Part VI

Infinite Data Types

Chapter 19

Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important classes of recursive types. 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 an *iterator*, 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*.

The motivating example of an inductive type is the type of natural numbers. It is the least type containing the introductory forms `z` and `s(`e`)`, where e is again an introductory form. To compute with a number we define a recursive procedure that returns a specified value on `z`, and, for `s(`e`)`, returns a value defined in terms of the recursive call to itself on e. Other examples of inductive types are strings, lists, trees, and any other type that may be thought of as finitely generated from its introductory forms.

The motivating example of a coinductive type is the type of streams of natural numbers. Every stream may be thought of as being in the process of generation of pairs consisting of a natural number (its head) and another stream (its tail). To create a stream we define a generator that, when

prompted, produces such a natural number and a co-recursive call to the generator. Other examples of coinductive types include the type of regular trees, which includes nodes whose descendants are also ancestors, and the type of co-natural numbers, which includes a "point at infinity" consisting of an infinite stack of successors.

19.1 Static Semantics

We will consider the language $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}\}$, which extends $\mathcal{L}\{\rightarrow\times+\}$ with inductive and co-inductive types.

19.1.1 Types and Operators

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	t	t
		$\mid$	$\mathtt{ind}(t.\tau)$	$\mu_{\mathsf{i}}(t.\tau)$
		$\mid$	$\mathtt{coi}(t.\tau)$	$\mu_{\mathsf{f}}(t.\tau)$

The subscripts on the inductive and coinductive types are intended to indicate "initial" and "final", respectively, with the meaning that the inductive types determine "least" solutions to certain type equations, and the coinductive types determine "greatest" solutions.

We will consider *type formation* judgements of the form

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

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

$$\frac{}{\Delta, t\ \mathsf{type} \mid t\ \mathsf{type}} \tag{19.1a}$$

$$\frac{}{\Delta \mid \mathtt{unit}\ \mathsf{type}} \tag{19.1b}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid \tau_2\ \mathsf{type}}{\Delta \mid \mathtt{prod}(\tau_1;\tau_2)\ \mathsf{type}} \tag{19.1c}$$

$$\frac{}{\Delta \mid \mathtt{void}\ \mathsf{type}} \tag{19.1d}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid \tau_2\ \mathsf{type}}{\Delta \mid \mathtt{sum}(\tau_1;\tau_2)\ \mathsf{type}} \tag{19.1e}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid \tau_2\ \mathsf{type}}{\Delta \mid \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{19.1f}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \tau\ \mathsf{type} \quad \Delta \mid t.\tau\ \mathsf{pos}}{\Delta \mid \mathtt{ind}(t.\tau)\ \mathsf{type}} \tag{19.1g}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \tau\ \mathsf{type} \quad \Delta \mid t.\tau\ \mathsf{pos}}{\Delta \mid \mathtt{coi}(t.\tau)\ \mathsf{type}} \tag{19.2}$$

The premises on Rules (19.1g) and (19.2) involve a judgement of the form $t.\tau$ pos, which will be explained in Section 19.2 on the following page.

A *type operator* is an abstractor of the form $t.\tau$ such that t type $\mid$ τ type. Thus a type operator may be thought of as a type, τ, with a distinguished free variable, t, possibly occurring in it. It follows from the meaning of the hypothetical judgement that if $t.\tau$ is a well-formed type operator, and σ type, then $[\sigma/t]\tau$ type. Thus, a type operator may also be thought of as a mapping from types to types given by substitution.

As an example of a type operator, consider the abstractor $t.\mathtt{unit} + t$, which will be used in the definition of the natural numbers as an inductive type. Other examples include $t.\mathtt{unit} + (\mathtt{nat} \times t)$, which underlies the definition of the inductive type of lists of natural numbers, and $t.\mathtt{nat} \times t$, which underlies the coinductive type of streams of natural numbers.

19.1.2 Expressions

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

Category	*Item*		*Abstract*	*Concrete*
Expr	e	::=	$\mathtt{in}[t.\tau](e)$	$\mathtt{in}(e)$
		$\mid$	$\mathtt{rec}[t.\tau](x.e;e')$	$\mathtt{rec}(x.e;e')$
		$\mid$	$\mathtt{out}[t.\tau](e)$	$\mathtt{out}(e)$
		$\mid$	$\mathtt{gen}[t.\tau](x.e;e')$	$\mathtt{gen}(x.e;e')$

The expression $\mathtt{rec}(x.e;e')$ is called an *iterator*, and the expression $\mathtt{gen}(x.e;e')$ is called a *co-iterator*, or *generator*. The expression $\mathtt{in}(e)$ is called a *fold* operation, or *constructor*, and $\mathtt{out}(e)$ is called an *unfold* operation, or *destructor*.

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

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{in}[t.\tau](e) : \mathtt{ind}(t.\tau)} \tag{19.3a}$$

$$\frac{\Gamma \vdash e' : \mathtt{ind}(t.\tau) \quad \Gamma, x : [\rho/t]\tau \vdash e : \rho}{\Gamma \vdash \mathtt{rec}[t.\tau](x.e; e') : \rho} \tag{19.3b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{out}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{19.3c}$$

$$\frac{\Gamma \vdash e' : \rho \quad \Gamma, x : \rho \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau](x.e; e') : \mathtt{coi}(t.\tau)} \tag{19.3d}$$

The dynamic semantics of these constructs is given in terms of the action of a positive type operator, which we now define.

19.2 Positive Type Operators

The formation of inductive and coinductive types is restricted to a special class of type operators, called the *(strictly) positive type operators*.[1] These are type operators of the form $t.\tau$ in which t is restricted so that its occurrences within τ do not lie within the domain of a function type. For example, the type operator $t.\mathtt{nat} \to t$ is positive, as is $t.u \to t$, where u type is some type variable other than t. On the other hand, the type operator $t.t \to t$ is not positive, because t occurs in the domain of a function type.

The judgement $\Delta \mid t.\tau \ \mathsf{pos}$, where $\Delta, t \ \mathsf{type} \mid \tau \ \mathsf{type}$, is inductively defined by the following rules:

$$\frac{}{\Delta \mid t.t \ \mathsf{pos}} \tag{19.4a}$$

$$\frac{u \neq t}{\Delta \mid t.u \ \mathsf{pos}} \tag{19.4b}$$

$$\frac{}{\Delta \mid t.\mathtt{unit} \ \mathsf{pos}} \tag{19.4c}$$

$$\frac{\Delta \mid t.\tau_1 \ \mathsf{pos} \quad \Delta \mid t.\tau_2 \ \mathsf{pos}}{\Delta \mid t.\tau_1 \times \tau_2 \ \mathsf{pos}} \tag{19.4d}$$

[1] A more permissive notion of *positive type operator* is sometimes considered, but we shall make use only of the strict form.

$$\frac{}{\Delta \mid t.\mathtt{void}\ \mathsf{pos}} \tag{19.4e}$$

$$\frac{\Delta \mid t.\tau_1\ \mathsf{pos} \quad \Delta \mid t.\tau_2\ \mathsf{pos}}{\Delta \mid t.\tau_1 + \tau_2\ \mathsf{pos}} \tag{19.4f}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid t.\tau_2\ \mathsf{pos}}{\Delta \mid t.\tau_1 \to \tau_2\ \mathsf{pos}} \tag{19.4g}$$

Notice that in Rule (19.4g), the type variable t is not permitted to occur in τ_1, the domain type of the function type.

Positivity is preserved under substitution.

Lemma 19.1. *If* $t.\sigma\ \mathsf{pos}$ *and* $u.\tau\ \mathsf{pos}$, *then* $t.[\sigma/u]\tau\ \mathsf{pos}$.

Proof. By rule induction on Rules (19.4). □

Strictly positive type operators admit a *covariant action*, or *map* operation, that transforms types and expressions in tandem. Specifically, if $t.\tau\ \mathsf{pos}$, then

1. If $\sigma\ \mathsf{type}$, then $\mathtt{Map}[t.\tau](\sigma)\ \mathsf{type}$.
2. If $x : \sigma_1 \vdash e : \sigma_2$ and $\mathtt{map}[t.\tau](x.e) = x'.e'$, then $x' : \mathtt{Map}[t.\tau](\sigma_1) \vdash e' : \mathtt{Map}[t.\tau](\sigma_2)$.

