16 System F of Polymorphic Types

The languages we have considered so far are all *monomorphic* in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in **T** there is a *distinct* identity function for each type τ, namely $\lambda\,(x:\tau)\,x$, even though the behavior is the same for each choice of τ. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f:\tau_2 \to \tau_3)\,\lambda\,(g:\tau_1 \to \tau_2)\,\lambda\,(x:\tau_1)\,f(g(x)).$$

Each choice of the three types requires a *different* program, even though they all have the same behavior when executed.

Obviously, it would be useful to capture the pattern once and for all, and to instantiate this pattern each time we need it. The expression patterns codify generic (type-independent) behaviors that are shared by all instances of the pattern. Such generic expressions are *polymorphic*. In this chapter, we will study the language **F**, which was introduced by Girard under the name *System F* and by Reynolds under the name *polymorphic typed λ-calculus*. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 17), and the definability of product, sum, inductive, and coinductive types considered in the preceding chapters. (Only general recursive types extend the expressive power of the language.)

16.1 Polymorphic Abstraction

The language **F** is a variant of **T** in which we eliminate the type of natural numbers, but add, in compensation, polymorphic types:[1]

Typ	τ	::=	t	t	variable
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Exp	e	::=	x	x	
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{Lam}(t.e)$	$\Lambda(t)\,e$	type abstraction
			$\mathtt{App}\{\tau\}(e)$	$e[\tau]$	type application

A *type abstraction* $\texttt{Lam}(t.e)$ defines a *generic*, or *polymorphic*, function with *type variable* t standing for an unspecified type within e. A *type application*, or *instantiation* $\texttt{App}\{\tau\}(e)$, applies a polymorphic function to a specified type, which is plugged in for the type variable to obtain the result. The *universal type*, $\texttt{all}(t.\tau)$, classifies polymorphic functions.

The statics of **F** consists of two judgment forms, the *type formation* judgment,

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

and the *typing judgment*,

$$\Delta\ \Gamma \vdash e : \tau.$$

The hypotheses Δ have the form t type, where t is a variable of sort Typ, and the hypotheses Γ have the form $x : \tau$, where x is a variable of sort Exp.

The rules defining the type formation judgment are as follows:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{16.1a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{16.1b}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Delta \vdash \texttt{all}(t.\tau) \ \mathsf{type}} \tag{16.1c}$$

The rules defining the typing judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{16.2a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{16.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{16.2c}$$

$$\frac{\Delta, t \ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \texttt{Lam}(t.e) : \texttt{all}(t.\tau)} \tag{16.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \texttt{all}(t.\tau') \quad \Delta \vdash \tau \ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{App}\{\tau\}(e) : [\tau/t]\tau'} \tag{16.2e}$$

Lemma 16.1 (Regularity). *If $\Delta\ \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i$ type for each assumption $x_i : \tau_i$ in Γ, then $\Delta \vdash \tau$ type.*

Proof By induction on rules (16.2). □

The statics admits the structural rules for a general hypothetical judgment. In particular, we have the following critical substitution property for type formation and expression typing.

Lemma 16.2 (Substitution). *1. If Δ, t type $\vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau'$ type.*

2. *If* Δ, t *type* $\Gamma \vdash e' : \tau'$ *and* $\Delta \vdash \tau$ *type, then* $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.
3. *If* $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ *and* $\Delta\ \Gamma \vdash e : \tau$, *then* $\Delta\ \Gamma \vdash [e/x]e' : \tau'$.

The second part of the lemma requires substitution into the context Γ as well as into the term and its type, because the type variable t may occur freely in any of these positions.

Returning to the motivating examples from the introduction, the polymorphic identity function, I, is written

$$\Lambda(t)\,\lambda\,(x : t)\,x;$$

it has the polymorphic type

$$\forall(t.t \to t).$$

Instances of the polymorphic identity are written $I[\tau]$, where τ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, C, is written

$$\Lambda(t_1)\,\Lambda(t_2)\,\Lambda(t_3)\,\lambda\,(f : t_2 \to t_3)\,\lambda\,(g : t_1 \to t_2)\,\lambda\,(x : t_1)\,f(g(x)).$$

