicts the assumption that POR defines parallel-or. ■

The reader may reasonably ask whether the non-definability of parallel-or is a peculiarity of PCF or a basic property shared by sequential programming languages such as Pascal and Lisp. The main complicating factor in comparing PCF to these languages is the order of evaluation. Since a Pascal function call `P(M,N)` is compiled so that expressions `M` and `N` are evaluated *before* `P` is called, it is clear that no boolean function `P` could compute parallel-or. However, we can ask whether parallel-or is definable by a Pascal (or Lisp) context with two boolean positions. More precisely, consider a Pascal program context `P`$[\cdot\,,\cdot]$ with two "holes" for boolean expressions (or bodies of functions that return boolean values). An important part of the execution of `P[M, N]` is what happens when `M` or `N` may run forever. Therefore, it is sensible and nontrivial to ask whether parallel-or is definable by a Pascal or Lisp context. If we were to go to the effort of formalizing the execution of Pascal programs in the way that we have formalized PCF evaluation, it seems very likely that we could prove that parallel-or is not definable by any context. The reader may enjoy trying to demonstrate otherwise by constructing a Pascal context that defines parallel-or and seeing what obstacles are involved. In Lisp, it is possible to define parallel-or by defining your own `eval` function. However, the apparent need to define a non-standard `eval` illustrates the sequentiality of standard Lisp evaluation.

Proof of Lemma 2.5.20 We use induction on the proof, in the system of Table 2.4, that $M \stackrel{lazy}{\to} N$. The base case is that $M \to N$ is one of the reduction axioms (these are listed in Table 2.2). If this is so, then the lemma holds with $\textsc{ev}[\,] \equiv [\,]$. The induction steps for the inference rules in Table 2.4 are all similar and essentially straightforward. For example, if we have $P + R \stackrel{lazy}{\to} Q + R$ by the rule

$$\frac{P \stackrel{lazy}{\to} Q}{P + R \stackrel{lazy}{\to} Q + R}\ ,$$

then by the induction hypothesis there is a unique evaluation context $\textsc{ev}'[\,]$ such that $P \equiv \textsc{ev}'[P']$, $P' \to Q'$ is a reduction axioms, and $Q \equiv \textsc{ev}'[Q']$. The lemma follows by taking $\textsc{ev}[\,] \equiv \textsc{ev}'[\,] + R$. ■

Proof of Lemma 2.5.21 We prove the lemma by induction on the structure of evaluation contexts. In the base case, we have an evaluation context $\textsc{ev}[\,] \equiv [\,]$. Since $\textsc{ev}[M] \equiv M$ and $\textsc{ev}[M'] \equiv M'$, it is easy to see that under the hypotheses of the lemma, $\textsc{ev}[M] \stackrel{lazy}{\to} \textsc{ev}[M']$.

There are a number of induction cases, corresponding to the possible syntactic forms of $\textsc{ev}[\,]$. Since the analysis of each of these is similar, we consider three representative cases.

The case $\textsc{ev}[\,] \equiv \textsc{ev}'[\,] + M_1$ is similar to most of the compound evaluation contexts. In this case, $\textsc{ev}[M] \equiv \textsc{ev}'[M] + M_1$ and, by the induction hypothesis, $\textsc{ev}'[M] \stackrel{lazy}{\to} \textsc{ev}'[M']$. Since $\textsc{ev}'[M]$ is not syntactically a numeral, no axiom can apply to the entire term $\textsc{ev}'[M] + M_1$. Therefore the

only rule in Table 2.4 that applies is

$$\frac{\text{EV}'[M] \overset{lazy}{\to} \text{EV}'[M']}{\text{EV}'[M] + M_1 \overset{lazy}{\to} \text{EV}'[M'] + M_1} \ .$$

This gives us $\text{EV}[M] \overset{lazy}{\to} \text{EV}[M']$.

The two other cases we will consider are projection and function application. A context $\text{EV}[\ ] \equiv \mathbf{Proj}_i\text{EV}'[\ ]$ is an evaluation context only if $\text{EV}'[\ ]$ does not have the syntactic form of a pair. Since $M \overset{lazy}{\to} M'$ implies that M is not a pair, we know that $\text{EV}[M]$ and $\text{EV}'[M]$ do not have the syntactic form of a pair. It follows that no reduction axiom applies to the entire term $\mathbf{Proj}_i\text{EV}'[M]$. Therefore, by the induction hypothesis that $\text{EV}'[M] \overset{lazy}{\to} \text{EV}'[M']$, we conclude that $\text{EV}[M] \equiv \mathbf{Proj}_i\text{EV}'[M] \overset{lazy}{\to} \mathbf{Proj}_i\text{EV}'[M'] \equiv \text{EV}[M']$.

The last case we consider is $\text{EV}[\ ] \equiv \text{EV}'[\ ]\,N$. This is similar to the projection case. The context $\text{EV}'[\ ]$ cannot be a lambda abstraction, by the definition of evaluation contexts, and M is not a lambda abstraction since $M \overset{lazy}{\to} M'$. It follows that $\text{EV}'[M]$ does not have the syntactic form of a lambda abstraction. Since the reduction axiom (β) does not apply to the entire term $\text{EV}'[M]\,N$, we use the induction hypothesis $\text{EV}'[M] \overset{lazy}{\to} \text{EV}'[M']$ and the definition of lazy reduction in Table 2.4 to conclude that $\text{EV}[M] \overset{lazy}{\to} \text{EV}[M']$. ∎

Proof of Lemma 2.5.23 We prove the lemma by induction on the structure of contexts. If $\mathcal{C}$ is a single symbol, then it is either one of the placeholders, $[\ ]_i$, a variable or a constant. It is easy to see that $\mathcal{C}$ has property (ii) if it is a placeholder and property (i) if it is some other symbol.

For a context $\mathcal{C}_1[\cdot, \ldots, \cdot] + \mathcal{C}_2[\cdot, \ldots, \cdot]$, we consider two cases, each dividing into two subcases when we apply the induction hypothesis. If there exist terms $M_1, \ldots, M_k$ with $\mathcal{C}_1[M_1, \ldots, M_k]$ not in lazy normal form, then we apply the induction hypothesis to $\mathcal{C}_1$. If $\mathcal{C}_1[M_1, \ldots, M_k] \overset{lazy}{\to} \mathcal{C}_1'[M_1, \ldots, M_k]$ for all $M_1, \ldots, M_k$, then it is easy to see from the definition of lazy reduction that

$$\mathcal{C}_1[M_1, \ldots, M_k] + \mathcal{C}_2[M_1, \ldots, M_k] \overset{lazy}{\to} \mathcal{C}_1'[M_1, \ldots, M_k] + \mathcal{C}_2[M_1, \ldots, M_k]$$

for all $M_1, \ldots, M_k$. On the other hand, if for every $M_1, \ldots, M_k$ we can write $\mathcal{C}_1[M_1, \ldots, M_k] \equiv \text{EV}[M_i]$, then

$$\mathcal{C}_1[M_1, \ldots, M_k] + \mathcal{C}_2[M_1, \ldots, M_k] \equiv \text{EV}[M_i] + \mathcal{C}_2[M_1, \ldots, M_k]$$

and we have an evaluation context for M_i. The second case is that $\mathcal{C}_1[M_1, \ldots, M_k]$ is a numeral and therefore $M_1, \ldots, M_k$ do not occur in $\mathcal{C}_1[M_1, \ldots, M_k]$. In this case, we apply the induction hypothesis to $\mathcal{C}_2$ and reason as above.

The induction steps for $Eq?$ and `if ... then ... else ...` are similar to the addition case. The induction steps for pairing and lambda abstraction are trivial, since these are lazy normal forms.

For a context $\mathbf{Proj}_i\mathcal{C}[\cdot, \ldots, \cdot]$, we must consider the form of $\mathcal{C}$. If $\mathcal{C}[\cdot, \ldots, \cdot] \equiv [\ ]_j$, then $\mathbf{Proj}_i\mathcal{C}[\cdot, \ldots, \cdot]$ satisfies condition (ii) of the lemma since this is an evaluation context. If $\mathcal{C}[\cdot, \ldots, \cdot] \equiv \langle \mathcal{C}_1[\cdot, \ldots, \cdot], \mathcal{C}_2[\cdot, \ldots, \cdot] \rangle$, then $\mathbf{Proj}_i\mathcal{C}[\cdot, \ldots, \cdot]$ satisfies condition (i). If $\mathcal{C}[\cdot, \ldots, \cdot]$ has some other form, then $\mathbf{Proj}_i\mathcal{C}[M_1, \ldots, M_k]$ cannot be a $(proj)$ redex. Therefore, we apply the induction hypothesis to $\mathcal{C}$ and reason as in the earlier cases.

The induction step for a context $\mathcal{C}_1[\cdot, \ldots, \cdot]\,\mathcal{C}_2[\cdot, \ldots, \cdot]$ is similar to the $\mathbf{Proj}_i$ case, except that we will need to use the assumption that all the terms we place in contexts are closed. Specifically, if $\mathcal{C}_1[\cdot, \ldots, \cdot] \equiv \lambda x{:}\sigma.\, \mathcal{C}_3[\cdot, \ldots, \cdot]$, then let $\mathcal{C}'$ be the context $\mathcal{C}'[\cdot, \ldots, \cdot] \equiv [\mathcal{C}_2[\cdot, \ldots, \cdot]/x]\mathcal{C}_1[\cdot, \ldots, \cdot]$.

For all *closed* $M_1, \ldots, M_k$ we have

$$(\lambda x{:}\sigma.\, \mathcal{C}_3[M_1, \ldots, M_k])\mathcal{C}_2[M_1, \ldots, M_k] \overset{lazy}{\to} \mathcal{C}'[M_1, \ldots, M_k]$$

and the context satisfies condition (i) of the lemma. If $\mathcal{C}_1[\cdot, \ldots, \cdot] \equiv [\,]_i$, then as in the projection case, the application context satisfies condition (ii) of the lemma. Finally, if $\mathcal{C}_1[\cdot, \ldots, \cdot]$ is some context that is not of one of these two forms, we reason as in the addition case, applying the induction hypothesis to either $\mathcal{C}_1$ or, if $\mathcal{C}_1[M_1, \ldots, M_k]$ is a lazy a normal form for all $M_1, \ldots, M_k$, the context $\mathcal{C}_2$. This completes the proof. ∎

Proof of Lemma 2.5.24 Let $\mathcal{C}[\cdot, \ldots, \cdot]$ be a context with k holes and let $M_1, \ldots, M_k$ be closed terms of the appropriate types such that $\mathcal{C}[M_1, \ldots, M_k]$ has normal form N of observable type. Then $\mathcal{C}[M_1, \ldots, M_k] \overset{lazy}{\twoheadrightarrow}_n N$, where n indicates the number of reduction steps. We prove the lemma by induction on n.

