Chapter 1

Polymorphism

In programming language theory, *polymorphism* (where *poly* means "many" and *morph* "shape") refers to the mechanism by which the same piece of code can be reused for different types of objects. C++ templates are a good example of a language construct for polymorphism: the same C++ template can be instantiated to different classes which operate on different types of objects all in a uniform way. The recent version of Java (J2SE 5.0) also supports *generics* which provides polymorphism in a similar way to C++ templates.

There are two kinds of polymorphism: *parametric polymorphism* and *ad hoc polymorphism*. Parametric polymorphism enables us to write a piece of code that operates on *all* types of objects all in a *uniform* way. Such a piece of code provides a high degree of generality by accepting all types of objects, but cannot exploit specific properties of these objects.[1] *Ad hoc* polymorphism, in contrast, allows a piece of code to exhibit different behavior depending on the type of objects it operates on. The operator + of SML is an example of *ad hoc* polymorphism: both `int * int -> int` and `real * real -> real` are valid types for `+`, which manipulates integers and floating point numbers differently. In this chapter, we restrict ourselves to parametric polymorphism.

We begin with *System F*, an extension of the untyped λ-calculus with polymorphic types. Despite its syntactic simplicity and rich expressivity, System F is not a good framework for practical functional languages because the problem of assigning a polymorphic type (of System F) to an expression in the untyped λ-calculus is undecidable (*i.e.*, there is no algorithm for solving the problem for all input expressions). We will then take an excursion to the *predicate polymorphic λ-calculus*, another extension of the untyped λ-calculus with polymorphic types which is a sublanguage of System F and is thus less expressive than System F. (Interestingly it uses slightly more complex syntax.) Our study of polymorphism will culminate in the formulation of the polymorphic type system of SML, called *let-polymorphism*, which is a variant of the type system of the predicate polymorphic λ-calculus. Hence the study of System F is our first step toward the polymorphic type system of SML!

1.1 System F

Consider a λ-abstraction $\lambda x.\, x$ of the untyped λ-calculus. We wish to extend the definition of the untyped λ-calculus so that we can assign a type to $\lambda x.\, x$. Assigning a type to $\lambda x.\, x$ involves two tasks: binding variable x to a type and deciding the type of the resultant expression.

In the case of the simply typed λ-calculus, we have to choose a specific type for x, say, bool. Then the resultant λ-abstraction $\lambda x{:}\mathsf{bool}.\, x$ has type $\mathsf{bool} \rightarrow \mathsf{bool}$. Ideally, however, we do not want to stipulate a specific type for x because $\lambda x.\, x$ is an identity function which works for any type. For example, $\lambda x.\, x$ is an identify function for an integer type int, but once the type of $\lambda x.\, x$ is fixed as $\mathsf{bool} \rightarrow \mathsf{bool}$, we cannot use it

[1]Generics in the Java language does not fully support parametric polymorphism: it accepts only Java objects (of class `Object`), and does not accept primitive types such as `int`.

for integers. Hence a better answer would be to bind variable x to an "any type" α and assign type $\alpha \rightarrow \alpha$ to $\lambda x : \alpha.\, x$.

Now every variable in the untyped λ-calculus is assigned an "any type," and there arises a need to distinguish between different "any types." As an example, consider a λ-abstraction $\lambda x.\, \lambda y.\, (x, y)$ where (x, y) denotes a pair of x and y. Since both x and y may assume an "any type," we could assign the same "any type" to x and y as follows:

$$\lambda x : \alpha.\, \lambda y : \alpha.\, (x, y) : \alpha \rightarrow \alpha \rightarrow \alpha \times \alpha$$

Although it is fine to assign the same "any type" to both x and y, it does not give the most general type for $\lambda x.\, \lambda y.\, (x, y)$ because x and y do not have to assume the same type in general. Instead we need to assign a different "any type" β to y so that x and y remain independent of each other:

$$\lambda x : \alpha.\, \lambda y : \beta.\, (x, y) : \alpha \rightarrow \beta \rightarrow \alpha \times \beta$$

Since each variable in a given expression may need a fresh "any type," we introduce a new construct $\lambda \alpha.\, e$, called a *type abstraction*, for declaring α as a fresh "any type," or a fresh *type variable*. That is, a type abstraction $\lambda \alpha.\, e$ declares a type variable α for use in expression e; we may rename α in a way analogous to α-conversions on λ-abstractions. If e has type A, then $\lambda \alpha.\, e$ is assigned a polymorphic type $\forall \alpha.A$ which reads *"for all* α*,* A*."* Note that A in $\forall \alpha.A$ may use α (*e.g.*, $A = \alpha \rightarrow \alpha$). Then $\lambda x.\, \lambda y.\, (x, y)$ is converted to the following expression:

$$\lambda \alpha.\, \lambda \beta.\, \lambda x : \alpha.\, \lambda y : \beta.\, (x, y) : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta$$

$\rightarrow$ has a higher operator precedence than $\forall$, so $\forall \alpha.A \rightarrow B$ is equal to $\forall \alpha.(A \rightarrow B)$, not $(\forall \alpha.A) \rightarrow B$. Hence $\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta$ is equal to $\forall \alpha. \forall \beta.(\alpha \rightarrow \beta \rightarrow \alpha \times \beta)$.