The function C has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Instances of C are obtained by applying it to a triple of types, written $C[\tau_1][\tau_2][\tau_3]$. Each such instance has the type

$$(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3).$$

Dynamics

The dynamics of **F** is given as follows:

$$\frac{}{\texttt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{16.3a}$$

$$\frac{}{\texttt{Lam}(t.e)\ \mathsf{val}} \tag{16.3b}$$

$$\frac{[e_2\ \mathsf{val}]}{\texttt{ap}(\texttt{lam}\{\tau_1\}(x.e); e_2) \longmapsto [e_2/x]e} \tag{16.3c}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{ap}(e_1; e_2) \longmapsto \texttt{ap}(e_1'; e_2)} \tag{16.3d}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{ap}(e_1; e_2) \longmapsto \texttt{ap}(e_1; e_2')}\right] \tag{16.3e}$$

$$\frac{}{\texttt{App}\{\tau\}(\texttt{Lam}(t.e)) \longmapsto [\tau/t]e} \tag{16.3f}$$

$$\frac{e \longmapsto e'}{\texttt{App}\{\tau\}(e) \longmapsto \texttt{App}\{\tau\}(e')} \tag{16.3g}$$

The bracketed premises and rule are included for a call-by-value interpretation and omitted for a call-by-name interpretation of **F**.

It is a simple matter to prove safety for **F**, using familiar methods.

Lemma 16.3 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \texttt{arr}(\tau_1; \tau_2)$, then $e = \texttt{lam}\{\tau_1\}(x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \texttt{all}(t.\tau')$, then $e = \texttt{Lam}(t.e')$ with t type $\vdash e' : \tau'$.*

Proof By rule induction on the statics. □

Theorem 16.4 (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof By rule induction on the dynamics. □

Theorem 16.5 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \longmapsto e'$.*

Proof By rule induction on the statics. □

16.2 Polymorphic Definability

The language **F** is astonishingly expressive. Not only are all (lazy) finite products and sums definable in the language, but so are all (lazy) inductive and coinductive types. Their definability is most naturally expressed using definitional equality, which is the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash (\lambda\,(x : \tau)\,e_2)(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{16.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t)\,e[\rho] \equiv [\rho/t]e : [\rho/t]\tau} \tag{16.4b}$$

In addition, there are rules omitted here specifying that definitional equality is a congruence relation (that is, an equivalence relation respected by all expression-forming operations).

16.2.1 Products and Sums

The nullary product, or unit, type is definable in **F** as follows:

$$\begin{aligned} \texttt{unit} &\triangleq \forall(r.r \to r) \\ \langle\rangle &\triangleq \Lambda(r)\,\lambda\,(x : r)\,x \end{aligned}$$

The identity function plays the role of the null tuple, because it is the only closed value of this type.

Binary products are definable in **F** by using encoding tricks similar to those described in Chapter 21 for the untyped λ-calculus:

$$\begin{aligned} \tau_1 \times \tau_2 &\triangleq \forall(r.(\tau_1 \to \tau_2 \to r) \to r) \\ \langle e_1, e_2\rangle &\triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to \tau_2 \to r)\,x(e_1)(e_2) \\ e \cdot \texttt{l} &\triangleq e[\tau_1](\lambda\,(x : \tau_1)\,\lambda\,(y : \tau_2)\,x) \\ e \cdot \texttt{r} &\triangleq e[\tau_2](\lambda\,(x : \tau_1)\,\lambda\,(y : \tau_2)\,y) \end{aligned}$$

The statics given in Chapter 10 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in **F** from these definitions:

$$\langle e_1, e_2\rangle \cdot \texttt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle \cdot \texttt{r} \equiv e_2 : \tau_2.$$

The nullary sum, or void, type is definable in **F**:

$$\begin{aligned} \texttt{void} &\triangleq \forall(r.r) \\ \texttt{abort}\{\rho\}(e) &\triangleq e[\rho] \end{aligned}$$

Binary sums are also definable in **F**:

$$\begin{aligned} \tau_1 + \tau_2 &\triangleq \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r) \\ \texttt{l} \cdot e &\triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to r)\,\lambda\,(y : \tau_2 \to r)\,x(e) \\ \texttt{r} \cdot e &\triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to r)\,\lambda\,(y : \tau_2 \to r)\,y(e) \\ \texttt{case}\,e\,\{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} &\triangleq \\ e[\rho](\lambda\,(x_1 : \tau_1)\,e_1)(\lambda\,(x_2 : \tau_2)\,e_2) \end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in **F**:

$$\texttt{case}\,\texttt{l} \cdot d_1\,\{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\texttt{case}\,\texttt{r} \cdot d_2\,\{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

Thus, the dynamic behavior specified in Chapter 11 is correctly implemented by these definitions.

