Polymorphism

Perhaps the most serious drawback of early type systems was that they precluded polymorphic functions, which are functions that can be used with arguments and results of an infinite number of types. In this chapter we will explore a type system that overcomes this problem; it is based on the polymorphic lambda calculus, which was discovered in the early 1970's.

An example of a polymorphic function is mapcar, which can have the type \( (\theta \to \theta') \rightarrow \text{list } \theta \to \text{list } \theta' \) for any types \( \theta \) and \( \theta' \). Thus, for example, it should be possible to write

\[
\text{letrec mapcar } \equiv \lambda f. \lambda x. \text{listcase } x \text{ of (nil, } \lambda i. \lambda r. f i :: \text{mapcar } f r) \text{ in }
\lambda x. \text{mapcar}(\lambda n. n = 0)(\text{mapcar}(\lambda n. n - 3) x).
\]

But this expression cannot be typed according to the simple type discipline, since the two occurrences of mapcar in the last line have different types:

\[
(\text{int } \to \text{ bool}) \rightarrow \text{list int } \to \text{list bool} \quad (\text{int } \to \text{ int}) \rightarrow \text{list int } \to \text{list int}.
\]

(Of course, one could type this expression using intersection types as described in Section 16.3, by giving mapcar the intersection of the above types. But polymorphism goes beyond intersection types in giving an infinite family of types to a function such as mapcar, rather than just some finite set of types.)

Two other examples of the need for polymorphism are a function that sums a list of lists of integers:

\[
\text{letrec reduce } \equiv \lambda x. \lambda f. \lambda a. \text{listcase } x \text{ of (a, } \lambda i. \lambda r. f i (\text{reduce } r f a) \text{ in }
\lambda y. \text{reduce } y (\lambda x. \lambda s. (\text{reduce } x (\lambda m. \lambda n. m + n) 0) + s) 0,
\]

where the two last occurrences of reduce have types

\[
\text{list list int } \to (\text{list int } \to \text{int } \to \text{int} ) \to \text{int } \to \text{int}
\]

and

\[
\text{list int } \to (\text{int } \to \text{int } \to \text{int}) \to \text{int } \to \text{int}.
\]
and a self-application of a function that composes its argument with itself:

\[
\text{let } \text{double} \equiv \lambda f. \lambda x. f(f\ x) \ \text{in double double},
\]

where the two last occurrences of double might have types

\[
((\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \to ((\text{int} \to \text{int}) \to (\text{int} \to \text{int}))
\]

and

\[
(\text{int} \to \text{int}) \to (\text{int} \to \text{int}).
\]

There are two basic approaches to accommodating polymorphism: a limited but implicitly typed polymorphism, typified by ML, and a broader explicitly typed polymorphism, typified by the polymorphic lambda calculus. Here we will focus on the latter, which requires explicit type information but gives polymorphic functions “first-class citizenship”, in particular allowing them to be passed as arguments to other functions.

### 17.1 Syntax and Inference Rules

We start by extending our explicitly typed functional language with type variables (ranging over the predefined set \{tvar\}), so that types become type expressions:

\[
\langle \text{type} \rangle ::= \langle \text{tvar} \rangle
\]

(Notice that this includes the type expressions that occur within ordinary expressions in an explicitly typed language.) Then we introduce the ability to define polymorphic functions that accept types as arguments, by abstracting ordinary expressions on type variables (instead of ordinary variables), and the ability to apply such functions to type expressions:

\[
\langle \text{exp} \rangle ::= \Lambda \langle \text{tvar} \rangle. \langle \text{exp} \rangle \ |
\langle \text{exp} \rangle[(\langle \text{type} \rangle)]
\]

In the first form \(\Lambda \tau. e\), the initial occurrence of the type variable \(\tau\) binds the occurrences of \(\tau\) in (the type expressions in) the expression \(e\).

This new kind of function is explained by a new kind of \(\beta\)-reduction that substitutes for type variables instead of ordinary variables:

\[
(\Lambda \tau. e)[\theta] \to (e/\tau \to \theta).
\]

Then, for example, one can write

\[
\text{let } \text{double} \equiv \Lambda \alpha. \lambda f_{\alpha \to \alpha}. \lambda x_{\alpha}. f(f\ x) \ \text{in double}[\text{int} \to \text{int}](\text{double}[\text{int}]),
\]

which reduces in a few steps to

\[
(\lambda f_{(\text{int} \to \text{int}) \to (\text{int} \to \text{int})}. \lambda x_{\text{int} \to \text{int}}. f(f\ x)) (\lambda f_{\text{int} \to \text{int}}. \lambda x_{\text{int}}. f(f\ x))
\]
17.1 Syntax and Inference Rules

(at which stage the polymorphism has disappeared). Similarly, our examples involving mapcar and reduce can be written as follows:

\[
\text{letrec mapcar} \equiv \lambda \alpha. \lambda \beta. \lambda f_{\alpha \rightarrow \beta}. \lambda x_{\text{list } \alpha}.
\]

\[
\text{listcase } x \text{ of } (\text{nil}_{\beta}, \lambda i_{\alpha}. \lambda r_{\text{list } \alpha}. f_i :: \text{mapcar}[[\alpha][\beta]] f r) \text{ in}
\]

\[
\lambda x_{\text{list int}}. \text{mapcar}[\text{int}][\text{bool}]((\lambda n_{\text{int}}. n = 0)
\]

\[
(\text{mapcar}[\text{int}][\text{int}]((\lambda n_{\text{int}}. n - 3) x))
\]

\[
\text{letrec reduce} \equiv \lambda \alpha. \lambda \beta. \lambda x_{\text{list } \alpha}. \lambda f_{\alpha \rightarrow \beta \rightarrow \beta}. \lambda r_{\beta}.
\]

\[
\text{listcase } x \text{ of } (a, \lambda i_{\alpha}. \lambda r_{\text{list } \alpha}. f_i (\text{reduce}[[\alpha][\beta]] f a)) \text{ in}
\]

\[
\lambda y_{\text{list int}}. \text{reduce}[\text{list int}][\text{int}] y ((\lambda x_{\text{list int}}. \lambda s_{\text{int}}.
\]

\[
(\text{reduce}[\text{int}][\text{int}] x (\lambda m_{\text{int}}. \lambda n_{\text{int}}. m + n) 0) + s) 0.
\]

The final step is to introduce new types for the polymorphic functions. (Actually, we should have included such types in the letrec definitions above.) We introduce the type expression \( \forall \tau. \theta \), which binds the occurrences of \( \tau \) in \( \theta \) and describes the type of polymorphic function that maps a type \( \tau \) into a value of type \( \theta \). For example, double (strictly speaking, the \( \Lambda \)-expression defining double) has the type

\[\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha,\]

mapcar has the type

\[\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta,\]

and reduce has the type

\[\forall \alpha. \forall \beta. \text{list } \alpha \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta.\]

