Chapter 17

Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important forms of recursive type. Inductive types correspond to *least*, or *initial*, solutions of certain type isomorphism equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, the elements of an inductive type are those that may be obtained by a finite composition of its introductory forms. Consequently, if we specify the behavior of a function on each of the introductory forms of an inductive type, then its behavior is determined for all values of that type. Such a function is called a *recursor*, or *catamorphism*. Dually, the elements of a coinductive type are those that behave properly in response to a finite composition of its elimination forms. Consequently, if we specify the behavior of an element on each elimination form, then we have fully specified that element as a value of that type. Such an element is called an *generator*, or *anamorphism*.

17.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 11. The type `nat` is defined to be the *least* type containing `z` and closed under `s(`$-$`)`. The minimality condition is witnessed by the existence of the recursor, $\texttt{natiter}\, e\, \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$, which transforms a natural number into a value of type τ, given its value for zero, and a transformation from its value on a number to its value on the successor of that number. This operation is well-defined precisely because there are no other natural numbers. Put the other way around, the existence

of this operation expresses the inductive nature of the type nat.

With a view towards deriving the type nat as a special case of an inductive type, it is useful to consolidate zero and successor into a single introductory form, and to correspondingly consolidate the basis and inductive step of the recursor. This following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \mathtt{unit} + \mathtt{nat}}{\Gamma \vdash \mathtt{fold}_{\mathtt{nat}}(e) : \mathtt{nat}} \tag{17.1a}$$

$$\frac{\Gamma, x : \mathtt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2) : \tau} \tag{17.1b}$$

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introductory form of the type nat. Using this, the expression z is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{s}(e)$ is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor, $\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2)$, takes as argument the abstractor $x.e_1$ that consolidates the basis and inductive step into a single computation that is given a value of type $\mathtt{unit} + \tau$ yields a value of type τ. Intuitively, if x is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\mathtt{r} \cdot e$, then e_1 computes the inductive step as a function of the result, e, of the recursive call.

The dynamics of the consolidated representation of natural numbers is given by the following rules:

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathtt{val}} \tag{17.2a}$$

$$\frac{e_2 \mapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2) \mapsto \mathtt{rec}_{\mathtt{nat}}[x.e_1](e_2')} \tag{17.2b}$$

$$\frac{}{\mathtt{rec}_{\mathtt{nat}}[x.e_1](\mathtt{fold}_{\mathtt{nat}}(e_2)) \mapsto [\mathtt{map}[t.\mathtt{unit} + t](y.\mathtt{rec}_{\mathtt{nat}}[x.e_1](y); e_2)/x]e_1} \tag{17.2c}$$

Rule (17.2c) makes use of generic extension (see Chapter 7) to apply the recursor to the predecessor, if any, of a natural number. The idea is that the result of extending the recursor from the type $\mathtt{unit} + \mathtt{nat}$ to the type $\mathtt{unit} + \tau$ is substituted into the inductive step, given by the expression e_1. If we expand the definition of the generic extension in place, we obtain the

following reformulation of this rule:

$$\frac{}{\mathtt{rec}_{\mathtt{nat}}[x.e_1](\mathtt{fold}_{\mathtt{nat}}(e_2)) \mapsto [\mathtt{case}\, e_2\, \{\mathtt{l}\cdot \_ \Rightarrow \mathtt{l}\cdot\langle\rangle \mid \mathtt{r}\cdot y \Rightarrow \mathtt{r}\cdot \mathtt{rec}_{\mathtt{nat}}[x.e_1](y)\}/x]e_1}$$

An illustrative example of a coinductive type is the type of *streams* of natural numbers. A stream is an infinite sequence of natural numbers such that an element of the stream can be computed only after computing all preceding elements in that stream. That is, the computations of successive elements of the stream are sequentially dependent in that the computation of one element influences the computation of the next. This characteristic of the introductory form for streams is *dual* to the analogous property of the eliminatory form for natural numbers whereby the result for a number is determined by its result for all preceding numbers.

