20 System FPC of Recursive Types

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

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

20.1 Solving Type Equations

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

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

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

The subscript on the concrete syntax of `fold` is often omitted when it is clear from context.

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

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

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

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

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

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

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

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

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

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

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

Theorem 20.1 (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

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

20.2 Inductive and Coinductive Types

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

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

satisfies the type equation

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

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

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

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

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

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

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

satisfies the same type equation,

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

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

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

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

which satisfies the type equation

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

The list introduction operations are given by the following equations:

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

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

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

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

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

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

has the form $\mathtt{fold}(e)$, where e is an unevaluated computation of the sum type, whose values are injections of unevaluated computations of either the unit type or of the product type $\mathtt{nat} \times t$. And the latter consists of pairs of an unevaluated computation of a (lazy!) natural number, and an unevaluated computation of another value of this type. In particular, this type contains infinite lists whose tails go on without end, as well as finite lists that

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

eventually reach an end. The type is, in fact, a version of the type of infinite streams defined in Chapter 15, rather than a type of finite lists as is the case under an eager dynamics.

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

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

20.3 Self-Reference

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

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

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

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

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

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

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

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathtt{unfold}(e)(e).$$

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

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

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

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

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

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

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

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

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

20.4 The Origin of State

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

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

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

The `X` and `Q` components of the latch represent its current outputs (of which `Q` represents the current state of the latch), and the `N` component represents the next state of the latch. If e is of type τ_{rsl}, then we define e @ `X` to mean $\texttt{unfold}(e) \cdot \texttt{X}$, and define e @ `Q` and e @ `N` similarly. The expressions e @ `X` and e @ `Q` evaluate to the boolean outputs of the latch e, and e @ `N` evaluates to another latch representing its evolution over time based on these inputs.

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

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

where e_{rsl} is the expression

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

where e_{nor} is the obvious binary function on booleans. The outputs of the latch are computed in terms of the r and s inputs and the outputs of the previous state of the latch. To get the

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

construction started, we define an initial state of the latch in which the outputs are arbitrarily set to `false`, and whose next state is determined by applying the recursive function *rsl* to that state:

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

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

20.5 Notes

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

Exercises

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

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

such that

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

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

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

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

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press

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

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

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

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

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

Notes

1 The "underscore" stands for a variable that does not occur free in e_0.
2 For convenience, we assume that `fold` is evaluated lazily.

https://doi.org/10.1017/CBO9781316576892.022 Published online by Cambridge University Press