To accomplish this final step, we augment the syntax of type expressions:

\[
\langle \text{type} \rangle ::= \forall (\text{tvar}. \langle \text{type} \rangle)
\]

(Concretely, as with other quantifiers, we assume that \( \theta \) in \( \forall \tau. \theta \) extends to the first stopping symbol or the end of the enclosing phrase.) We also add two new inference rules (to the rules given in Section 15.2, as modified for explicit typing in Section 15.3):

\[
\text{TY RULE: Explicit } \forall \text{ Introduction}
\]

\[
\pi \vdash e : \theta \quad \text{when } \tau \text{ does not occur free in any type expression in } \pi,
\]

\[
\pi \vdash \Lambda \tau. \theta
\]

\[
\pi \vdash e : \forall \tau. \theta
\]

\[
\pi \vdash e[\theta'] : (\theta' / \tau \rightarrow \theta').
\]

(17.1)

(17.2)
The condition in the first of these rules prevents the binding of a type variable that would still have free occurrences in the conclusion of the rule.

The following is an example of a typing proof using these inference rules. Let \( \pi \) be the context

\[
\text{double: } \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha).
\]

Then:

1. \( f: \alpha \to \alpha, x: \alpha \vdash f : \alpha \to \alpha \) \hspace{1cm} (15.3)
2. \( f: \alpha \to \alpha, x: \alpha \vdash x : \alpha \) \hspace{1cm} (15.3)
3. \( f: \alpha \to \alpha, x: \alpha \vdash f \cdot x : \alpha \) \hspace{1cm} (15.5)
4. \( f: \alpha \to \alpha, x: \alpha \vdash f (f \cdot x) : \alpha \) \hspace{1cm} (15.5)
5. \( f: \alpha \to \alpha \vdash \lambda \alpha. (f (f \cdot x)) : \alpha \to \alpha \) \hspace{1cm} (15.19)
6. \( \vdash \lambda \alpha. (f (f \cdot x)) : (\alpha \to \alpha) \to (\alpha \to \alpha) \) \hspace{1cm} (15.19)
7. \( \vdash \lambda \alpha. (f (f \cdot x)) : (\alpha \to \alpha) \to (\alpha \to \alpha) \) \hspace{1cm} (17.1)
8. \( \pi \vdash \text{double} : \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \) \hspace{1cm} (15.3)
9. \( \pi \vdash \text{double} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \) \hspace{1cm} (17.2)
10. \( \pi \vdash \text{double} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \to ((\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \to ((\text{int} \to \text{int}) \to (\text{int} \to \text{int})) \to (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \) \hspace{1cm} (17.2)
11. \( \pi \vdash \text{double} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \) \hspace{1cm} (15.5)
12. \( \vdash \text{let double = } \lambda \alpha. (f (f \cdot x)) \text{ in double} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \to (\text{int} \to \text{int}) \) \hspace{1cm} (15.11)

In conclusion, it should be noted that one can also formulate an implicitly typed polymorphic language where, to eliminate all types from expressions, one erases the type binders \( \Lambda \tau \) and the type operands \( [\theta] \), as well as the types that appear as subscripts:

**TY RULE: Implicit \( \forall \) Introduction**

\[
\frac{\pi \vdash e : \theta}{\pi \vdash e : \forall \tau. \theta} \quad \text{when } \tau \text{ does not occur free in any type expression in } \pi,
\]

**TY RULE: Implicit \( \forall \) Elimination**

\[
\frac{\pi \vdash e : \forall \tau. \theta}{\pi \vdash e : (\theta/\tau \to \theta')},
\]
17.2 Polymorphic Programming

It is known that there is no algorithm that can infer typing judgements for this implicitly typed language. The language ML, however, provides a restricted form of implicit polymorphism for which type inference can be performed by an extension of the Hindley-Milner algorithm. Essentially, ML permits polymorphic functions to be introduced by let and letrec definitions, as in the examples at the beginning of this chapter. But it does not permit polymorphic functions to be passed as arguments to other functions.

17.2 Polymorphic Programming

Examples such as mapcar and reduce make a strong case for being able to declare polymorphic functions using a let or letrec definition. But at first sight, the idea of passing polymorphic functions to other functions seems very exotic. Nevertheless, the work of several researchers suggests that using polymorphic functions as data may be the key to a novel programming style. They have studied the pure lambda calculus with polymorphic typing (without explicit recursive definition via rec or letrec) and have shown that this restricted language has extraordinary properties. On the one hand, all well-typed expressions have normal forms (indeed, they are “strongly” normalizing, which means that they have no infinite reduction sequences), although the proof of this fact requires “second-order arithmetic” and cannot be obtained from Peano’s axioms. On the other hand, the variety of functions that can be expressed goes far beyond the class of primitive recursive functions to include any program whose termination can be proved in second-order arithmetic. Beyond this they have shown that certain types of polymorphic functions behave like algebras.

Our purpose here is not to give the details of this theoretical work, but to illustrate the unusual programming style that underlies it. The starting point is a variant of the way that early investigators of the untyped lambda calculus encoded truth values and natural numbers, which we discussed briefly in Section 10.6.

Suppose we regard bool as an abbreviation for a polymorphic type, and true and false as abbreviations for certain functions of this type:

\[
\begin{align*}
\text{bool} & \equiv \forall \alpha. \alpha \to \alpha \to \alpha \\
\text{true} & \equiv \Lambda \alpha. \lambda x_\alpha. \lambda y_\alpha. x \\
\text{false} & \equiv \Lambda \alpha. \lambda x_\alpha. \lambda y_\alpha. y.
\end{align*}
\]

Then, when \( e \) and \( e' \) have type \( \theta \), we can define the conditional expression by

\[
\text{if } b \text{ then } e \text{ else } e' \overset{\text{def}}{=} b[\theta] e e',
\]
since the reduction rules prescribe the right behavior for the conditional:

\[
\text{if true then } e \text{ else } e' \overset{\text{def}}{=} (\Lambda \alpha. \lambda x_\alpha. \lambda y_\alpha. x)[\theta] e e' \\
\rightarrow (\lambda x_\theta. \lambda y_\theta. x) e e' \\
\rightarrow (\lambda y_\theta. e) e' \\
\rightarrow e
\]

and

\[
\text{if false then } e \text{ else } e' \overset{\text{def}}{=} (\Lambda \alpha. \lambda x_\alpha. \lambda y_\alpha. y)[\theta] e e' \\
\rightarrow (\lambda x_\theta. \lambda y_\theta. y) e e' \\
\rightarrow (\lambda y_\theta. y) e' \\
\rightarrow e'.
\]