A stream is characterized by its behavior under the elimination forms for the stream type: $\mathtt{hd}(e)$ returns the next, or head, element of the stream, and $\mathtt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of a recursor, that determines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{nat}} \tag{17.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}(e) : \mathtt{stream}} \tag{17.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\, e\, \mathtt{<hd}(x) \Rightarrow e_1\, \mathtt{\&}\, \mathtt{tl}(x) \Rightarrow e_2\mathtt{>} : \mathtt{stream}} \tag{17.3c}$$

In Rule (17.3c) the current state of the stream is given by the expression e of some type τ, and the head and tail of the stream are determined by the expressions e_1 and e_2, respectively, as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\, e\, \mathtt{<hd}(x) \Rightarrow e_1\, \mathtt{\&}\, \mathtt{tl}(x) \Rightarrow e_2\mathtt{>}\ \mathtt{val}} \tag{17.4a}$$

$$\frac{e \mapsto e'}{\mathtt{hd}(e) \mapsto \mathtt{hd}(e')} \tag{17.4b}$$

$$\mathtt{hd}(\mathtt{strgen}\, e\, \mathtt{<hd}(x) \Rightarrow e_1\, \&\, \mathtt{tl}(x) \Rightarrow e_2\mathtt{>}) \mapsto [e/x]e_1 \tag{17.4c}$$

$$\frac{e \mapsto e'}{\mathtt{tl}(e) \mapsto \mathtt{tl}(e')} \tag{17.4d}$$

$$\mathtt{tl}(\mathtt{strgen}\, e\, \mathtt{<hd}(x) \Rightarrow e_1\, \&\, \mathtt{tl}(x) \Rightarrow e_2\mathtt{>}) \mapsto \mathtt{strgen}\, [e/x]e_2\, \mathtt{<hd}(x) \Rightarrow e_1\, \&\, \mathtt{tl}(x) \Rightarrow e_2\mathtt{>} \tag{17.4e}$$

Rules (17.4c) and (17.4e) express the dependency of the head and tail of the stream on its current state. Observe that the tail is obtained by applying the generator to the new state determined by e_2 as a function of the current state.

To derive streams as a special case of a coinductive type, we consolidate the head and the tail into a single eliminatory form, and reorganize the generator correspondingly. This leads to the following statics:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{unfold}_{\mathtt{stream}}(e) : \mathtt{nat} \times \mathtt{stream}} \tag{17.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathtt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2) : \mathtt{stream}} \tag{17.5b}$$

Rule (17.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head, $\mathtt{hd}(e)$, and tail, $\mathtt{tl}(e)$, of a stream, e, are defined to be the projections $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{l}$ and $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{r}$, respectively. Rule (17.5b) states that a stream may be generated from the state element, e_2, by an expression e_1 that yields the head element and the next state as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)\ \mathtt{val}} \tag{17.6a}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}_{\mathtt{stream}}(e) \mapsto \mathtt{unfold}_{\mathtt{stream}}(e')} \tag{17.6b}$$

$$\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)) \mapsto \mathtt{map}[t.\mathtt{nat} \times t](y.\mathtt{gen}_{\mathtt{stream}}[x.e_1](y); [e_2/x]e_1) \tag{17.6c}$$

Rule (17.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\frac{}{\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}[x.e_1](e_2)) \mapsto \langle([e_2/x]e_1)\cdot\mathtt{l}, \mathtt{gen}_{\mathtt{stream}}[x.e_1](([e_2/x]e_1)\cdot\mathtt{r})\rangle}$$

17.2 Statics

We may now give a fully general account of inductive and coinductive types, which are defined in terms of positive type operators. We will consider the language $\mathcal{L}\{\mu_i\mu_f\}$, which extends $\mathcal{L}\{\rightarrow\times+\}$ with inductive and co-inductive types.

