Part VIII

Variable Types

Chapter 23

Girard's System F

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 $\mathcal{L}\{\mathtt{nat} \to\}$ 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 exhibit the same behavior when executed.

Obviously it would be useful to capture the general 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 said to be *polymorphic*. In this chapter we will study a language 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 24), 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.)

23.1 System F

System F, or the *polymorphic λ-calculus*, or $\mathcal{L}\{\rightarrow\forall\}$, is a minimal functional language that illustrates the core concepts of polymorphic typing, and permits us to examine its surprising expressive power in isolation from other language features. The syntax of System **F** is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	t	t
		|	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$
		|	$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$
Expr	e	::=	x	x
		|	$\mathtt{lam}[\tau](x.e)$	$\lambda(x{:}\tau.e)$
		|	$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$
		|	$\mathtt{Lam}(t.e)$	$\Lambda(t.e)$
		|	$\mathtt{App}[\tau](e)$	$e[\tau]$

The meta-variable t ranges over a class of *type variables*, and x ranges over a class of *expression variables*. The *type abstraction*, $\mathtt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* t standing for an unspecified type within e. The *type application*, or *instantiation*, $\mathtt{App}[\tau](e)$, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type*, $\mathtt{all}(t.\tau)$, that determines the type, τ, of the result as a function of the argument, t.

The static semantics of $\mathcal{L}\{\rightarrow\forall\}$ consists of two judgement forms, the *type formation* judgement,

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

and the *typing judgement*,

$$\mathcal{T}\ \mathcal{X} \mid \Delta\ \Gamma \vdash e : \tau.$$

These are generic judgements over the parameter set $\mathcal{T}$ of *type variables* and the parameter set $\mathcal{X}$ of *expression variables*. They are also hypothetical in a set Δ of *type assumptions* of the form $t\ \mathsf{type}$, where $t \in \mathcal{T}$, and *typing assumptions* of the form $x : \tau$, where $x \in \mathcal{T}$ and $\Delta \vdash \tau\ \mathsf{type}$. As usual we drop explicit mention of the parameter sets, relying on typographical conventions to determine them.

The rules defining the type formation judgement are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{23.1a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ type}} \tag{23.1b}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \mathtt{all}(t.\tau) \text{ type}} \tag{23.1c}$$

The rules defining the typing judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{23.2a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{23.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{23.2c}$$

$$\frac{\Delta, t \text{ type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \text{ type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{23.2e}$$

Lemma 23.1 (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 (23.2). □

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

Lemma 23.2 (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, writing $C[\tau_1][\tau_2][\tau_3]$. Each such instance has the type

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

Dynamic Semantics

The dynamic semantics of $\mathcal{L}\{\to\forall\}$ is given as follows:

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{23.3a}$$

$$\frac{}{\mathtt{Lam}(t.e)\ \mathsf{val}} \tag{23.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau_1](x.e);e_2) \mapsto [e_2/x]e} \tag{23.3c}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{23.3d}$$

$$\frac{}{\mathtt{App}[\tau](\mathtt{Lam}(t.e)) \mapsto [\tau/t]e} \tag{23.3e}$$

$$\frac{e \mapsto e'}{\mathtt{App}[\tau](e) \mapsto \mathtt{App}[\tau](e')} \tag{23.3f}$$

These rules endow $\mathcal{L}\{\to\forall\}$ with a call-by-name interpretation of application, but one could as well consider a call-by-value variant.

It is a simple matter to prove safety for $\mathcal{L}\{\to\forall\}$, using familiar methods.

Lemma 23.3 (Canonical Forms)**.** *Suppose that $e : \tau$ and e val, then*

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

Proof. By rule induction on the static semantics. □

Theorem 23.4 (Preservation)**.** *If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.*

Proof. By rule induction on the dynamic semantics. □

Theorem 23.5 (Progress)**.** *If $e : \sigma$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on the static semantics. □

23.2 Polymorphic Definability

The language $\mathcal{L}\{\rightarrow\forall\}$ is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types, including both the eager and the lazy natural numbers! This is most naturally expressed using definitional equivalence, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash \lambda(x{:}\tau.\, e_2)(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{23.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \sigma\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.e)[\sigma] \equiv [\sigma/t]e : [\sigma/t]\tau} \tag{23.4b}$$

The remaining rules specify that definitional equivalence is reflexive, symmetric, and transitive, and that it is compatible with both forms of application and abstraction.

23.2.1 Products and Sums

The nullary product, or unit, type is definable in $\mathcal{L}\{\rightarrow\forall\}$ as follows:

$$\begin{aligned}\mathtt{unit} &= \forall(r.r \rightarrow r)\\ \langle\rangle &= \Lambda(r.\lambda(x{:}r.x))\end{aligned}$$

It is easy to check that the static semantics given in Chapter 16 is derivable. There being no elimination rule, there is no requirement on the dynamic semantics.

Binary products are definable in $\mathcal{L}\{\rightarrow\forall\}$ by using encoding tricks similar to those described in Chapter 21 for the untyped λ-calculus:

$$\begin{aligned}\tau_1 \times \tau_2 &= \forall(r.(\tau_1 \rightarrow \tau_2 \rightarrow r) \rightarrow r)\\ \langle e_1, e_2\rangle &= \Lambda(r.\lambda(x{:}\tau_1 \rightarrow \tau_2 \rightarrow r.\,x(e_1)(e_2)))\\ \mathtt{pr_l}(e) &= e[\tau_1](\lambda(x{:}\tau_1.\lambda(y{:}\tau_2.x)))\\ \mathtt{pr_r}(e) &= e[\tau_2](\lambda(x{:}\tau_1.\lambda(y{:}\tau_2.y)))\end{aligned}$$

The static semantics given in Chapter 16 is derivable according to these definitions. Moreover, the following definitional equivalences are derivable in $\mathcal{L}\{\rightarrow\forall\}$ from these definitions:

$$\mathtt{pr_l}(\langle e_1, e_2\rangle) \equiv e_1 : \tau_1$$

and

$$\mathtt{pr_r}(\langle e_1, e_2\rangle) \equiv e_2 : \tau_2.$$

The nullary sum, or void, type is definable in $\mathcal{L}\{\rightarrow\forall\}$:

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

There is no definitional equivalence to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned}\tau_1 + \tau_2 &= \forall(r.(\tau_1 \rightarrow r) \rightarrow (\tau_2 \rightarrow r) \rightarrow r)\\ \mathtt{in[l]}(e) &= \Lambda(r.\lambda(x{:}\tau_1 \rightarrow r.\lambda(y{:}\tau_2 \rightarrow r.\,x(e))))\\ \mathtt{in[r]}(e) &= \Lambda(r.\lambda(x{:}\tau_1 \rightarrow r.\lambda(y{:}\tau_2 \rightarrow r.\,y(e))))\\ \mathtt{case}\,e\,\{\mathtt{in[l]}(x_1) \Rightarrow e_1 \mid \mathtt{in[r]}(x_2) \Rightarrow e_2\} &=\\ e[\rho](\lambda(x_1{:}\tau_1.e_1))(\lambda(x_2{:}\tau_2.e_2))&\end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in $\mathcal{L}\{\to\forall\}$:

$$\texttt{case}\ \texttt{in[l]}(d_1)\ \{\texttt{in[l]}(x_1) \Rightarrow e_1 \mid \texttt{in[r]}(x_2) \Rightarrow e_2\} \equiv [e/x_1]e_1 : \rho$$

and

$$\texttt{case}\ \texttt{in[r]}(d_2)\ \{\texttt{in[l]}(x_1) \Rightarrow e_1 \mid \texttt{in[r]}(x_2) \Rightarrow e_2\} \equiv [e/x_2]e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 17 is correctly implemented by these definitions.