The action on types is given by substitution:

$$\mathtt{Map}[t.\tau](\sigma) := [\sigma/t]\tau.$$

The action of a type operator on an expression is an example of *generic programming* in which the type of a computation determines its behavior. Specifically, the action of the type operator $t.\tau$ on an abstraction $x.e$ transforms an element e_1 of type $\mathtt{Map}[t.\tau](\sigma_1)$ into an element of e_2 of type $\mathtt{Map}[t.\tau](\sigma_2)$. This is achieved by replacing each sub-expression, d, of e_1 corresponding to an occurrence of t in τ by the expression $[d/x]e_2$. (This is well-defined provided that $t.\tau$ is a positive type operator.)

For example, consider the type operator

$$t.\tau = t.\mathtt{unit} + (\mathtt{nat} \times t).$$

The action of this operator on $x.e$ such that

$$x : \sigma_1 \vdash e : \sigma_2$$

is the abstractor $x'.e'$ with type

$$x' : \mathtt{unit} + (\mathtt{nat} \times \sigma_1) \vdash e' : \mathtt{unit} + (\mathtt{nat} \times \sigma_2).$$

The expression e' is such that if we instantiate x' by $\mathtt{in[l]}(\langle\rangle)$, then e' evaluates to $\mathtt{in[l]}(\langle\rangle)$, and if we instantiate x' by $\mathtt{in[r]}(\langle d_1, d_2\rangle)$, it evaluates to $\mathtt{in[r]}(\langle d_1, [d_2/x]e\rangle)$. Note that this action is independent of the choice of σ_1 and σ_2. Even if σ_1 happens to be the type $\mathtt{nat}$, the action in the second case above remains the same. In particular, the first component, d_1, of the pair is passed through untouched, whereas d_2 is replaced by $[d_2/x]e$, even though it, too, has type $\mathtt{nat}$. This is because the action is guided by the operator $t.\tau$, and not by $[\sigma_1/t]\tau$.

The action of a strictly positive type operator on an abstraction is given by the judgement

$$\mathtt{map}[t.\tau](x.e) = x'.e',$$

which is inductively defined by the following rules:

$$\frac{}{\mathtt{map}[t.t](x.e) = x.e} \tag{19.5a}$$

$$\frac{u \neq t}{\mathtt{map}[t.u](x.e) = x.x} \tag{19.5b}$$

$$\frac{}{\mathtt{map}[t.\mathtt{unit}](x.e) = x'.\langle\rangle} \tag{19.5c}$$

$$\frac{\begin{array}{c}\mathtt{map}[t.\tau_1](x.\mathtt{proj[l]}(e)) = x'.e_1' \\ \mathtt{map}[t.\tau_2](x.\mathtt{proj[r]}(e)) = x'.e_2'\end{array}}{\mathtt{map}[t.\tau_1 \times \tau_2](x.e) = x'.\mathtt{pair}(e_1'; e_2')} \tag{19.5d}$$

$$\frac{}{\mathtt{map}[t.\mathtt{void}](x.e) = x'\mathtt{abort}(x')} \tag{19.5e}$$

$$\frac{\begin{array}{c}\mathtt{map}[t.\tau_1](x_1.[\mathtt{in[l]}(x_1)/x]e) = x_1'.e_1' \\ \mathtt{map}[t.\tau_2](x_2.[\mathtt{in[r]}(x_2)/x]e) = x_1'.e_2'\end{array}}{\mathtt{map}[t.\tau_1 + \tau_2](x.e) = x'.\mathtt{case}(x'; x_1'.e_1'; x_2'.e_2')} \tag{19.5f}$$

$$\frac{\mathtt{map}[t.\tau_2](x.e) = x_2'.e_2'}{\mathtt{map}[t.\tau_1 \to \tau_2](x.e) = x'.\lambda(x_1' : \tau_1.[x'(x_1')/x_2']e_2')} \tag{19.5g}$$

Lemma 19.2. *If $x : \sigma \vdash e : \sigma'$, and $\mathtt{map}[t.\tau](x.e) = x'.e'$, then $x' : \mathtt{Map}[t.\tau](\sigma) \vdash e' : \mathtt{Map}[t.\tau](\sigma')$.*

Proof. By rule induction on Rules (19.5). □

19.3 Dynamic Semantics

The dynamic semantics of inductive and coinductive types is given in terms of the covariant action of the associated type operator. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\}$:

$$\frac{}{\mathtt{in}(e)\ \mathsf{val}} \tag{19.6a}$$

$$\frac{e' \mapsto e''}{\mathtt{rec}(x.e; e') \mapsto \mathtt{rec}(x.e; e'')} \tag{19.6b}$$

$$\frac{\mathtt{map}[t.\tau](x'.\mathtt{rec}(x.e; x')) = x''.e''}{\mathtt{rec}(x.e; \mathtt{in}(e')) \mapsto [[e'/x'']e''/x]e} \tag{19.6c}$$

$$\frac{}{\mathtt{gen}(x.e; e')\ \mathsf{val}} \tag{19.6d}$$

$$\frac{e \mapsto e'}{\mathtt{out}(e) \mapsto \mathtt{out}(e')} \tag{19.6e}$$

$$\frac{\mathtt{map}[t.\tau](x'.\mathtt{gen}(x.e; x')) = x''.e''}{\mathtt{out}(\mathtt{gen}(x.e; e')) \mapsto [[e'/x]e/x'']e''} \tag{19.6f}$$

Rule (19.6c) states that to evaluate the iterator on a value of recursive type, we inductively apply the iterator as guided by the type operator to the value, and then perform the inductive step on the result. Rule (19.6f) is simply the dual of this rule for coinductive types.

Lemma 19.3. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on Rules (19.6). □

Lemma 19.4. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on Rules (19.3). □

Although we shall not give the proof here, the language $\mathcal{L}\{\mu_i\mu_f\}$ is terminating, and all functions defined within it are total.

Theorem 19.5. *If $e : \tau$ in $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}\}$, then there exists e' val such that $e \mapsto^* e'$.*

The judgement $\Gamma \vdash e_1 \equiv e_2 : \tau$ of *definitional equivalence* (or *symbolic evaluation*) is defined to be the strongest congruence containing the extension of the dynamic semantics to open expressions. In particular the following two rules are admissible as principles of definitional equivalence:

$$\frac{\mathtt{map}[t.\tau]\,(x'.\mathtt{rec}\,(x.e;x')) = x''.e''}{\Gamma \vdash \mathtt{rec}\,(x.e;\mathtt{in}\,(e')) \equiv [[e'/x'']e''/x]e : \rho} \tag{19.7a}$$

$$\frac{\mathtt{map}[t.\tau]\,(x'.\mathtt{gen}\,(x.e;x')) = x''.e''}{\Gamma \vdash \mathtt{out}\,(\mathtt{gen}\,(x.e;e')) \equiv [[e'/x]e/x'']e'' : [\mathtt{coi}\,(t.\tau)/t]\tau} \tag{19.7b}$$

In addition to these rules we also have rules specifying that definitional equivalence is an equivalence relation, and that it is a congruence with respect to all expression-forming operators of the language. These rules license the replacement of any sub-expression of an expression by a definitionally equivalent one to obtain a definitionally equivalent result.