Back to the example of the identity function which is now written as $\lambda \alpha.\, \lambda x : \alpha.\, x$ of type $\forall \alpha. \alpha \rightarrow \alpha$, let us apply it to a boolean truth true of type bool. First we need to convert $\lambda \alpha.\, \lambda x : \alpha.\, x$ to an identity function $\lambda x : \mathsf{bool}.\, x$ by instantiating α to a specific type bool. To this end, we introduce a new construct $e\, [\![A]\!]$, called a *type application*, such that $(\lambda \alpha.\, e)\, [\![A]\!]$ reduces to $[A/\alpha]e$ which substitutes A for α in e:

$$(\lambda \alpha.\, e)\, [\![A]\!] \mapsto [A/\alpha]e$$

(Thus the only difference of a type application $e\, [\![A]\!]$ from an ordinary application $e_1\ e_2$ is that a type application substitutes a type for a type variable instead of an expression for an ordinary variable.) Then $(\lambda \alpha.\, \lambda x : \alpha.\, x)\, [\![\mathsf{bool}]\!]$ reduces to an identity function specialized for type bool, and an ordinary application $(\lambda \alpha.\, \lambda x : \alpha.\, x)\, [\![\mathsf{bool}]\!]\ \mathsf{true}$ finishes the job.

System F is essentially an extension of the untyped λ-calculus with type abstractions and type applications. Although variables in λ-abstractions are always annotated with their types, we do not consider System F as an extension of the simply typed λ-calculus because System F does not have to assume base types. The abstract syntax for System F is as follows:

$$\begin{array}{rrcl}
\mathsf{type} & A & ::= & A \rightarrow A \mid \alpha \mid \forall \alpha.A \\
\mathsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \lambda \alpha.\, e \mid e\, [\![A]\!] \\
\mathsf{value} & v & ::= & \lambda x : A.\, e \mid \lambda \alpha.\, e
\end{array}$$

The reduction rules for type applications are analogous to those for ordinary applications except that there is no reduction rule for types:

$$\frac{e \mapsto e'}{e\, [\![A]\!] \mapsto e'\, [\![A]\!]}\ \mathit{Tlam} \qquad \frac{}{(\lambda \alpha.\, e)\, [\![A]\!] \mapsto [A/\alpha]e}\ \mathit{Tapp}$$

$[A/\alpha]e$ substitutes A for α in e; we omit its pedantic definition here. For ordinary applications, we reuse the reductions rules for the simply typed λ-calculus.

There are two important observations to make about the abstract syntax. First the syntax for type applications implies that type variables may be instantiated to all kinds of types, including even polymorphic types. For example, the identity function $\lambda\alpha.\,\lambda x\!:\!\alpha.\,x$ may be applied to its own type $\forall\alpha.\alpha\!\to\!\alpha$! Such flexibility in type applications is the source of rich expressivity of System F, but on the other hand, it also makes System F a poor choice as a framework for practical functional languages. Second we define a type abstraction $\lambda\alpha.\,e$ as a value, even though it appears to be computationally equivalent to e.

As an example, let us write a function compose for composing two functions. One approach is to require that all type variables in the type of compose be instantiated before producing a λ-abstraction:

$$\begin{array}{lcl} \mathsf{compose} & : & \forall\alpha.\forall\beta.\forall\gamma.(\alpha\!\to\!\beta)\!\to\!(\beta\!\to\!\gamma)\!\to\!(\alpha\!\to\!\gamma) \\ \mathsf{compose} & = & \lambda\alpha.\,\lambda\beta.\,\lambda\gamma.\,\lambda f\!:\!\alpha\!\to\!\beta.\,\lambda g\!:\!\beta\!\to\!\gamma.\,\lambda x\!:\!\alpha.\,g\ (f\ x) \end{array}$$

Alternatively we may require that only the first two type variables α and β be instantiated before producing a λ-abstraction which returns a type abstraction expecting the third type variable γ:

$$\begin{array}{lcl} \mathsf{compose} & : & \forall\alpha.\forall\beta.(\alpha\!\to\!\beta)\!\to\!\forall\gamma.(\beta\!\to\!\gamma)\!\to\!(\alpha\!\to\!\gamma) \\ \mathsf{compose} & = & \lambda\alpha.\,\lambda\beta.\,\lambda f\!:\!\alpha\!\to\!\beta.\,\lambda\gamma.\,\lambda g\!:\!\beta\!\to\!\gamma.\,\lambda x\!:\!\alpha.\,g\ (f\ x) \end{array}$$

As for the type system, System F is *not* a straightforward extension of the simply typed λ-calculus because of the inclusion of type variables. In the simply typed λ-calculus, $x : A$ qualifies as a valid type binding regardless of type A, and the order of type bindings in a typing context does not matter by Proposition **??**. In System F, $x : A$ may not qualify as a valid type binding if type A may contain type variables, which becomes valid types only when declared in type abstractions. For example, without type abstractions declaring type variables α and β, we may not use $\alpha\!\to\!\beta$ as a type and hence $x : \alpha\!\to\!\beta$ is not a valid type binding. This observation leads to the conclusion that in System F, a typing context consists not only of type bindings but also of a new form of declarations for indicating which type variables are valid and which are not; moreover the order of elements in a typing context now *does* matter because of type variables.