Moreover, we can define

\[
\text{not} \overset{\text{def}}{=} A\alpha . \lambda x_\alpha . \lambda y_\alpha . b[\alpha] y x
\]

and

\[
\text{and} \overset{\text{def}}{=} A\alpha . \lambda x_\alpha . \lambda y_\alpha . b[\alpha](c[\alpha] x y) y.
\]

Similarly, we can define \text{nat} and the natural numbers by a typed form of Church numerals:

\[
\text{nat} \overset{\text{def}}{=} \forall \alpha . (\alpha \to \alpha) \to \alpha \to \alpha
\]

\[
0 \overset{\text{def}}{=} \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . x
\]

\[
1 \overset{\text{def}}{=} \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . f x
\]

\[
2 \overset{\text{def}}{=} \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . f(f x)
\]

\[
:\vdots
\]

\[
n \overset{\text{def}}{=} \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . f^n x
\]

(where \(f^n x\) denotes \(f (\cdots (f x) \cdots)\) with \(n\) occurrences of \(f\)), so that the number \(n\) becomes a polymorphic function accepting a function and giving its \(n\)-fold composition. For example, 2 is double. (Strictly speaking, we should use some notation such as \(\text{NUM}_n\) for the \(n\)th Church numeral, as in Section 10.6, to indicate that the numeral is an expression denoting a number rather than a number itself. But such formality would make our arguments much harder to follow.)

Using this encoding of the natural numbers, we can define the successor function in either of two ways:

\[
\text{succ} \overset{\text{def}}{=} \lambda n_{\text{nat}} . \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . f(n[\alpha] f x)
\]

or

\[
\text{succ} \overset{\text{def}}{=} \lambda n_{\text{nat}} . \Lambda \alpha . \lambda f_\alpha \to \alpha . \lambda x_\alpha . n[\alpha] f(f x).
\]
Now suppose \( g : \text{nat} \to \theta \) satisfies
\[
g \circ 0 = c \quad \text{and} \quad \forall n \geq 0. \ g(n + 1) = h(g(n)), \tag{17.5}
\]
where \( c : \theta \) and \( h : \theta \to \theta \). Then \( g n = h^n c \), so that we can define \( g n \) by applying the \( n \)th Church numeral:
\[
g \overset{\text{def}}{=} \lambda n_{\text{nat}}. \ n[\theta] h c.
\]
For example, since
\[
\text{add } m \ 0 = m \quad \text{and} \quad \text{add } m \ (n + 1) = \text{succ} (\text{add } m \ n)
\]
fit the mold of Equations (17.5), we can define
\[
\text{add } m \overset{\text{def}}{=} \lambda n_{\text{nat}}. \ n[\text{nat}] \text{succ } m
\]
or, more abstractly,
\[
\text{add } \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ n[\text{nat}] \text{succ } m.
\]
Similarly,
\[
\text{mult } \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ n[\text{nat}] \text{add } m \ 0
\]
\[
\text{exp } \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ n[\text{nat}] \text{mult } m \ 1.
\]
In fact, this approach can be generalized to define any primitive recursive function on the natural numbers. A function \( f : \text{nat} \to \theta \) is said to be primitive recursive if it satisfies
\[
f 0 = c \quad \text{and} \quad \forall n \geq 0. \ f(n + 1) = h(n \ f(n)),
\]
where \( c : \theta \) and \( h : \text{nat} \to \theta \to \theta \) are defined nonrecursively in terms of previously defined primitive recursive functions.

Let \( g : \text{nat} \to \text{nat} \times \theta \) be such that \( g n = \langle n, f n \rangle \). Then
\[
g 0 = \langle 0, f 0 \rangle
\]
\[
= \langle 0, c \rangle
\]
\[
g(n + 1) = \langle n + 1, f(n + 1) \rangle
\]
\[
= \langle \text{succ } n, h(n \ f(n)) \rangle
\]
\[
= (\lambda \langle k, z \rangle_{\text{nat} \times \theta}. \ \langle \text{succ } k, h \ k \ z \rangle \rangle (n, f n)
\]
\[
= (\lambda \langle k, z \rangle_{\text{nat} \times \theta}. \ \langle \text{succ } k, h \ k \ z \rangle \rangle (g n).
\]
This matches the form of Equations (17.5), so that \( g n \) can be defined by applying the \( n \)th Church numeral:
\[
g = \lambda n_{\text{nat}}. \ n[\text{nat} \times \theta] (\lambda \langle k, z \rangle_{\text{nat} \times \theta}. \ \langle \text{succ } k, h \ k \ z \rangle \rangle (0, c),
\]
which gives

\[ f = \lambda n_{\text{nat}}. \ (n[\text{nat} \times \theta](\lambda (k, z)_{\text{nat} \times \theta}. \ (\text{succ} \ k, h \ k \ z))(0, c)).1. \]

For example, since the predecessor function satisfies

\[
\begin{align*}
\text{pred} 0 & = 0 \\
\text{pred}(n + 1) & = n = (\lambda n_{\text{nat}}. \ \lambda m_{\text{nat}}. \ n \ (\text{pred} \ n)),
\end{align*}
\]

we can define

\[
\begin{align*}
\text{pred} & = \lambda n_{\text{nat}}. \ (n[\text{nat} \times \text{nat}]
\ (\lambda (k, z)_{\text{nat} \times \text{nat}}. \ (\text{succ} \ k, (\lambda n_{\text{nat}}. \ \lambda m_{\text{nat}}. \ n \ k \ z))(0, 0)).1
\end{align*}
\]

(Unfortunately, this predecessor function requires time that is proportional to its argument. It is not known how to program a constant-time predecessor function in this formalism.) Similarly, since

\[
\begin{align*}
\text{fact} 0 & = 1 \\
\text{fact}(n + 1) & = (n + 1) \mathrm{\ast} \text{fact} \ n = (\lambda n_{\text{nat}}. \ \lambda m_{\text{nat}}. \ (n + 1) \times m) \ n \ (\text{fact} \ n),
\end{align*}
\]

we can define

\[
\begin{align*}
\text{fact} & = \lambda n_{\text{nat}}. \ (n[\text{nat} \times \text{nat}]
\ (\lambda (k, z)_{\text{nat} \times \text{nat}}. \ (\text{succ} \ k, (\lambda n_{\text{nat}}. \ \lambda m_{\text{nat}}. \ (n + 1) \times m) \ k \ z))(0, 1)).1
\end{align*}
\]

