19 System PCF of Recursive Functions

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

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

For example, consider the equations

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

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

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

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

Re-writing yet again, we seek f given by

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

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

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

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

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

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

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

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

19.1 Statics

The syntax of **PCF** is given by the following grammar:

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

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

The statics of **PCF** is inductively defined by the following rules:

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

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

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

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \texttt{parr}(\tau_1; \tau_2)} \tag{19.1e}$$

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

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

Rule (19.1g) reflects the self-referential nature of general recursion. To show that $\texttt{fix}\{\tau\}(x.e)$ has type τ, we *assume* that it is the case by assigning that type to the variable x, which stands for the recursive expression itself, and checking that the body, e, has type τ under this very assumption.

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

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

19.2 Dynamics

The dynamic semantics of **PCF** is defined by the judgments e val, specifying the closed values, and $e \longmapsto e'$, specifying the steps of evaluation.

The judgment e val is defined by the following rules:

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

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

$$\frac{}{\texttt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{19.2c}$$

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

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

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

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

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

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

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

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

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

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

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

Theorem 19.2 (Safety)**.**

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

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

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

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

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

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

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

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

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

19.3 Definability

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

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

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

Recursive functions are defined in **PCF** using recursive functions, writing

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

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

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

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

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

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

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

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

It is informative to classify those partial functions ϕ that are definable in **PCF**. The *partial recursive functions* are defined to be the primitive recursive functions extended with the *minimization* operation: given $\phi(m, n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m', n)$ is defined and non-zero, and (2) $\phi(m, n) = 0$. If no such m exists, then $\psi(n)$ is undefined.

Theorem 19.3. *A partial function ϕ on the natural numbers is definable in* **PCF** *iff it is partial recursive.*

Proof sketch Minimization is definable in **PCF**, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with some tedium, define an evaluator for expressions of **PCF** as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Therefore, **PCF** does not exceed the power of the set of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the set of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language that is or will ever be defined.[1] Therefore, **PCF** is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function ϕ_{univ} for **PCF** is the partial function on the natural numbers defined by

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

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

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

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

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

19.4 Finite and Infinite Data Structures

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

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

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

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

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

19.5 Totality and Partiality

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

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

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

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

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

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

We can, but at a cost. We have already seen one limitation of total programming languages: they are not universal. You cannot write an interpreter for **T** within **T**, and this limitation extends to any total language whatever. If this does not seem important, then consider the *Blum Size Theorem (BST)*, which places another limitation on total languages. Fix *any* total language $\mathcal{L}$ that permits writing functions on the natural numbers. Pick any blowup factor, say 2^{2^n}. The BST states that there is a total function on the natural numbers that is programmable in $\mathcal{L}$, but whose shortest program in $\mathcal{L}$ is larger by the given blowup factor than its shortest program in **PCF**!

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

19.6 Notes

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

Exercises

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

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

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

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

19.5. Suppose that we wished to define, in the lazy variant of **PCF**, a version of the *parallel or* function specified a function of two arguments that returns z if either of its arguments is z, and s(z) otherwise. That is, we wish to find an expression e satisfying the following properties:

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

Thus, e defines a total function of its two arguments, *even if* one of the arguments diverges. Clearly, such a function cannot be defined in the call-by-value variant of

PCF, but can it be defined in the call-by-name variant? If so, show how; if not, prove that it cannot be, and suggest an extension of **PCF** that would allow it to be defined.

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

Note

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