We define a typing context as an ordered set of type bindings and *type declarations*; a type declaration α type declares α as a type, or equivalently, α as a valid type variable:

$$\text{typing context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \mathsf{type}$$

We simplify the presentation by assuming that variables and type variables in a typing context are all distinct. We consider a type variable α as valid only if its type declaration appears to its left. For example, $\Gamma_1, \alpha\ \mathsf{type}, x : \alpha\!\to\!\alpha$ is a valid typing context because α in $x : \alpha\!\to\!\alpha$ has been declared as a type variable in $\alpha\ \mathsf{type}$ (provided that Γ_1 is also a valid typing context). $\Gamma_1, x : \alpha\!\to\!\alpha, \alpha\ \mathsf{type}$ is, however, not a valid typing context.

The type system of System F uses two forms of judgments: a typing judgment $\Gamma \vdash e : A$ whose meaning is the same as in the simply typed λ-calculus, and a *type judgment* $\Gamma \vdash A\ \mathsf{type}$ which means that A is a valid type under typing context Γ.[2] We need type judgments because the definition of syntactic category type in the abstract syntax is incapable of differentiating valid type variables from invalid ones. We refer to an inference rule deducing a type judgment as a *type rule*.

The type system of System F uses the following rules in addition to the typing rules Var, →I, and →E:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash A\!\to\!B\ \mathsf{type}}\ \mathsf{Ty}\!\to \qquad \frac{\alpha\ \mathsf{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \mathsf{TyVar} \qquad \frac{\Gamma, \alpha\ \mathsf{type} \vdash A\ \mathsf{type}}{\Gamma \vdash \forall\alpha.A\ \mathsf{type}}\ \mathsf{Ty}\forall$$

$$\frac{\Gamma, \alpha\ \mathsf{type} \vdash e : A}{\Gamma \vdash \lambda\alpha.\,e : \forall\alpha.A}\ \forall\mathsf{I} \qquad \frac{\Gamma \vdash e : \forall\alpha.B \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash e\,[\![A]\!] : [A/\alpha]B}\ \forall\mathsf{E}$$

Note that a proof of a type judgment $\Gamma \vdash A\ \mathsf{type}$ does not use type bindings in Γ. The rule ∀I, called the ∀ Introduction rule, introduces a polymorphic type $\forall\alpha.A$ from the judgment in the premise. The rule ∀E, called the ∀ Elimination rule, eliminates a polymorphic type $\forall\alpha.B$ by substituting a valid type A for type

[2] A type judgment $\Gamma \vdash A\ \mathsf{type}$ is also an example of a hypothetical judgment which deduces a "judgment" A type using each "judgment" A_i type in Γ as a hypothesis.

variable α. Note that the typing rule $\to$E uses a substitution of a type into another *type* where as the reduction rule *Tapp* uses a substitution of a type into an *expression*.

As an example of a typing derivation, let us decide the type of an identity function specialized for type bool; we assume that $\Gamma \vdash \mathsf{bool}\ \mathsf{type}$ holds for any typing context Γ (see the type rule TyBool below):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha\ \mathsf{type}, x : \alpha \vdash x : \alpha}\ \mathsf{Var}}{\alpha\ \mathsf{type} \vdash \lambda x{:}\alpha.\, x : \alpha \to \alpha}\ {\to}\mathsf{I}}{\cdot \vdash \lambda\alpha.\, \lambda x{:}\alpha.\, x : \forall\alpha.\alpha \to \alpha}\ \forall\mathsf{I} \qquad \dfrac{}{\cdot \vdash \mathsf{bool}\ \mathsf{type}}\ \mathsf{TyBool}}{\cdot \vdash (\lambda\alpha.\, \lambda x{:}\alpha.\, x)\, [\![\mathsf{bool}]\!] : [\mathsf{bool}/\alpha](\alpha \to \alpha)}$$

Since $[\mathsf{bool}/\alpha](\alpha \to \alpha)$ is equal to $\mathsf{bool} \to \mathsf{bool}$, the type application has type $\mathsf{bool} \to \mathsf{bool}$.

The proof of type safety of System F needs three substitution lemmas as there are three kinds of substitutions: $[A/\alpha]B$ for the rule $\forall$E, $[A/\alpha]e$ for the rule *Tapp*, and $[e'/x]e$ for the rule *App*. We write $[A/\alpha]\Gamma$ for substituting A for α in all type bindings in Γ.

Lemma 1.1 (Type substitution into types).
If $\Gamma \vdash A\ \mathsf{type}$ *and* $\Gamma, \alpha\ \mathsf{type}, \Gamma' \vdash B\ \mathsf{type}$, *then* $\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]B\ \mathsf{type}$.

Lemma 1.2 (Type substitution into expressions).
If $\Gamma \vdash A\ \mathsf{type}$ *and* $\Gamma, \alpha\ \mathsf{type}, \Gamma' \vdash e : B$, *then* $\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]e : [A/\alpha]B$.

Lemma 1.3 (Expression substitution).
If $\Gamma \vdash e : A$ *and* $\Gamma, x : A, \Gamma' \vdash e' : C$, *then* $\Gamma, \Gamma' \vdash [e/x]e' : C$.

In Lemmas 1.1 and 1.2, we have to substitute A into Γ', which may contain types involving α. In Lemma 1.2, we have to substitute A into e and B, both of which may contain types involving α. Lemma 1.3 reflects the fact that typing contexts are ordered sets.