Our ability to define numerical functions, however, is not limited to the scheme of primitive recursion. For example, the exponentiation laws \( f^m \cdot f^n = f^{m+n} \) and \( (f^m)^n = f^{m \times n} \) lead directly to the definitions

\[
\begin{align*}
\text{add} & \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ \lambda \alpha. \ \lambda f_{\alpha \rightarrow \alpha}. \ \lambda x_{\alpha}. \ n \ [\alpha] \ f \ (m \ [\alpha] \ f \ x) \\
\text{mult} & \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ \lambda \alpha. \ \lambda f_{\alpha \rightarrow \alpha}. \ n \ [\alpha] \ (m \ [\alpha] \ f),
\end{align*}
\]

and the law \( \lambda f. \ f(\text{f}^n) = (\lambda f. \ f^m)^n \) (which can be proved by induction on \( n \)) leads to

\[
\text{exp} \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ \lambda n_{\text{nat}}. \ \lambda \alpha. \ n \ [\alpha \rightarrow \alpha] \ (m \ [\alpha]).
\]

Moreover, if we define

\[
\text{aug} \overset{\text{def}}{=} \lambda f_{\text{nat} \rightarrow \text{nat}}. \ \lambda n_{\text{nat}}. \ \text{succ} \ n \ [\text{nat}] \ f \ 1,
\]

which implies \( \text{aug} \ f \ n = f^{n+1} \), then we can define

\[
\text{ack} \overset{\text{def}}{=} \lambda m_{\text{nat}}. \ m \ [\text{nat} \rightarrow \text{nat}] \ \text{aug} \ \text{succ},
\]
so that we have

\[
\text{ack } 0 \ n = n + 1
\]

\[
\text{ack } (m + 1) \ 0 = \text{aug } (\text{ack } m) \ 0
\]

\[
= \text{ack } m \ 1
\]

\[
\text{ack } (m + 1) (n + 1) = \text{aug } (\text{ack } m) (n + 1)
\]

\[
= \text{ack } m (\text{aug } (\text{ack } m) n)
\]

\[
= \text{ack } m (\text{ack } (m + 1) n).
\]

This establishes that \( \lambda n_{\text{nat}}. \text{ack } n \ n \) is Ackermann’s function, which is known to grow so rapidly that it is not primitive recursive. Thus the functions definable in the polymorphic lambda calculus go beyond primitive recursion (which is extraordinary in a language where all expressions terminate).

In addition to primitive types such as \textbf{nat} and \textbf{bool}, various type constructors can be defined in terms of polymorphic functions. For binary products, we can define

\[
\theta_0 \times \theta_1 \overset{\text{def}}{=} \forall \alpha. (\theta_0 \rightarrow \theta_1 \rightarrow \alpha) \rightarrow \alpha
\]

and, where \( e_0 : \theta_0, \ e_1 : \theta_1 \), and \( p : \theta_0 \times \theta_1 \),

\[
\langle e_0, e_1 \rangle \overset{\text{def}}{=} \Lambda \alpha. \lambda f_{\theta_0 \rightarrow \theta_1 \rightarrow \alpha}. f \ e_0 \ e_1
\]

\[
p.0 \overset{\text{def}}{=} p \ [\theta_0] (\lambda x_{\theta_0}. \lambda y_{\theta_1}. x)
\]

\[
p.1 \overset{\text{def}}{=} p \ [\theta_1] (\lambda x_{\theta_0}. \lambda y_{\theta_1}. y),
\]

since we have the reduction

\[
\langle e_0, e_1 \rangle .0 \overset{\text{def}}{=} (\Lambda \alpha. \lambda f_{\theta_0 \rightarrow \theta_1 \rightarrow \alpha}. f \ e_0 \ e_1) \ [\theta_0] (\lambda x_{\theta_0}. \lambda y_{\theta_1}. x)
\]

\[
\rightarrow (\lambda f_{\theta_0 \rightarrow \theta_1 \rightarrow \theta_0}. f \ e_0 \ e_1) (\lambda x_{\theta_0}. \lambda y_{\theta_1}. x)
\]

\[
\rightarrow (\lambda x_{\theta_0}. \lambda y_{\theta_1}. x) \ e_0 \ e_1
\]

\[
\rightarrow (\lambda y_{\theta_1}. \ e_0) \ e_1
\]

\[
\rightarrow e_0,
\]

and similarly

\[
\langle e_0, e_1 \rangle .1 \rightarrow^* e_1.
\]

For binary disjoint unions, we can define

\[
\theta_0 + \theta_1 \overset{\text{def}}{=} \forall \alpha. (\theta_0 \rightarrow \alpha) \rightarrow (\theta_1 \rightarrow \alpha) \rightarrow \alpha
\]
and, where \( e_0 : \theta_0, e_1 : \theta_1, f_0 : \theta_0 \to \theta, f_1 : \theta_1 \to \theta \) and \( e : \theta_0 + \theta_1 \),

\[
\begin{align*}
@ 0 e_0 & \overset{\text{def}}{=} \Lambda \alpha. \lambda \theta_0 \to \alpha. \lambda \theta_1 \to \alpha. g
e_0 \\
@ 1 e_1 & \overset{\text{def}}{=} \Lambda \alpha. \lambda \theta_0 \to \alpha. \lambda \theta_1 \to \alpha. h
e_1 \\
\text{sumcase } e \text{ of } (f_0, f_1) & \overset{\text{def}}{=} e[\theta]f_0f_1,
\end{align*}
\]

since we have

\[
\begin{align*}
\text{sumcase } (@ 0 e_0) \text{ of } (f_0, f_1) & \overset{\text{def}}{=} (\Lambda \alpha. \lambda \theta_0 \to \alpha. \lambda \theta_1 \to \alpha. g
e_0)[\theta]f_0f_1 \\
& \to (\lambda \theta_0 \to \theta. \lambda \theta_1 \to \theta. g
e_0)f_0f_1 \\
& \to (\lambda \theta_1 \to \theta. f_0
e_0)f_1 \\
& \to f_0
e_0,
\end{align*}
\]

and similarly

\[
\text{sumcase } (@ 1 e_1) \text{ of } (f_0, f_1) \to f_1 e_1.
\]

Finally, we consider representing lists by polymorphic functions. If we define the type

\[
\text{list } \theta \overset{\text{def}}{=} \forall \alpha. (\theta \to \alpha \to \alpha) \to \alpha \to \alpha,
\]

and the individual lists of this type by

\[
x_0 :: x_1 :: \cdots :: x_{n-1} :: \text{nil} \overset{\text{def}}{=} \Lambda \alpha. \lambda f_\theta \to \alpha \to \alpha. \lambda a_\alpha. f x_0 (f x_1 \cdots (f x_{n-1} a) \cdots)
\]