19.4 Fixed Point Properties

Inductive and coinductive types enjoy an important property that will play a prominent role in Chapter 20, called a *fixed point property*, that characterizes them as solutions to recursive type equations. Specifically, the inductive type $\mu_{\mathsf{i}}(t.\tau)$ is isomorphic to its unrolling,

$$\mu_{\mathsf{i}}(t.\tau) \cong [\mu_{\mathsf{i}}(t.\tau)/t]\tau,$$

and, similarly, the coinductive type is isomorphic to its unrolling,

$$\mu_{\mathsf{f}}(t.\tau) \cong [\mu_{\mathsf{f}}(t.\tau)/t]\tau$$

The isomorphism arises from the invertibility of $\mathtt{in}(-)$ in the inductive case and of $\mathtt{out}(-)$ in the coinductive case, with the required inverses given as follows:

$$x.\mathtt{in}^{-1}_{t.\tau}(x) = x.\mathtt{rec}_{t.\tau}(\mathtt{map}[t.\tau]\,(y.\mathtt{in}\,(y));x) \tag{19.8}$$

$$x.\mathtt{out}^{-1}_{t.\tau}(x) = x.\mathtt{gen}_{t.\tau}(\mathtt{map}[t.\tau]\,(y.\mathtt{out}\,(y));x) \tag{19.9}$$

Rule (19.7a) of definitional equivalence specifies that $x.\mathtt{in}^{-1}_{t.\tau}(x)$ is post-inverse to $y.\mathtt{in}(y)$, and Rule (19.7b) of definitional equivalence specifies

that $x\,.\,\mathtt{out}^{-1}_{t\,.\,\tau}(x)$ is pre-inverse to $y\,.\,\mathtt{out}(y)$. This is to say that these properties are consequences solely of the dynamic semantics of the operators involved.

It is natural to ask whether these pairs of abstractors are, in fact, two-sided inverses of each other. This is the case, but only up to *observational equivalence*, which is defined to be the coarsest consistent congruence on expressions. This relation equates as many expressions as possible subject to the conditions that it be a congruence (to permit replacing equals by equals anywhere in an expression) and that it be consistent (not equate all expressions). It is difficult, in general, to show that two expressions are observationally equivalent. In most cases some form of inductive proof is required, rather than being simply a matter of direct calculation. (Please see Chapter 50 for further discussion of observational equivalence for $\mathcal{L}\{\mathtt{nat}\to\}$, a special case of $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}\}$.)

One consequence of these inverse relationships (up to observational equivalence) is that both the inductive and the coinductive type are two solutions to the type isomorphism

$$X \cong \mathtt{Map}[t\,.\,\tau](X) = [X/t]\tau.$$

This is to say that we have two isomorphisms,

$$\mu_{\mathsf{i}}(t\,.\,\tau) \cong [\mu_{\mathsf{i}}(t\,.\,\tau)/t]\tau$$

and

$$\mu_{\mathsf{f}}(t\,.\,\tau) \cong [\mu_{\mathsf{f}}(t\,.\,\tau)/t]\tau,$$

witnessed by the two pairs of mutually inverse abstractors given above. What distinguishes the two solutions is that the inductive type is the *initial* solution, whereas the coinductive type is the *final* solution to the isomorphism equation. Initiality means that the iterator is a general means of defining functions that act on values of inductive type; finality means that the generator is a general means of creating values of coinductive types.

To understand better what is happening here, let us consider a specific example. Let $\mathtt{nat}_{\mathsf{i}}$ be the type of inductive natural numbers, $\mu_{\mathsf{i}}(t\,.\,\mathtt{unit}+t)$, and let $\mathtt{nat}_{\mathsf{f}}$ be the type of coinductive natural numbers, $\mu_{\mathsf{f}}(t\,.\,\mathtt{unit}+t)$. Intuitively, $\mathtt{nat}_{\mathsf{i}}$ is the smallest (most restrictive) type containing zero, which is defined by the expression

$$\mathtt{in}(\mathtt{in}[\mathtt{l}](\langle\rangle)),$$

and, if e is of type $\mathtt{nat}_\mathtt{i}$, its successor, which is defined by the expression

$$\mathtt{in(in[r]}(e)\mathtt{)}.$$

Dually, $\mathtt{nat}_\mathtt{f}$ is the largest (most permissive) type of expressions e such that $\mathtt{out}(e)$ is either equivalent to zero, which is defined by $\mathtt{in[l]}(\langle\rangle)$, or to the successor of some expression $e' : \mathtt{nat}_\mathtt{f}$, which is defined by $\mathtt{in[r]}(e')$.

It is not hard to embed the inductive natural numbers into the coinductive natural numbers, but the converse is impossible. In particular, the expression

$$\omega = \mathtt{gen}(x\,.\,\mathtt{in[r]}(x); \langle\rangle)$$

is a coinductive natural number that is greater than the embedding of all inductive natural numbers. Intuitively, this is because ω is an infinite stack of successors, and hence is larger than any finite stack of successors, which is to say that it is larger than any finite natural number. Any embedding of the coinductive into the inductive natural numbers would place ω among the finite natural numbers, making it larger than some and smaller than others, in contradiction to the preceding remark. (To make all this precise requires that we specify what we mean by an embedding, and to argue formally that no such embedding exists.)

19.5 Exercises

1. Extend the covariant action to nullary and binary products and sums.

2. Prove progress and preservation.

3. Show that the required abstractor mapping the inductive to the coinductive type associated with a type operator is given by the equation

   $$x\,.\,\mathtt{gen}(y\,.\,\mathtt{in}_t^{-1}{}_{\cdot\tau}(y); x).$$

   Characterize the behavior of this term when x is replaced by an element of the inductive type.

Chapter 20

Recursive Types

Inductive and coinductive types may be seen as initial and final solutions to certain forms of recursive type equations. Both the inductive type, $\mu_{\mathsf{i}}(t.\tau)$, and the coinductive type, $\mu_{\mathsf{f}}(t.\tau)$, are fixed points of the type operator $t.\tau$. Thus both are solutions to the recursion equation $t \cong \tau$ "up to isomorphism" in that both

$$\mu_{\mathsf{i}}(t.\tau) \cong [\mu_{\mathsf{i}}(t.\tau)/t]\tau$$

and

$$\mu_{\mathsf{f}}(t.\tau) \cong [\mu_{\mathsf{f}}(t.\tau)/t]\tau.$$

However, inductive and coinductive types provide solutions to type isomorphisms only for positive type operators. In many situations this restriction cannot be met. For example, to model self-reference we require a solution to the type isomorphism $t \cong t \rightharpoonup \sigma$ for which the associated type operator $t.\sigma$ is not positive.

In this chapter we study the language $\mathcal{L}\{\mu\}$, which provides solutions to general type isomorphism equations, without positivity restrictions. 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

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

and

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

which are mutually inverse to each other.

Postulating solutions to arbitrary type isomorphism equations may seems suspicious, since we know by Cantor's Theorem that isomorphisms such as $X \cong \wp(X)$ do not exist, as long as we interpret types as sets and $\wp(X)$ as the set of all subsets of X. But this only means that types cannot be understood as sets! In particular, we consider only *partial* function types (as in Chapter 15), and not *total* function types, so that the "powerset" cannot be properly expressed in the language. It is, however, powerful enough to support a rich variety of programming idioms, including general forms of self-reference.

20.1 Recursive Types

A *recursive type* has the form $\mu t.\tau$, where $t.\tau$ is any type operator. Intuitively, the recursive type is a fixed point, up to isomorphism, of the associated type operator. The isomorphism is witnessed by two operations, `fold(`e`)` and `unfold(`e`)`, that relate the recursive type $\mu t.\tau$ to its unfolding, $[\mu t.\tau/t]\tau$.

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

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	t	t
		|	`rec(`$t.\tau$`)`	$\mu t.\tau$
Expr	e	::=	`fold[`$t.\tau$`](`e`)`	`fold(`e`)`
		|	`unfold(`e`)`	`unfold(`e`)`

The expression `fold(`e`)` is the introductory form for the recursive type, and `unfold(`e`)` is its eliminatory form.

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

$$\mathcal{T} \mid \Delta \vdash \tau \ \mathsf{type},$$

where $\mathcal{T} = \{ t_1, \ldots, t_k \}$ and Δ is t_1 type, $\ldots, t_k$ type. As usual we drop explicit mention of $\mathcal{T}$, relying on typographical conventions to make clear which are the type variables of the judgement.