23.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\to\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

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

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

$$\texttt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument n, yields a polymorphic function that, for any result type, t, if given the initial result for $\texttt{z}$, and if given a function transforming the result for x into the result for $\texttt{s}(x)$, then it returns the result of iterating the transformer n times starting with the initial result.

Since the *only* operation we can perform on a natural number is to iterate up to it in this manner, we may simply *identify* a natural number, n, with the polymorphic iterate-up-to-n function just described. This means that we may define the type of natural numbers in $\mathcal{L}\{\to\forall\}$ by the following equations:

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

It is a straightforward exercise to check that the static and dynamic semantics given in Chapter 14 is derivable in $\mathcal{L}\{\to\forall\}$ under these definitions.

This shows that $\mathcal{L}\{\to\forall\}$ is *at least as expressive* as $\mathcal{L}\{\mathtt{nat}\to\}$. But is it *more* expressive? Yes! It is possible to show that the evaluation function for $\mathcal{L}\{\mathtt{nat}\to\}$ is definable in $\mathcal{L}\{\to\forall\}$, even though it is not definable in $\mathcal{L}\{\mathtt{nat}\to\}$ itself. However, the same diagonal argument given in Chapter 14 applies here, showing that the evaluation function for $\mathcal{L}\{\to\forall\}$ is not definable in $\mathcal{L}\{\to\forall\}$. We may enrich $\mathcal{L}\{\to\forall\}$ a bit more to define the evaluator for $\mathcal{L}\{\to\forall\}$, but as long as the enriched language is itself total, we will once again have an undefinable function, the evaluation function for that extension! The extension process will never close as long as the language remains total.

23.3 Parametricity

A remarkable property of polymorphic typing is that it strongly constrains the behavior of an expression of that type. For example, if i is *any* expression of type $\forall(t.t\to t)$, then it must behave like the identity function in the following sense. For an arbitrary type τ and an arbitrary expression $e:\tau$, it must be that $i[\tau](e)\equiv e$. The informal reason is that i, being polymorphic, must, when applied to an arbitrary argument of arbitrary type must return a result of that type. Since not even the type, much less the value, of the argument is known in advance, the function i has no choice but to return the argument as result if it is to achieve the specified typing. Similarly, if c is *any* expression of type $\forall(t.t\to t\to t)$, then for any type τ and any $e_1:\tau$ and $e_2:\tau$, it must be that either $c(e_1)(e_2)\equiv e_1$ or $c(e_1)(e_2)\equiv e_2$.

A rigorous justification of these claims is deferred to Chapter 52. Meanwhile we content ourselves with a brief summary of the argument developed there. The crucial idea is that types may be interpreted as relations, and we may prove that every well-typed expression of $\mathcal{L}\{\to\forall\}$ preserves any such relational interpretation. This is best explained by example. The upshot of Theorem 52.8 on page 472, specialized to the type $i:\forall(t.t\to t)$, is that for any type τ, any predicate P on expressions of type τ, and any $e:\tau$, if $P(e)$, then $P(i(e))$. Fix τ and $e:\tau$, and define $P(x)$ to hold iff $x\equiv e$. By Theorem 52.8 on page 472 we have that for any $e':\tau$, if $e'\equiv e$, then $i(e')\equiv e$. Noting that definitional equivalence is reflexive, it follows that $i(e)\equiv e$. Similarly, if $c:\forall(t.t\to t\to t)$, then, fixing τ, $e_1:\tau$, and $e_2:\tau$, we may define $P(e)$ to hold iff either $e\equiv e_1$ or $e\equiv e_2$. It follows from Theorem 52.8 on page 472 that either $c(e_1)(e_2)\equiv e_1$ or $c(e_1)(e_2)\equiv e_2$.

The important point here is that the properties of i and c are derived without knowing anything about these expressions themselves beyond their types. That is, based solely on the types of these expressions we are able to derive theorems about their behavior without ever having seen the code for either of them! Such theorems are sometimes called *free theorems* because they come "for free" as a consequence of typing, and require no program analysis or verification to derive (beyond the once-and-for-all proof of Theorem 52.8 on page 472). Free theorems such as those illustrated above underly the experience that in a polymorphic language, well-typed programs tend to behave as expected no further debugging or analysis required. Parametricity so constrains the behavior of a program that it is relatively easy to ensure that the code works just by checking its type. Free theorems also underly the principal of representation independence for abstract types, which is discussed further in Chapter 24.

23.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of $\mathcal{L}\{\rightarrow\forall\}$. These are obtained in one of two ways:

1. Restricting type quantification to unquantified types.
2. Restricting the occurrence of quantifiers within types.

23.4.1 Predicative Fragment

The remarkable expressive power of the language $\mathcal{L}\{\rightarrow\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let τ be the type $\forall(t.t \rightarrow t)$, and, assuming that $e : \tau$, we may apply e to its own type, obtaining the expression $e[\tau]$ of type $\tau \rightarrow \tau$. Written out in full, this is the type

$$\forall(t.t \rightarrow t) \rightarrow \forall(t.t \rightarrow t),$$

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of e itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to e again, obtaining the expression $e[\tau](e)$, which is again of type τ — the very type of e!

This property of $\mathcal{L}\{\to\forall\}$ is called *impredicativity*[1]; the language $\mathcal{L}\{\to\forall\}$ is said to permit *impredicative (type) quantification*. The distinguishing characteristic of impredicative polymorphism is that it involves a kind of circularity in that the meaning of a quantified type is given in terms of its instances, including the quantified type itself. This quasi-circularity is responsible for the surprising expressive power of $\mathcal{L}\{\to\forall\}$, and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of $\mathcal{L}\{\to\forall\}$ terminate).