16.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in **F**. The key is the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}\{e_1 ; x.e_2\}(e_0) : \tau} .$$

Because the result type τ is arbitrary, this means that if we have an iterator, then we can use it to define a function of type

$$\mathtt{nat} \rightarrow \forall(t.t \rightarrow (t \rightarrow t) \rightarrow t).$$

This function, when applied to an argument n, yields a polymorphic function that, for any result type, t, given the initial result for $\mathtt{z}$ and a transformation from the result for x into the result for $\mathtt{s}(x)$, yields the result of iterating the transformation n times, starting with the initial result.

Because the *only* operation we can perform on a natural number is to iterate up to it, we may simply *identify* a natural number, n, with the polymorphic iterate-up-to-n function just described. Thus, we may define the type of natural numbers in **F** by the following equations:

$$\begin{aligned}
\mathtt{nat} &\triangleq \forall(t.t \rightarrow (t \rightarrow t) \rightarrow t) \\
\mathtt{z} &\triangleq \Lambda(t)\,\lambda\,(z : t)\,\lambda\,(s : t \rightarrow t)\,z \\
\mathtt{s}(e) &\triangleq \Lambda(t)\,\lambda\,(z : t)\,\lambda\,(s : t \rightarrow t)\,s(e[t](z)(s)) \\
\mathtt{iter}\{e_1 ; x.e_2\}(e_0) &\triangleq e_0[\tau](e_1)(\lambda\,(x : \tau)\,e_2)
\end{aligned}$$

It is easy to check that the statics and dynamics of the natural numbers type given in Chapter 9 are derivable in **F** under these definitions. The representations of the numerals in **F** are called the *polymorphic Church numerals*.

The encodability of the natural numbers shows that **F** is *at least as expressive* as **T**. But is it *more* expressive? Yes! It is possible to show that the evaluation function for **T** is definable in **F**, even though it is not definable in **T** itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for **F** is not definable in **F**. We may enrich **F** a bit more to define the evaluator for **F**, but as long as all programs in the enriched language terminate, we will once again have an undefinable function, the evaluation function for that extension.

16.3 Parametricity Overview

A remarkable property of **F** is that polymorphic types severely constrain the behavior of their elements. We may prove useful theorems about an expression knowing *only* its type—that is, without ever looking at the code. For example, if i is any expression of type $\forall(t.t \rightarrow t)$, then it is the identity function. Informally, when i is applied to a type, τ, and an argument of type τ, it returns a value of type τ. But because τ is not specified until i is called, the function has no choice but to return its argument, which is to say that it is essentially the identity function. Similarly, if b is any expression of type $\forall(t.t \to t \to t)$, then b is equivalent to either $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,x$ or $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,y$. Intuitively, when b is applied to two arguments of a given type, the only value it can return is one of the givens.

Properties of a program in **F** that can be proved knowing only its type are called *parametricity properties*. The facts about the functions i and b stated above are examples of parametricity properties. Such properties are sometimes called "free theorems," because they come from typing "for free," without any knowledge of the code itself. It bears repeating that in **F** we prove non-trivial behavioral properties of programs without ever examining the program text. The key to this incredible fact is that we are able to prove a deep property, called *parametricity*, about the language **F**, that then applies to every program written in **F**. One may say that the type system "pre-verifies" programs with respect to a broad range of useful properties, eliminating the need to prove those properties about every program separately. The parametricity theorem for **F** explains the remarkable experience that if a piece of code type checks, then it "just works." Parametricity narrows the space of well-typed programs sufficiently that the opportunities for programmer error are reduced to almost nothing.

So how does the parametricity theorem work? Without getting into too many technical details (but see Chapter 48 for a full treatment), we can give a brief summary of the main idea. Any function $i : \forall(t.t \to t)$ in **F** enjoys the following property:

> *For any type τ and any property $\mathcal{P}$ of the type τ, then if $\mathcal{P}$ holds of $x : \tau$, then $\mathcal{P}$ holds of $i[\tau](x)$.*

To show that for any type τ, and any x of type τ, the expression $i[\tau](x)$ is equivalent to x, it suffices to fix $x_0 : \tau$, and consider the property $\mathcal{P}_{x_0}$ that holds of $y : \tau$ iff y is equivalent to x_0. Obviously, $\mathcal{P}$ holds of x_0 itself, and hence by the above-displayed property of i, it sends any argument satisfying $\mathcal{P}_{x_0}$ to a result satisfying $\mathcal{P}_{x_0}$, which is to say that it sends x_0 to x_0. Because x_0 is an arbitrary element of τ, it follows that $i[\tau]$ is the identity function, $\lambda\,(x:\tau)\,x$, on the type τ, and because τ is itself arbitrary, i is the polymorphic identity function, $\Lambda(t)\,\lambda\,(x:t)\,x$.