Type formation is inductively defined by the following rules:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{20.1a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{20.1b}$$

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

The second form of judgement comprising the static semantics is the *typing judgement*, which is a general hypothetical judgement of the form

$$\mathcal{X} \mid \Gamma \vdash e : \tau,$$

where we assume that τ type. The parameter set, $\mathcal{X}$, is a finite set of *variables*, each of which is governed by a typing hypothesis in Γ. We ordinarily suppress the parameter set, $\mathcal{X}$, in favor of relying on the form of Γ to make clear what is intended.

Typing for $\mathcal{L}\{\mu\}$ is inductively defined by the following rules:

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

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

The dynamic semantics of $\mathcal{L}\{\mu\}$ is specified by one axiom stating that the elimination form is inverse to the introduction form, together with rules specifying the order of evaluation (eager or lazy, according to whether the bracketed rules and premises are included or omitted):

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

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

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

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

Definitional equivalence for $\mathcal{L}\{\mu\}$ is the least congruence containing the following rule:

$$\frac{}{\Gamma \vdash \mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \equiv e : [\mathtt{rec}(t.\tau)/t]\tau} \tag{20.4}$$

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

Theorem 20.1 (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 $e \mapsto e'$.

20.2 Recursive Data Structures

One important application of recursive types is to the representation of data structures such as lists and trees whose size and content is determined during the course of execution of a program.

One example is the type of natural numbers, which we have taken as primitive in Chapter 15. We may instead treat `nat` as a recursive type by thinking of it as a solution (up to isomorphism) of the type equation $t \cong 1 + t$, which is to say that every natural number is either zero or the successor of another natural number. More formally, we may define `nat` to be the recursive type

$$\mu t.\,[\mathtt{z}:\mathtt{unit}, \mathtt{s}:t], \tag{20.5}$$

which specifies that

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

The zero and successor operations are correspondingly defined by the following equations:

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

The conditional branch on zero is defined by the following equation:

$$\begin{aligned} &\mathtt{ifz}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\} = \\ &\qquad \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{in}[\mathtt{z}](\_) \Rightarrow e_0 \mid \mathtt{in}[\mathtt{s}](x) \Rightarrow e_1\}, \end{aligned}$$

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 in that they correctly simulate the dynamic semantics given in Chapter 15.

As another example, the type `nat 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

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

The list formation operations are represented by the following equations:

$$\begin{aligned} \texttt{nil} &= \texttt{fold(in[n](}\langle\rangle\texttt{))} \\ \texttt{cons(}e_1\texttt{;}e_2\texttt{)} &= \texttt{fold(in[c](}\langle e_1, e_2\rangle\texttt{))}. \end{aligned}$$

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

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

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

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.

20.3 Self-Reference

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

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

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 $\texttt{self(}\tau\texttt{)}$. The introduction form of this type is (a variant of) general recursion, written $\texttt{self[}\tau\texttt{]}(x\texttt{.}e)$, and the elimination form is an operation to unroll the recursion by one step,

written $\texttt{unroll}(e)$. The static semantics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \texttt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{self}[\tau](x.e) : \texttt{self}(\tau)} \tag{20.6a}$$

$$\frac{\Gamma \vdash e : \texttt{self}(\tau)}{\Gamma \vdash \texttt{unroll}(e) : \tau} \tag{20.6b}$$

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

$$\frac{}{\texttt{self}[\tau](x.e)\ \mathsf{val}} \tag{20.7a}$$

$$\frac{e \mapsto e'}{\texttt{unroll}(e) \mapsto \texttt{unroll}(e')} \tag{20.7b}$$

$$\frac{}{\texttt{unroll}(\texttt{self}[\tau](x.e)) \mapsto [\texttt{self}[\tau](x.e)/x]e} \tag{20.7c}$$

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 $t \# \tau$ 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

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

We may easily check that Rule (20.6a) 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 (20.6b) is derivable from this definition. Moreover, we may check that the definitional equivalence

$$\texttt{unroll(self}\,y\,\texttt{is}\,e\texttt{)} \equiv [\texttt{self}\,y\,\texttt{is}\,e/y]e$$

also holds by expanding the definitions and applying the rules of definitional equivalence for recursive types.

This completes the derivation of the type $\texttt{self}(\tau)$ of self-referential expressions of type τ. Using this type we may define general recursion at any type τ by simply inserting unrolling operations that are implicit in the semantics of general recursion. Specifically, we may define $\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e$ to be the expression

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

It is easy to check that this verifies the static semantics of general recursion given in Chapter 15. Moreover, it also validates the dynamic semantics, as evidenced by the following derivation:

$$\begin{aligned}\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e &= \texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll}(y)/x]e\texttt{)}\\ &\equiv [\texttt{unroll(self}\,y\,\texttt{is}\,[\texttt{unroll}(y)/x]e\texttt{)}/x]e\\ &= [\texttt{fix}\,x\texttt{:}\tau\,\texttt{is}\,e/x]e.\end{aligned}$$

By replacing x in e by $\texttt{unroll}(e)$, and wrapping the entire self-referential expression similarly, we ensure that the self-reference is unrolled implicitly as in Chapter 15, rather than explicitly, as here.

One consequence of this derivation is that adding recursive types to a programming language is a *non-conservative extension*. For suppose that we add recursive types to a terminating language such as $\mathcal{L}\{\texttt{nat}\rightarrow\}$ defined in Chapter 14. The foregoing argument shows that general recursion is definable in this extension, and hence that the termination property of the language has been destroyed. This is in contrast to extensions with, say, product and sum types, which do not disrupt the termination properties of the language. In short, adding new language features (new forms of type) can have subtle, and often surprising, consequences!

20.4 Exercises

Part VII

Dynamic Types

Chapter 21

The Untyped λ-Calculus

Types are the central organizing principle in the study of programming languages. Yet many languages of practical interest are said to be *untyped*. Have we missed something important? The answer is *no*! The supposed opposition between typed and untyped languages turns out to be illusory. In fact, untyped languages are special cases of typed languages with a single, pre-determined recursive type. Far from being *untyped*, such languages are instead *uni-typed*.[1]

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) λ-calculus*. This formalism was introduced by Church in the 1930's as a universal language of computable functions. It is distinctive for its austere elegance. The λ-calculus has but one "feature", the higher-order function, with which to compute. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a well-worn phrase, in the λ-calculus it's functions all the way down!

21.1 The λ-Calculus

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

Category	*Item*		*Abstract*	*Concrete*
Term	u	$::=$	x	x
		$\mid$	$\lambda(x.u)$	$\lambda x.u$
		$\mid$	$\mathtt{ap}(u_1;u_2)$	$u_1(u_2)$

[1] An apt description suggested by Dana Scott.

The second form of expression is called a *λ-abstraction*, and the third is called *application*.

The static semantics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical judgements of the form $x_1, \ldots, x_n \mid x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok} \vdash u \ \mathsf{ok}$, stating that u is a well-formed expression involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the parameters when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x \ \mathsf{ok} \vdash x \ \mathsf{ok}} \tag{21.1a}$$

$$\frac{\Gamma \vdash u_1 \ \mathsf{ok} \quad \Gamma \vdash u_2 \ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(u_1; u_2) \ \mathsf{ok}} \tag{21.1b}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \ \mathsf{ok}}{\Gamma \vdash \lambda(x.u) \ \mathsf{ok}} \tag{21.1c}$$

The dynamic semantics is given by the following rules:

$$\frac{}{\lambda(x.u) \ \mathsf{val}} \tag{21.2a}$$

$$\frac{}{\mathtt{ap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{21.2b}$$

$$\frac{u_1 \mapsto u_1'}{\mathtt{ap}(u_1; u_2) \mapsto \mathtt{ap}(u_1'; u_2)} \tag{21.2c}$$

In the λ-calculus literature this judgement is called *weak head reduction*. The first rule is called *β-reduction*; it defines the meaning of function application as substitution of argument for parameter.

Despite the apparent lack of types, $\mathcal{L}\{\lambda\}$ is nevertheless type safe!