Contrast this with $\mathcal{L}\{\to\}$, in which the type of an application of a function is evidently smaller than the type of the function itself. For if $e : \tau_1 \to \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of e. This situation extends to polymorphism, provided that we impose the restriction that a quantified type can only be instantiated by an un-quantified type. For in that case passage from $\forall(t.\tau)$ to $[\sigma/t]\tau$ decreases the number of quantifiers (even if the size of the type expression viewed as a tree grows). For example, the type $\forall(t.t \to t)$ may be instantiated with the type $u \to u$ to obtain the type $(u \to u) \to (u \to u)$. This type has more symbols in it than τ, but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called *predicative*[2] *polymorphism*. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the natural numbers are no longe definable in it.

The formalization of $\mathcal{L}\{\to\forall_{\mathsf{p}}\}$ is left to Chapter 25, where the appropriate technical machinery is available.

23.4.2 Prenex Fragment

A rather more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is often imposed for the sake of type inference, which permits type annotations to be omitted entirely in the knowledge that they can be recovered from the way the expression is used. We will not discuss type inference here, but we will give a formulation of the prenex fragment of $\mathcal{L}\{\to\forall\}$, because it plays an important role in the design of practical polymorphic languages.

[1] pronounced *im-PRED-ic-a-tiv-it-y*
[2] pronounced *PRED-i-ca-tive*

The prenex fragment of $\mathcal{L}\{\rightarrow\forall\}$ is designated $\mathcal{L}^1\{\rightarrow\forall\}$, for reasons that will become clear in the next subsection. It is defined by *stratifying* types into two classes, the *monotypes* (or *rank*-0 types) and the *polytypes* (or *rank*-1 types). The monotypes are those that do not involve any quantification, and may be used to instantiate the polymorphic quantifier. The polytypes include the monotypes, but also permit quantification over monotypes. These classifications are expressed by the judgements $\Delta \vdash \tau$ mono and $\Delta \vdash \tau$ poly, where Δ is a finite set of hypotheses of the form t mono, where t is a type variable not otherwise declared in Δ. The rules for deriving these judgements are as follows:

$$\frac{}{\Delta, t \text{ mono} \vdash t \text{ mono}} \tag{23.5a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta \vdash \tau_2 \text{ mono}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ mono}} \tag{23.5b}$$

$$\frac{\Delta \vdash \tau \text{ mono}}{\Delta \vdash \tau \text{ poly}} \tag{23.5c}$$

$$\frac{\Delta, t \text{ mono} \vdash \tau \text{ poly}}{\Delta \vdash \mathtt{all}(t.\tau) \text{ poly}} \tag{23.5d}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The static semantics of $\mathcal{L}^1\{\rightarrow\forall\}$ is given by rules for deriving hypothetical judgements of the form $\Delta\ \Gamma \vdash e : \sigma$, where Δ consists of hypotheses of the form t mono, and Γ consists of hypotheses of the form $x : \sigma$, where $\Delta \vdash \sigma$ poly. The rules defining this judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{23.6a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam[\tau_1](x.e_2)} : \mathtt{arr}(\tau_1;\tau_2)} \tag{23.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{23.6c}$$

$$\frac{\Delta, t \text{ mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.6d}$$

$$\frac{\Delta \vdash \tau \text{ mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App[\tau](e)} : [\tau/t]\tau'} \tag{23.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgements have the form $e : \sigma$ for some expression e and polytype σ.

The restriction on the domain of a λ-abstraction to be a monotype means that a fully general `let` construct is no longer definable—there is no means of binding an expression of polymorphic type to a variable. For this reason it is usual to augment $\mathcal{L}\{\rightarrow\forall_{\mathsf{p}}\}$ with a primitive `let` construct whose static semantics is as follows:

$$\frac{\Delta \vdash \tau_1 \ \mathsf{poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}\,[\tau_1]\,(e_1; x.e_2) : \tau_2}\ . \tag{23.7}$$

For example, the expression

$$\mathtt{let}\ I : \forall(t.t \rightarrow t)\ \mathtt{be}\ \Lambda(t.\lambda(x:t.x))\ \mathtt{in}\ I[\tau \rightarrow \tau](I[\tau])$$

has type $\tau \rightarrow \tau$ for any polytype τ.

23.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathcal{L}^1\{\rightarrow\forall\}$ may be generalized to form a hierarchy of languages in which the occurrences of polymorphic types are restricted in relation to function types. The key feature of the prenex fragment is that quantified types are not permitted to occur in the domain of a function type. The prenex fragment also prohibits polymorphic types from the range of a function type, but it would be harmless to admit it, there being no significant difference between the type $\sigma \rightarrow \forall(t.\tau)$ and the type $\forall(t.\sigma \rightarrow \tau)$ (where $t \mathbin{\#} \sigma$). This motivates the definition of a hierarchy of fragments of $\mathcal{L}\{\rightarrow\forall\}$ that subsumes the prenex fragment as a special case.

We will define a judgement of the form $\tau\ \mathsf{type}\,[k]$, where $k \geq 0$, to mean that τ is a type of *rank* k. Informally, types of rank 0 have no quantification, and types of rank $k+1$ may involve quantification, but the domains of function types are restricted to be of rank k. Thus, in the terminology of Section 23.4.2 on page 212, a monotype is a type of rank 0 and a polytype is a type of rank 1.

The definition of the types of rank k is defined simultaneously for all k by the following rules. These rules involve hypothetical judgements of the form $\Delta \vdash \tau\ \mathsf{type}\,[k]$, where Δ is a finite set of hypotheses of the form $t_i\ \mathsf{type}\,[k_i]$ for some pairwise distinct set of type variables t_i. The rules defining these judgements are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\,[k] \vdash t\ \mathsf{type}\,[k]} \tag{23.8a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[0] \quad \Delta \vdash \tau_2 \ \mathsf{type}\,[0]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}\,[0]} \tag{23.8b}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[k] \quad \Delta \vdash \tau_2 \ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}\,[k+1]} \tag{23.8c}$$

$$\frac{\Delta \vdash \tau \ \mathsf{type}\,[k]}{\Delta \vdash \tau \ \mathsf{type}\,[k+1]} \tag{23.8d}$$

$$\frac{\Delta, t \ \mathsf{type}\,[k] \vdash \tau \ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{type}\,[k+1]} \tag{23.8e}$$

With these restrictions in mind, it is a good exercise to define the static semantics of $\mathcal{L}^k\{\to\forall\}$, the restriction of $\mathcal{L}\{\to\forall\}$ to types of rank k (or less). It is most convenient to consider judgements of the form $e : \tau\,[k]$ specifying simultaneously that $e : \tau$ and $\tau \ \mathsf{type}\,[k]$. For example, the rank-limited rules for λ-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}(x.e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[0]} \tag{23.9a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}(x.e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[k+1]} \tag{23.9b}$$

The remaining rules follow a similar pattern.