The proof of type safety of System F is similar to the proof for the simply typed λ-calculus. We need to extend the canonical forms lemma (Lemma **??**) and the inversion lemma (Lemma **??**):

Lemma 1.4 (Canonical forms).
If v is a value of type $\forall\alpha.A$, then v is a type abstraction $\lambda\alpha.\, e$.

Lemma 1.5 (Inversion). *Suppose* $\Gamma \vdash e : C$.
If $e = \lambda\alpha.\, e'$, *then* $C = \forall\alpha.A$ *and* $\Gamma, \alpha\ \mathsf{type} \vdash e' : A$.

Theorem 1.6 (Progress). *If $\cdot \vdash e : A$ for some type A, then either e is a value or there exists e' such that $e \mapsto e'$.*

Theorem 1.7 (Type preservation). *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$, *then* $\Gamma \vdash e' : A$.

1.2 Type reconstruction

The type systems of both the simply typed λ-calculus and System F require that all variables in λ-abstractions be annotated with their types. While it certainly simplifies the proof of type safety (and the study of type-theoretic properties in general), such a requirement on variables is not a good idea when it comes to designing practical functional languages. One reason is that annotating all variables with their types does not always improve code readability. On the contrary, excessive type annotations often reduces code readability! For example, one would write an SML function adding two integers as `fn x => fn y => x + y`, which is no less readable than a fully type-annotated function `fn x : int => fn y : int => x + y`. A more important reason is that in many cases, types of variables can be inferred, or *reconstructed*, from the context. For example, the presence of `+` in `fn x => fn y => x + y` gives enough information to decide a unique type `int` for both `x` and `y`. Thus we wish to eliminate such a requirement on variables, so as to provide programmers with more flexibility in type annotations, by developing a *type reconstruction* algorithm which automatically infers types for variables.

In the case of System F, the problem of type reconstruction is to convert an expression e in the untyped λ-calculus to a well-typed expression e' in System F such that erasing type annotations (including type abstractions and type applications) in e' yields the original expression e. That is, by reconstructing types for all variables in e, we obtain a new well-typed expression e' in System F. Formally we define an *erasure* function $erase(\cdot)$ which takes an expression in System F and erases all type annotations in it:

$$\begin{array}{lcl} erase(x) & = & x \\ erase(\lambda x{:}A.\,e) & = & \lambda x.\,erase(e) \\ erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\ erase(\lambda\alpha.\,e) & = & erase(e) \\ erase(e\ [\![A]\!]) & = & erase(e) \end{array}$$

The erasure function respects the reduction rules for System F in the following sense:

Proposition 1.8. *If $e \mapsto e'$ holds in System F, then $erase(e) \mapsto^* erase(e')$ holds in the untyped λ-calculus.*

The problem of type reconstruction is then to convert an expression e in the untyped λ-calculus to a well-typed expression e' in System F such that $erase(e') = e$. We say that an expression e in the untyped λ-calculus is *typable* in System F if there exists such a well-typed expression e'.

As an example, let us consider an untyped λ-abstraction $\lambda x.\,x\ x$. It is not typable in the simply typed λ-calculus because the first x in $x\ x$ must have a type strictly larger than the second x, which is impossible. It is, however, typable in System F because we can replace the first x in $x\ x$ by a type application. Specifically $\lambda x{:}\forall\alpha.\alpha\to\alpha.\,x\ [\![\forall\alpha.\alpha\to\alpha]\!]\ x$ is a well-typed expression in System F which erases to $\lambda x.\,x\ x$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash x:\forall\alpha.\alpha\to\alpha}\,\mathsf{Var}\quad \Gamma\vdash\forall\alpha.\alpha\to\alpha\ \mathsf{type}}{\Gamma\vdash x\ [\![\forall\alpha.\alpha\to\alpha]\!] : (\forall\alpha.\alpha\to\alpha)\to(\forall\alpha.\alpha\to\alpha)}\,\forall\mathsf{E}\quad \dfrac{}{\Gamma\vdash x:\forall\alpha.\alpha\to\alpha}\,\mathsf{Var}}{\Gamma\vdash x\ [\![\forall\alpha.\alpha\to\alpha]\!]\ x:\forall\alpha.\alpha\to\alpha\quad \mathit{where}\quad \Gamma = x:\forall\alpha.\alpha\to\alpha}\,{\to}\mathsf{E}}{\cdot\vdash\lambda x{:}\forall\alpha.\alpha\to\alpha.\,x\ [\![\forall\alpha.\alpha\to\alpha]\!]\ x : (\forall\alpha.\alpha\to\alpha)\to(\forall\alpha.\alpha\to\alpha)}\,{\to}\mathsf{I}$$

The proof of $x:\forall\alpha.\alpha\to\alpha\vdash\forall\alpha.\alpha\to\alpha$ type is shown below:

$$\dfrac{\dfrac{\dfrac{\alpha\ \mathsf{type}\in x:\forall\alpha.\alpha\to\alpha,\alpha\ \mathsf{type}}{x:\forall\alpha.\alpha\to\alpha,\alpha\ \mathsf{type}\vdash\alpha\ \mathsf{type}}\,\mathsf{TyVar}\quad \dfrac{\alpha\ \mathsf{type}\in x:\forall\alpha.\alpha\to\alpha,\alpha\ \mathsf{type}}{x:\forall\alpha.\alpha\to\alpha,\alpha\ \mathsf{type}\vdash\alpha\ \mathsf{type}}\,\mathsf{TyVar}}{x:\forall\alpha.\alpha\to\alpha,\alpha\ \mathsf{type}\vdash\alpha\to\alpha\ \mathsf{type}}\,\mathsf{Ty}{\to}}{x:\forall\alpha.\alpha\to\alpha\vdash\forall\alpha.\alpha\to\alpha\ \mathsf{type}}\,\mathsf{Ty}\forall$$

Hence a type reconstruction algorithm for System F, *if any*, would convert $\lambda x.\,x\ x$ to $\lambda x{:}\forall\alpha.\alpha\to\alpha.\,x\ [\![\forall\alpha.\alpha\to\alpha]\!]\ x$.