Theorem 21.1. *If u ok, then either u val, or there exists u' such that $u \mapsto u'$ and u' ok.*

Proof. Exactly as in preceding chapters. We may show by induction on transition that well-formation is preserved by the dynamic semantics. Since every closed value of $\mathcal{L}\{\lambda\}$ is a λ-abstraction, every closed expression is either a value or can make progress. □

Definitional equivalence for $\mathcal{L}\{\lambda\}$ is a judgement of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok}$ for some $n \geq 0$, and e and e' are terms

having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{21.3a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{21.3b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{21.3c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2')} \tag{21.3d}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x\,.\,u) \equiv \lambda(x\,.\,u')} \tag{21.3e}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\lambda(x\,.\,e_2); e_1) \equiv [e_1/x]e_2} \tag{21.3f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

21.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressive power: it is a *Turing-complete* language in the sense that it has the same capability to expression computations on the natural numbers as does any other known programming language. Church's Law states that any conceivable notion of computable function on the natural numbers is equivalent to the λ-calculus. This is certainly true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration due to the gravitational field of a massive object.[2]

[2] Unfortunately, it is common in Computer Science to put forth as "laws" assertions that are not scientific laws at all. For example, Moore's Law is merely an observation about a near-term trend in microprocessor fabrication that is certainly not valid over the long term, and Amdahl's Law is but a simple truth of arithmetic. Worse, Church's Law, which is a true scientific law, is usually called *Church's Thesis*, which, to the author's ear, suggests something less than the full force of a scientific law.

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 15. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. This means, in particular, that we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} = \lambda b.\, \lambda s.\, b \tag{21.4a}$$

$$\overline{n+1} = \lambda b.\, \lambda s.\, s(\overline{n}\,(b)\,(s)) \tag{21.4b}$$

It follows that

$$\overline{n}\,(u_1)\,(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

the n-fold application of u_2 to u_1. That is, $\overline{n}$ iterates its second argument (the induction step) n times, starting with its first argument (the basis).

Using this definition it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped λ-terms:

$$\mathtt{succ} = \lambda x.\, \lambda b.\, \lambda s.\, s(x\,(b)\,(s)) \tag{21.5}$$

$$\mathtt{plus} = \lambda x.\, \lambda y.\, y\,(x)\,(\mathtt{succ}) \tag{21.6}$$

$$\mathtt{times} = \lambda x.\, \lambda y.\, y(\overline{0})\,((\mathtt{plus}(x))) \tag{21.7}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

We may readily define $\mathtt{ifz}(u; u_0; u_1)$ to be the application $u\,(u_0)\,(\lambda\_.\, u_1)$, where the underscore stands for a dummy variable chosen apart from u_1. We can use this to define $\mathtt{ifz}(u; u_0; x\,.\,u_1)$, provided that we can compute the predecessor of a natural number. Doing so requires a bit of ingenuity. We wish to find a term $\mathtt{pred}$ such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{21.8}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{21.9}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ as a function of its value for $\overline{n}$. At first glance

this seems straightforward—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the nth iteration the registers contain the predecessor of n and n itself, respectively. Given the result for n, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of n we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2 \rangle = \lambda f.\, f(u_1)(u_2) \tag{21.10}$$

$$\mathtt{pr_l}(u) = u(\lambda x.\, \lambda y.\, x) \tag{21.11}$$

$$\mathtt{pr_r}(u) = u(\lambda x.\, \lambda y.\, y) \tag{21.12}$$

It is easy to check that under this encoding $\mathtt{pr_l}(\langle u_1, u_2 \rangle) \equiv u_1$, and similarly for the second projection. We may now define the required term u representing the predecessor:

$$u'_p = \lambda x.\, x(\langle \overline{0}, \overline{0} \rangle)(\lambda y.\, \langle \mathtt{pr_r}(y), \mathtt{s}(\mathtt{pr_r}(y)) \rangle) \tag{21.13}$$

$$u_p = \lambda x.\, \mathtt{pr_l}(u(x)) \tag{21.14}$$

It is then easy to check that this gives us the required behavior. Finally, we may define $\mathtt{ifz}(u; u_0; x)\, u_1$ to be the untyped term

$$u(u_0)(\lambda \_.\, [u_p(u)/x]u_1).$$

This gives us all the apparatus of PCF, apart from general recursion. But this is also definable using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the **Y** *combinator*:

$$\mathbf{Y} = \lambda F.\, (\lambda f.\, F(f(f)))(\lambda f.\, F(f(f))). \tag{21.15}$$

Observe that

$$\mathbf{Y}(F) \equiv F(\mathbf{Y}(F)).$$

Using the **Y** combinator, we may define general recursion by writing $\mathbf{Y}(\lambda x.\, u)$, where x stands for the recursive expression itself.

21.3 Scott's Theorem

Definitional equivalence for the untyped λ-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equivalent. The proof of this result is based on two key lemmas:

1. For any untyped λ-term u, we may find an untyped term v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of v, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 14 for a discussion of Gödel-numbering.)

2. Any two non-trivial[3] properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms that respect definitional equivalence are *inseparable*. This means that there is no decidable property $\mathcal{B}$ of untyped terms such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ and $\mathcal{A}_1\ u$ implies that it is *not* the case that $\mathcal{B}\ u$. In particular, if $\mathcal{A}_0$ and $\mathcal{A}_1$ are inseparable, then neither is decidable.

For a property $\mathcal{B}$ of untyped terms to respect definitional equivalence means that if $\mathcal{B}\ u$ and $u \equiv u'$, then $\mathcal{B}\ u'$.

Lemma 21.2. *For any u there exists v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$.*

Proof Sketch. The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. $\mathbf{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.
2. $\mathbf{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms, and builds the representation of the application of one to the other. The second takes a numeral for n, and yields the representation of $\overline{n}$. Given these, we may find the required term v by defining $v = w(\overline{\ulcorner w \urcorner})$, where $w = \lambda x.\, u(\mathbf{ap}(x)(\mathbf{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathbf{ap}(\overline{\ulcorner w \urcorner})(\mathbf{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

[3] A property of untyped terms is said to be *trivial* if it either holds for all untyped terms or never holds for any untyped term.

The definition is very similar to that of $\mathbf{Y}(u)$, except that u takes as input the representation of a term, and we find a v such that, when applied to the representation of v, the term u yields v itself. □

Lemma 21.3. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-vacuous properties of untyped terms that respect definitional equivalence. Then there is no untyped term w such that*

1. *For every u either $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*
2. *If $\mathcal{A}_0\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*
3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

Proof. Suppose there is such an untyped term w. Let v be the untyped term $\lambda x.\,\mathtt{ifz}(w(x); u_1; u_0)$, where $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. By Lemma 21.2 on the preceding page there is an untyped term t such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, since $\mathcal{A}_1$ respects definitional equivalence and $\mathcal{A}_1\ u_1$. But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of w, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

Corollary 21.4. *There is no algorithm to decide whether or not $u \equiv u'$.*

Proof. For fixed u consider the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$. This is non-vacuous and respects definitional equivalence, and hence is undecidable. □

21.4 Untyped Means Uni-Typed

The untyped λ-calculus may be faithfully embedded in the typed language $\mathcal{L}\{\mu\}$, enriched with recursive types. This means that every untyped λ-term has a representation as an expression in $\mathcal{L}\{\mu\}$ in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. If the execution model of the λ-calculus is call-by-name, this correspondence holds for the call-by-name variant of $\mathcal{L}\{\mu\}$, and similarly for call-by-value.

It is important to understand that this form of embedding is *not* a matter of writing an interpreter for the λ-calculus in $\mathcal{L}\{\mu\}$ (which we could

surely do), but rather a direct representation of untyped λ-terms as certain typed expressions of $\mathcal{L}\{\mu\}$. It is for this reason that we say that untyped languages are just a special case of typed languages, provided that we have recursive types at our disposal.