The rank-limited languages $\mathcal{L}^k\{\to\forall\}$ clarifies the requirement for a primitive `let` construct in $\mathcal{L}^1\{\to\forall\}$. The prenex fragment of $\mathcal{L}\{\to\forall\}$ corresponds to the rank-one fragment $\mathcal{L}^1\{\to\forall\}$. The `let` construct for rank-one types is definable in $\mathcal{L}^2\{\to\forall\}$ from λ-abstraction and application. This definition only makes sense at rank two, since it abstracts over a rank-one polymorphic type.

23.5 Exercises

1. Show that primitive recursion is definable in $\mathcal{L}\{\to\forall\}$ by exploiting the definability of iteration and binary products.

2. Investigate the representation of eager products and sums in eager and lazy variants of $\mathcal{L}\{\to\forall\}$.

3. Show how to write an interpreter for $\mathcal{L}\{\mathtt{nat}\to\}$ in $\mathcal{L}\{\to\forall\}$.

Chapter 24

Abstract Types

Data abstraction is perhaps the most important technique for structuring programs. The main idea is to introduce an *interface* that serves as a contract between the *client* and the *implementor* of an abstract type. The interface specifies what the client may rely on for its own work, and, simultaneously, what the implementor must provide to satisfy the contract. The interface serves to isolate the client from the implementor so that each may be developed in isolation from the other. In particular one implementation may be replaced by another without affecting the behavior of the client, provided that the two implementations meet the same interface and are, in a sense to be made precise below, suitably related to one another. (Roughly, each simulates the other with respect to the operations in the interface.) This property is called *representation independence* for an abstract type.

Data abstraction may be formalized by extending the language $\mathcal{L}\{\rightarrow\forall\}$ with *existential types*. Interfaces are modelled as existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are modelled as packages, the introductory form for existentials, and clients are modelled as uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is modelled so naturally and directly by the logical concept of existential type quantification. Existential types are closely connected with universal types, and hence are often treated together. The superficial reason is that both are forms of type quantification, and hence both require the machinery of type variables. The deeper reason is that existentials are *definable* from universals — surprisingly, data abstraction is actually just a form of polymorphism! One consequence of this observation is that representation independence is just a use of the parametricity properties of polymorphic

functions discussed in Chapter 23.

24.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

Category	*Item*		*Abstract*	*Concrete*
Types	τ	::=	$\mathtt{some}(t.\tau)$	$\exists(t.\tau)$
Expr	e	::=	$\mathtt{pack}[t.\tau]\,[\rho]\,(e)$	$\mathtt{pack}\,\rho\,\mathtt{with}\,e\,\mathtt{as}\,\exists(t.\tau)$
		|	$\mathtt{open}[t.\tau]\,[\rho]\,(e_1; t, x.e_2)$	$\mathtt{open}\,e_1\,\mathtt{as}\,t\,\mathtt{with}\,x\!:\!\tau\,\mathtt{in}\,e_2$

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a *package* of the form $\mathtt{pack}\,\rho\,\mathtt{with}\,e\,\mathtt{as}\,\exists(t.\tau)$, where ρ is a type and e is an expression of type $[\rho/t]\tau$. The type ρ is called the *representation type* of the package, and the expression e is called the *implementation* of the package. The eliminatory form for existentials is the expression $\mathtt{open}\,e_1\,\mathtt{as}\,t\,\mathtt{with}\,x\!:\!\tau\,\mathtt{in}\,e_2$, which *opens* the package e_1 for use within the *client* e_2 by binding its representation type to t and its implementation to x for use within e_2. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the `open` construct specifies that the type variable, t, and the expression variable, x, are bound within the client. They may be renamed at will by α-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, t, may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to α-equivalence, and is not particularly tied to data abstraction.

24.1.1 Static Semantics

The static semantics of existential types is specified by rules defining when an existential is well-formed, and by giving typing rules for the associated introductory and eliminatory forms.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{some}(t.\tau)\ \mathsf{type}} \tag{24.1a}$$

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \mathtt{pack}[t.\tau]\,[\rho]\,(e) : \mathtt{some}(t.\tau)} \tag{24.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{some}(t.\tau) \quad \Delta, t \text{ type } \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta\ \Gamma \vdash \mathtt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) : \tau_2} \tag{24.1c}$$

Rule (24.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, τ_2, must not involve the abstract type t. This restriction prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.
2. The body of the client, e_2, is type checked without knowledge of the representation type, t. The client is, in effect, polymorphic in the type variable t.

Lemma 24.1 (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$*. If* $\Delta \vdash \tau_i$ *type for each* $x_i : \tau_i$ *in* Γ*, then* $\Delta \vdash \tau$ *type.*

Proof. By induction on Rules (24.1). □

24.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified as follows:

$$\frac{\{e \text{ val}\}}{\mathtt{pack}[t.\tau]\,[\rho]\,(e) \text{ val}} \tag{24.2a}$$

$$\left\{ \frac{e \mapsto e'}{\mathtt{pack}[t.\tau]\,[\rho]\,(e) \mapsto \mathtt{pack}[t.\tau]\,[\rho]\,(e')} \right\} \tag{24.2b}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{open}[t.\tau]\,[\tau_2]\,(e_1; t, x.e_2) \mapsto \mathtt{open}[t.\tau]\,[\tau_2]\,(e_1'; t, x.e_2)} \tag{24.2c}$$

$$\frac{\{e \text{ val}\}}{\mathtt{open}[t.\tau]\,[\tau_2]\,(\mathtt{pack}[t.\tau]\,[\rho]\,(e); t, x.e_2) \mapsto [\rho, e/t, x]e_2} \tag{24.2d}$$

These rules endow $\mathcal{L}\{\rightarrow\forall\exists\}$ with a lazy semantics for packages. More importantly, these rules specify that *there are no abstract types at run time*! The representation type is exposed to the client by substitution when the package is opened. In other words, data abstraction is a *compile-time discipline* that leaves no traces of its presence at execution time.

24.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\rightarrow\forall\}$ to the new constructs.

Theorem 24.2 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on $e \mapsto e'$, making use of substitution for both expression- and type variables. □

Lemma 24.3 (Canonical Forms)**.** *If $e : \texttt{some}(t.\tau)$ and e val, then $e = \texttt{pack}[t.\tau][\rho](e')$ for some type ρ and some e' val such that $e' : [\rho/t]\tau$.*

Proof. By rule induction on the static semantics, making use of the definition of closed values. □

Theorem 24.4 (Progress)**.** *If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on $e : \tau$, making use of the canonical forms lemma. □

24.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of queues of natural numbers supporting three operations:

1. Formation of the empty queue.
2. Inserting an element at the tail of the queue.
3. Remove the head of the queue.

This is clearly a bare-bones interface, but is sufficient to illustrate the main ideas of data abstraction. Queue elements may be taken to be of any type, τ, of our choosing; we will not be specific about this choice, since nothing depends on it.