In the base case, $\mathcal{C}[M_1, \ldots, M_k] \equiv N$ is in normal form. There are two cases to consider. The degenerate one is that $\mathcal{C}[M'_1, \ldots, M'_k]$ is in normal form for all closed terms $M'_1, \ldots, M'_k$ of the appropriate types. But since the only results are numerals and boolean constants, $M'_1, \ldots, M'_k$ must not appear in N and the lemma easily follows. The second case is that $\mathcal{C}[M'_1, \ldots, M'_k]$ is not in normal form for some $M'_1, \ldots, M'_k$. Since $\mathcal{C}[M_1, \ldots, M_k]$ is in normal form, condition (i) of Lemma 2.5.23 cannot hold. Therefore $\mathcal{C}[M'_1, \ldots, M'_k] \equiv \textsc{ev}[M'_i]$.

In the induction step, with $\mathcal{C}[M_1, \ldots, M_k] \overset{lazy}{\twoheadrightarrow}_{m+1} N$, Lemma 2.5.23 gives us two cases. The simpler is that there is some i such that for all closed terms $M'_1, \ldots, M'_k$ of the appropriate types $\mathcal{C}[M'_1, \ldots, M'_k]$ has the form $\textsc{ev}[M'_i]$. But then clearly $\mathcal{C}[M'_1, \ldots, M'_k] \overset{lazy}{\twoheadrightarrow} \textsc{ev}[M'_i]$ and the lemma holds. The remaining case is that there is some context $\mathcal{C}'[\cdot, \ldots, \cdot]$ such that for all $M'_1, \ldots, M'_k$ of appropriate types, we have a reduction of the form

$$\mathcal{C}[M_1, \ldots, M_k] \overset{lazy}{\to} \mathcal{C}'[M_1, \ldots, M_k] \overset{lazy}{\twoheadrightarrow}_m N.$$

The lemma follows by the induction hypothesis for $\mathcal{C}'[\cdot \ldots, \cdot]$. This concludes the proof. ∎

Exercise 2.5.25 Show that Lemma 2.5.24 fails if we drop the assumption that $\mathcal{C}[M_1, \ldots, M_k]$ is a program.

Exercise 2.5.26 This exercise asks you to prove that parallel-conditional is not definable in PCF. Use Lemmas 2.5.21 and 2.5.24 to show that there is no PCF expression $PIF_{nat} : bool \to nat \to nat \to nat$ such that for all closed boolean expressions $M{:}\, bool$ and $N, P, Q{:}\, nat$ with Q in normal form,

$$PIF_{nat}\, M\, N\, P \twoheadrightarrow Q \;\text{ iff }\; \left[\begin{array}{l} M \twoheadrightarrow \mathit{true} \text{ and } N \text{ has normal form } Q \text{ or,} \\ M \twoheadrightarrow \mathit{false} \text{ and } P \text{ has normal form } Q \text{ or,} \\ N, P \text{ both have normal form } Q. \end{array}\right]$$

(If we extend Lemmas 2.5.21 and 2.5.24 to left-most reduction, the same proof shows that PIF_σ is not definable for any type σ.)

Exercise 2.5.27 The operational equivalence relation $=_{op}$ on terms is defined in Section 2.3.5. Show that if M and N are either closed terms of type nat or closed terms of type $bool$, neither having a normal form, then $M =_{op} N$.

Exercise 2.5.28 A plausible statement that is stronger than Lemma 2.5.24 is this:

> Let $\mathcal{C}[\,\cdot, \ldots, \cdot\,]$ be a context with k holes, let $M_1, \ldots, M_k$ be closed terms of the appropriate types for $\mathcal{C}$ and suppose $\mathcal{C}[M_1, \ldots, M_k] \overset{lazy}{\twoheadrightarrow} N$, where N does not further reduce by lazy reduction. Then either $\mathcal{C}[M'_1, \ldots, M'_k]$ reduces to N for all closed $M'_1, \ldots, M'_k$ of the appropriate types or there is some integer i such that for all closed $M'_1, \ldots, M'_k$ of the appropriate types there is an evaluation context $\textsc{ev}[\,]$ with $\mathcal{C}[M'_1, \ldots, M'_k] \overset{lazy}{\twoheadrightarrow} \textsc{ev}[M'_i]$.

The only difference is that we do not require N to be in normal form. Show that this statement is *false* by giving a counterexample. This may be done using a context $\mathcal{C}[\,]$ with only one placeholder.

Exercise 2.5.29 [Sto91a] This problem asks you to show that parallel-or, boolean parallel-conditional and natural-number parallel-conditional are all interdefinable. Parallel-or is defined in the statement of Theorem 2.5.19 and parallel-conditional is defined in Exercise 2.5.26. An intermediate step involves parallel-and, defined below.

(a) Show how to define POR from PIF_{bool} by writing a PCF expression containing PIF_{bool}.

(b) Show how to define PIF_{bool} from PIF_{nat} using an expression of the form

$$\lambda x\colon bool.\ \lambda y\colon bool.\ \lambda z\colon bool.\ Eq?\ 1\ (PIF_{nat}\ x\ M\ N).$$

(c) Parallel-and, $PAND$, has the following behavior:

$$PAND\ M\ N \twoheadrightarrow \begin{cases} true & \text{if } M \twoheadrightarrow true \text{ and } N \twoheadrightarrow true, \\ false & \text{if } M \twoheadrightarrow false \text{ or } N \twoheadrightarrow false, \\ \text{no normal form} & \text{otherwise.} \end{cases}$$

Show how to define $PAND$ from POR by writing a PCF expression containing POR.

(d) Show how to define PIF_{nat} from POR using an expression of the form

$$\lambda x\colon bool.\ \lambda y\colon nat.\ \lambda z\colon nat.\ search\ P$$

where $P\colon nat \to bool$ has x, y, z free and contains POR, $PAND$, and $Eq?$. The essential properties of *search* are summarized in Proposition 2.5.3 on page 95.

2.6 Variations and Extensions of PCF

2.6.1 Summary of extensions

This section briefly summarizes several important extensions of PCF. All are obtained by adding new types. The first extension is a very simple one, a type *unit* with only one element. The second extension is *sum,* or disjoint union, types. With unit and disjoint unions, we can define *bool* as the disjoint union of *unit* and *unit*. This makes the primitive type *bool* unnecessary. The next extension is recursive type definitions. With recursive type definitions, *unit* and sum types, we can define *nat* and its operations, making the primitive type *nat* also unnecessary. Other commonly-used types that have straightforward recursive definitions are stacks, lists and trees. Another use of recursive types is that we can define the fixed-point operator *fix* on any type. Thus with *unit*, sums and recursive types, we may define all of the type and term constants of PCF.

The final language is a variant of PCF with lifted types, which give us a different view of nontermination. With lifted types, written in the form $\sigma_\perp$, we can distinguish between natural-number expressions that necessarily have a normal form and those that may not. Specifically, natural-number terms that do not involve *fix* may have type *nat*, while terms that involve *fix*, and therefore may not have a normal form, have the lifted type $nat_\perp$. Thus the type *nat* of PCF corresponds to type $nat_\perp$ in this language. The syntax of many common programs becomes more complicated, since the distinction between lifted and unlifted types forces us to include additional lifting operations in terms. However, the refinement achieved with lifted types has some theoretical advantages. One is that many more equational or reduction axioms become consistent with recursion. Another is that we may study different evaluation orders within a single framework. For this reason, explicit lifting seems useful in meta-languages for studying other programming languages.

All of the extensions summarized here may be combined with polymorphism, treated in Chapter 9 and other typing ideas in Chapters 10 and 11.

2.6.2 Unit and sum types

We add the one-element type *unit* to PCF, or any language based on typed lambda calculus, by adding *unit* to the type expressions and adding the constant $*\colon unit$ to the syntax of terms. The equational axiom for $*$ is

$$M = *\colon unit$$

for any term $M\colon unit$. Intuitively, this axiom says that every element of type *unit* is equal to $*$. This may be used as a reduction rule, read left to right. While *unit* may not seem very interesting, it is surprisingly useful in combination with sums and other forms of types.

It should be mentioned that the reduction rule for *unit* terms causes confluence to fail, when combined with η-reduction [CD91, LS86]. This does not have an immediate consequence for PCF, since we do not use η-reduction.

Intuitively, the *sum type* $\sigma + \tau$ is the disjoint union of types σ and τ. The difference between a disjoint union and an ordinary set union is that with a disjoint union, we can tell which type any element comes from. This is particularly noticeable when the two types overlap or are identical. For example, if we take the set union of *int* and *int*, we just get *int*. In contrast, the disjoint union of *int* and *int* has two "copies" of *int*. Informally, we think of the sum type $int + int$ as the collection of "tagged" integers, with each tag indicating whether the integer comes from the left or right *int* in the sum.

The term forms associated with sums are injection and case functions. For any types σ and τ, the injection functions $\mathbf{Inleft}^{\sigma,\tau}$ and $\mathbf{Inright}^{\sigma,\tau}$ have types

$$\begin{aligned} \mathbf{Inleft}^{\sigma,\tau} &: \sigma \to \sigma + \tau \\ \mathbf{Inright}^{\sigma,\tau} &: \tau \to \sigma + \tau \end{aligned}$$

Intuitively, the injection functions map σ or τ to $\sigma+\tau$ by "tagging" elements. However, since the operations on tags are encapsulated in the injection and case functions, we never say exactly what the tags actually are.

The **Case** function applies one of two functions to an element of a sum type. The choice between functions depends on which type of the sum the element comes from. For all types σ, τ and ρ, the case function $\mathbf{Case}^{\sigma,\tau,\rho}$ has type

$$\mathbf{Case}^{\sigma,\tau,\rho} : (\sigma + \tau) \to (\sigma \to \rho) \to (\tau \to \rho) \to \rho$$

Intuitively, $\mathbf{Case}^{\sigma,\tau,\rho}\, x\, f\, g$ inspects the tag on x and applies f if x is from σ and g if x is from τ. The main equational axioms are

$$\begin{aligned} \mathbf{Case}^{\sigma,\tau,\rho}\,(\mathbf{Inleft}^{\sigma,\tau}\, x)\, f\, g &= f x \\ \mathbf{Case}^{\sigma,\tau,\rho}\,(\mathbf{Inright}^{\sigma,\tau}\, x)\, f\, g &= g x \end{aligned}$$

Both of these give us reduction axioms, read left to right. There is also an extensionality axiom for sum types,

$$\mathbf{Case}^{\sigma,\tau,\rho}\, x\, (f \circ \mathbf{Inleft}^{\sigma,\tau})\, (f \circ \mathbf{Inright}^{\sigma,\tau}) = f\, x,$$

where $f\colon(\sigma+\tau)\to(\sigma+\tau)$. Some consequences of this axiom are given in Exercise 2.6.3. Since the extensionality axiom leads to inconsistency, when combined with *fix*, we do not include this equational axiom in the extension of PCF with sum types (see Exercise 2.6.4). We will drop the type superscripts from injection and case functions when the type is either irrelevant or determined by context.