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-calculus! It is not the *absence* of types that gives it its power, but rather that it has *only one* type, namely the recursive type

$$D = \mu t\,.\,t \to t.$$

A value of type D is of the form $\texttt{fold}(e)$ where e is a value of type $D \to D$ — a function whose domain and range are both D. Any such function can be regarded as a value of type D by "rolling", and any value of type D can be turned into a function by "unrolling". As usual, a recursive type may be seen as a solution to a type isomorphism equation, which in the present case is the equation

$$D \cong D \to D.$$

This specifies that D is a type that is isomorphic to the space of functions on D itself, something that is impossible in conventional set theory, but is feasible in the computationally-based setting of the λ-calculus.

This isomorphism leads to the following embedding, $u^\dagger$, of u into $\mathcal{L}\{\mu\}$:

$$x^\dagger = x \tag{21.16a}$$
$$\lambda x.\,u^\dagger = \texttt{fold}(\lambda(x\!:\!D.\,u^\dagger)) \tag{21.16b}$$
$$u_1(u_2)^\dagger = \texttt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{21.16c}$$

Observe that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

$$\begin{aligned}\lambda x.\,u_1(u_2)^\dagger &= \texttt{unfold}(\texttt{fold}(\lambda(x\!:\!D.\,u_1^\dagger)))(u_2^\dagger)\\ &\equiv \lambda(x\!:\!D.\,u_1^\dagger)(u_2^\dagger)\\ &\equiv [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of u_1. Thus β-reduction is faithfully implemented by evaluation of the embedded terms.

Thus we see that the canonical *untyped* language, $\mathcal{L}\{\lambda\}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all! Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the expense of incurring substantial dynamic overhead to coerce values to and from the recursive type. In Chapter 22 we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus a traditional distinction can hardly be considered an opposition, since dynamic languages are but particular forms of static language in which (undue) emphasis is placed on a single recursive type.

21.5 Exercises

Chapter 22

Dynamic Typing

We saw in Chapter 21 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped λ-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. Obviously this property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped λ-calculus (rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number.

One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be *dynamically typed*, in supposed opposition to the *statically typed* languages we have studied thus far. In this chapter we show that the supposed opposition between static and dynamic languages is fallacious: dynamic typing is but a mode of use of static typing, and, moreover, it is profitably seen as such. Dynamic typing can hardly be in opposition to that of which it is a special case!

22.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{\mathit{dyn}\}$. The abstract syntax of $\mathcal{L}\{\mathit{dyn}\}$ is given by the

following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	d	::=	x	x
		|	$\texttt{num}(\overline{n})$	$\overline{n}$
		|	$\texttt{zero}$	$\texttt{zero}$
		|	$\texttt{succ}(d)$	$\texttt{succ}(d)$
		|	$\texttt{ifz}(d; d_0; x.d_1)$	$\texttt{ifz}\, d\, \{\texttt{zero} \Rightarrow d_0 \mid \texttt{succ}(x) \Rightarrow d_1\}$
		|	$\texttt{fun}(\lambda(x.d))$	$\lambda x.d$
		|	$\texttt{dap}(d_1; d_2)$	$d_1(d_2)$
		|	$\texttt{fix}(x.d)$	$\texttt{fix}\, x\, \texttt{is}\, d$

There are two *classes* of values in $\mathcal{L}\{dyn\}$, the *numbers*, which have the form $\overline{n}$,[1] and the *functions*, which have the form $\lambda x.d$. The elimination forms of $\mathcal{L}\{dyn\}$ operate on classified values, and must check that their arguments are of the appropriate class at run-time. The expressions $\texttt{zero}$ and $\texttt{succ}(d)$ are not in themselves values, but rather are operations that evaluate to classified values, as we shall see shortly.

The concrete syntax of $\mathcal{L}\{dyn\}$ is somewhat deceptive, in keeping with common practice in dynamic languages. For example, the concrete syntax for a number is a bare numeral, $\overline{n}$, but in fact it is just a convenient notation for the classified value, $\overline{n}$, of class $\texttt{num}$. Similarly, the concrete syntax for a function is a bare λ-abstraction, $\lambda x.d$, which must be regarded as standing for the classified value $\lambda x.d$ of class $\texttt{fun}$. It is the responsibility of the parser to translate the surface syntax into the abstract syntax, adding class information to values in the process.

The static semantics of $\mathcal{L}\{dyn\}$ is essentially the same as that of $\mathcal{L}\{\lambda\}$ given in Chapter 21; it merely checks that there are no free variables in the expression. The judgement

$$x_1\ \mathsf{ok}, \ldots x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

states that d is a well-formed expression with free variables among those in the hypothesis list.

The dynamic semantics for $\mathcal{L}\{dyn\}$ checks for errors that would never arise in a safe statically typed language. For example, function application must ensure that its first argument is a function, signaling an error in the case that it is not, and similarly the case analysis construct must ensure that its first argument is a number, signaling an error if not. The reason for

[1] The numerals, $\overline{n}$, are n-fold compositions of the form $\texttt{s(s(}\ldots\texttt{s(z)}\ldots\texttt{))}$.

having classes labelling values is precisely to make this run-time check possible. One could argue that the required check may be made by inspection of the unlabelled value itself, but this is unrealistic. At run-time both numbers and functions might be represented by machine words, the former a two's complement number, the latter an address in memory. But given an arbitrary word, one cannot determine whether it is a number or an address!

The value judgement, d val, states that d is a fully evaluated (closed) expression:

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{val}} \tag{22.1a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{val}} \tag{22.1b}$$

The dynamic semantics makes use of judgements that check the class of a value, and recover the underlying λ-abstraction in the case of a function.

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{22.2a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{is\_fun}\ \lambda(x.d)} \tag{22.2b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\texttt{num(\_)}\ \mathsf{isnt\_fun}} \tag{22.3a}$$

$$\frac{}{\texttt{fun(\_)}\ \mathsf{isnt\_num}} \tag{22.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, d err, are defined simultaneously by the following rules.

$$\frac{}{\texttt{zero} \mapsto \texttt{num(z)}} \tag{22.4a}$$

$$\frac{d \mapsto d'}{\texttt{succ}(d) \mapsto \texttt{succ}(d')} \tag{22.4b}$$

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\texttt{succ}(d) \mapsto \texttt{num(s}(\overline{n}))} \tag{22.4c}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{succ}(d)\ \mathsf{err}} \tag{22.4d}$$

$$\frac{d \mapsto d'}{\mathtt{ifz}(d; d_0; x\,.\,d_1) \mapsto \mathtt{ifz}(d'; d_0; x\,.\,d_1)} \tag{22.4e}$$

$$\frac{d\ \mathsf{is\_num}\ \mathsf{z}}{\mathtt{ifz}(d; d_0; x\,.\,d_1) \mapsto d_0} \tag{22.4f}$$

$$\frac{d\ \mathsf{is\_num}\ \mathsf{s}(\overline{n})}{\mathtt{ifz}(d; d_0; x\,.\,d_1) \mapsto [\mathtt{num}(\overline{n})\,/\,x]d_1} \tag{22.4g}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{ifz}(d; d_0; x\,.\,d_1)\ \mathsf{err}} \tag{22.4h}$$

$$\frac{d_1 \mapsto d_1'}{\mathtt{dap}(d_1; d_2) \mapsto \mathtt{dap}(d_1'; d_2)} \tag{22.4i}$$

$$\frac{d_1\ \mathsf{is\_fun}\ \lambda(x\,.\,d)}{\mathtt{dap}(d_1; d_2) \mapsto [d_2\,/\,x]d} \tag{22.4j}$$

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{dap}(d_1; d_2)\ \mathsf{err}} \tag{22.4k}$$

$$\frac{}{\mathtt{fix}(x\,.\,d) \mapsto [\mathtt{fix}(x\,.\,d)\,/\,x]d} \tag{22.4l}$$

Rule (22.4g) labels the predecessor with the class `num` to maintain the invariant that variables are bound to expressions of $\mathcal{L}\{dyn\}$.

The language $\mathcal{L}\{dyn\}$ enjoys essentially the same safety properties as $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, except that there are more opportunities for errors to arise at run-time.