A similar argument suffices to show that the function b, defined above, is either $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,x$ or $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,y$. By virtue of its type, the function b enjoys the parametricity property

> *For any type τ and any property $\mathcal{P}$ of τ, if $\mathcal{P}$ holds of $x : \tau$ and of $y : \tau$, then $\mathcal{P}$ holds of $b[\tau](x)(y)$.*

Choose an arbitrary type τ and two arbitrary elements x_0 and y_0 of type τ. Define $\mathcal{Q}_{x_0,y_0}$ to hold of $z : \tau$ iff either z is equivalent to x_0 or z is equivalent to y_0. Clearly $\mathcal{Q}_{x_0,y_0}$ holds of both x_0 and y_0 themselves, so by the quoted parametricity property of b, it follows that $\mathcal{Q}_{x_0,y_0}$ holds of $b[\tau](x_0)(y_0)$, which is to say that it is equivalent to either x_0 or y_0. Since τ, x_0, and y_0 are arbitrary, it follows that b is equivalent to either $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,x$ or $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,y$.

The parametricity theorem for **F** implies even stronger properties of functions such as i and b considered above. For example, the function i of type $\forall(t.t \to t)$ also satisfies the following condition:

> *If τ and τ' are any two types, and $\mathcal{R}$ is a binary relation between τ and τ', then for any $x : \tau$ and $x' : \tau'$, if $\mathcal{R}$ relates x to x', then $\mathcal{R}$ relates $i[\tau](x)$ to $i[\tau'](x')$.*

Using this property, we may again prove that i is equivalent to the polymorphic identity function. Specifically, if τ is any type and $g : \tau \to \tau$ is any function on that type, then it follows from the type of i alone that $i[\tau](g(x))$ is equivalent to $g(i[\tau](x))$ for any $x : \tau$. To prove this, simply choose $\mathcal{R}$ to the be graph of the function g, the relation $\mathcal{R}_g$ that holds of x and x' iff x' is equivalent to $g(x)$. The parametricity property of i, when specialized to $\mathcal{R}_g$, states that if x' is equivalent to $g(x)$, then $i[\tau](x')$ is equivalent to $g(i[\tau](x))$, which is to say that $i[\tau](g(x))$ is equivalent to $g(i[\tau](x))$. To show that i is equivalent to the identity function, choose $x_0 : \tau$ arbitrarily, and consider the constant function g_0 on τ that always returns x_0. Because x_0 is equivalent to $g_0(x_0)$, it follows that $i[\tau](x_0)$ is equivalent to x_0, which is to say that i behaves like the polymorphic identity function.

16.4 Notes

System F was introduced by Girard (1972) in the context of proof theory and by Reynolds (1974) in the context of programming languages. The concept of parametricity was originally isolated by Strachey but was not fully developed until the work of Reynolds (1983). The phrase “free theorems” for parametricity theorems was introduced by Wadler (1989).

Exercises

16.1. Give polymorphic definitions and types to the `s` and `k` combinators defined in Exercise **3.1**.

16.2. Define in **F** the type `bool` of *Church booleans*. Define the type `bool`, and define `true` and `false` of this type, and the conditional `if` e `then` e_0 `else` e_1, where e is of this type.

16.3. Define in **F** the inductive type of lists of natural numbers as defined in Chapter 15. *Hint*: Define the representation in terms of the recursor (elimination form) for lists, following the pattern for defining the type of natural numbers.

16.4. Define in **F** an arbitrary inductive type, $\mu(t.\tau)$. *Hint*: generalize your answer to Exercise **16.3**.

16.5. Define the type t `list` as in Exercise **16.3**, with the element type, t, unspecified. Define the finite set of *elements* of a list l to be those x given by the head of some number of tails of l. Now suppose that $f : \forall(t.t\,\texttt{list} \to t\,\texttt{list})$ is an arbitrary function of the stated type. Show that the elements of $f[\tau](l)$ are a subset of those of l. Thus, f may only permute, replicate, or drop elements from its input list to obtain its output list.

Note

1 Girard's original version of System F included the natural numbers as a basic type.