As an illustration of *unit* and sum types, we will show how *bool* may be eliminated if *unit* and sum types are added to PCF. We define *bool* by

$$bool \stackrel{def}{=} unit + unit$$

and boolean values *true* and *false* by

$$\begin{aligned} true &\stackrel{def}{=} \mathbf{Inleft}\, * \\ false &\stackrel{def}{=} \mathbf{Inright}\, * \end{aligned}$$

The remaining basic boolean expression is conditional, `if ... then ... else ...`. We may consider this sugar for **Case**, as follows:

$$\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P \stackrel{def}{=} \mathbf{Case}^{unit,unit,\rho}\, M\, (K_{\rho,unit} N)\, (K_{\rho,unit} P),$$

where $N, P\colon \rho$ and $K_{\rho,unit}$ is the lambda term $\lambda x\colon \rho.\, \lambda y\colon unit.\, x$ that produces a constant function. To show that this works, we must check the two equational axioms for conditional:

$$\begin{aligned} \texttt{if}\ true\ \texttt{then}\ M\ \texttt{else}\ N &= M, \\ \texttt{if}\ false\ \texttt{then}\ M\ \texttt{else}\ N &= N. \end{aligned}$$

The reader may easily verify that when we eliminate syntactic sugar, the first equational axiom for **Case** yields `if` *true* `then` M `else` $N = K_{\rho,unit} M * = M$, and similarly for the second equation.

Other uses of sum types include *variants* and *enumeration types,* considered in Exercises 2.6.1 and 2.6.2.

Exercise 2.6.1 A *variant type* is a form of labeled sum, bearing the same relationship to sum types as records do to product types. A variant type defining a sum of types $\sigma_1, \ldots, \sigma_k$ is written $[\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k]$, where $\ell_1, \ldots, \ell_k$ are distinct syntactic labels. As for records, the order in which we write the label/type pairs does not matter. For example, $[A\colon int, B\colon bool]$ is the same type as $[B\colon bool, A\colon int]$.

Intuitively, an element of the variant type $[\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k]$ is an element of one of the types σ_i, for $1 \leq i \leq k$, tagged with label ℓ_i. More precisely, if $M\colon \sigma_i$, then the expression $\ell_i^{[\ell_1:\sigma_1,\ldots,\ell_k:\sigma_k]}(M_i)$ is an expression of type $[\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k]$. The case function for $[\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k]$ has type

$$\mathbf{Case}^{[\ell_1:\sigma_1,\ldots,\ell_k:\sigma_k],\,\rho} : [\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k] \to (\sigma_1 \to \rho) \to \ldots \to (\sigma_k \to \rho) \to \rho$$

and satisfies the equation

$$\mathbf{Case}^{[\ell_1:\sigma_1,\ldots,\ell_k:\sigma_k],\,\rho}\, \ell_i(M_i)\, f_1 \ \ldots\ f_k \ = \ f_i\, M_i.$$

(Although the label/type pairs are unordered in $[\ell_1\colon \sigma_1, \ldots, \ell_k\colon \sigma_k]$, the order of arguments clearly matters for $\mathbf{Case}^{[\ell_1:\sigma_1,\ldots,\ell_k:\sigma_k],\,\rho}$.) Following the pattern we use to treat records as sugar for products (see Section 2.5.1), show how to treat variants as syntactic sugar for sums. Illustrate this by translating an expression containing occurrences of two different variant types into PCF with sums.

Exercise 2.6.2 *Enumeration types* appear in Pascal and some subsequent programming languages. The elements of an enumeration type $(\ell_1, \ldots, \ell_n)$ are the literals $\ell_1, \ldots, \ell_n$, which are considered distinct values not equal to any values of any other type. (We do not consider subranges or subtypes of enumeration types here.) Show how to regard the enumeration type $(\ell_1, \ldots, \ell_n)$ as syntactic sugar for the variant $[\ell_1\colon unit, \ldots, \ell_k\colon unit]$ and illustrate your technique by showing how to translate an expression containing enumeration types and literals into a PCF expression with *unit* and variants.

Exercise 2.6.3 Prove the following consequences of the extensionality axiom for sum types, using the equational axioms and inference rules of PCF as needed.

(a) $\mathbf{Case}\, x\, \mathbf{Inleft}\, \mathbf{Inright} \ = \ x.$

(b) $f(\mathbf{Case}\, x\, (g \circ \mathbf{Inleft})\, (g \circ \mathbf{Inright})) \ = \ \mathbf{Case}\, x\, (f \circ g \circ \mathbf{Inleft})\, (f \circ g \circ \mathbf{Inright}).$

(c) $f(\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P) = \texttt{if}\ M\ \texttt{then}\ fN\ \texttt{else}\ fP$, when desugared into sum operations as described in this section.

(d) $M \ = \ \lambda z\colon \sigma + \tau.\, \mathbf{Case}\, z\, N\, P : (\sigma + \tau) \to (\sigma + \tau)$ is provable whenever $\lambda x\colon \sigma.\, M(\mathbf{Inleft}\, x) = N : \sigma \to (\sigma + \tau)$ and $\lambda y\colon \tau.\, M(\mathbf{Inright}\, y) = P : \tau \to (\sigma + \tau)$ are both provable and z is not free in N or P.

Exercise 2.6.4 This exercise asks you to show that the extensionality axiom for sums is inconsistent with the equational axiom for *fix,* when combined with the rest of the equational proof system of PCF. The first observation of this phenomenon is generally credited to Lawvere [Law69].

(a) Define functions $not\colon bool \to bool$ and $eq?\colon bool \to bool \to bool$ and show that when we regard $bool$ as sugar for $unit + unit$, we can prove the following three equations with boolean variable x:

$$\begin{array}{lcl} not\,(not\,x) & = & x \\ eq?\,x\,(not\,x) & = & false \\ eq?\,x\,x & = & true \end{array}$$

You may use the results of Exercise 2.6.3.

(b) Recall from Exercise 2.3.2 that every equation between well-formed terms of the same type is provable from $true = false$ by the axioms and inference rules of PCF. Use the three equations proved in part (a) to prove $true = false$. You can do this by substituting a boolean expression of the form $fix\,B$ for x in two of the equations.

2.6.3 Recursive types

In some programming languages, it is possible to define types recursively. An example is ML, which has recursive `datatype` declarations (see Exercise 2.6.8). We may add recursively-defined types to PCF, or any related language based on lambda calculus, by adding type variables $r, s, t, \ldots$ to the syntax of type expressions and a new form of type expression, $\mu t.\sigma$. Intuitively, $\mu t.\sigma$ is the smallest type (collection of values) satisfying the equation

$$t \;=\; \sigma,$$

where t will generally occur in σ. As with recursive definitions in PCF, we could introduce a fixed-point operator and write $fix\,(\lambda t.\,\sigma)$ for a solution to the equation $t = \sigma$. The syntax $\mu t.\sigma$ could be considered sugar for $fix\,(\lambda t.\,\sigma)$. However, we will avoid lambda abstraction in types by taking μ as primitive. We consider the denotational semantics of recursive type declarations in Section 7.4.

The operator μ binds t in $\mu t.\sigma$, so we have both free and bound type variables in type expressions. However, since no term constructs bind type variables, we only use closed types in terms. More specifically, the type expressions of PCF with nat and $bool$ eliminated but $unit$, sum and recursive types added are the *closed* type expressions generated by the following grammar:

$$\sigma \;::=\; t \mid unit \mid \sigma + \sigma \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \mu t.\sigma$$

Since terms that contain types with free type variables lead to polymorphism, these are discussed in Chapter 9. We consider type expressions that differ in the names of bound variables equivalent.

With recursive types, we must consider the issue of type equality with some care. Although intuitively $\mu t.\sigma$ introduces a solution to the equation $t = \sigma$, there are two possible interpretations. One is that these are truly indistinguishable types. In this view, type equality becomes relatively complicated, since $t = \sigma$ implies that $t = [\sigma/t]\sigma$. Written using the μ syntax, this gives us the equation

$$\mu t.\sigma \;=\; [\mu t.\sigma/t]\sigma$$

Many other equations follow, including some that are not directly derivable from an axiom of this form [AC91]. While the type equality view has some appeal, this makes it more difficult to determine whether terms are well typed. Specifically, we must consider type equality in terms, allowing a term of one type to be used wherever a term of equal but syntactically different type is required. We may avoid this by taking the second view of type recursion.

The alternate view is to solve an equation $t = \sigma$ by finding a type t that is *isomorphic* to σ. Isomorphic, in this context, means that there is a function from $\mu t.\sigma$ to $[\mu t.\sigma\,/\,t]\sigma$, and one in the opposite direction, each the inverse of the other. We express this by writing

$$\mu t.\sigma \;\cong\; [\mu t.\sigma/t]\sigma$$

where $\cong$ indicates an isomorphism. In the isomorphism view, $\mu t.\sigma$ is not *equal* to $[\mu t.\sigma/t]\sigma$, but there are functions that allow us to "convert" a value of type $\mu t.\sigma$ to $[\mu t.\sigma/t]\sigma$, and vice versa. With recursive types satisfying isomorphisms instead of equations, we may continue to use syntactic equality for type equality (except for renaming of bound type variables). The price is that we must write the conversion functions between $\mu t.\sigma$ and $[\mu t.\sigma/t]\sigma$ in terms so that we know the syntactic type of each term exactly.

The term forms associated with recursive types allow us to convert a term of type $\mu t.\sigma$ to type $[\mu t.\sigma/t]\sigma$, and vice versa.

$$\text{If } M\colon [\mu t.\sigma/t]\sigma \text{ then } \mathit{up}\, M\colon \mu t.\sigma.$$

$$\text{If } M\colon \mu t.\sigma \text{ then } \mathit{dn}\, M\colon [\mu t.\sigma/t]\sigma.$$

These two functions are inverses, as stated by the following equational axioms.

$$\mathit{dn}\,(\mathit{up}\, M) = M, \quad \mathit{up}\,(\mathit{dn}\, M) = M.$$

Only the first axiom is traditionally used as a reduction rule (read left to right). While the operations *up* and *dn* disambiguate the types of terms, these are often cumbersome to write. In later chapters, we will occasionally omit *up* and *dn* from examples to improve readability.

We will show that with recursive type definitions, *unit* and sum types, we can define *nat*, numerals for $0, 1, 2, \ldots$, and the successor, predecessor and zero-test functions. As shown in Exercise 2.5.7, we may replace addition and equality test ($Eq?$) of PCF by successor, predecessor and zero-test without changing the set of definable functions. Therefore, we can translate any expression of PCF into the language with recursive type definitions, *unit* and sum types.