17.2.1 Types

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over types. The abstract syntax of inductive and coinductive types is given by the following grammar:

Type	τ	::=	t	t	self-reference
			$\mathtt{ind}(t.\tau)$	$\mu_i(t.\tau)$	inductive
			$\mathtt{coi}(t.\tau)$	$\mu_f(t.\tau)$	coinductive

Type formation judgements have the form

$$t_1\ \mathsf{type}, \ldots, t_n\ \mathsf{type} \vdash \tau\ \mathsf{type},$$

where $t_1, \ldots, t_n$ are type names. We let Δ range over finite sets of hypotheses of the form t type, where t name is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \quad (17.7a)$$

$$\frac{}{\Delta \vdash \mathtt{unit}\ \mathsf{type}} \quad (17.7b)$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{prod}(\tau_1;\tau_2)\ \mathsf{type}} \quad (17.7c)$$

$$\frac{}{\Delta \vdash \mathtt{void}\ \mathsf{type}} \quad (17.7d)$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{sum}(\tau_1;\tau_2) \text{ type}} \tag{17.7e}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ type}} \tag{17.7f}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{ind}(t.\tau) \text{ type}} \tag{17.7g}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{coi}(t.\tau) \text{ type}} \tag{17.8}$$

17.2.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Expr	e	::=	$\mathtt{fold}[t.\tau](e)$	$\mathtt{fold}(e)$	constructor
			$\mathtt{rec}[t.\tau][x.e_1](e_2)$	$\mathtt{rec}[x.e_1](e_2)$	recursor
			$\mathtt{unfold}[t.\tau](e)$	$\mathtt{unfold}(e)$	destructor
			$\mathtt{gen}[t.\tau][x.e_1](e_2)$	$\mathtt{gen}[x.e_1](e_2)$	generator

The statics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{ind}(t.\tau)} \tag{17.9a}$$

$$\frac{\Gamma, x : [\rho/t]\tau \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \mathtt{ind}(t.\tau)}{\Gamma \vdash \mathtt{rec}[t.\tau][x.e_1](e_2) : \rho} \tag{17.9b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{unfold}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{17.9c}$$

$$\frac{\Gamma \vdash e_2 : \rho \quad \Gamma, x : \rho \vdash e_1 : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau][x.e_1](e_2) : \mathtt{coi}(t.\tau)} \tag{17.9d}$$

17.3 Dynamics

The dynamics of these constructs is given in terms of the generic extension operation described in Chapter 16. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\}$:

$$\frac{}{\mathtt{fold}(e) \text{ val}} \tag{17.10a}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{rec}[x.e_1](e_2) \mapsto \texttt{rec}[x.e_1](e_2')} \tag{17.10b}$$

$$\texttt{rec}[x.e_1](\texttt{fold}(e_2)) \mapsto [\texttt{map}[t.\tau](y.\texttt{rec}[x.e_1](y); e_2)/x]e_1 \tag{17.10c}$$

$$\texttt{gen}[x.e_1](e_2)\ \texttt{val} \tag{17.10d}$$

$$\frac{e \mapsto e'}{\texttt{unfold}(e) \mapsto \texttt{unfold}(e')} \tag{17.10e}$$

$$\texttt{unfold}(\texttt{gen}[x.e_1](e_2)) \mapsto \texttt{map}[t.\tau](y.\texttt{gen}[x.e_1](y); [e_2/x]e_1) \tag{17.10f}$$

Rule (17.10c) states that to evaluate the recursor on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value, and then perform the inductive step on the result. Rule (17.10f) is simply the dual of this rule for coinductive types.

Lemma 17.1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on Rules (17.10). □

Lemma 17.2. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on Rules (17.9). □

17.4 Notes

The general formulation of inductive and coinductive types for programming was introduced by Mendler [57], making use of the functorial action of a type constructor described in Chapter 16. Mendler's account is based on the interpretation of inductive types as initial algebras, and coinductive types as final co-algebras, for a functor [52, 92].