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. This is captured

by the following existential type, $\exists(t.\tau)$, which serves as the interface of the queue abstraction:

$$\exists(t.\langle \texttt{emp}:t, \texttt{ins}:\texttt{nat}\times t\to t, \texttt{rem}:t\to\texttt{nat}\times t\rangle).$$

The representation type, t, of queues is *abstract* — all that is specified about it is that it supports the operations `emp`, `ins`, and `rem`, with the specified types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations. Here is a very simple implementation, e_l, in which queues are represented as lists:

$$\texttt{pack}\,\texttt{list}\,\texttt{with}\,\langle \texttt{emp}=\texttt{nil}, \texttt{ins}=e_i, \texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\tau),$$

where

$$e_i : \texttt{nat}\times\texttt{list}\to\texttt{list} = \lambda(x{:}\texttt{nat}\times\texttt{list}.\, e_i'),$$

and

$$e_r : \texttt{list}\to\texttt{nat}\times\texttt{list} = \lambda(x{:}\texttt{list}.\, e_r').$$

Here the expression e_i' conses the first component of x, the element, onto the second component of x, the queue. Correspondingly, the expression e_r' reverses its argument, and returns the head element paired with the reversal of the tail. These operations "know" that queues are represented as values of type `list`, and are programmed accordingly.

It is also possible to give another implementation, e_p, of the same interface, $\exists(t.\tau)$, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half". This representation avoids the need for reversals on each call, and, as a result, achieves amortized constant-time behavior:

$$\texttt{pack}\,\texttt{list}\times\texttt{list}\,\texttt{with}\,\langle \texttt{emp}=\langle\texttt{nil},\texttt{nil}\rangle, \texttt{ins}=e_i, \texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\tau).$$

In this case e_i has type

$$\texttt{nat}\times(\texttt{list}\times\texttt{list})\to(\texttt{list}\times\texttt{list}),$$

and e_r has type

$$(\texttt{list}\times\texttt{list})\to\texttt{nat}\times(\texttt{list}\times\texttt{list}).$$

These operations "know" that queues are represented as values of type $\texttt{list} \times \texttt{list}$, and are implemented accordingly.

The important point is that the *same* client type checks regardless of which implementation of queues we choose. This is because the representation type is hidden, or *held abstract*, from the client during type checking. Consequently, it cannot rely on whether it is `list` or $\texttt{list} \times \texttt{list}$ or some other type. That is, the client is *independent* of the representation of the abstract type.

24.3 Definability of Existentials

It turns out that it is not necessary to extend $\mathcal{L}\{\rightarrow\forall\}$ with existential types to model data abstraction, because they are already definable using only universal types! Before giving the details, let us consider why this should be possible. The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\texttt{open}\, e\, \texttt{as}\, t\, \texttt{with}\, x{:}\tau\, \texttt{in}\, e' : \tau',$$

where $e : \exists(t.\tau)$, specifies that $e' : \tau'$ under the assumptions t type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau \rightarrow \tau'),$$

where t may occur in τ (the type of the operations), but not in τ' (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned}
\exists(t.\sigma) &= \forall(t'.\forall(t.\sigma \rightarrow t') \rightarrow t') \\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\sigma) &= \Lambda(t'.\lambda(x{:}\forall(t.\sigma \rightarrow t').x[\rho](e))) \\
\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e' &= e[\tau'](\Lambda(t.\lambda(x{:}\sigma.e')))
\end{aligned}$$

An existential is encoded as a polymorphic function taking the overall result type, t', as argument, followed by a polymorphic function representing the client with result type t', and yielding a value of type t' as overall result. Consequently, the `open` construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, τ, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, τ, of the `open` construct.) Finally, a package consisting of a representation type ρ and an implementation e is a

polymorphic function that, when given the result type, t, and the client, x, instantiates x with ρ and passes to it the implementation e.

It is then a straightforward exercise to show that this translation correctly reflects the static and dynamic semantics of existential types.

24.4 Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types. More precisely, there is a criterion, called *bisimilarity*, for relating two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation by another that is bisimilar to it. This leads to a simple methodology for proving the correctness of *candidate* implementation of an abstract type, which is to show that it is bisimilar to an obviously correct *reference* implementation of it. Since the candidate and the reference implementations are bisimilar, no client may distinguish them from one another, and hence if the client behaves properly with the reference implementation, then it must also behave properly with the candidate.

To derive the definition of bisimilarity of implementations, it is helpful to examine the definition of existentials in terms of universals given in Section 24.3 on the facing page. It is an immediate consequence of the definition that the client of an abstract type is polymorphic in the representation of the abstract type. A client, c, of an abstract type $\exists(t.\sigma)$ has type $\forall(t.(\sigma \to \tau) \to \tau)$, where $t \# \tau$ (but t may, of course, occur in σ). Applying the parametricity property described informally in Chapter 23 (and developed rigorously in Chapter 52), this says that if R is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on both of them. The fact that t does not occur in the result type ensures that the behavior of the client is independent of the choice of relation between the implementations, provided that this relation is preserved by the operation that implement it.

To see what this means requires that we specify what is meant by a bisimulation. This is best done by example. So suppose that σ is the type

$$\langle \mathtt{emp}:t, \mathtt{ins}:\tau \times t \to t, \mathtt{rem}:t \to \tau \times t \rangle.$$

Theorem 52.8 on page 472 ensures that if ρ and ρ' are any two closed types, R is a relation between expressions of these two types, then if any the implementations $e : [\rho/x]\sigma$ and $e' : [\rho'/x]\sigma$ respect R, then $c\,[\rho]\,e$ behaves the

same as $c\,[\rho]\,e'$. It remains to define when two implementations respect the relation R. Let

$$e = \langle \mathtt{emp} = e_{\mathtt{m}}, \mathtt{ins} = e_{\mathtt{i}}, \mathtt{rem} = e_{\mathtt{r}} \rangle$$

and

$$e' = \langle \mathtt{emp} = e'_{\mathtt{m}}, \mathtt{ins} = e'_{\mathtt{i}}, \mathtt{rem} = e'_{\mathtt{r}} \rangle.$$

For these implementations to respect R means that the following three conditions hold:

1. The empty queues are related: $R(e_{\mathtt{m}}, e'_{\mathtt{m}})$.

2. Inserting the same element on each of two related queues yields related queues: if $d : \tau$ and $R(q, q')$, then $R(e_{\mathtt{i}}\,(d)\,(q), e_{\mathtt{i}}\,(d)\,(q'))$.

3. If two queues are related, their front elements are the same and their back elements are related: if $R(q, q')$, $e_{\mathtt{r}}\,(q) \equiv \langle d, r \rangle$, $e_{\mathtt{r}}\,(q) \equiv \langle d', r' \rangle$, then d is d' and $R(r, r')$.