A distinguishing feature of the natural numbers is that if we add one more element to the set of natural numbers, we obtain a set that is in one-to-one correspondence, or isomorphic, to the set of natural numbers. Since the sum $\mathit{unit} + \tau$ has one more element than τ, we therefore expect *nat* to satisfy the isomorphism

$$\mathit{nat} \;\cong\; \mathit{unit} + \mathit{nat}$$

This leads us to the following definition of *nat* as a recursive type

$$\mathit{nat} \;\stackrel{\mathit{def}}{=}\; \mu t.\, \mathit{unit} + t$$

Intuitively, this may be understood by considering the isomorphisms

$$\begin{array}{rcl}
\mathit{nat} & \cong & \mathit{unit} + \mathit{nat} \\
& \cong & \mathit{unit} + (\mathit{unit} + \mathit{nat}) \\
& \cong & \mathit{unit} + (\mathit{unit} + (\mathit{unit} + \mathit{nat})) \\
& \cong & \ldots \\
& \cong & \mathit{unit} + (\mathit{unit} + (\mathit{unit} + (\mathit{unit} + \ldots + \mathit{nat}\ldots))))
\end{array}$$

Since we can continue this expansion as long as we like, we can think intuitively of *nat* as the disjoint union of infinitely many one-element types. We can think of the natural number 0 as coming from the first *unit* in this sum, 1 from the second *unit*, and so on for each natural number.

To avoid confusion between natural numbers and the terms that we use to represent natural numbers, we will write $\lceil n \rceil$ for the numeral (term) for n. Following the intuitive picture above, we let the numeral $\lceil 0 \rceil$ for natural number 0 be the term

$$\lceil 0 \rceil \stackrel{def}{=} up\ (\mathbf{Inleft} *).$$

For any natural number $n > 0$, we may similarly define the numeral $\lceil n \rceil$ by

$$\lceil n \rceil \stackrel{def}{=} up\ (\mathbf{Inright}\ up\ (\mathbf{Inright}\ \ldots\ up\ (\mathbf{Inleft} *) \ldots))$$

with n occurrences of **Inright** and $n+1$ applications of up. The successor function just applies up and **Inright**:

$$succ \stackrel{def}{=} \lambda x{:}\, nat.\, up\,(\mathbf{Inright}\, x)$$

We can check that this has the right type by noting that if $x{:}\, nat$, then $\mathbf{Inright}\, x$ belongs to $unit + nat$. Therefore, $up\,(\mathbf{Inright}\, x){:}\, nat$. The reader may verify that $S\lceil n \rceil = \lceil n+1 \rceil$ for any natural number n.

The zero-test operation works by taking $x{:}\, nat$ and applying dn to obtain an element of $unit + nat$, then using **Case** to see whether the result is an element of $unit$ (*i.e.*, equal to 0) or not.

$$zero? \stackrel{def}{=} \lambda x{:}\, nat.\, \mathbf{Case}^{unit,nat,bool}\,(dn\ x)\,(\lambda y{:}\, unit.\, true)\,(\lambda z{:}\, nat.\, false)$$

The reader may easily check that $zero?\,\lceil 0 \rceil = true$ and $zero?\,\lceil n \rceil = false$ for $n > 0$.

The final operation we need is predecessor. Recall that although there is no predecessor of 0, it is conventional to take $pred\, 0 = 0$. The predecessor function is similar to *zero?* since it works by taking $x{:}\, nat$ and applying dn, then using **Case** to see whether the result is an element of $unit$ (*i.e.*, equal to 0) or not. If so, then the predecessor is 0; otherwise, the element of nat already obtained by applying dn is the predecessor.

$$pred \stackrel{def}{=} \lambda x{:}\, nat.\, \mathbf{Case}^{unit,nat,bool}\,(dn\ x)\,(\lambda y{:}\, unit.\, 0)\,(\lambda z{:}\, nat.\, z)$$

The reader may verify that for any $x{:}\, nat$ we have $pred(succ\, x) = x$. (This is part of Exercise 2.6.5 below.)

A second illustration of the use of recursive types has a very different flavor. While most common uses of recursive types involve data structures, we may also write recursive definitions over any type using recursive types. More specifically, we may define a function fix_σ with the reduction property $fix_\sigma \twoheadrightarrow \lambda f{:}\, \sigma \to \sigma.\, f(fix_\sigma\, f)$ using only typed lambda calculus with recursive types. The main idea is to use "self-application," which is not possible without recursive types. More specifically, if M is any PCF term, the application MM cannot be well-typed. The reason is simply that if $M{:}\, \tau$, then τ does not have enough symbols to have the form $\tau \to \tau'$. However, with recursive types, $(dn\, M)M$ may be well-typed, since the type of M could have the form $\mu t.t \to \tau$.

We will define fix_σ using a term of the form $dn\, MM$. The main property we want for $dn\, MM$ is that

$$dn\, MM \quad \twoheadrightarrow \quad \lambda f{:}\, \sigma \to \sigma.\, f(dn\, MM\, f).$$

This reduction gives us a good clue about the form of M. Specifically, we can let M have the form

$$M \stackrel{def}{=} up\,(\ \lambda x{:}\, \tau.\, \lambda f{:}\, \sigma \to \sigma.\, f(dn\, x\, x\, f)\ )$$

if we can find an appropriate type τ. Since we want $dn\,MM : (\sigma \to \sigma) \to \sigma$, the expression $dn\,x\,x\,f$ in the body of M must have type σ. Therefore, τ must satisfy the isomorphism

$$\tau \;\cong\; \tau \to (\sigma \to \sigma) \to \sigma.$$

We can easily achieve this by letting

$$\tau \;\stackrel{def}{=}\; \mu t.\, t \to (\sigma \to \sigma) \to \sigma.$$

We leave it to the reader to verify that if we define $fix_\sigma \stackrel{def}{=} dn\,MM$, or more simply

$$fix_\sigma \;\stackrel{def}{=}\; (\lambda x{:}\tau.\, \lambda f{:}\sigma \to \sigma.\, f(dn\,x\,x\,f))\;(up\;(\lambda x{:}\tau.\, \lambda f{:}\sigma \to \sigma.\, f(dn\,x\,x\,f)))$$

we have a well-typed expression with $fix_\sigma \;\twoheadrightarrow\; \lambda f{:}\sigma \to \sigma.\, f(fix_\sigma\, f)$.

The fixed-point operator we have just derived is a typed version of the fixed-point operator Θ from untyped lambda calculus [Bar84]. As outlined in Exercise 2.6.7 below, any untyped lambda term may be written in a typed way using recursive types.

Exercise 2.6.5 Using the definitions given in this section, show that we have the following reductions on natural number terms.

(a) For every natural number n, we have $S\lceil n \rceil \twoheadrightarrow \lceil n+1 \rceil$.

(b) $zero?\,\lceil 0 \rceil \twoheadrightarrow true$ and $zero?\,\lceil n \rceil \twoheadrightarrow false$ for $n > 0$.

(c) If x is a natural number variable, then $pred(succ\,x) \twoheadrightarrow x$.

Exercise 2.6.6 We may define the type of lists of natural numbers using the following recursive type:

$$list \;\stackrel{def}{=}\; \mu t.\, unit + (nat \times t)$$

The intuition behind this definition is that a list is either empty, which we represent using the element $*{:}\,unit$, or may be regarded as a pair consisting of a natural number and a list.

(a) Define the empty list $nil{:}\,list$ and the function $cons{:}\,nat \times list \to list$ that adds a natural number to a list.

(b) The function car returns the first element of a list and the function cdr returns the list following the first element. We can define versions of these functions that make sense when applied to the empty list by giving them types

$$\begin{array}{rcl} car & : & list \to unit + nat \\ cdr & : & list \to unit + list \end{array}$$

and establishing the convention that when applied to the empty list, each returns $\mathbf{Inleft}\,*$. Write terms defining car and cdr functions so that the following equations are provable:

$$\begin{array}{lcl} car\;nil & = & \mathbf{Inleft}\,* \\ car\;(cons\,x\,\ell) & = & x \\ cdr\;nil & = & \mathbf{Inleft}\,* \\ cdr\;(cons\,x\,\ell) & = & \ell \end{array}$$

for $x{:}\,nat$ and $\ell{:}\,list$.

(c) Using recursion, it is possible to define "infinite" lists of type *list*. Show how to use *fix* to define a list L_n, for any numeral n, such that $car\, L_n = n$ and $cdr\, L_n = L_n$.

Exercise 2.6.7 The terms of pure untyped lambda calculus (without constant symbols) are given by the grammar

$$U \quad ::= \quad x \mid UU \mid \lambda x.\, U$$

The main equational properties of untyped lambda terms are untyped versions of (α), (β) and (η). We may define untyped lambda terms in typed lambda calculus with recursive types using the type $\mu t.\, t \to t$. Intuitively, the reason this type works is that terms of type $\mu t.\, t \to t$ may be used as functions on this type, and conversely. This is exactly what we need to make sense of the untyped lambda calculus.

(a) Show that if we give every free or bound variable type $\mu t.\, t \to t$, we may translate any untyped term into a typed lambda term of type $\mu t.\, t \to t$ by inserting up , dn , and type designations on lambda abstractions. Verify that untyped (α) follows from typed (α), untyped (β) follows from $dn\,(up\, M) = M$ and untyped (β) and, finally, untyped (η) follows from typed (η) and $up\,(dn\, M) = M$.

(b) Without recursive types, every typed lambda term without constants has a unique normal form. Show that by translating $(\lambda x.\, xx)\, \lambda x.\, xx$ into typed lambda calculus, we can use recursive types to write terms with no normal form.

(c) A well-known term in untyped lambda calculus is the fixed-point operator

$$Y \stackrel{def}{=} \lambda f.(\lambda x.\, f(xx))(\lambda x.\, f(xx)).$$

Unlike Θ, this term is only an equational fixed-point operator, *i.e.*, $Yf = f(Yf)$ but Yf does not reduce to $f(Yf)$. Use a variant of the translation given in part (a) to obtain a typed equational fixed point operator from Y. Your typed term should have type $(\sigma \to \sigma) \to \sigma$. Show that although Yf does not reduce to $f(Yf)$, Yf reduces to a term M_f such that $M_f \twoheadrightarrow f\, M_f$.

Exercise 2.6.8 The programming language ML has a form of recursive type declaration, called `datatype`, that combines sums and type recursion. In general, a datatype declaration has the form

$$\texttt{datatype}\ t \ = \ \ell_1\ \texttt{of}\ \sigma_1 \mid \ \ldots\ \mid \ell_k\ \texttt{of}\ \sigma_k$$

where the syntactic labels $\ell_1, \ldots, \ell_k$ are used in the same way as in variant types (labeled sums). Intuitively, this declaration defines a type t that is the sum of $\sigma_1, \ldots, \sigma_k$, with labels $\ell_1, \ldots, \ell_k$ used as injection functions as in Exercise 2.6.1. If the declared type t occurs in $\sigma_1, \ldots, \sigma_k$, then this is a recursive declaration of type t. Note that the vertical bar $\mid$ is part of the syntax of ML, not the meta-language we are using to describe ML. A similar exercise on ML datatype declarations, using algebraic specifications instead of recursive types and sums, appears in Chapter 3 (Exercise 3.6.4).