Chapter 18

Recursive Types

Inductive and coinductive types, such as natural numbers and streams, may be seen as examples of *fixed points* of type operators *up to isomorphism*. An isomorphism between two types, τ_1 and τ_2, is given by two expressions

1. $x_1 : \tau_1 \vdash e_2 : \tau_2$, and
2. $x_2 : \tau_2 \vdash e_1 : \tau_1$

that are mutually inverse to each other.[1] For example, the types $\mathtt{nat}$ and $\mathtt{unit} + \mathtt{nat}$ are isomorphic, as witnessed by the following two expressions:

1. $x : \mathtt{unit} + \mathtt{nat} \vdash \mathtt{case}\, x \,\{\mathtt{l} \cdot \_ \Rightarrow \mathtt{z} \mid \mathtt{r} \cdot x_2 \Rightarrow \mathtt{s}(x_2)\} : \mathtt{nat}$, and
2. $x : \mathtt{nat} \vdash \mathtt{ifz}\, x \,\{\mathtt{z} \Rightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{s}(x_2) \Rightarrow \mathtt{r} \cdot x_2\} : \mathtt{unit} + \mathtt{nat}$.

These are called, respectively, the $\mathtt{fold}$ and $\mathtt{unfold}$ operations of the isomorphism $\mathtt{nat} \cong \mathtt{unit} + \mathtt{nat}$. Thinking of $\mathtt{unit} + \mathtt{nat}$ as $[\mathtt{nat}/t](\mathtt{unit} + t)$, this means that $\mathtt{nat}$ is a *fixed point* of the type operator $t\,.\,\mathtt{unit} + t$.

In this chapter we study the language $\mathcal{L}\{+ \times \rightharpoonup \mu\}$, which provides solutions to all type isomorphism equations. The *recursive type* $\mu t\,.\,\tau$ is defined to be a solution to the type isomorphism

$$\mu t\,.\,\tau \cong [\mu t\,.\,\tau/t]\tau.$$

This is witnessed by the operations

$$x : \mu t\,.\,\tau \vdash \mathtt{unfold}(x) : [\mu t\,.\,\tau/t]\tau$$

[1] To make this precise requires a discussion of equivalence of expressions to be taken up in Chapter 49. For now we will rely on an intuitive understanding of when two expressions are equivalent.

and

$$x : [\mu t.\tau/t]\tau \vdash \mathtt{fold}(x) : \mu t.\tau,$$

which are mutually inverse to each other.

Requiring solutions to all type equations may seem suspicious, since we know by Cantor's Theorem that an isomorphisms such as $X \cong (X \to \mathbf{2})$ is impossible. This negative result tells us not that our requirement is untenable, but rather that *types are not sets*. To permit solution of arbitrary type equations, we must take into account that types describe computations, some of which may not even terminate. Consequently, the function space does not coincide with the set-theoretic function space, but rather is analogous to it (in a precise sense that we shall not go into here).

18.1 Solving Type Isomorphisms

The *recursive type* $\mu t.\tau$, where $t.\tau$ is a type operator, represents a solution for t to the isomorphism $t \cong \tau$. The solution is witnessed by two operations, $\mathtt{fold}(e)$ and $\mathtt{unfold}(e)$, that relate the recursive type $\mu t.\tau$ to its unfolding, $[\mu t.\tau/t]\tau$, and serve, respectively, as its introduction and elimination forms.

The language $\mathcal{L}\{+\times\rightharpoonup\mu\}$ extends $\mathcal{L}\{\rightharpoonup\}$ with recursive types and their associated operations.

Type	τ	$::=$	t	self-reference
		$\mid$ $\mathtt{rec}(t.\tau)$	$\mu t.\tau$	recursive
Expr	e	$::=$	$\mathtt{fold}[t.\tau](e)$	constructor
		$\mid$ $\mathtt{unfold}(e)$	$\mathtt{unfold}(e)$	destructor

The statics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ consists of two forms of judgement. The first, called *type formation*, is a general hypothetical judgement of the form

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

where Δ has the form $t_1 \text{ type}, \ldots, t_k \text{ type}$. Type formation is inductively defined by the following rules:

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}} \tag{18.1a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ type}} \tag{18.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{18.1c}$$

The second form of judgement comprising the statics is the *typing judgement*, which is a hypothetical judgement of the form

$$\Gamma \vdash e : \tau,$$

where we assume that τ type. Typing for $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is inductively defined by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \tag{18.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{18.2b}$$

The dynamics of $\mathcal{L}\{+\times\rightharpoonup\mu\}$ is specified by one axiom stating that the elimination form is inverse to the introduction form.