Theorem 22.1. *If d ok, then either d val, or d err, or there exists d' such that $d \mapsto d'$.*

Proof. By rule induction on Rules (22.4). The rules are designed so that if d ok, then some rule, possibly an error rule, applies, ensuring progress. Since well-formedness is closed under substitution, the result of a transition is always well-formed. □

22.2 Critique of Dynamic Typing

The safety of $\mathcal{L}\{dyn\}$ is often promoted as an *advantage* of dynamic over static typing. Unlike static languages, essentially every piece of abstract syntax has a well-defined dynamic semantics. But this can also be seen as a *disadvantage*, since errors that could be ruled out at compile time by type checking are not signalled until run time in $\mathcal{L}\{dyn\}$. To make this possible, the dynamic semantics of $\mathcal{L}\{dyn\}$ incurs considerable overhead at execution time to classify values.

Consider, for example, the addition function written in $\mathcal{L}\{dyn\}$, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:

$$\texttt{fun}(\lambda(x\texttt{.fix}\,p\,\texttt{is}\,\texttt{fun}(\lambda(y\texttt{.ifz}(y;x;y'\texttt{.succ}(\texttt{dap}(p;y'))))))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda x.\texttt{fix}\,p\,\texttt{is}\,\lambda y.\texttt{ifz}\,y\,\{\texttt{zero}\Rightarrow x \mid \texttt{succ}(y')\Rightarrow \texttt{succ}(p(y'))\}.$$

It is deceptive, because the concrete syntax obscures the class tags on values, and obscures the use of primitives that check those tags. Let us examine the costs of these operations in a bit more detail.

First, observe that the body of the fixed point expression is labelled with class `fun`. The semantics of the fixed point construct binds p to this function. This means that the dynamic class check incurred by the application of p in the recursive call is guaranteed to succeed. But there is no way to suppress it by rewriting the program within $\mathcal{L}\{dyn\}$.

Second, observe that the result of applying the inner λ-abstraction is either x, the argument of the outer λ-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class `num`, even though this is guaranteed for all but the base case, which returns the given x, which can be of any class at all. In principle we can check that x is of class `num` once, and observe that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, $\mathcal{L}\{dyn\}$ gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

Third, the argument, y, to the inner function is either the original argument to the addition function, or is the predecessor of some earlier recursive call. But as long as the original call is to a value of class `num`, then

the semantics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in $\mathcal{L}\{dyn\}$, and hence there is no way to avoid the class check imposed by the conditional branch.

Class checking and labelling is not free—storage is required for the label itself, and the marking of a value with a class takes time as well as space. But while the overhead is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated whenever possible. But within $\mathcal{L}\{dyn\}$ itself there is no way to avoid the overhead, because there are no "unchecked" operations in the language—to have these without sacrificing safety requires a static type system!

22.3 Hybrid Typing

Let us consider the language $\mathcal{L}\{\texttt{nat}\ \texttt{dyn}\ \rightharpoonup\}$, whose syntax extends that of the language $\mathcal{L}\{\texttt{nat}\ \rightharpoonup\}$ defined in Chapter 15 with the following additional constructs:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`dyn`	`dyn`
Expr	e	::=	`new[`l`](`e`)`	l `!` e
		$\mid$	`cast[`l`](`e`)`	e `?` l
Class	l	::=	`num`	`num`
		$\mid$	`fun`	`fun`

The type `dyn` represents the type of labelled values. Here we have only two classes of data object, numbers and functions. Observe that the `cast` operation takes as argument a class, not a type! That is, casting is concerned with an object's *class*, which is indicated by a label, not with its *type*, which is always `dyn`.

The static semantics for $\mathcal{L}\{\texttt{nat}\ \texttt{dyn}\ \rightharpoonup\}$ is the extension of that of $\mathcal{L}\{\texttt{nat}\ \rightharpoonup\}$ with the following rules governing the type `dyn`.

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}(e) : \texttt{dyn}} \tag{22.5a}$$

$$\frac{\Gamma \vdash e : \texttt{parr(dyn;dyn)}}{\Gamma \vdash \texttt{new[fun]}(e) : \texttt{dyn}} \tag{22.5b}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[num]}(e) : \texttt{nat}} \tag{22.5c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast[fun]}(e) : \mathtt{parr(dyn;dyn)}} \tag{22.5d}$$

The static semantics ensures that class labels are applied to objects of the appropriate type, namely `num` for natural numbers, and `fun` for functions defined over labelled values.

The dynamic semantics of $\mathcal{L}\{\mathtt{nat\ dyn} \rightharpoonup\}$ is given by the following rules:

$$\frac{e\ \mathsf{val}}{\mathtt{new}[l]\,(e)\ \mathsf{val}} \tag{22.6a}$$

$$\frac{e \mapsto e'}{\mathtt{new}[l]\,(e) \mapsto \mathtt{new}[l]\,(e')} \tag{22.6b}$$

$$\frac{e \mapsto e'}{\mathtt{cast}[l]\,(e) \mapsto \mathtt{cast}[l]\,(e')} \tag{22.6c}$$

$$\frac{\mathtt{new}[l]\,(e)\ \mathsf{val}}{\mathtt{cast}[l]\,(\mathtt{new}[l]\,(e)) \mapsto e} \tag{22.6d}$$

$$\frac{\mathtt{new}[l']\,(e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l]\,(\mathtt{new}[l']\,(e))\ \mathsf{err}} \tag{22.6e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 22.2 (Canonical Forms)**.** *If $e : \mathtt{dyn}$ and e val, then $e = \mathtt{new}[l]\,(e')$ for some class l and some e' val. If $l = \mathtt{num}$, then $e' : \mathtt{nat}$, and if $l = \mathtt{fun}$, then $e' : \mathtt{parr(dyn;dyn)}$.*

Proof. By a straightforward rule induction on static semantics of $\mathcal{L}\{\mathtt{nat\ dyn} \rightharpoonup\}$. □

Theorem 22.3 (Safety)**.** *The language $\mathcal{L}\{\mathtt{nat\ dyn} \rightharpoonup\}$ is safe:*

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

Proof. Preservation is proved by rule induction on the dynamic semantics, and progress is proved by rule induction on the static semantics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for $\mathcal{L}\{\mathit{dyn}\}$—a well-typed cast might fail at run-time if the class of the case does not match the class of the value. □

22.4 Optimization of Dynamic Typing

The type dyn—whether primitive or derived—supports the smooth integration of dynamic with static typing. This means that we can take full advantage of the expressive power of static types whenever possible, while permitting the flexibility of dynamic typing whenever desirable.

One application of the hybrid framework is that it permits the optimization of dynamically typed programs by taking advantage of statically evident typing constraints. Let us examine how this plays out in the case of the addition function, which is rendered in $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ by the expression

$$\texttt{fun !}\,\lambda(x\texttt{:dyn}.\,\texttt{fix}\,p\texttt{:dyn is fun !}\,\lambda(y\texttt{:dyn}.\,e_{x,p,y})),$$

where

$$x : \texttt{dyn}, p : \texttt{dyn}, y : \texttt{dyn} \vdash e_{x,p,y} : \texttt{dyn}$$

is defined to be the expression

$$\texttt{ifz}\,(y\,\texttt{?}\,\texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num !}\,(\texttt{s}((p\,\texttt{?}\,\texttt{fun})(\texttt{num !}\,y')\,\texttt{?}\,\texttt{num}))\}.$$

This is a re-formulation of the dynamic addition function given in Section 22.2 on page 193 in which we have made explicit the checking and imposition of classes on values. We will exploit the static type system of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ to optimize this dynamically typed implementation of addition in accordance with the specification given in Section 22.2 on page 193.