The type of binary trees, with natural numbers at the leaves, may be declared using `datatype` by

$$\texttt{datatype}\ \mathit{tree} \ = \ \mathit{leaf}\ \texttt{of}\ \mathit{nat} \mid \mathit{node}\ \texttt{of}\ \mathit{tree} \times \mathit{tree}.$$

Using the notation for variants given in Exercise 2.6.1, this declares a type *tree* satisfying

$$\mathit{tree} \ \cong \ [\mathit{leaf}\colon \mathit{nat},\ \mathit{node}\colon \mathit{tree} \times \mathit{tree}]$$

(a) Explain how to regard a `datatype` declaration as a recursive type expression of the form $\mu t.[\ldots]$ using variants as in Exercise 2.6.1. Illustrate your general method on the type of trees above.

(b) A convenient feature of ML is the way that pattern matching may be used to define functions over declared datatypes. A function over trees may be declared using two "clauses," one for each form of tree. In our variant of ML syntax, a function over *tree*'s defined by pattern matching will have the form

$$\begin{array}{rlcl} \texttt{letrec fun} & f(\mathit{leaf}(x\colon \mathit{nat})) & = & M \\ \mid & f(\mathit{node}(t_1\colon \mathit{tree}, t_2\colon \mathit{tree})) & = & N \\ & \texttt{in } P & & \end{array}$$

where variables $x\colon \mathit{nat}$ and $f\colon \mathit{tree} \to \sigma$ are bound in M, $t_1, t_2\colon \mathit{tree}$ and f are bound in N and f is bound in the the declaration body P. The compiler checks to make sure there is one clause for each constructor of the datatype. For example, we may declare a function f that sums up the values of all the leaves as follows.

$$\begin{array}{rlcl} \texttt{letrec fun} & f(\mathit{leaf}(x\colon \mathit{nat})) & = & x \\ \mid & f(\mathit{node}(t_1\colon \mathit{tree}, t_2\colon \mathit{tree})) & = & f(t_1) + f(t_2) \end{array}$$

Explain how to regard a function definition with pattern matching as a function with **Case** and illustrate your general method on the function that sums up the values of all the leaves of a tree.

2.6.4 Lifted types

Lifted types may be used to distinguish possibly nonterminating computations from ones that are guaranteed to terminate. We can see how lifted types could have been used in PCF by recalling that every term that does not contain *fix* has a normal form. If we combine basic boolean and natural number expressions, pairs and functions, we therefore have a language in which every term has a normal form. Let us call this language PCF_0. In Section 2.2.5, we completed the definition of PCF by adding *fix* to PCF_0. In the process, we extended the set of terms of each type in a nontrivial way. For example, while every PCF_0 term of type *nat* reduces to one of the numerals $0, 1, 2, \ldots$, PCF contains terms such as $\mathit{fix}\,(\lambda x\colon \mathit{nat}.\, x)$ of type *nat* that do not have any normal form and are not provably equal to any numeral. This means that if we want to associate a mathematical value with every PCF term, we cannot think of the values of type *nat* as simply the natural numbers. We must consider some superset of the natural numbers that contains values for all the terms without normal form. (Since it is very reasonable to give all *nat* terms without normal form the same value, we only need one new value, representing nontermination, in addition to the usual natural numbers.) An alternative way of extending PCF_0 with a fixed-point operator is to use lifted types. Although lifted types are sometimes syntactically cumbersome, when it comes to writing common functions or programs, lifted types have some theoretical advantages since they let us clearly identify the sources of nontermination, or "undefinedness," in expressions.

One reason to use lifted types is that many equational axioms that are inconsistent with *fix* may be safely combined with lifted types. One example, which is related to the inconsistency of sum types and *fix* outlined in Exercise 2.6.4, is the pair of equations

$$\begin{array}{lcll} \mathit{Eq}?\ x\ x & = & \mathit{true}, & \\ \mathit{Eq}?\ x\ (n + x) & = & \mathit{false} & \text{numeral } n \text{ different from } 0. \end{array}$$

While these equations make good sense for the ordinary natural numbers, it is inconsistent to add them to PCF. To see this, the reader may substitute $\mathit{fix}\,(\lambda x\colon \mathit{nat}.\, 1 + x)$ for x in both equations and derive $\mathit{true} = \mathit{false}$. In contrast, it follows from the semantic construction in Section 5.2 that we may consistently add these two equations to the variant of PCF with lifted types (see Example 2.6.10).

Another important reason to consider lifted types is that this provides an interesting and insightful way to incorporate alternative reduction orders into a single system. In particular, as we shall see below, both ordinary (nondeterministic or leftmost) PCF and Eager PCF (defined in Section 2.4.5) can be expressed in PCF with lifted types. This has an interesting consequence for equational reasoning about eager reduction. The equational theory of Eager PCF is not given in Section 2.4.5 since it is relatively subtle and has a different form from the equational theory of PCF. An advantage of lifted types is that we may essentially express eager reduction, and at the same time allow ordinary equational reasoning. In particular, equational axioms may be used to "optimize" a term without changing its normal form. This follows from confluence of PCF with lifted types [How92]. These advantages notwithstanding, the use of lifted types in programming language analysis is a relatively recent development and some syntactic and semantic issues remain unresolved. Consequently, this section will be a high-level and somewhat informal overview.

The types of the language $\mathrm{PCF}_\perp$, called "PCF with lifted types," are given by the grammar

$$\sigma \quad ::= \quad \mathit{bool} \mid \mathit{nat} \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \sigma_\perp$$

These are the types of PCF, plus the form $\sigma_\perp$ which is read "σ lifted." Intuitively, the elements of $\sigma_\perp$ are the same as the elements of σ, plus the possibility of "nontermination" or "divergence".

Mathematically, we can interpret $\sigma_\perp$ as the union of σ and one extra element, written $\perp_\sigma$, which serves as the value of any "divergent" term that is not equal to an element of type σ.

The terms of $\text{PCF}_\perp$ include all of the general forms of PCF, extended to $\text{PCF}_\perp$ types. Specifically, we have pairing, projection, lambda abstraction and application for all product and function types. Since we can use `if ... then ... else ...` on all types in PCF, we extend this to all $\text{PCF}_\perp$ types. More specifically, `if` M `then` N `else` P is well-formed whenever $M\colon \mathit{bool}$ and $N, P\colon \sigma$, for any σ. However, the basic natural number and boolean functions of PCF (numerals, boolean constants, addition and equality test) keep the same types as in PCF. For example, $5 + x$ is a nat expression of $\text{PCF}_\perp$ if $x\colon \mathit{nat}$; it is not an expression of type $\mathit{nat}_\perp$. The equational and reduction axioms for all of these operations are the same as for PCF. The language $\text{PCF}_\perp$ also has two operations associated with lifted types and a restricted fixed-point operator. Since both involve a class of types called the "pointed" types, we discuss these before giving the remaining basic operations of the language.

Intuitively, the pointed types of $\text{PCF}_\perp$ are the types that contain terms which do not yield any meaningful computational value. More precisely, we can say that a closed term $M\colon\sigma$ is *noninformative* if, for every context $\mathcal{C}[\ ]$ of observable type, if $\mathcal{C}[M]$ has a normal form, then $\mathcal{C}[N]$ has the same normal form for every closed $N\colon\sigma$. (An example noninformative term is $\mathit{fix}(\lambda x\colon\sigma.\, x)$; see Lemma 2.5.24.) Formally, we say a type σ *is pointed* if either $\sigma \equiv \tau_\perp$, $\sigma \equiv \sigma_1 \times \sigma_2$ and both σ_1, σ_2 are pointed, or $\sigma \equiv \sigma_1 \to \sigma_2$ and σ_2 is pointed. The reason $\tau_\perp$ has noninformative terms will become apparent from the type of the fixed-point operator. For a product $\sigma_1 \times \sigma_2$, we can write a noninformative term by combining noninformative terms of types σ_1 and σ_2. If $M\colon\sigma_2$ is noninformative, and $x\colon\sigma_1$ is not free in M, then $\lambda x\colon\sigma_1.\, M$ is a noninformative term of type $\sigma_1 \to \sigma_2$. The name "pointed," which is not very descriptive, comes from the fact that a pointed type has a distinguished element (or "point") representing the value of any noninformative term.

The language $\text{PCF}_\perp$ has a fixed-point constant for each pointed type:

$$\mathit{fix}_\sigma\colon (\sigma \to \sigma) \to \sigma \qquad \sigma \text{ pointed}$$

The equational axiom and reduction axiom for fix are as in PCF.

The first term form associated with lifted types is that if $M\colon\sigma$, then

$$\lfloor M \rfloor\colon \sigma_\perp.$$

Intuitively, since the elements of σ are included in $\sigma_\perp$, the term $\lfloor M \rfloor$ defines an element of σ, considered as an element of $\sigma_\perp$. The second term form associated with lifting allows us to evaluate expressions of $\sigma_\perp$ to obtain expressions of type σ. Since some terms of type $\sigma_\perp$ do not define elements of σ, this is formalized in a careful way. If $M\colon\sigma_\perp$ and $N\colon\tau$, with τ pointed and N possibly containing a variable x of type σ, then

$$\texttt{let } \lfloor x\colon\sigma \rfloor = M \texttt{ in } N$$

is a term of type τ. The reader familiar with ML may recognize this syntax as a form of "pattern-matching" `let`. Intuitively, the intent of `let` $\lfloor x\colon\sigma \rfloor = M$ `in` N is that we "hope" that M is equal to $\lfloor M' \rfloor$ for some M'. If so, then the value of this expression is the value of N when x has the value of M'.

The main equational axiom for $\lfloor\ \rfloor$ and `let` is

$$(\texttt{let } \lfloor\ \rfloor) \qquad \texttt{let } \lfloor x\colon\sigma \rfloor = \lfloor M \rfloor \texttt{ in } N \;=\; [M/x]N$$

This is also used as a reduction axiom, read left to right. Intuitively, the value of $M{:}\,\sigma_\perp$ is either $\perp$ or an element of σ. If $M = \perp$, then $\texttt{let } \lfloor x{:}\,\sigma \rfloor = M \texttt{ in } N$ will have value $\perp_\tau$. In operational terms, if M cannot be reduced to the form $\lfloor M' \rfloor$, then $\texttt{let } \lfloor x{:}\,\sigma \rfloor = M \texttt{ in } N$ not be reducible to a useful form of type τ. If M is equal to some $\lfloor M' \rfloor$, then the value of $\texttt{let } \lfloor x{:}\,\sigma \rfloor = M \texttt{ in } N$ is equal to $[M'/x]N$.