(so that a list is its own reduce function), then we can define

\[
\text{nil} \overset{\text{def}}{=} \Lambda \alpha. \lambda f_\theta \to \alpha \to \alpha. \lambda a_\alpha. a \\
e :: e' \overset{\text{def}}{=} \Lambda \alpha. \lambda f_\theta \to \alpha \to \alpha. \lambda a_\alpha. f (e'[\alpha]f a).
\]

Now suppose \( g : \text{list } \theta \to \theta' \) satisfies

\[
g\text{ nil} = c \quad \text{and} \quad g(i :: r) = h i (g\ r), \quad (17.6)
\]

where \( c : \theta' \) and \( h : \theta \to \theta' \to \theta' \). Then \( g x = \text{reduce } x \ h \ c \) so that we can define \( g \) by

\[
g \overset{\text{def}}{=} \lambda x_\text{list } \theta. x[\theta']h\ c.
\]

For example, if \( \text{rappend } y \ x = \text{append } x \ y \), then

\[
\text{rappend } y \ \text{nil} = y \\
\text{rappend } y (i :: r) = i :: (\text{rappend } y r),
\]

which fits the form of Equations (17.6), so that we can define

\[
\text{rappend } y \overset{\text{def}}{=} \lambda x_\text{list } \theta. x[\theta] \ (\lambda i_\theta. \ \lambda x_\text{list } \theta. i :: z) y
\]
17.2 Polymorphic Programming

or

\[
\text{append} \equiv \lambda \text{list } \theta \cdot \lambda y \text{list } \theta \cdot \text{list } \theta \cdot (\lambda i \theta \cdot \lambda z \text{list } \theta \cdot i :: z) y.
\]

Similarly,

\[
\text{length} \equiv \lambda \text{list } \theta \cdot \text{nat } \theta \cdot (\lambda i \theta \cdot \lambda z \text{nat } \theta \cdot z + 1) 0
\]

\[
\text{sumlist} \equiv \lambda \text{list nat } \theta \cdot \text{nat } \theta \cdot (\lambda i \text{nat } \theta \cdot \lambda z \text{nat } \theta \cdot i + z) 0
\]

\[
\text{mapcar} \equiv \lambda f \theta \rightarrow \theta' \cdot \lambda \text{list } \theta \cdot \text{list } \theta' \cdot (\lambda i \theta \cdot \lambda z \text{list } \theta \cdot (f i) :: z) \text{nil}.
\]

These are all functions that recur uniformly down a single list. At first sight, one might expect that it would be more difficult to define a function where the recursion switches back and forth between two lists, such as a function for merging ordered lists of integers. But in fact one can define a curried merging function that uses uniform recursion in its first argument to produce the appropriate function of its second argument.

The first step is to introduce a subsidiary function

\[
\text{insertapp} : \text{int } \rightarrow (\text{list int } \rightarrow \text{list int}) \rightarrow \text{list int } \rightarrow \text{list int}
\]

such that, when \( x \) is an ordered list of integers, \( \text{insertapp } m f x \) inserts \( m \) into the proper position of \( x \) and applies \( f \) to the portion of \( x \) following this position. (To keep our argument simple, we are using the primitive type \text{int}, rather than the representation by Church numerals discussed earlier in this section.) Then \( \text{merge} \) can be expressed in terms of \( \text{insertapp} \) by

\[
\text{merge } \text{nil } y = y
\]

\[
\text{merge } (i :: r) y = \text{insertapp } i (\lambda y \text{list int } \cdot \text{merge } r y) y.
\]

These equations can be written more abstractly as

\[
\text{merge } \text{nil } = \lambda y \text{list int } : y
\]

\[
\text{merge } (i :: r) = \text{insertapp } i (\text{merge } r),
\]

which matches the form of Equations (17.6), so that we can define

\[
\text{merge} \equiv \lambda \text{list int } : \text{list int } \rightarrow \text{list int } \rightarrow \text{list int } \text{list int } \rightarrow \text{list int } \text{list int } \rightarrow \text{list int } \text{list int }
\]

The remaining task is to program \( \text{insertapp} \), which satisfies

\[
\text{insertapp } m f \text{nil } = m :: (f \text{nil})
\]

\[
\text{insertapp } m f (i :: r) = \text{if } m \leq i \text{ then } m :: (f(i :: r)) \text{ else } i :: (\text{insertapp } m f r).
\]

This does not match the form of Equations (17.6), but it can be treated by the same approach that we used for primitive recursive functions on numbers. Let

\[
g : \text{int } \rightarrow (\text{list int } \rightarrow \text{list int}) \rightarrow \text{list int } \rightarrow (\text{list int } \times \text{list int })
\]
be such that \( g \, m \, f \, x = (x, \text{insertapp} \, m \, f \, x) \). Then

\[
g \, m \, f \, \text{nil} = (\text{nil}, \text{insertapp} \, m \, f \, \text{nil})
= (\text{nil}, m :: (f \, \text{nil}))
\]

\[
g \, m \, f \, (i :: r) = (i :: r, \text{insertapp} \, m \, f \, (i :: r))
= (i :: r, \text{if } m \leq i \text{ then } m :: (f(i :: r)) \text{ else } i :: \text{insertapp} \, m \, f \, r)
= (i :: r, \text{if } m \leq i \text{ then } m :: (f(i :: r)) \text{ else } i :: z)
\]

This fits the form of Equations (17.6), so that we can define

\[
g \equiv \lambda \text{int}. \lambda \text{list int} \to \text{list int}. \lambda \text{list int}. x \to \text{list int} \times \text{list int}
\]

\[
(\lambda \text{int}. \lambda \langle y, z \rangle \text{list int} \times \text{list int}. (i :: y, \text{if } m \leq i \text{ then } m :: (f(i :: y)) \text{ else } i :: z))
\]

\[
(i :: y, \text{if } m \leq i \text{ then } m :: (f(i :: y)) \text{ else } i :: z)
\]

and therefore

\[
\text{insertapp} \equiv \lambda m \text{int}. \lambda \text{list int} \to \text{list int}. \lambda \text{list int}. x \to \text{list int} \times \text{list int}
\]

\[
(\lambda \text{int}. \lambda \langle y, z \rangle \text{list int} \times \text{list int}. (i :: y, \text{if } m \leq i \text{ then } m :: (f(i :: y)) \text{ else } i :: z))
\]

\[
(i :: y, \text{if } m \leq i \text{ then } m :: (f(i :: y)) \text{ else } i :: z)
\]

\[
\langle \text{nil}, m :: (f \, \text{nil}) \rangle.
\]

17.3 Extrinsic Semantics

The extrinsic semantics of polymorphic types reveals their similarity to intersection types: Just as \( \theta_0 \& \theta_1 \) denotes the intersection of the partial equivalence relations denoted by \( \theta_0 \) and \( \theta_1 \), so the quantified type \( \forall \tau . \theta \) denotes the intersection of all of the partial equivalence relations that are denoted by \( \theta \) as one varies the denotation of the type variable \( \tau \).