It turns out that not every expression in the untyped λ-calculus is typable in System F. For example, omega $= (\lambda x.\,x\ x)\ (\lambda x.\,x\ x)$ is not typable: there is no well-typed expression in System F that erases to omega. The proof exploits the *normalization* property of System F which states that the reduction of a well-typed expression in System F always terminates. Thus a type reconstruction algorithm for System F first decides if a given expression e is typable or not in System F; if e is typable, the algorithm yields a corresponding expression in System F.

Unfortunately the problem of type reconstruction in System F is undecidable: there is no algorithm for deciding whether a given expression in the untyped λ-calculus is typable or not in System F. Our plan now is to find a compromise between rich expressivity and decidability of type reconstruction — we wish to identify a sublanguage of System F that supports polymorphic types and also has a decidable type construction algorithm. Section 1.4 presents such a sublanguage, called the *predicative polymorphic λ-calculus*, which is extended to the polymorphic type system of SML in Section 1.5.

1.3 Programming in System F

We have seen in Section **??** how to encode common datatypes in the untyped λ-calculus. While these expressions correctly encode their respective datatypes, unavailability of a type system makes it difficult to express the intuition behind the encoding of each datatype. Besides it is often tedious and even unreliable to check the correctness of an encoding without recourse to a type system.

In this section, we rewrite these untyped expressions as well-typed expressions in System F. A direct definition of a datatype in terms of types in System F provides the intuition behind its encoding, and availability of type annotations within expressions makes it easy to check the correctness of the encoding.

Let us begin with base types bool and nat for Church booleans and Church numerals, respectively. The intuition behind Church booleans is that a boolean value chooses one of two different options. The following definition of the base type bool is based on the decision to assign the same type α to both options:

$$\mathsf{bool} \;=\; \forall\alpha.\alpha\to\alpha\to\alpha$$

Then boolean values true and false, both of type bool, are encoded as follows:

$$\begin{array}{rcl} \mathsf{true} & = & \lambda\alpha.\,\lambda t\!:\!\alpha.\,\lambda f\!:\!\alpha.\,t \\ \mathsf{false} & = & \lambda\alpha.\,\lambda t\!:\!\alpha.\,\lambda f\!:\!\alpha.\,f \end{array}$$

The intuition behind Church numerals is that a Church numeral $\widehat{n}$ takes a function f and returns another function f^n which applies f exactly n times. In order for f^n to be well-typed, its argument type and return type must be identical. Hence we define the base type nat in System F as follows:[3]

$$\mathsf{nat} \;=\; \forall\alpha.(\alpha\to\alpha)\to(\alpha\to\alpha)$$

Then a zero zero of type nat and a successor function succ of type nat $\to$ nat are encoded as follows:

$$\begin{array}{rcl} \mathsf{zero} & = & \lambda\alpha.\,\lambda f\!:\!\alpha\to\alpha.\,\lambda x\!:\!\alpha.\,x \\ \mathsf{succ} & = & \lambda n\!:\!\mathsf{nat}.\,\lambda\alpha.\,\lambda f\!:\!\alpha\to\alpha.\,\lambda x\!:\!\alpha.\,(n\;[\![\alpha]\!]\;f)\;(f\;x) \end{array}$$

The definition of a product type $A\times B$ in System F exploits the fact that in essence, a value of type $A\times B$ contains a value of type A and another value of type B. If we think of $A\to B\to\alpha$ as a type for a function taking two arguments of types A and B and returning a value of type α, a value of type $A\times B$ contains everything necessary for applying such a function, which is expressed in the following definition of $A\times B$:

$$A\times B = \forall\alpha.(A\to B\to\alpha)\to\alpha$$

Pairs and projections are encoded as follows; note that without type annotations, these expressions degenerate to pairs and projections for the untyped λ-calculus given in Section **??**:

$$\begin{array}{rclcl} \mathsf{pair} & : & \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha\times\beta & = & \lambda\alpha.\,\lambda\beta.\,\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,\lambda\gamma.\,\lambda f\!:\!\alpha\to\beta\to\gamma.\,f\;x\;y \\ \mathsf{fst} & : & \forall\alpha.\forall\beta.\alpha\times\beta\to\alpha & = & \lambda\alpha.\,\lambda\beta.\,\lambda p\!:\!\alpha\times\beta.\,p\;[\![\alpha]\!]\;(\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,x) \\ \mathsf{snd} & : & \forall\alpha.\forall\beta.\alpha\times\beta\to\beta & = & \lambda\alpha.\,\lambda\beta.\,\lambda p\!:\!\alpha\times\beta.\,p\;[\![\beta]\!]\;(\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,y) \end{array}$$