If we add a constant $\perp_\tau{:}\ \tau$ representing "undefinedness" or nontermination at each pointed type τ, we can also state some other equational properties.

$$(\texttt{let } \lfloor x \rfloor = \perp_{\sigma_\perp} \texttt{ in } M) \quad = \quad \perp_\tau$$

$$\frac{M \perp_{\sigma_\perp} = N \perp_{\sigma_\perp} \qquad M\lfloor x \rfloor = N \lfloor x \rfloor}{M = N} \ x \text{ not free in } M, N{:}\,\sigma_\perp \to \tau$$

Intuitively, the first equation reflects the fact that if we cannot reduce M to the form $\lfloor M' \rfloor$, then $\texttt{let } \lfloor x \rfloor = M \texttt{ in } N$ is "undefined." The inference rule is a form of reasoning by cases, based on the intuitive idea that any element of a lifted type $\sigma_\perp$ is either the result of lifting an element of type σ or the value $\perp_{\sigma\perp}$ representing "undefinedness" or nontermination at type σ. However, we will not use constants of the form $\perp_\tau$ or either of these properties in the remainder of this section. In general, they are not especially useful for proving equations between expressions unless we have a more powerful proof system (such as the one using fixed-point induction in Section 5.3) for proving that terms are equal to $\perp$.

We will illustrate the use of lifted types by translating both PCF and Eager PCF into $\text{PCF}_\perp$. A number of interesting relationships between lifted types and sums and recursive types are beyond the scope of this section but may be found in [How92], for example. Before proceeding, we consider some simple examples.

Example 2.6.9 While there is only one natural way to write factorial in PCF, there are some choices with explicit lifting. This example shows two essentially equivalent ways of writing factorial. For simplicity, we assume we have natural number subtraction and multiplication operations so that if $M, N{:}\,nat$, then $M - N{:}\,nat$ and $M * N{:}\,nat$. Using these, we can write a factorial function as $fact_1 \stackrel{def}{=} fix_{nat \to nat_\perp} F_1$, where

$$\begin{aligned} F_1 \ \stackrel{def}{=} \ & \lambda f{:}\,nat \to nat_\perp.\, \lambda x{:}\,nat. \\ & \texttt{if } Eq?\, x\, 0 \texttt{ then } \lfloor 1 \rfloor \texttt{ else let } \lfloor y \rfloor = f(x-1) \texttt{ in } \lfloor x * y \rfloor \end{aligned}$$

In the definition here, the type of $fact$ is $nat \to nat_\perp$. This is a pointed type, which is important since we only have fixed points at pointed types. This type for $fact_1$ is also the type of the lambda-bound variable f in F_1. Since $f{:}\,nat \to nat_\perp$, we can apply f to arguments of type nat. However, the result $f(x-1)$ of calling f will be an element of $nat_\perp$, and therefore we must use `let` to extract the natural number value from the recursive call before multiplying by x. If we wish to apply $fact_1$ to an argument M of type $nat_\perp$, we do so by writing

$$\texttt{let } \lfloor x \rfloor = M \texttt{ in } fact_1\, x$$

An alternative definition of factorial is $fact_2 \stackrel{def}{=} fix_{nat_\perp \to nat_\perp} F_2$, where

$$\begin{aligned} F_2 \ \stackrel{def}{=} \ & \lambda f{:}\,nat_\perp \to nat_\perp.\, \lambda x{:}\,nat_\perp.\, \texttt{let } \lfloor z \rfloor = x \texttt{ in} \\ & \texttt{if } Eq?\, z\, 0 \texttt{ then } \lfloor 1 \rfloor \texttt{ else let } \lfloor y \rfloor = f \lfloor z - 1 \rfloor \texttt{ in } \lfloor z * y \rfloor \end{aligned}$$

Here, the type of factorial is different since the domain of the function is $nat_\perp$ instead of nat. The consequence is that we could apply $fact_2$ to a possibly nonterminating expression. However, since the first thing $fact_2$ does with an argument $x\colon nat_\perp$ is force it to some value $z\colon nat$, there is not much flexibility gained in this definition of factorial. In particular, as illustrated below, both functions calculate 3! in essentially the same way. The reason $fact_2$ immediately forces $x\colon nat_\perp$ to some natural number $z\colon nat$ is that equality test $Eq?$ requires a nat argument, instead of a $nat_\perp$.

For the first definition of factorial, we can compute 3! by expanding out three recursive calls and simplifying as follows:

$$
\begin{array}{ll}
 & (\mathit{fix}\, F_1)\, 3 \\
\twoheadrightarrow & \texttt{if } \mathit{Eq?}\, 3\, 0 \texttt{ then } \lfloor 1 \rfloor \texttt{ else let } \lfloor y \rfloor = (\mathit{fix}\, F_1)(3-1) \texttt{ in } \lfloor 3 * y \rfloor \\
\twoheadrightarrow & \texttt{let } \lfloor y \rfloor = \\
 & \quad \texttt{let } \lfloor y' \rfloor = (\mathit{fix}\, F_1)(1) \texttt{ in } \lfloor 2 * y' \rfloor \\
 & \texttt{in } \lfloor 3 * y \rfloor \\
\twoheadrightarrow & \texttt{let } \lfloor y \rfloor = \\
 & \quad \texttt{let } \lfloor y' \rfloor = \\
 & \quad\quad \texttt{let } \lfloor y'' \rfloor = (\mathit{fix}\, F_1)0 \texttt{ in } \lfloor 1 * y'' \rfloor \\
 & \quad\quad \texttt{in } \lfloor 2 * y' \rfloor \\
 & \texttt{in } \lfloor 3 * y \rfloor
\end{array}
$$

Then, once we have reduced $(\mathit{fix}\, F_1)0 \twoheadrightarrow \lfloor 1 \rfloor$, we can set y'' to 1, then y' to 1, then y to 2, then perform the final multiplication and produce the final value 6.

For the alternative definition, we have essentially the same expansion, beginning with $\lfloor 3 \rfloor$ instead of 3 for typing reasons:

$$
\begin{array}{ll}
 & (\mathit{fix}\, F_2)\, \lfloor 3 \rfloor \\
\twoheadrightarrow & \texttt{let } \lfloor z \rfloor = \lfloor 3 \rfloor \texttt{ in} \\
 & \quad \texttt{if } \mathit{Eq?}\, z\, 0 \texttt{ then } \lfloor 1 \rfloor \texttt{ else let } \lfloor y \rfloor = (\mathit{fix}\, F_2) \lfloor z-1 \rfloor \texttt{ in } \lfloor z * y \rfloor \\
\twoheadrightarrow & \ldots \\
\twoheadrightarrow & \texttt{let } \lfloor y \rfloor = \\
 & \quad \texttt{let } \lfloor y' \rfloor = \\
 & \quad\quad \texttt{let } \lfloor y'' \rfloor = (\mathit{fix}\, F_2) \lfloor 0 \rfloor \texttt{ in } \lfloor 1 * y'' \rfloor \\
 & \quad\quad \texttt{in } \lfloor 2 * y' \rfloor \\
 & \texttt{in } \lfloor 3 * y \rfloor
\end{array}
$$

The only difference here is that in each recursive call, the argument is lifted to match the type of the function. However, the body F_2 immediately uses `let` to "unlift" the argument, so the net effect is the same as for $fact_1$. ∎

Example 2.6.10 As mentioned above, the equations

$$
\begin{array}{lcll}
\mathit{Eq?}\ x\ x & = & \mathit{true}, & \\
\mathit{Eq?}\ x\ (n+x) & = & \mathit{false} & \text{numeral } n \text{ different from } 0.
\end{array}
$$

are consistent with $\mathrm{PCF}_\perp$, when $x\colon nat$ is a natural number variable, but inconsistent when added to PCF. We will see how these equations could be used in $\mathrm{PCF}_\perp$ by considering the PCF expression

$$
\begin{array}{l}
\texttt{letrec } f(x\colon nat)\colon nat = \texttt{if } \mathit{Eq?}\, x\, 0 \texttt{ then } 1 \texttt{ else} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{if } \mathit{Eq?}\, f(x-1)\, f(x-1) \texttt{ then } 2 \texttt{ else } 3 \\
\texttt{in } f3
\end{array}
$$

If we apply left-most reduction to this expression, we end up with an expression of the form

$$\texttt{if } \mathit{Eq?}\,((\mathit{fix}\,F)(3-1))\,((\mathit{fix}\,F)(3-1)) \texttt{ then 2 else 3}$$

which requires us to simplify two applications of the recursive function in order to determine that $f(2) = f(2)$. We might like to apply some "optimization" which says that whenever we have a subexpression of the form $\mathit{Eq?}\,M\,M$ we can simplify this to *true*. However, the inconsistency of the two equations above shows that this optimization cannot be combined with other reasonable optimizations of this form.

We cannot eliminate the need to compute at least one recursive call, but we can eliminate the equality test using lifted types. A natural and routine translation of this expression into $\mathrm{PCF}_\perp$ is

$$\begin{array}{l}\texttt{letrec } f(x\colon \mathit{nat})\colon \mathit{nat}_\perp = \texttt{if } \mathit{Eq?}\,x\,0 \texttt{ then } \lfloor 1\rfloor \texttt{ else}\\ \qquad\qquad\qquad\qquad \texttt{let } \lfloor y\rfloor = f(x-1) \texttt{ in if } \mathit{Eq?}\,y\,y \texttt{ then } \lfloor 2\rfloor \texttt{ else } \lfloor 3\rfloor\\ \texttt{in } f3\end{array}$$

In this form, we have a subexpression of the form $\mathit{Eq?}\,y\,y$ where $y\colon \mathit{nat}$. With the equations above, we can replace this $\mathrm{PCF}_\perp$ expression by an equivalent one, obtaining the provably equivalent expression

$$\begin{array}{l}\texttt{letrec } f(x\colon \mathit{nat})\colon \mathit{nat}_\perp = \texttt{if } \mathit{Eq?}\,x\,0 \texttt{ then } \lfloor 1\rfloor \texttt{ else}\\ \qquad\qquad\qquad\qquad \texttt{let } \lfloor y\rfloor = f(x-1) \texttt{ in } \lfloor 2\rfloor\\ \texttt{in } f3\end{array}$$

The reason we cannot expect to simplify $\texttt{let } \lfloor y\rfloor = f(x-1) \texttt{ in } \lfloor 2\rfloor$ to $\lfloor 2\rfloor$ by any local optimization that does not analyze the entire declaration of f is that when $f(x-1)$ does not terminate (*i.e.*, cannot be reduced to the form $\lfloor M\rfloor$), the two are not equivalent. ∎

Translation of PCF into $\mathrm{PCF}_\perp$

The translation of PCF into $\mathrm{PCF}_\perp$ has two parts. The first is a translation of types. We will map expressions of type σ from PCF to expressions of type $\overline{\sigma}$ in $\mathrm{PCF}_\perp$, where $\overline{\sigma}$ is the result of replacing *nat* by $\mathit{nat}_\perp$ and *bool* by $\mathit{bool}_\perp$. More specifically, we may define $\overline{\sigma}$ by induction on types:

$$\begin{array}{lcl}\overline{\mathit{bool}} & = & \mathit{bool}_\perp\\ \overline{\mathit{nat}} & = & \mathit{nat}_\perp\\ \overline{\sigma_1 \times \sigma_2} & = & \overline{\sigma_1} \times \overline{\sigma_2}\\ \overline{\sigma_1 \to \sigma_2} & = & \overline{\sigma_1} \to \overline{\sigma_2}\end{array}$$

It is easy to check that for any PCF type σ, the type $\overline{\sigma}$ is pointed. This gives us a fixed-point operator of each type $\overline{\sigma}$.