To make this precise, the first step is to accommodate type variables. Let \( \text{PER} \) stand for the set of chain-complete partial equivalence relations whose domain contains \( \bot \) but not \( \text{tyerr} \) (i.e. which satisfy the conditions of Proposition 15.1 in Section 15.4). In Sections 15.4 and 16.4, we defined a function \( \mathcal{P} \in \langle \text{type} \rangle \to \text{PER} \) that mapped types into their denotations. But when types can contain type variables, what they denote depends (just as with ordinary expressions) on what the type variables denote. Thus we must introduce the concept of type environments, which map type variables into partial equivalence relations, and redefine \( \mathcal{P} \) to satisfy

\[
\mathcal{P} \in \langle \text{type} \rangle \to (\text{PER}^{(\text{tvar})} \to \text{PER} ).
\]

We will use the metavariable \( \xi \) for type environments in \( \text{PER}^{(\text{tvar})} \).
The various cases defining $\mathcal{P}$ in Sections 15.4 and 16.4 change in a trivial way: The occurrences of $\mathcal{P}(-)$ become occurrences of $\mathcal{P}(-)\xi$. In addition, there are two new cases, which describe type variables and polymorphic types:

(h) $\langle x, x' \rangle \in \mathcal{P}(\tau)\xi$ if and only if $\langle x, x' \rangle \in \tau$.

(i) $\langle x, x' \rangle \in \mathcal{P}(\forall \tau. \theta)\xi$ if and only if $\langle x, x' \rangle \in \mathcal{P}(\theta)[\xi \mid \tau: \rho]$ for all $\rho \in \text{PER}$.

More succinctly,

(h) $\mathcal{P}(\tau)\xi = \xi \tau$.

(i) $\mathcal{P}(\forall \tau. \theta)\xi = \bigcap_{\rho \in \text{PER}} \mathcal{P}(\theta)[\xi \mid \tau: \rho]$.

With the introduction of type variables and a binding operator, types exhibit the same behavior as did assertions in Section 1.4 or expressions of the lambda calculus in Section 10.5. The set of free type variables in a type is given by

$$FTV(\tau) = \{\tau\}$$

$$FTV(\forall \tau. \theta) = FTV(\theta) - \{\tau\}$$

and, for all other forms, by

$$FTV(\theta) = FTV(\theta_0) \cup \cdots \cup FTV(\theta_{n-1}),$$

where $\theta_0, \ldots, \theta_{n-1}$ are the subphrases of $\theta$. Then one can show

**Proposition 17.1 (Coincidence Theorem)** If $\xi \tau = \xi' \tau$ for all $\tau \in FTV(\theta)$, then $\mathcal{P}(\theta)\xi = \mathcal{P}(\theta)\xi'$.

**Proposition 17.2 (Substitution Theorem)** If $[\delta \tau]\xi' = \xi \tau$ for all $\tau \in FTV(\theta)$, then $\mathcal{P}(\theta/\delta)\xi' = \mathcal{P}(\theta)\xi$.

**Proposition 17.3 (Finite Substitution Theorem)**

$$\mathcal{P}(\theta/\tau_0 \rightarrow \theta_0, \ldots, \tau_{n-1} \rightarrow \theta_{n-1})\xi'$$

$$= \mathcal{P}(\theta)[\xi' \mid \tau_0: \mathcal{P}(\theta_0)\xi' \mid \cdots \mid \tau_{n-1}: \mathcal{P}(\theta_{n-1})\xi'][.]$$

**Proposition 17.4 (Soundness of Renaming)** If

$$\tau_{\text{new}} \notin FTV(\theta) - \{\tau\},$$

then

$$\mathcal{P}(\forall \tau_{\text{new}}. (\theta/\tau \rightarrow \tau_{\text{new}})) = \mathcal{P}(\forall \tau. \theta).$$

Moreover, since an intersection of members of PER is a member of PER, Proposition 15.1 in Section 15.4 remains true (with the replacement of $\mathcal{P}(\theta)$ by $\mathcal{P}(\theta)\xi$, where $\xi$ is an arbitrary type environment in $\text{PER}^{(\text{tvar})}$).
The appropriate generalization of Proposition 15.3 is

**Proposition 17.5 (Soundness of Polymorphic Typing)** If \( \pi \vdash e : \theta \) is a valid typing judgement, \( \xi \) is a type environment, and \( \eta \) and \( \eta' \) are environments such that

\[
\langle \eta v, \eta' v \rangle \in \mathcal{P}(\pi v) \xi \text{ for all } v \in \text{dom } \pi,
\]

then

\[
\langle [e] \eta, [e] \eta' \rangle \in \mathcal{P}(\theta) \xi.
\]

**Proof** As before, the proof is by structural induction on the formal proof of \( \pi \vdash e : \theta \), with a case analysis on the inference rule used in the final step of the proof. The cases in the proof in Section 15.4 (and its continuation in Section 16.4) remain essentially the same. But now there are two more inference rules to consider. (Since the earlier version of the proposition was stated for implicit typing, we will use the implicitly typed version of the new rules as well.)

Suppose the rule used in the final step is the rule for \( \forall \) introduction ((17.3) in Section 17.1),

\[
\frac{\pi \vdash e : \theta}{\pi \vdash e : \forall \tau. \theta}
\]

when \( \tau \) does not occur free in any type expression in \( \pi \).