If such a relation R exists, then the implementations e and e' are said to be *bisimilar*. The terminology stems from the requirement that the operations of the abstract type preserve the relation: if it holds before an operation is performed, then it must also hold afterwards, and the relation must hold for the initial state of the queue. Thus each implementation *simulates* the other up to the relationship specified by R.

To see how this works in practice, let us consider informally two implementations of the abstract type of queues specified above. For the reference implementation we choose ρ to be the type `list`, and define the empty queue to be the empty list, insert to add the specified element to the front of the list, and remove to remove the last element of the list. (A remove therefore takes time linear in the length of the list.) For the candidate implementation we choose ρ' to be the type $\mathtt{list} \times \mathtt{list}$ consisting of two lists, $\langle b, f \rangle$, where b represents the "back" of the queue, and f represents the "front" of the queue represented in reverse order of insertion. The empty queue consists of two empty lists. To insert d onto $\langle b, f \rangle$, we simply return $\langle \mathtt{cons}(d; b), f \rangle$, placing it on the "back" of the queue as expected. To remove an element from $\langle b, f \rangle$ breaks into two cases. If the front, f, of the queue is non-empty, say $\mathtt{cons}(d; f')$, then return $\langle d, \langle b, f' \rangle \rangle$ consisting of the front element and the queue with that element removed. If, on the other hand, f is empty, then we must move elements from the "back" to the "front" by reversing b and re-performing the remove operation on $\langle \mathtt{nil}, \mathtt{rev}(b) \rangle$, where `rev` is the obvious list reversal function.

To show that the candidate implementation is correct, we show that it is bisimilar to the reference implementation. This reduces to specifying a relation, R, between the types `list` and `list` $\times$ `list` such that the three simulation conditions given above are satisfied by the two implementations just described. The relation in question states that $R(l, \langle b, f \rangle)$ iff the list l is the list `app`(b)`(rev`(f)`)`, where `app` is the evident append function on lists. That is, thinking of l as the reference representation of the queue, the candidate must maintain that the elements of b followed by the elements of f in reverse order form precisely the list l. It is easy to check that the implementations just described preserve this relation. Having done so, we are assured that the client, c, behaves the same regardless of whether we use the reference or the candidate. Since the reference implementation is obviously correct (albeit inefficient), the candidate must also be correct in that the behavior of any client is unaffected by using it instead of the reference.

24.5 Exercises

Chapter 25

Constructors and Kinds

Types such as $\tau_1 \to \tau_2$ or τ `list` may be thought of as being built from other types by the application of a *type constructor*, or *type operator*. These two examples differ from each other in that the function space type constructor takes two arguments, whereas the list type constructor takes only one. We may, for the sake of uniformity, think of types such as `nat` as being built by a type constructor of *no* arguments. More subtly, we may even think of the types $\forall(t.\tau)$ and $\exists(t.\tau)$ as being built up in the same way by regarding the quantifiers as *higher-order* type operator.

These seemingly disparate cases may be treated uniformly by enriching the syntactic structure of a language with a new layer of *constructors*. To ensure that constructors are used properly (for example, that the list constructor is given only one argument, and that the function constructor is given two), we classify constructors by *kinds*. Constructors of a distinguished kind, `Type`, are types, which may be used to classify expressions. To allow for multi-argument and higher-order constructors, we will also consider finite product and function kinds. (Later we shall consider even richer kinds.)

The distinction between constructors and kinds on one hand and types and expressions on the other reflects a fundamental separation between the static and dynamic *phase* of processing of a programming language, called the *phase distinction*. The static phase implements the static semantics, and the dynamic phase implements the dynamic semantics. Constructors may be seen as a form of *static data* that is manipulated during the static phase of processing. Expressions are a form of *dynamic data* that is manipulated at run-time. Since the dynamic phase follows the static phase (we only execute well-typed programs), we may also manipulate constructors at run-

time.

Adding constructors and kinds to a language introduces more technical complications than might at first be apparent. The main difficulty is that as soon as we enrich the kind structure beyond the distinguished kind of types, it becomes essential to simplify constructors to determine whether they are equivalent. For example, if we admit product kinds, then a pair of constructors is a constructor of product kind, and projections from a constructor of product kind are also constructors. But what if we form the first projection from the pair consisiting of the constructors `nat` and `str`? This should be equivalent to `nat`, since the elimination form if post-inverse to the introduction form. Consequently, any expression (say, a variable) of the one type should also be an expression of the other. That is, typing should respect definitional equivalence of constructors.

There are two main ways to deal with this. One is to introduce a concept of definitional equivalence for constructors, and to demand that the typing judgement for expressions respect definitional equivalence of constructors of kind `Type`. This means, however, that we must show that definitional equivalence is decidable if we are to build a complete implementation of the language. The other is to prohibit formation of awkward constructors such as the projection from a pair so that there is never any issue of when two constructors are equivalent (only when they are identical). But this complicates the definition of substitution, since a projection from a constructor variable is well-formed, until you substitute a pair for the variable. Both approaches have their benefits, but the second is simplest, and is adopted here.

25.1 Statics

The syntax of kinds is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Kind	κ	::=	`Type`	`Type`
		|	`Unit`	1
		|	`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$
		|	`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \rightarrow \kappa_2$

The kinds consist of the kind of types, `Type`, the unit kind, `Unit`, and are closed under formation of product and function kinds.

The syntax of constructors is divided into two categories, the *neutral*

and the *canonical*, according to the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Neutral	a	$::=$	u	u
		$\mid$	$\mathtt{proj[l]}(a)$	$\mathtt{pr_l}(a)$
		$\mid$	$\mathtt{proj[r]}(a)$	$\mathtt{pr_r}(a)$
		$\mid$	$\mathtt{app}(a_1;c_2)$	$a_1[c_2]$
Canonical	c	$::=$	$\mathtt{atom}(a)$	$\widehat{a}$
		$\mid$	$\mathtt{unit}$	$\star$
		$\mid$	$\mathtt{pair}(c_1;c_2)$	$\langle c_1, c_2\rangle$
		$\mid$	$\mathtt{lam}(u.c)$	$\lambda\, u.c$

The meta-variable u ranges over *constructor variables*.

The reason to distinguish neutral from canonical constructors is to ensure that it is impossible to apply an elimination form to an introduction form, which demands an equation to capture the inversion principle. For example, the putative constructor $\mathtt{pr_l}(\langle c_1, c_2\rangle)$, which would be definitionally equivalent to c_1, is ill-formed according to Grammar (25.1). This is because the argument to a projection must be neutral, but a pair is only canonical, not neutral.