First, note that that the body of the fix expression is an explicitly labelled function. This means that when the recursion is unwound, the variable p is bound to this value of type dyn. Consequently, the check that p is labelled with class fun is redundant, and can be eliminated. This is achieved by re-writing the function as follows:

$$\texttt{fun !}\,\lambda(x\texttt{:dyn}.\,\texttt{fun !}\,\texttt{fix}\,p\texttt{:dyn} \rightharpoonup \texttt{dyn is}\,\lambda(y\texttt{:dyn}.\,e'_{x,p,y})),$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,\texttt{?}\,\texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num !}\,(\texttt{s}(p(\texttt{num !}\,y')\,\texttt{?}\,\texttt{num}))\}.$$

We have "hoisted" the function class label out of the loop, and suppressed the cast inside the loop. Correspondingly, the type of p has changed to $\texttt{dyn} \rightharpoonup \texttt{dyn}$, reflecting that the body is now a "bare function", rather than a labelled function value of type dyn.

Next, observe that the parameter y of type dyn is cast to a number on each iteration of the loop before it is tested for zero. Since this function

is recursive, the bindings of y arise in one of two ways, at the initial call to the addition function, and on each recursive call. But the recursive call is made on the predecessor of y, which is a true natural number that is labelled with `num` at the call site, only to be removed by the class check at the conditional on the next iteration. This suggests that we hoist the check on y outside of the loop, and avoid labelling the argument to the recursive call. Doing so changes the type of the function, however, from $\texttt{dyn} \rightharpoonup \texttt{dyn}$ to $\texttt{nat} \rightharpoonup \texttt{dyn}$. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class `num`, and, if so, the underlying value is incremented and labelled with class `num`. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the class check is redundant, because we know that it must have class `num`. But what if the result came from the other branch of the conditional? In that case the function returns x, which need not be of class `num`! However, one might reasonably insist that this is only a theoretical possibility—after all, we are defining the addition function, and its arguments might reasonably be restricted to have class `num`. This can be achieved by replacing x by $x\,\texttt{?}\,\texttt{num}$, which checks that x is of class `num`, and returns the underlying number.

Combining these optimizations we obtain the inner loop e_x'' defined as follows:

$$\texttt{fix}\,p\texttt{:nat} \rightharpoonup \texttt{nat}\,\texttt{is}\,\lambda(y\texttt{:nat}.\,\texttt{ifz}\,y\,\{\texttt{zero} \Rightarrow x\,\texttt{?}\,\texttt{num} \mid \texttt{succ}(y') \Rightarrow \texttt{s}(p(y'))\}).$$

This function has type $\texttt{nat} \rightharpoonup \texttt{nat}$, and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Finally, recall that the overall goal is to define a version of addition that works on values of type `dyn`. Thus we require a value of type $\texttt{dyn} \rightharpoonup \texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. This can be converted to the required form by pre-composing with a cast to `num` and post-composing with a coercion to `num`:

$$\texttt{fun}\,!\,\lambda(x\texttt{:dyn}.\,\texttt{fun}\,!\,\lambda(y\texttt{:dyn}.\,\texttt{num}\,!\,(e_x''(y\,\texttt{?}\,\texttt{num})))).$$

The innermost λ-abstraction converts the function e_x'' from type $\texttt{nat} \rightharpoonup \texttt{nat}$ to type $\texttt{dyn} \rightharpoonup \texttt{dyn}$ by composing it with a class check that ensures that y is a natural number at the initial call site, and applies a label to the result to restore it to type `dyn`.

22.5 Static "Versus" Dynamic Typing

There have been many attempts to explain the distinction between dynamic and static typing, most of which are misleading or wrong. For example, it is often said that static type systems associate types with variables, but dynamic type systems associate types with values. This oft-repeated characterization appears to be justified by the absence of type annotations on λ-abstractions, and the presence of classes on values. But it is based on a confusion of classes with types—the *class* of a value (`num` or `fun`) is not its *type*. Moreover, a static type system assigns types to values just as surely as it does to variables, so the description fails on this account as well. Thus, this supposed distinction between dynamic and static typing makes no sense, and is best disregarded.

Another way to differentiate dynamic from static languages is to say that whereas static languages check types at compile time, dynamic languages check types at run time. While this description seems superficially accurate, it does not bear scrutiny. To say that static languages check types statically is to state a tautology, and to say that dynamic languages check types at run-time is to utter a falsehood. Dynamic languages perform *class checking*, not *type checking*, at run-time. For example, application checks that its first argument is labelled with `fun`; it does not type check the body of the function. Indeed, at no point does the dynamic semantics compute the *type* of a value, rather it checks its class against its expectations before proceeding. Here again, a supposed contrast between static and dynamic languages evaporates under careful analysis.

Another characterization is to assert that dynamic languages admit *heterogeneous* lists, whereas static languages admit only *homogeneous* lists. (The distinction applies to other collections as well.) To see why this description is wrong, let us consider briefly how one might add lists to $\mathcal{L}\{dyn\}$. One would add two constructs, `nil`, representing the empty list, and `cons`$(d_1; d_2)$, representing the non-empty list with head d_1 and tail d_2. The origin of the supposed distinction lies in the observation that each element of a list represented in this manner might have a different class. For example, one might form the list

$$\texttt{cons(s(z);cons(}\lambda x.\, x\texttt{;nil))},$$

whose first element is a number, and whose second element is a function. Such a list is said to be heterogeneous. In contrast static languages commit to a single *type* for each element of the list, and hence are said to be homogeneous. But here again the supposed distinction breaks down on

close inspection, because it is based on the confusion of the type of a value with its class. Every labelled value has type `dyn`, so that the lists are *type* homogeneous. But since values of type `dyn` may have different classes, lists are *class* heterogenoues—regardless of whether the language is statically or dynamically typed!

What, then, are we to make of the traditional distinction between dynamic and static languages? Rather than being in opposition to each other, we see that *dynamic languages are a mode of use of static languages*. If we have a type `dyn` in the language, then we have all of the apparatus of dynamic languages at our disposal, so there is no loss of expressive power. But there is a very significant gain from embedding dynamic typing within a static type discipline! We can avoid much of the overhead of dynamic typing by simply limiting our use of the type `dyn` in our programs, as was illustrated in Section 22.4 on page 196.

22.6 Dynamic Typing From Recursive Types

The type `dyn` codifies the use of dynamic typing within a static language. Its introduction form labels an object of the appropriate type, and its elimination form is a (possibly undefined) casting operation. Rather than treating `dyn` as primitive, we may derive it as a particular use of recursive types, according to the following definitions:[2]

$$\texttt{dyn} = \mu t.\,[\texttt{num}:\texttt{nat}, \texttt{fun}: t \rightharpoonup t] \tag{22.7}$$

$$\texttt{new}[\texttt{num}](e) = \texttt{fold}(\texttt{in}[\texttt{num}](e)) \tag{22.8}$$

$$\texttt{new}[\texttt{fun}](e) = \texttt{fold}(\texttt{in}[\texttt{fun}](e)) \tag{22.9}$$

$$\texttt{cast}[\texttt{num}](e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in}[\texttt{num}](x) \Rightarrow x \mid \texttt{in}[\texttt{fun}](x) \Rightarrow \texttt{error}\} \tag{22.10}$$

$$\texttt{cast}[\texttt{fun}](e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{in}[\texttt{num}](x) \Rightarrow \texttt{error} \mid \texttt{in}[\texttt{fun}](x) \Rightarrow x\} \tag{22.11}$$

One may readily check that the static and dynamic semantics for the type `dyn` are derivable according to these definitions.

This observation strengthens the argument that dynamic typing is but a mode of use of static typing. This encoding shows that we need not include a special-purpose type `dyn` in a statically typed language in order to

[2] Here we have made use of a special expression `error` to signal an error condition. In a richer language we would use exceptions, which are introduced in Chapter 28.

admit dynamic typing. Instead, one may use the general concepts of recursive types and sum types to define special-purpose dynamically typed sub-languages on a per-program basis. For example, if we wish to admit strings into our dynamic sub-language, then we may simply expand the type definition above to admit a third summand for strings, and so on for any type we may wish to consider. Classes emerge as labels of the summands of a sum type, and recursive types ensure that we can represent class-heterogeneous aggregates. Thus, not only is dynamic typing a special case of static typing, but we need make no special provision for it in a statically typed language, since we already have need of recursive types independently of this particular application.

22.7 Exercises