Assume \( \langle \eta v, \eta' v \rangle \in \mathcal{P}(\pi v) \xi \) for all \( v \in \text{dom } \pi \). Since \( \tau \) does not occur free in any type expression in \( \pi \), the coincidence theorem gives \( \langle \eta v, \eta' v \rangle \in \mathcal{P}(\pi v)[\xi | \tau : \rho] \) for all \( v \in \text{dom } \pi \), where \( \rho \) is any relation in \( \text{PER} \). Then the induction hypothesis gives \( \langle [e] \eta, [e] \eta' \rangle \in \mathcal{P}(\theta)[\xi | \tau : \rho] \), and since this holds for any \( \rho \in \text{PER} \),

\[
\langle [e] \eta, [e] \eta' \rangle \in \bigcap_{\rho \in \text{PER}} \mathcal{P}(\theta)[\xi | \tau : \rho] = \mathcal{P}(\forall \tau. \theta) \xi.
\]

On the other hand, suppose the rule used in the final step is the rule for \( \forall \) elimination ((17.4) in Section 17.1),

\[
\frac{\pi \vdash e : \forall \tau. \theta}{\pi \vdash e : (\theta/\tau \rightarrow \theta')}.
\]

Assume \( \langle \eta v, \eta' v \rangle \in \mathcal{P}(\pi v) \xi \) for all \( v \in \text{dom } \pi \). Then the induction hypothesis gives

\[
\langle [e] \eta, [e] \eta' \rangle \in \mathcal{P}(\forall \tau. \theta) \xi = \bigcap_{\rho \in \text{PER}} \mathcal{P}(\theta)[\xi | \tau : \rho].
\]

The intersection here contains \( \mathcal{P}(\theta)[\xi | \tau : \mathcal{P}(\theta') \xi] \), which equals \( \mathcal{P}(\theta/\tau \rightarrow \theta') \xi \) by the finite substitution theorem. Thus \( \langle [e] \eta, [e] \eta' \rangle \in \mathcal{P}(\theta/\tau \rightarrow \theta') \xi \).

**End of Proof**
Bibliographic Notes

The intuitive concept of polymorphism is due to Strachey [1967]. The polymorphic typed lambda calculus (called the “second-order typed lambda calculus” or “System F” by logicians) was devised by Girard [1971; 1972], who showed that, in the absence of explicit recursion, all expressions reduce to normal form. (In fact, Girard showed “strong normalization”—that there are no infinite reduction sequences, regardless of any evaluation strategy.) The language was rediscovered independently, and connected to Strachey’s concept of polymorphism, by Reynolds [1974]. Girard’s development also included the existential types that we will discuss in Section 18.2, and an extension called the “ω-order typed lambda calculus” or “System Fω” that permits quantification over higher “kinds” (such as the kind of type constructors, which can be viewed as functions from types to types).

The representation of lists in Section 17.2 is a special case of the representation of anarchic algebras that was discovered independently by Leivant [1983] and Böhm and Berarducci [1985].

The undecidability of type inference for the full polymorphic typed lambda calculus was shown by Wells [1994; 1998]. In contrast, if one permits the definition of polymorphic functions but prohibits them as arguments to other functions, as in Standard ML [Milner, Tofte, and Harper, 1990; Milner and Tofte, 1991], then type inference can be performed by the Hindley-Milner algorithm [Milner, 1978; Damas and Milner, 1982]. Type inference for more general sublanguages of the polymorphic calculus is discussed by Giannini and Ronchi della Rocca [1994].

Extrinsic models of polymorphism, based on partial equivalence relations, have been developed by Girard [1972], Troelstra [1973a], and, more recently, by Longo and Moggi [1991], Pitts [1987], Freyd and Scedrov [1987], Hyland [1988], Breazutannen and Coquand [1988], and Mitchell [1986]. A variety of domain-based intrinsic models have also been devised; in the earliest, McCracken [1979] interpreted types as closures of a universal domain. She later used finitary retractions in place of closures [McCracken, 1982], while Amadio, Bruce, and Longo [1986] used finitary projections. More recently, Girard [1986] used qualitative domains and stable functions. Other domain-theoretic intrinsic models have been proposed by Coquand, Gunter, and Winskel [1988; 1989].

The general concept of what constitutes a model of the polymorphic calculus has been explicated using category theory by Seely [1987] and Ma and Reynolds [1992]. The impossibility of any model that extends the set-theoretic model of the simply typed lambda calculus has been shown by Reynolds and Plotkin [1993].

When Strachey introduced polymorphism, he distinguished between ad hoc polymorphic functions, which can behave in arbitrarily different ways for different types, and parametric polymorphic functions, which behave uniformly for all types. Reynolds [1983] formalized parametricity in terms of Plotkin’s [1973] con-
cept of logical relations, and Wadler [1989] showed that this formalization can be used to prove a variety of useful properties of parametric polymorphic functions.

Although only parametric functions can be defined in the polymorphic lambda calculus, most of the semantic models of the language include ad hoc functions as well. However, a purely parametric extrinsic semantics, using partial equivalence relations on natural numbers, has been given by Bainbridge, Freyd, Scedrov, and Scott [1990]. On the other hand, although Ma and Reynolds [1992] have managed to define what it would mean for an arbitrary intrinsic semantics to be parametric, no purely parametric domain-theoretic semantics is known.

For a selection of research papers and further references about the polymorphic lambda calculus, see Huet [1990].

In Cardelli and Wegner [1985], polymorphism and subtyping are combined in a language called "bounded fun" (or $F_\leq$) by introducing bounded quantification over all subtypes of a specified type. The coherence of this language was shown by Breazu-Tannen, Coquand, Gunter, and Scedrov [1991] and by Curien and Ghelli [1992]; the latter also gave a semidecision procedure for type checking. However, Ghelli [1991; 1995] gave a complex example showing that this procedure does not always terminate, and Pierce [1992] showed that there is no full decision procedure for type checking. A further extension of $F_\leq$ (called $F_\land$) that includes intersection types was also investigated by Pierce [1997].

Although the discussion of polymorphism in this book has been limited to purely functional languages with normal-order evaluation, there has been considerable research on the interaction of polymorphism with features of eager-evaluation and Iswim-like languages, including assignment to references [Tofte, 1990; Harper, 1994] and continuations as values [Harper, Duba, and MacQueen, 1993b]. Interactions with the transformation from direct to continuation semantics have been considered by Harper and Lillibridge [1993b; 1993a].

In the implementation of early typed languages, type information was used to determine the representation of values used by the compiled program. A naïve view of polymorphism seems to prevent this usage: Since a polymorphic function may accept arguments of an infinite variety of types, all values must be represented in a uniform way (say as an address). Indeed, current compilers for languages such as SML discard type information after the type-checking phase and compile the same kind of target code as one would for a untyped language. Recently, however, it has been found that polymorphism does not preclude the type-directed selection of data representations [Harper and Morrisett, 1995; Morrisett, 1995], which can provide a substantial improvement in the efficiency of polymorphic languages.