The canonical constructor $\mathtt{atom}(a)$ is the inclusion of neutral constructors into canonical constructors. However, the grammar does not capture a crucial property of the static semantics that ensures that only neutral constructors of kind $\mathtt{Type}$ may be treated as canonical. This requirement is imposed to limit the forms of canonical contructors of the other kinds. In particular, variables of function, product, or unit kind will turn out *not* to be canonical, but only neutral.

The static semantics of constructors and kinds is specified by the judgements

$\Delta \vdash a \Uparrow \kappa$	neutral constructor formation
$\Delta \vdash c \Downarrow \kappa$	canonical constructor formation

In each of these judgements Δ is a finite set of hypotheses of the form

$$u_1 \Uparrow \kappa_1, \ldots, u_n \Uparrow \kappa_n$$

for some $n \geq 0$. The form of the hypotheses expresses the principle that variables are neutral constructors. The formation judgements are to be understood as parametric hypothetical judgements with parameters $u_1, \ldots, u_n$ that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow \kappa} \tag{25.1a}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash \mathtt{pr_1}(a) \Uparrow \kappa_1} \tag{25.1b}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash \mathtt{pr_r}(a) \Uparrow \kappa_2} \tag{25.1c}$$

$$\frac{\Delta \vdash a_1 \Uparrow \kappa_2 \to \kappa \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash a_1[c_2] \Uparrow \kappa} \tag{25.1d}$$

$$\frac{\Delta \vdash a \Uparrow \mathtt{Type}}{\Delta \vdash \widehat{a} \Downarrow \mathtt{Type}} \tag{25.1e}$$

$$\frac{}{\Delta \vdash \star \Downarrow 1} \tag{25.1f}$$

$$\frac{\Delta \vdash c_1 \Downarrow \kappa_1 \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \Downarrow \kappa_1 \times \kappa_2} \tag{25.1g}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \lambda\, u\, .\, c_2 \Downarrow \kappa_1 \to \kappa_2} \tag{25.1h}$$

Rule (25.1e) specifies that the only neutral constructors that are canonical are those with kind `Type`. This ensures that the language enjoys the following canonical forms property, which is easily proved by inspection of Rules (25.1).

Lemma 25.1. *Suppose that* $\Delta \vdash c \Downarrow \kappa$.

1. *If* $\kappa = 1$, *then* $c = \star$.
2. *If* $\kappa = \kappa_1 \times \kappa_2$, *then* $c = \langle c_1, c_2 \rangle$ *for some* c_1 *and* c_2 *such that* $\Delta \vdash c_i \Downarrow \kappa_i$ *for* $i = 1, 2$.
3. *If* $\kappa = \kappa_1 \to \kappa_2$, *then* $c = \lambda\, u\, .\, c_2$ *with* $\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2$.

25.2 Adding Constructors and Kinds

To equip a language, $\mathcal{L}$, with constructors and kinds requires that we augment its static semantics with hypotheses governing constructor variables, and that we relate constructors of kind `Type` (types as static data) to the classifiers of dynamic expressions (types as classifiers). To achieve this the

static semantics of $\mathcal{L}$ must be defined to have judgements of the following two forms:

$\Delta \vdash \tau\ \mathsf{type}$	type formation
$\Delta\ \Gamma \vdash e : \tau$	expression formation

where, as before, Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k$$

for some $k \geq 0$ such that $\Delta \vdash \tau_i\ \mathsf{type}$ for each $1 \leq i \leq k$.

As a general principle, every constructor of kind `Type` is a classifier:

$$\frac{\Delta \vdash \tau \Uparrow \mathtt{Type}}{\Delta \vdash \tau\ \mathsf{type}}\ . \tag{25.2}$$

In many cases this is the sole rule of type formation, so that every classifier is a constructor of kind `Type`. However, this need not be the case. In some situations we may wish to have strictly more classifiers than constructors of the distinguished kind.

To see how this might arise, let us consider two extensions of $\mathcal{L}\{\rightarrow\forall\}$ from Chapter 23. In both cases we extend the universal quantifier $\forall(t.\tau)$ to admit quantification over an arbitrary kind, written $\forall_\kappa\, u.\tau$, but the two languages differ in what constitutes a constructor of kind `Type`. In one case, the *impredicative*, we admit quantified types as constructors, and in the other, the *predicative*, we exclude quantified types from the domain of quantification.

The impredicative fragment includes the following two constructor constants:

$$\frac{}{\Delta \vdash \rightarrow \Uparrow \mathtt{Type} \rightarrow \mathtt{Type} \rightarrow \mathtt{Type}} \tag{25.3a}$$

$$\frac{}{\Delta \vdash \forall_\kappa \Uparrow (\kappa \rightarrow \mathtt{Type}) \rightarrow \mathtt{Type}} \tag{25.3b}$$

We regard the classifier $\tau_1 \rightarrow \tau_2$ to be the application $\rightarrow[\tau_1][\tau_2]$. Similarly, we regard the classifier $\forall_\kappa\, u.\tau$ to be the application $\forall_\kappa[\lambda\, u.\tau]$.

The predicative fragment excludes the constant specified by Rule (25.3b) in favor of a separate rule for the formation of universally quantified types:

$$\frac{\Delta, u \Uparrow \kappa \vdash \tau\ \mathsf{type}}{\Delta \vdash \forall_\kappa\, u.\tau\ \mathsf{type}}\ . \tag{25.4}$$

The important point is that $\forall_\kappa\, u.\tau$ is a type (as classifier), but is *not* a constructor of kind type.

The signficance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

$$\frac{\Delta, u \Uparrow \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(u::\kappa.e) : \forall_\kappa u.\tau} \tag{25.5a}$$

$$\frac{\Delta\ \Gamma \vdash e : \forall_\kappa u.\tau \quad \Delta \vdash c \Downarrow \kappa}{\Delta\ \Gamma \vdash e[c] : [c/u]\tau} \tag{25.5b}$$

(Rule (25.5b) makes use of substitution, whose definition requires some care. We will return to this point in Section 25.3.)

Rule (25.5b) makes clear that a polymorphic abstraction quantifies over the constructors of kind κ. When κ is $\mathtt{Type}$ this kind may or may not include all of the classifiers of the language, according to whether we are working with the impredicative formulation of quantification (in which the quantifiers are distinguished constants for building constructors of kind $\mathtt{Type}$) or the predicative formulation (in which quantifiers arise only as classifiers and not as constructors).

The important principle here is that *constructors are static data*, so that a constructor abstraction $\Lambda(u::\kappa.e)$ of type $\forall_\kappa u.\tau$ is a mapping from static data c of kind κ to dynamic data $[c/u]e$ of type $[c/u]\tau$. Rule (25.1e) tells us that every constructor of kind $\mathtt{Type}$ determines a classifier, but it may or may not be the case that every classifier arises in this manner.