The translation of compound PCF terms into $\mathrm{PCF}_\perp$ is essentially straightforward. The most interesting part of the translation lies in the basic natural number and boolean operations. If $M\colon \sigma$ in PCF, we must find some suitable term $\overline{M}$ of type $\overline{\sigma}$ in $\mathrm{PCF}_\perp$. The translation $\overline{n}$ of a numeral n is $\lfloor n\rfloor$ and the translations of *true* and *false* are similarly $\lfloor \mathit{true}\rfloor$ and $\lfloor \mathit{false}\rfloor$. It is easy to see these have the right types. For each of the operations $\mathit{Eq?}, +$ and conditional, we will write a lambda term of the appropriate $\mathrm{PCF}_\perp$ type. Since $\mathit{Eq?}$ compares two natural number terms and produces a boolean, we will write a $\mathrm{PCF}_\perp$ term $\overline{\mathit{Eq?}}$ of type $\mathit{nat}_\perp \to \mathit{nat}_\perp \to \mathit{bool}_\perp$. The term for equality test is

$$\overline{\mathit{Eq?}} \;=\; \lambda x\colon \mathit{nat}_\perp.\,\lambda y\colon \mathit{nat}_\perp.\,\texttt{let } \lfloor x'\rfloor = x \texttt{ in let } \lfloor y'\rfloor = y \texttt{ in } \lfloor \mathit{Eq?}\,x'\,y'\rfloor$$

An intuitive explanation of this function is that $\overline{Eq?}$ takes two arguments of type $nat_\perp$, evaluates them to obtain elements of type nat (if possible), and then compares the results for equality. If either argument does not reduce to the form $\lfloor M \rfloor$ for $M\colon nat$, then the corresponding `let` cannot be reduced. But if both arguments do reduce to the form $\lfloor M \rfloor$, then we are certain to be able to reduce both to numerals, and hence we will be able to obtain $\lfloor true \rfloor$ or $\lfloor false \rfloor$ by the reduction axiom for $Eq?$. The reader is encouraged to try a few examples in Exercise 2.6.14.

The terms $\overline{+}$ for addition and $\overline{cond}$ for `if ... then ... else ...` are similar in spirit to $\overline{Eq?}$.

$$\begin{aligned} \overline{+} &= \lambda x\colon nat_\perp.\, \lambda y\colon nat_\perp.\, \texttt{let } \lfloor x' \rfloor = x \texttt{ in let } \lfloor y' \rfloor = y \texttt{ in } \lfloor x' + y' \rfloor \\ \overline{cond_\sigma} &= \lambda x\colon bool_\perp.\, \lambda y\colon \overline{\sigma}.\, \lambda z\colon \overline{\sigma}.\, \texttt{let } \lfloor x' \rfloor = x \texttt{ in if } x' \texttt{ then } y \texttt{ else } z \end{aligned}$$

In the translation of conditional, it is important to notice that $\overline{\sigma}$ is pointed. As a result, the function body `let` $\lfloor x' \rfloor = x$ `in if` x' `then` y `else` z is well-typed.

For pairing and lambda abstraction, we let

$$\begin{aligned} \overline{\langle M, N \rangle} &= \langle \overline{M}, \overline{N} \rangle \\ \overline{\lambda x\colon \sigma.\, M} &= \lambda x\colon \overline{\sigma}.\, \overline{M} \end{aligned}$$

and similarly for application, projections and fixed points. Some examples appear in Exercises 2.6.14 and 2.6.16.

The main properties of this translation are

(i) If $M\colon \sigma$ in PCF, then $\overline{M}\colon \overline{\sigma}$ in $\mathrm{PCF}_\perp$.

(ii) If M, N are syntactically distinct terms of PCF, then $\overline{M}, \overline{N}$ are syntactically distinct terms of $\mathrm{PCF}_\perp$.

(iii) If $M \to N$ in PCF, then $\overline{M} \to \overline{N}$ in $\mathrm{PCF}_\perp$. Conversely, if M is a normal form in PCF, then $\overline{M}$ is a normal form in $\mathrm{PCF}_\perp$.

(iv) If $M = N$ is provable in PCF, then $\overline{M} = \overline{N}$ is provable in $\mathrm{PCF}_\perp$.

While there are a number of cases, it is essentially straightforward to verify these properties. It follows from properties (ii) and (iii) that if M, N are distinct normal forms of PCF, then $\overline{M}, \overline{N}$ are distinct normal forms of $\mathrm{PCF}_\perp$.

Translation of Eager PCF into $\mathrm{PCF}_\perp$

Like the translation of PCF into $\mathrm{PCF}_\perp$, the translation of Eager PCF into $\mathrm{PCF}_\perp$ has two parts. We begin with the translation of types.

For any type σ of Eager PCF, we define the associated type $\underline{\sigma}$. Intuitively, $\underline{\sigma}$ is the type of Eager PCF values (in the technical sense of Section 2.4.5) of type σ. Since any Eager PCF term will either reduce to a value, or diverge under eager reduction, an Eager PCF term of type σ will be translated to a $\mathrm{PCF}_\perp$ term of type $\underline{\sigma}_\perp$. After defining $\underline{\sigma}$ by induction on the structure of type expressions, we give an informal explanation.

$$\begin{aligned} \underline{nat} &= nat \\ \underline{bool} &= bool \\ \underline{\sigma \to \tau} &= \underline{\sigma} \to \underline{\tau}_\perp \\ \underline{\sigma \times \tau} &= \underline{\sigma} \times \underline{\tau} \end{aligned}$$

The Eager PCF values of type *nat* or *bool* correspond to $\mathrm{PCF}_\bot$ terms of type *nat* or *bool*. An Eager PCF function value of type $\sigma \to \tau$ has the form $\lambda x{:}\sigma.\, M$, where for any argument V, the body $[V/x]M$ may either reduce to a value of type τ or diverge under eager reduction. Therefore, in $\mathrm{PCF}_\bot$, the values of type $\sigma \to \tau$ are functions from $\underline{\sigma}$ to $\underline{\tau}_\bot$. Finally, the Eager PCF values of type $\sigma \times \tau$ are pairs of values, and therefore have type $\underline{\sigma} \times \underline{\tau}$ in $\mathrm{PCF}_\bot$.

We translate a term $M{:}\sigma$ of Eager PCF with free variables $x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k$ to a term $\underline{M}{:}\underline{\sigma}_\bot$ of $\mathrm{PCF}_\bot$ with free variables $x_1{:}\underline{\sigma}_1, \ldots, x_k{:}\underline{\sigma}_k$, as follows:

$$
\begin{array}{lcl}
\underline{x} & = & \lfloor x \rfloor \\
\underline{n} & = & \lfloor n \rfloor \\
\underline{\mathit{true}} & = & \lfloor \mathit{true} \rfloor \\
\underline{\mathit{false}} & = & \lfloor \mathit{false} \rfloor \\
\underline{M + N} & = & \texttt{let } \lfloor x \rfloor = \underline{M} \texttt{ in let } \lfloor y \rfloor = \underline{N} \texttt{ in } \lfloor x + y \rfloor \\
\underline{\mathit{Eq}?MN} & = & \texttt{let } \lfloor x \rfloor = \underline{M} \texttt{ in let } \lfloor y \rfloor = \underline{N} \texttt{ in } \lfloor \mathit{Eq}?\, x\, y \rfloor \\
\underline{\texttt{if } M \texttt{ then } N \texttt{ else } P} & = & \texttt{let } \lfloor x \rfloor = \underline{M} \texttt{ in if } x \texttt{ then } \underline{N} \texttt{ else } \underline{P} \\
\underline{MN} & = & \texttt{let } \lfloor f \rfloor = \underline{M} \texttt{ in let } \lfloor x \rfloor = \underline{N} \texttt{ in } fx \\
\underline{\lambda x{:}\sigma.\, M} & = & \lfloor \lambda x{:}\underline{\sigma}.\, \underline{M} \rfloor \\
\underline{\mathit{fix}_{\sigma \to \tau}} & = & \lfloor \mathit{fix}(\lambda f{:}\underline{((\sigma \to \tau) \to (\sigma \to \tau)) \to (\sigma \to \tau)}. \\
 & & \quad \lambda g{:}\underline{(\sigma \to \tau) \to (\sigma \to \tau)}.\, g(\lambda x{:}\underline{\sigma}.\, \texttt{let } \lfloor h \rfloor = fg \texttt{ in } hx)) \rfloor
\end{array}
$$

The translation of the fixed-point operator needs some explanation. As discussed in Section 2.4.5, Eager PCF only has fixed-point operators on function types. The translation of the fixed-point operator on type $\sigma \to \tau$ is recursively defined to satisfy

$$\underline{\mathit{fix}_{\sigma \to \tau}} = \lambda g{:}\underline{(\sigma \to \tau) \to (\sigma \to \tau)}.\, g(\lambda x{:}\underline{\sigma}.\, \texttt{let } \lfloor h \rfloor = \underline{\mathit{fix}_{\sigma \to \tau}}\, g \texttt{ in } hx)$$

This is essentially the Eager PCF property "$\mathit{fix} = \lambda g{:}(\ldots).\, g(\mathit{delay}[\mathit{fix}\ g])$" from Section 2.4.5, where with $\mathit{delay}[M] \equiv \lambda x{:}(\ldots).\, Mx$, using the fact that

$$
\begin{array}{l}
\underline{Mx} \equiv \texttt{let } \lfloor f \rfloor = \underline{M} \texttt{ in let } \lfloor y \rfloor = \lfloor x \rfloor \texttt{ in } fy \\
= \quad \texttt{let } \lfloor f \rfloor = \underline{M} \texttt{ in } fx
\end{array}
$$

by axiom ($\texttt{let } \lfloor\ \rfloor$).

In the next two examples, we discuss the way that eager reduction is preserved by this translation. A general difference between the eager reduction strategy given in Section 2.4.5 and reduction in $\mathrm{PCF}_\bot$ is that eager reduction is deterministic. In particular, eager reduction of an application MN specifies reduction of M to a value (*e.g.*, $\lambda x{:}(\ldots).\, M'$) before reducing N to a value. However, from the point of view of accurately representing the termination behavior and values of any expression, the order of evaluation between M and N is not significant. When an application MN of Eager PCF is translated into $\mathrm{PCF}_\bot$, it will be necessary to reduce both to the form $\lfloor \ldots \rfloor$, intuitively corresponding to the translation of an Eager PCF value, before performing β-reduction. However, $\mathrm{PCF}_\bot$ allows either the function M or the argument N to be reduced first.