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{fold}[t.\tau](e)\ \mathsf{val}} \tag{18.3a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{fold}[t.\tau](e) \mapsto \mathtt{fold}[t.\tau](e')}\right\} \tag{18.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{18.3c}$$

$$\frac{\mathtt{fold}[t.\tau](e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \mapsto e} \tag{18.3d}$$

The bracketed premise and rule are to be included for an *eager* interpretation of the introduction form, and omitted for a *lazy* interpretation.

It is a straightforward exercise to prove type safety for $\mathcal{L}\{+\times\rightharpoonup\mu\}$.

Theorem 18.1 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

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

18.2 Recursive Data Structures

One important application of recursive types is to the representation of inductive data types such as the type of natural numbers. We may think of the type `nat` as a solution (up to isomorphism) of the type equation

$$\mathtt{nat} \cong [\mathtt{z} : \mathtt{unit}, \mathtt{s} : \mathtt{nat}]$$

According to this isomorphism every natural number is either zero or the successor of another natural number. A solution is given by the recursive type

$$\mu t.\,[\mathtt{z}:\mathtt{unit},\mathtt{s}:t]. \tag{18.4}$$

The introductory forms for the type `nat` are defined by the following equations:

$$\begin{aligned}\mathtt{z} &= \mathtt{fold}(\mathtt{z}\cdot\langle\rangle)\\ \mathtt{s}(e) &= \mathtt{fold}(\mathtt{s}\cdot e).\end{aligned}$$

The conditional branch may then be defined as follows:

$$\mathtt{ifz}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \;=\; \mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{z}\cdot\_\Rightarrow e_0 \mid \mathtt{s}\cdot x\Rightarrow e_1\},$$

where the "underscore" indicates a variable that does not occur free in e_0. It is easy to check that these definitions exhibit the expected behavior.

As another example, the type `list` of lists of natural numbers may be represented by the recursive type

$$\mu t.\,[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times t]$$

so that we have the isomorphism

$$\mathtt{list}\cong[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times\mathtt{list}].$$

The list formation operations are represented by the following equations:

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

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned}&\mathtt{listcase}\,e\,\{\mathtt{nil}\Rightarrow e_0 \mid \mathtt{cons}(x;y)\Rightarrow e_1\} =\\ &\qquad\mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{n}\cdot\_\Rightarrow e_0 \mid \mathtt{c}\cdot\langle x,y\rangle\Rightarrow e_1\},\end{aligned}$$

where we have used an underscore for a "don't care" variable, and used pattern-matching syntax to bind the components of a pair.

As long as sums and products are evaluated eagerly, there is a natural correspondence between this representation of lists and the conventional "blackboard notation" for linked lists. We may think of `fold` as an abstract

heap-allocated pointer to a tagged cell consisting of either (a) the tag `n` with no associated data, or (b) the tag `c` attached to a pair consisting of a natural number and another list, which must be an abstract pointer of the same sort. If sums or products are evaluated lazily, then the blackboard notation breaks down because it is unable to depict the suspended computations that are present in the data structure. In general there is no substitute for the type itself. Drawings can be helpful, but the type determines the semantics.

We may also represent coinductive types, such as the type of streams of natural numbers, using recursive types. The representation is particularly natural in the case that `fold(-)` is evaluated lazily, for then we may define the type `stream` to be the recursive type

$$\mu t.\mathtt{nat} \times t.$$

This states that every stream may be thought of as a computation of a pair consisting of a number and another stream. If `fold(-)` is evaluated eagerly, then we may instead consider the recursive type

$$\mu t.\mathtt{unit} \to (\mathtt{nat} \times t),$$

which expresses the same representation of streams. In either case streams cannot be easily depicted in blackboard notation, not so much because they are infinite, but because there is no accurate way to depict the delayed computation other than by an expression in the programming language. Here again we see that pictures can be helpful, but are not adequate for accurately defining a data structure.