25.3 Substitution

Rule (25.5b) involves substitution of a canonical constructor, c, of kind κ into a family of types $u \Uparrow \kappa \vdash \tau\ \mathtt{type}$. This operation is is written $[c/u]\tau$, as usual. Although the intended meaning is clear, it is in fact impossible to interpret $[c/u]\tau$ as the standard concept of substitution defined for arbitrary abt's in Chapter 6. The reason is that to do so would risk violating the distinction between neutral and canonical constructors. Consider, for example, the case of the family of types

$$u \Uparrow \mathtt{Type} \rightharpoonup \mathtt{Type} \vdash u[d] \Uparrow \mathtt{Type},$$

where $d \Uparrow \mathtt{Type}$. (It is not important what we choose for d, so we leave it abstract.) Now if $c \Downarrow \mathtt{Type} \rightharpoonup \mathtt{Type}$, then by Lemma 25.1 on page 230 we have that c is $\lambda u'.c'$. Thus, if interpreted conventionally, substitution of c

for u in the given family yields the "constructor" $(\lambda\, u'\,.\,c')\,[d]$, which is not well-formed.

The solution is to define a form of *canonizing substitution* that simplifies such "illegal" combinations as it performs the replacement of a variable by a constructor of the same kind. In the case just sketched this means that we must ensure that

$$[\lambda\, u'\,.\,c'/u]u\,[d] = [d/u']c'.$$

If viewed as a definition this equation is problematic because it switches from substituting for u in the constructor $u\,[d]$ to substituting for u' in the unrelated constructor c'. Why should such a process terminate? The answer lies in the observation that the kind of u' is definitely smaller than the kind of u, since the former's kind is the domain kind of the latter's function kind. In all other cases of substitution (as we shall see shortly) the size of the target of the substitution becomes smaller; in the case just cited the size may increase, but the type of the target variable decreases. Therefore by a lexicographic induction on the type of the target variable and the structure of the target constructor, we may prove that canonizing substitution is well-defined.

We now turn to the task of making this precise. We will define simultaneously two principal forms of substitution, one of which divides into two cases:

$[c/u:\kappa]a = a'$	canonical into neutral yielding neutral
$[c/u:\kappa]a = c' \Downarrow \kappa'$	canonical into neutral yielding canonical and kind
$[c/u:\kappa]c' = c''$	canonical into canonical yielding canonical

Substitution into a neutral constructor divides into two cases according to whether the substituted variable u occurs in *critical position* in a sense to be made precise below.

These forms of substitution are simultaneously inductively defined by the following rules, which are broken into groups for clarity.

The first set of rules defines substitution of a canonical constructor into a canonical constructor; the result is always canonical.

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\widehat{a'} = \widehat{a''}} \tag{25.6a}$$

$$\frac{[c/u:\kappa]a' = c'' \Downarrow \kappa''}{[c/u:\kappa]\widehat{a'} = c''} \tag{25.6b}$$

$$\frac{}{[u/\star:\kappa]=\star} \tag{25.6c}$$

$$\frac{[c/u:\kappa]c_1' = c_1'' \quad [c/u:\kappa]c_2' = c_2''}{[c/u:\kappa]\langle c_1', c_2'\rangle = \langle c_1'', c_2''\rangle} \tag{25.6d}$$

$$\frac{[c/u:\kappa]c' = c'' \quad (u \neq u') \quad (u' \# c)}{[c/u:\kappa]\lambda\, u'.\, c' = \lambda\, u'.\, c''} \tag{25.6e}$$

The conditions on variables in Rule (25.6e) may always be met by renaming the bound variable, u', of the abstraction.

The second set of rules defines substitution of a canonical constructor into a neutral constructor, yielding another neutral constructor.

$$\frac{(u \neq u')}{[c/u:\kappa]u' = u'} \tag{25.7a}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\mathtt{pr_l}(a') = \mathtt{pr_l}(a'')} \tag{25.7b}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\mathtt{pr_r}(a') = \mathtt{pr_r}(a'')} \tag{25.7c}$$

$$\frac{[c/u:\kappa]a_1 = a_1' \quad [c/u:\kappa]c_2 = c_2'}{[c/u:\kappa]a_1\,[c_2] = a_1'(c_2')} \tag{25.7d}$$

Rule (25.7a) pertains to a *non-critical* variable, which is not the target of substitution. The remaining rules pertain to situations in which the recursive call on a neutral constructor yields a neutral constructor.

The third set of rules defines substitution of a canonical constructor into a neutral constructor, yielding a canonical constructor and its kind.

$$\frac{}{[c/u:\kappa]u = c \Downarrow \kappa} \tag{25.8a}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]\mathtt{pr_l}(a') = c_1' \Downarrow \kappa_1'} \tag{25.8b}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]\mathtt{pr_r}(a') = c_2' \Downarrow \kappa_2'} \tag{25.8c}$$

$$\frac{[c/u:\kappa]a_1' = \lambda\, u'\,.\, c' \Downarrow \kappa_2' \to \kappa' \quad [c/u:\kappa]c_2' = c_2'' \quad [c_2''/u':\kappa_2']c' = c''}{[c/u:\kappa]a_1'\,[c_2'] = c'' \Downarrow \kappa'} \tag{25.8d}$$

Rule (25.8a) governs a *critical* variable, which is the target of substitution. The substitution transforms it from a neutral constructor to a canonical constructor. This has a knock-on effect in the remaining rules of the group, which analyze the canonical form of the result of the recursive call to determine how to proceed. Rule (25.8d) is the most interesting rule. In the third premise, all three arguments to substitution change as we substitute the (substituted) argument of the application for the parameter of the (substituted) function into the body of that function. Here we require the type of the function in order to determine the type of its parameter.

Theorem 25.2. *Suppose that $\Delta \vdash c \Downarrow \kappa$, and $\Delta, u \Uparrow \kappa \vdash c' \Downarrow \kappa'$, and $\Delta, u \Uparrow \kappa \vdash a' \Uparrow \kappa'$. There exists a unique $\Delta \vdash c'' \Downarrow \kappa'$ such that $[c/u:\kappa]c' = c''$. Either there exists a unique $\Delta \vdash a'' \Uparrow \kappa'$ such that $[c/u:\kappa]a' = a''$, or there exists a unique $\Delta \vdash c'' \Downarrow \kappa'$ such that $[c/u:\kappa]a' = c''$, but not both.*

Proof. Simultaneously by a lexicographic induction with major component the structure of the kind κ, and with minor component determined by Rules (25.1) governing the formation of c' and a'. For all rules except Rule (25.8d) the inductive hypothesis applies to the premise(s) of the relevant formation rules. For Rule (25.8d) we appeal to the major inductive hypothesis applied to κ_2', which is a component of the kind $\kappa_2' \to \kappa'$. □

25.4 Exercises

Chapter 26

Indexed Families of Types