Example 2.6.11 We may apply this translation to the term considered in Example 2.4.20. Since this will help preserve the structure of the expression, we use the syntactic sugar

$$M \cdot N \stackrel{def}{=} \texttt{let } \lfloor f \rfloor = M \texttt{ in let } \lfloor x \rfloor = N \texttt{ in } fx$$

for eager application. For readability, we will also use the simplification

$$\underline{u + v} = \texttt{let } \lfloor x \rfloor = \lfloor u \rfloor \texttt{ in } (\texttt{let } \lfloor y \rfloor = \lfloor v \rfloor \texttt{ in } \lfloor x + y \rfloor) = \lfloor u + v \rfloor$$

for any variables or constants u and v, since this consequence of our equational axioms does not change the behavior of the resulting term. Applying the translation to the term

$$(\mathit{fix}\ (\lambda x{:}\,nat \to nat.\ \lambda y{:}\,nat.\ y))\ ((\lambda z{:}\,nat.\ z+1)\ 2)$$

gives us

$$(\underline{\mathit{fix}} \cdot \lfloor\ \lambda x{:}\,nat \to nat_\perp.\lfloor \lambda y{:}\,nat.\lfloor y \rfloor \rfloor\ \rfloor)\ \cdot\ (\lfloor\ \lambda z{:}\,nat.\lfloor z+1 \rfloor\ \rfloor \cdot \lfloor 2 \rfloor)$$

A useful general observation is that

$$\lfloor\ M\ \rfloor \cdot \lfloor N \rfloor \quad \twoheadrightarrow \quad MN$$

but that we cannot simplify a form $M' \cdot N'$ unless M' has the form $\lfloor M \rfloor$ and N' has the form $\lfloor N \rfloor$. The need to produce "lifted values" (or, more precisely, the lifting of terms that reduce to values) is important in expressing eager reduction with lifted types. We can see that there are two possible reductions of the term above, one involving $\underline{\mathit{fix}}$ and the other $(\lfloor \lambda z{:}\,nat.\lfloor z+1 \rfloor \rfloor \cdot \lfloor 2 \rfloor) \twoheadrightarrow \lfloor 2+1 \rfloor$. The reader may enjoy leftmost reducing the entire expression and seeing how the steps correspond to the eager reduction steps carried out in Example 2.4.20. ∎

Example 2.6.12 We can see how divergence is preserved by considering the term

$$\begin{array}{l}\texttt{let}\ f(x{:}\,nat){:}\,nat = 3\ \texttt{in}\\ \quad \texttt{letrec}\ g(x{:}\,nat){:}\,nat = g(x+1)\ \texttt{in}\ f(g\,5)\end{array}$$

from Example 2.4.21. After replacing `let`'s by lambda abstraction and application, the translation of this term is

$$\begin{array}{l}\lfloor\ \lambda f{:}\,nat \to nat_\perp.\lfloor \lambda g{:}\,nat \to nat_\perp.\lfloor f \rfloor \cdot (g\,5)\ \rfloor \rfloor\\ \cdot \lfloor\ \lambda x{:}\,nat.\lfloor 3 \rfloor\ \rfloor\\ \cdot\ (\ \underline{\mathit{fix}} \cdot \lfloor\ \lambda g{:}\,nat \to nat_\perp.\lfloor \lambda x{:}\,nat.\ g(x+1)\ \rfloor \rfloor\ )\end{array}$$

where for simplicity we have used $\lfloor g \rfloor \cdot \lfloor 5 \rfloor = g\,5$ and $\lfloor g \rfloor \cdot (\underline{x+1}) = \lfloor g \rfloor \cdot \lfloor x+1 \rfloor = g(x+1)$. Recall from Examples 2.4.6 and 2.4.21 that if we apply leftmost reduction to the original term, we obtain 3, while eager reduction does not produce a normal form. We can see how the translation of Eager PCF into $\mathrm{PCF}_\perp$ preserves the absence of normal forms by leftmost-reducing the result of this translation.

Since the first argument and the main function have the form $\lfloor \ldots \rfloor$, we can apply β-reduction as described in Example 2.6.11. This gives us

$$\lfloor \lambda g{:}\,nat \to nat_\perp.\lfloor \lambda x{:}\,nat.\lfloor 3 \rfloor \rfloor \cdot (g\,5)\ \rfloor \cdot (\ \underline{\mathit{fix}} \cdot \lfloor G \rfloor\ )$$

where $G \stackrel{def}{=} \lambda g{:}\,nat \to nat_\perp.\lfloor \lambda x{:}\,nat.\,g(x+1) \rfloor$. Our task is to reduce the argument $(\ \underline{\mathit{fix}} \cdot \lfloor G \rfloor\ )$ to the form $\lfloor \ldots \rfloor$ before performing the leftmost β-reduction. We do this using the that

$$\begin{array}{l}\underline{\mathit{fix}_{nat \to nat}} \quad = \quad \lfloor\ \mathit{fix}_{nat \to nat_\perp}\ F\ \rfloor \quad \twoheadrightarrow\\ \lfloor\ \lambda g{:}\,(nat \to nat_\perp) \to (nat \to nat_\perp).\ g(\lambda x{:}\,nat.\ \texttt{let}\ \lfloor h \rfloor = \mathit{fix}\ F\ g\ \texttt{in}\ hx)\ \rfloor\end{array}$$

where

$$\begin{array}{rcl}F & \stackrel{def}{=} & (\lambda f{:}\,((nat \to nat_\perp) \to (nat \to nat_\perp)) \to (nat \to nat_\perp).\\ & & \quad \lambda g{:}\,(nat \to nat_\perp) \to (nat \to nat_\perp).\ g(\lambda x{:}\,nat.\ \texttt{let}\ \lfloor h \rfloor = fg\ \texttt{in}\ hx))\end{array}$$

However, we will see in the process that the application to 5 resulting from β-reduction cannot be reduced to the form $\lfloor \ldots \rfloor$, keeping us from every reaching the normal form $\lfloor 3 \rfloor$ for the entire term.

The expansion of $\underline{fix}$ above gives us

$$\begin{array}{ll} \underline{fix} \cdot \lfloor G \rfloor & \twoheadrightarrow \quad fix\ F\ G \\ \twoheadrightarrow & \quad G\ (\lambda x{:}\, nat.\, \texttt{let}\ \lfloor h \rfloor = fix\ F\ G\ \texttt{in}\ hx) \\ \twoheadrightarrow & \quad \lfloor \lambda x{:}\, nat.((\lambda x{:}\, nat.\, \texttt{let}\ \lfloor h \rfloor = fix\ F\ G\ \texttt{in}\ hx)\ (x+1)) \rfloor \\ \twoheadrightarrow & \quad \lfloor \lambda x{:}\, nat.\, \texttt{let}\ \lfloor h \rfloor = fix\ F\ G\ \texttt{in}\ h(x+1) \rfloor \end{array}$$

When the function inside the $\lfloor\ \rfloor$ is applied to 5, the result must be reduced to a form $\lfloor \ldots \rfloor$ to complete the reduction of the entire term. However, this leads us to again reduce $fix\ F\ G$ to this form and apply the result to 6. Since this process may be continued indefinitely, the entire term does not have a normal form. ■

The main properties of the translation from Eager PCF into $\mathrm{PCF}_\perp$ are:

(i) If $M{:}\sigma$ in Eager PCF, with free variables $x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k$, then $\underline{M}{:}\underline{\sigma}_\perp$ in $\mathrm{PCF}_\perp$ with free variables $x_1{:}\underline{\sigma}_1, \ldots, x_k{:}\underline{\sigma}_k$.

(ii) If $M \overset{eager}{\rightarrow} N$ in Eager PCF, then $\underline{M} \twoheadrightarrow \underline{N}$ in $\mathrm{PCF}_\perp$.

(iii) If M is a closed term of Eager PCF of type *nat* or *bool* and c is a numeral or *true* or *false*, then $M \overset{eager}{\twoheadrightarrow} c$ in Eager PCF iff $\underline{M} \twoheadrightarrow \lfloor c \rfloor$ in $\mathrm{PCF}_\perp$.

(iv) If we let $=_{eager}$ be operational equivalence of Eager PCF term, defined in the same was as $=_{op}$ in Section 2.3.5 but using eager evaluation, and let $=_{PCF\perp}$ be operational equivalence in $\mathrm{PCF}_\perp$, then for closed terms M, N we have $M =_{eager} N$ iff $\underline{M} =_{PCF\perp} \underline{N}$.

It is largely straightforward to verify properties (i)–(iii). Property (iv), however, involves construction of fully abstract models satisfying the properties described in Section 5.4.2. A proof has been developed by R. Viswanathan.

Exercise 2.6.13 Suppose that instead of having fix_σ for every pointed σ, we only have fixed-point operators for types of the form $\tau_\perp$. Show that it is still possible to write a term with no normal form of each pointed type.

Exercise 2.6.14 Reduce the application of $\overline{Eq?}$ to the following pairs of terms. You should continue until you obtain $\lfloor true \rfloor$ or $\lfloor false \rfloor$ or it is apparent that one of the `let`'s can never be reduced.

(a) $\lfloor 3+2 \rfloor$ and $\lfloor 5 \rfloor$.

(b) $(\lambda x{:}\, nat_\perp.\, x)\lfloor 3+2 \rfloor$ and $\lfloor (\lambda x{:}\, nat.\, x+1)\, 3 \rfloor$.

(c) $fix(\lambda x{:}\, nat_\perp.\, x)$ and $\lfloor 19 \rfloor$.

Exercise 2.6.15 Carry out the reduction of the term $\underline{fix} \cdot \lfloor\ \lambda x{:}\, nat \to nat_\perp.\lfloor \lambda y{:}\, nat.\lfloor y \rfloor\ \rfloor \rfloor$ from Example 2.6.11.

Exercise 2.6.16 The fibonacci function may be written in PCF as $\mathit{fib} \stackrel{def}{=} \mathit{fix}_{nat \to nat} F$, where

$$F \stackrel{def}{=} \lambda f\!: nat \to nat.\, \lambda x\!: nat.\; \texttt{if } (Eq?\, x\, 0) \textit{ or } (Eq?\, x\, 1) \texttt{ then } 1 \texttt{ else } f(x-1) + f(x-2)$$

We assume for simplicity that we have natural number subtraction and a boolean *or* function.

(a) Translate the definition from (ordinary) PCF into $\mathrm{PCF}_\perp$ and describe the reduction of $\mathit{fib}\ 3$ in approximately the same detail as Example 2.6.9. (Subtraction and *or* should be translated into $\mathrm{PCF}_\perp$ following the pattern given for $+$ and $Eq?$. You may simplify $\lfloor M \rfloor - \lfloor N \rfloor$ to $\lfloor M - N \rfloor$, or perform other simplifications that are similar to the ones used in Example 2.6.11.)

(b) Same as part (a), but use the translation from Eager PCF into $\mathrm{PCF}_\perp$.