18.3 Self-Reference

In the general recursive expression, $\mathtt{fix}[\tau](x.e)$, the variable, x, stands for the expression itself. This is ensured by the unrolling transition

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

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, which is to be thought of as a function of x that it implicitly implied to the recursive expression itself whenever it is used. In many well-known languages this implicit argument has a special name, such as `this` or `self`, that emphasizes its self-referential interpretation.

Using this intuition as a guide, we may derive general recursion from recursive types. This derivation shows that general recursion may, like other language features, be seen as a manifestation of type structure, rather than an *ad hoc* language feature. The derivation is based on isolating a type of self-referential expressions of type τ, written $\texttt{self}(\tau)$. The introduction form of this type is (a variant of) general recursion, written $\texttt{self}[\tau](x.e)$, and the elimination form is an operation to unroll the recursion by one step, written $\texttt{unroll}(e)$. The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \texttt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{self}[\tau](x.e) : \texttt{self}(\tau)} \tag{18.5a}$$

$$\frac{\Gamma \vdash e : \texttt{self}(\tau)}{\Gamma \vdash \texttt{unroll}(e) : \tau} \tag{18.5b}$$

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

$$\frac{}{\texttt{self}[\tau](x.e)\ \texttt{val}} \tag{18.6a}$$

$$\frac{e \mapsto e'}{\texttt{unroll}(e) \mapsto \texttt{unroll}(e')} \tag{18.6b}$$

$$\frac{}{\texttt{unroll}(\texttt{self}[\tau](x.e)) \mapsto [\texttt{self}[\tau](x.e)/x]e} \tag{18.6c}$$

The main difference, compared to general recursion, is that we distinguish a type of self-referential expressions, rather than impose self-reference at every type. However, as we shall see shortly, the self-referential type is sufficient to implement general recursion, so the difference is largely one of technique.

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

$$\texttt{self}(\tau) \cong \texttt{self}(\tau) \to \tau.$$

This means that we seek a fixed point of the type operator $t.t \to \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\texttt{rec}(t.t \to \tau),$$

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

The self-referential expression $\texttt{self}[\tau](x.e)$ is then defined to be the expression

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

We may easily check that Rule (18.5a) is derivable according to this definition. The expression $\texttt{unroll}(e)$ is correspondingly defined to be the expression

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

It is easy to check that Rule (18.5b) is derivable from this definition. Moreover, we may check that

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

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

One consequence of admitting the self-referential type $\texttt{self}(\tau)$ is that we may use it to define general recursion at *any* type. To be precise, we may define $\texttt{fix}[\tau](x.e)$ to stand for the expression

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

in which we have unrolled the recursion at each occurrence of x within e. It is easy to check that this verifies the statics of general recursion given in Chapter 12. Moreover, it also validates the dynamics, as evidenced by the following derivation:

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

It follows that recursive types may be used to define a non-terminating expression of every type, namely $\texttt{fix}[\tau](x.x)$. Unlike many other type constructs we have considered, recursive types change the meaning of *every* type, not just those that involve recursion. Recursive types are therefore said to be a *non-conservative extension* of languages such as $\mathcal{L}\{\texttt{nat}\to\}$, which otherwise admits no non-terminating computations.

18.4 Notes

The systematic study of recursive types in programming was initiated by Scott [86, 87] to provide a mathematical model of the untyped λ-calculus.

The derivation of recursion from recursive types is essentially an application of Scott's theory to find the interpretation of a fixed point combinator in a model of the λ-calculus given by a recursive type. The general theory of recursive types was studied by Smyth and Plotkin [88] from a category-theoretic perspective.