The type unit is a general product type with no element and is thus defined as $\forall\alpha.\alpha\to\alpha$ which is obtained by removing A and B from the definition of $A\times B$. The encoding of a unit () is obtained by removing x and y from the encoding of pair:

$$()\;:\;\mathsf{unit}\;=\;\lambda\alpha.\,\lambda x\!:\!\alpha.\,x$$

The definition of a sum type $A+B$ in System F reminds us of the typing rule $+\mathsf{E}$ for sum types: given a function f of type $A\to\alpha$ and another function g of type $B\to\alpha$, a value v of type $A+B$ applies the right function (either f or g) to the value contained in v:

$$A+B = \forall\alpha.(A\to\alpha)\to(B\to\alpha)\to\alpha$$

Injections and case expressions are a translation of the typing rules $+\mathsf{I_L}$, $+\mathsf{I_R}$, and $+\mathsf{E}$:

$$\begin{array}{rcl} \mathsf{inl} & : & \forall\alpha.\forall\beta.\alpha\to\alpha+\beta \\ \mathsf{inr} & : & \forall\alpha.\forall\beta.\beta\to\alpha+\beta \\ \mathsf{case} & : & \forall\alpha.\forall\beta.\forall\gamma.\alpha+\beta\to(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma \end{array}$$

[3] We may also interpret nat as $\mathsf{nat} = \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$ such that a Church numeral $\widehat{n}$ takes a successor function succ of type $\alpha\to\alpha$ and a zero zero of type α to return succ^n `zero` of type α.

Exercise 1.9. Encode inl, inr, and case in System F.

The type void is a general sum type with no element and is thus defined as $\forall\alpha.\alpha$ which is obtained by removing A and B from the definition of $A+B$. Needless to say, there is no expression of type void in System F. (Why?)

1.4 Predicative polymorphic λ-calculus

This section presents the predicative polymorphic λ-calculus which is a sublanguage of System F with a decidable type construction algorithm. It is still not a good framework for practical functional languages because polymorphic types are virtually useless! Nevertheless it helps us a lot to motivate the development of let-polymorphism, the most popular polymorphic type system found in modern functional languages.

The key observation is that undecidability of type reconstruction in System F is traced back to the self-referential nature of polymorphic types: we augment the set of types with new elements called type variables and polymorphic types, but the syntax for type applications allows type variables to range over not only existing types (such as function types) but also these new elements which include polymorphic types themselves. That is, there is no restriction on type A in a type application $e\,[\![A]\!]$ where type A, which is to be substituted for a type variable, can be not only a function type but also another polymorphic type.

The predicative polymorphic λ-calculus recovers decidability of type reconstruction by prohibiting type variables from ranging over polymorphic types. We stratify types into two kinds: *monotypes* which exclude polymorphic types and *polytypes* which include all kinds of types:

$$
\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall\alpha.U \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\;e \mid \lambda\alpha.\,e \mid e\,[\![A]\!] \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\,e \mid \lambda\alpha.\,e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \textsf{type}
\end{array}
$$

A polytype is always written as $\forall\alpha.\forall\beta.\cdots\forall\gamma.A \rightarrow A'$ where $A \rightarrow A'$ cannot contain polymorphic types. For example, $(\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\beta \rightarrow \beta)$ is not a polytype whereas $\forall\alpha.\forall\beta.(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$ is. We say that a polytype is written in *prenex form* because a *type quantifier* $\forall\alpha$ may appear only as part of its prefix.

The main difference of the predicative polymorphic λ-calculus from System F is that a type application $e\,[\![A]\!]$ now accepts only a monotype A. (In System F, there is no distinction between monotypes and polytypes, and a type application can accept polymorphic types.) A type application $e\,[\![A]\!]$ itself, however, has a polytype if e has a polytype $\forall\alpha.U$ where U is another polytype (see the typing rule $\forall\mathsf{E}$ below).

As in System F, the type system of the predicative polymorphic λ-calculus uses two forms of judgments: a typing judgment $\Gamma \vdash e : U$ and a type judgment $\Gamma \vdash A\ \mathsf{type}$. The difference is that $\Gamma \vdash A\ \mathsf{type}$ now checks if a given type is a valid *monotype*. That is, we do not use a type judgment $\Gamma \vdash U\ \mathsf{type}$ (which is actually unnecessary because every polytype is written in prenex form anyway). Thus the system system uses the following rules; note that the rule $\mathsf{Ty}\forall$ from System F is gone:

$$
\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash A \rightarrow B\ \mathsf{type}}\ \mathsf{Ty}{\rightarrow} \qquad \frac{\alpha\ \mathsf{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \mathsf{TyVar}
$$

$$
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathsf{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\,e : A \rightarrow B}\ {\rightarrow}\mathsf{I} \qquad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\;e' : B}\ {\rightarrow}\mathsf{E}
$$

$$
\frac{\Gamma, \alpha\ \mathsf{type} \vdash e : U}{\Gamma \vdash \lambda\alpha.\,e : \forall\alpha.U}\ \forall\mathsf{I} \qquad \frac{\Gamma \vdash e : \forall\alpha.U \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash e\,[\![A]\!] : [A/\alpha]U}\ \forall\mathsf{E}
$$