Primitive recursion is discussed in texts by Mendelson [1979, Section 3.3], Davis and Weyuker [1983, Chapter 3], and Andrews [1986, Section 65]. Mendelson [1979, Section 5.3, Exercise 5.16, Part (11)] and Davis and Weyuker [1983, Section 13.3] give proofs that Ackermann's function is not primitive recursive.
Exercises

17.1 Find an expression $e$ and type expressions $\theta_0$, $\theta_1$, $\theta_2$, $\theta_3$, $\theta_4$, and $\theta_5$ that make the following three typing judgements of the explicitly typed polymorphic lambda calculus true. For each judgement, give a formal proof of the judgement using the rules of Chapter 15 (in their explicitly typed versions) and Section 17.1.

$$
\vdash e : \forall \alpha. \alpha \rightarrow \alpha
$$

$$
\vdash \lambda x_{\forall \alpha}. \alpha \rightarrow \alpha \cdot x[\theta_0] (x[\theta_1]) : \theta_2
$$

$$
\vdash \lambda x_{\forall \alpha}. \alpha \rightarrow \alpha \cdot x[\theta_3] x : \theta_4
$$

$$
\vdash \lambda m_{\text{nat}} \cdot \lambda n_{\text{nat}} \cdot \Lambda \alpha. n[\alpha \rightarrow \alpha] (m[\alpha]) : \theta_5.
$$

In the last case, $\text{nat} \overset{\text{def}}{=} \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. (Note that $\alpha$ is a type variable of the object language, while the $\theta_i$ are metavariables denoting type expressions.)

17.2 Let $e$ and the $\theta_i$ be the expression and types found in Exercise 17.1. Give a reduction sequence to normal form, using the $\beta$-reduction for type applications described in Section 17.1 as well as ordinary $\beta$-reduction, for each of the following expressions:

$$
(\lambda x_{\forall \alpha}. \alpha \rightarrow \alpha \cdot x[\theta_0] (x[\theta_1])) e
$$

$$
(\lambda x_{\forall \alpha}. \alpha \rightarrow \alpha \cdot x[\theta_3] x) e.
$$

17.3 Give a formal proof of the following typing judgement, using the inference rules of Chapter 15 (in their explicitly typed versions) and Section 17.1.

mapcar: $\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta$

$$
\vdash \lambda f_{\text{int} \rightarrow \text{int}} \cdot \text{mapcar}[\text{list int}][\text{list int}](\text{mapcar}[\text{int}][\text{int}] f)
$$

: $(\text{int} \rightarrow \text{int}) \rightarrow \text{list} (\text{list int}) \rightarrow \text{list} (\text{list int})$.

17.4 In typing rule (17.1) (and also (17.3)) in Section 17.1, the proviso "When $\tau$ does not occur free in any type expression in $\pi$" is necessary for type soundness. To demonstrate this:

(a) Use rule (17.1) without this proviso to prove the spurious judgement

$$
\vdash \Lambda \alpha. \lambda x_{\alpha}. \Lambda \alpha. x : \forall \alpha. (\alpha \rightarrow \forall \alpha. \alpha).
$$

(b) Construct an expression that reduces to $\text{true}$ but can be proved, assuming the spurious judgement in part (a), to have type $\text{int}$.
17.5 Consider the definition of eqlist in terms of reduce described in Exercise 11.9.

(a) Convert the definition of this function into the explicitly typed polymorphic language of this chapter. Here eqlist should not be polymorphic, but reduce should be polymorphic, with the typing given in Section 17.1.

(b) Generalize your definition to that of a polymorphic function called geneqlist, which accepts a type \( \alpha \) and a function of type \( \alpha \to \alpha \to \text{bool} \) that compares values of type \( \alpha \) for equality, and which returns a function of type \( \text{list} \ \alpha \to \text{list} \ \alpha \to \text{bool} \) that compares two lists with elements of type \( \alpha \) for equality.

(c) Use the function defined in part (b) to define a typed function that compares two lists of lists of integers for equality.

17.6 Using the definition in Exercise 11.8, give and prove a polymorphic typing judgement for the function lazyreduce which is general enough that this function can be applied to a list with any type of elements and its result (after application to \( x \), \( f \), \( a \), and the dummy argument \( \langle \rangle \)) can have any type.

17.7 Using the representation of natural numbers and boolean values in Section 17.2, define the predicate iszero of type \( \text{nat} \to \text{bool} \).

17.8 Using the representation of natural numbers in Section 17.2, give a non-recursive definition of the function that maps a natural number \( n \) into \( n \div 2 \).

*Hint* Define a subsidiary function that maps the number \( n \) into the pair \( \langle n \div 2, (n + 1) \div 2 \rangle \).

17.9 Suppose we say that \( f : \text{list} \ \theta \to \theta' \) is a primitive recursive function on lists if it satisfies

\[
f (\text{nil}) = c \quad \text{and} \quad f (i :: r) = h \ i \ r \ (f \ r),
\]

where \( c : \theta' \) and \( h : \theta \to \text{list} \ \theta \to \theta' \to \theta' \) are defined nonrecursively in terms of previously defined primitive recursive functions. Using the kind of polymorphic programming in Section 17.2, where a list is its own reduce function, give a nonrecursive definition of \( f \) in terms of \( c \) and \( h \).

*Hint* The definition of insertapp at the end of Section 17.2 is a special example.
17.10 Unlike the version of merge defined in Section 11.5, the version in Section 17.2 does not eliminate duplicates between the lists being merged. Alter the definition of insertapp to remove this discrepancy.

17.11 In the kind of polymorphic programming described in Section 17.2, binary trees can be regarded as their own reduction functions (i.e. redtree in Exercise 11.7), just as with lists. Following this approach, define the polymorphic type tree and two functions for constructing trees: term, which maps an integer \( n \) into the tree consisting of a terminal node labelled with \( n \), and combine, which maps two trees into the tree whose main node is a nonterminal node with the two arguments as subtrees.

17.12 Generalize Exercise 15.9 to include the inference rules for polymorphism given in Section 17.1. You will need to use a definition of the substitution operation \(-/\delta\) where bound type variables are renamed to avoid the capture of free type variables.

17.13 The extrinsic semantics given in Section 17.3 is compatible with that of subtyping and intersection types given in Section 16.4 if one generalizes Proposition 16.2 to encompass type variables:

\[
\text{If } \theta \leq \theta', \text{ then } \forall \xi \in \text{PER}^{(\text{tvar})}. \mathcal{P}(\theta)\xi \subseteq \mathcal{P}(\theta')\xi. 
\]

Show that this generalized proposition remains true if one introduces the subtyping rule

\[
\forall \tau. \theta \leq \forall \tau_0. \cdots \forall \tau_{n-1}. (\theta/\tau \rightarrow \theta'),
\]

where \( \tau_0, \ldots, \tau_{n-1} \notin \text{FTV}(\theta) - \{\tau\} \).

Notice that one can derive the implicit rule for \( \forall \) elimination ((17.4) in Section 17.1) from the subsumption rule ((16.1) in Section 16.1) and the special case of the above rule where \( n = 0 \). This kind of combination of polymorphism and subtyping has been investigated by Mitchell [1988] and Giannini [1988].