Unfortunately the use of a monotype A in a λ-abstraction $\lambda x\!:\!A.\,e$ defeats the purpose of introducing polymorphic types into the type system: even though we can now write an expression of a polytype U, we can never instantiate type variables in U more than once! Suppose, for example, that we wish to apply a polymorphic identity function $\mathsf{id} = \lambda\alpha.\, \lambda x\!:\!\alpha.\, x$ to two different types, say, bool and int. In the untyped λ-calculus, we would bind a variable f to an identity function and then apply f twice:

$$(\lambda f.\, \mathsf{pair}\ (f\ \mathsf{true})\ (f\ \widehat{0}))\ (\lambda x.\, x)$$

In the predicative polymorphic λ-calculus, it is impossible to reuse id more than once in this way, since f must be given a monotype while id has a polytype:

$$(\lambda f\!:\!\forall\alpha.\alpha\!\to\!\alpha.\, (f\ [\![\mathsf{bool}]\!]\ \mathsf{true}, f\ [\![\mathsf{int}]\!]\ 0))\ \mathsf{id} \qquad \text{(ill-typed)}$$

(Here we use pairs for product types.) If we apply id to a monotype bool (or int), $f\ 0$ (or f true) in the body fails to typecheck:

$$(\lambda f\!:\!\mathsf{bool}\!\to\!\mathsf{bool}.\, (f\ \mathsf{true}, f\ 0))\ (\mathsf{id}\ [\![\mathsf{bool}]\!]) \qquad \text{(ill-typed)}$$

Thus the only interesting way to use a polymorphic function (of a polytype) is to use it "monomorphically" by converting it to a function of a certain monotype!

Let-polymorphism extends the predicative polymorphic λ-calculus with a new construct that enables us to use a polymorphic expression *polymorphically* in the sense that type variables in it can be instantiated more than once. The new construct preserves decidability of type reconstruction, so let-polymorphism is a good compromise between expressivity and decidability of type reconstruction.

1.5 Let-polymorphism

Let-polymorphism extends the predicative polymorphic λ-calculus with a new construct, called a let-*binding*, for declaring variables of polytypes. A let-binding let $x = U$ in ee' binds x to a polymorphic expression e of type U and allows multiple occurrences of x in e'. With a let-binding, we can apply a polymorphic identity function to two different (mono)types bool and int as follows:

$$\mathsf{let}\ f\!:\!\forall\alpha.\alpha\!\to\!\alpha = \lambda\alpha.\, \lambda x\!:\!\alpha.\, x\ \mathsf{in}\ (f\ [\![\mathsf{bool}]\!]\ \mathsf{true}, f\ [\![\mathsf{int}]\!]\ 0))$$

Since variables can now assume polytypes, we use type bindings of the form $x : U$ instead of $x : A$. We require that let-bindings themselves be of monotypes:

$$\begin{array}{rrcl} \text{expression} & e & ::= & \cdots \mid \mathsf{let}\ x\!:\!U = e\ \mathsf{in}\ e \\ \text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \mid \Gamma, \alpha\ \mathsf{type} \end{array}$$

$$\frac{x : U \in \Gamma}{\Gamma \vdash x : U}\ \mathsf{Var} \qquad \frac{\Gamma \vdash e : U \quad \Gamma, x : U \vdash e' : A}{\Gamma \vdash \mathsf{let}\ x\!:\!U = e\ \mathsf{in}\ e' : A}\ \mathsf{Let}$$

The reduction of a let-binding let $x\!:\!U = e$ in e' proceeds by substituting e for x in e':

$$\mathsf{let}\ x\!:\!U = e\ \mathsf{in}\ e' \mapsto [e/x]e'$$

Depending on the reduction strategy, we may choose to fully evaluate e before performing the substitution.

Although let $x\!:\!U = e$ in e' reduces to the same expression that an application $(\lambda x\!:\!A.\, e')\ e$ reduces to, it is *not* syntactic sugar for $(\lambda x\!:\!A.\, e')\ e$: when e has a polytype U, let $x\!:\!U = e$ in e' may typecheck under the rule Let, but in general, $(\lambda x\!:\!A.\, e')\ e$ does not typecheck because monotype A does not match the type of e. Therefore, in order to use a polymorphic function polymorphically, we must bind it to a variable *using a* let-*binding* and then instantiate the variable as necessary.

Then why do we not just allow a λ-abstraction $\lambda x\!:\!U.\, e$ binding x to a polytype (which would degenerate let $x\!:\!U = e$ in e' into syntactic sugar)? The reason is that with an additional assumption that e may have a polytype (*e.g.*, $\lambda x\!:\!U.\, x$), such a λ-abstraction collapses the distinction between monotypes and polytypes. That is, polytypes constitute types of System F:

$$\begin{array}{rrcl} \text{monotype} & A & ::= & U \!\to\! U \mid \alpha \\ \text{polytype} & U & ::= & A \mid \forall\alpha.U \end{array} \quad\Longleftrightarrow\quad \text{type}\ \ U \ ::= \ U \!\to\! U \mid \alpha \mid \forall\alpha.U$$

We may construe a let-binding as a restricted use of a λ-abstraction $\lambda x\!:\!U.\, e$ (binding x to a polytype) such that it never stands alone as a first-class object and must be applied to a polymorphic expression *immediately*. At the cost of flexibility in applying such λ-abstractions, let-polymorphism retains decidability of type reconstruction without destroying the distinction between monotypes and polytypes and also without sacrificing too much expressivity. After all, we can still enjoy both polymorphism and decidability of type reconstruction — this is the reason why let-polymorphism is so popular among mainstream functional languages!

1.6 Implicit polymorphism

The polymorphic type systems considered so far are all "explicit" in that polymorphic types are introduced explicitly by type abstractions and type variables are instantiated explicitly by type applications. An explicit polymorphic type system has the property that every well-typed polymorphic expression has a unique polymorphic type.

The type system of SML uses a different approach to polymorphism: it makes no use of type abstractions and type applications, but allows an expression to have multiple types by requiring no type annotations in λ-abstractions. That is, polymorphic types arise "implicitly" from lack of type annotations in λ-abstractions.

As an example, consider an identity function $\lambda x.\, x$. It can be assigned such types as $\mathsf{bool} \rightarrow \mathsf{bool}$, $\mathsf{int} \rightarrow \mathsf{int}$, $(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$, and so on. These types are all distinct, but are subsumed by the same polytype $\forall \alpha.\alpha \rightarrow \alpha$ in the sense that they are results of instantiating α in $\forall \alpha.\alpha \rightarrow \alpha$. We refer to $\forall \alpha.\alpha \rightarrow \alpha$ as the *principal type* of $\lambda x.\, x$, which may be thought of as the most general type for $\lambda x.\, x$ (as opposed to specific types such as $\mathsf{bool} \rightarrow \mathsf{bool}$ and $\mathsf{int} \rightarrow \mathsf{int}$).

The type reconstruction algorithm of SML infers a unique principal type for every well-typed expression. We do not discuss details of the type reconstruction algorithm, and briefly discuss the type system of SML. In essence, the type system of SML uses let-polymorphism without type annotations in λ-abstractions, type abstractions, and type applications:

$$\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall \alpha.U \\
\textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
\textsf{value} & v & ::= & \lambda x.\, e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \mid \Gamma, \alpha\ \mathsf{type}
\end{array}$$

We use a new typing judgment $\Gamma \triangleright e : U$ to express that untyped expression e is typable with a polytype U. The intuition (which will be made clear in Theorem 1.11) is that if $\Gamma \triangleright e : U$ holds (where e is untyped), there exists a *typed* expression e' such that $\Gamma \vdash e' : U$ and e' erases to e under the following erasure function:

$$\begin{array}{lcl}
erase(x) & = & x \\
erase(\lambda x\!:\!A.\, e) & = & \lambda x.\, erase(e) \\
erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\
erase(\lambda \alpha.\, e) & = & erase(e) \\
erase(e\ [\![A]\!]) & = & erase(e) \\
erase(\mathsf{let}\ x\!:\!U = e\ \mathsf{in}\ e') & = & \mathsf{let}\ x = erase(e)\ \mathsf{in}\ erase(e')
\end{array}$$

That is, if $\Gamma \triangleright e : U$ holds, e has its counterpart in let-polymorphism in Section 1.5.

The rules for the typing judgment $\Gamma \triangleright e : U$ are as follows:

$$\frac{x : U \in \Gamma}{\Gamma \triangleright x : U}\ \mathsf{Var} \qquad \frac{\Gamma, x : A \triangleright e : B}{\Gamma \triangleright \lambda x.\, e : A \rightarrow B}\ {\rightarrow}\mathsf{I} \qquad \frac{\Gamma \triangleright e : A \rightarrow B \quad \Gamma \triangleright e' : A}{\Gamma \triangleright e\ e' : B}\ {\rightarrow}\mathsf{E}$$

$$\frac{\Gamma \triangleright e : U \quad \Gamma, x : U \triangleright e' : A}{\Gamma \triangleright \mathsf{let}\ x = e\ \mathsf{in}\ e' : A}\ \mathsf{Let}$$

$$\frac{\Gamma, \alpha \text{ type} \triangleright e : U}{\Gamma \triangleright e : \forall \alpha . U} \text{Gen} \qquad \frac{\Gamma \triangleright e : \forall \alpha . U \quad \Gamma \vdash A \text{ type}}{\Gamma \triangleright e : [A/\alpha]U} \text{Spec}$$

In the rule Gen (for generalizing a type), expression e in the conclusion plays the role of a type abstraction; in the rule Spec (for specializing a type), expression e in the conclusion plays the role of a type application. According to these rules, $\lambda x.\, x$ can be assigned $\forall \alpha . \alpha \rightarrow \alpha$:

$$\dfrac{\dfrac{\overline{\Gamma, \alpha \text{ type}, x : \alpha \triangleright x : \alpha}\ \text{Var}}{\Gamma, \alpha \text{ type} \triangleright \lambda x.\, x : \alpha \rightarrow \alpha}\ \rightarrow\mathsf{I}}{\Gamma \triangleright \lambda x.\, x : \forall \alpha . \alpha \rightarrow \alpha}\ \text{Gen}$$

By the rule Spec, then, $\lambda x.\, x$ can also be assigned such types as $\mathsf{bool} \rightarrow \mathsf{bool}$, $\mathsf{int} \rightarrow \mathsf{int}$, $(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$, and so on.

The implicit polymorphic type system of SML is connected with let-polymorphism in Section 1.5 via the following theorems:

Theorem 1.10. *If* $\Gamma \vdash e : U$, *then* $\Gamma \triangleright \mathit{erase}(e) : U$.

Theorem 1.11. *If* $\Gamma \triangleright e : U$, *then there exists a typed expression* e' *such that* $\Gamma \vdash e' : U$ *and* $\mathit{erase}(e') = e$.

1.7 Value restriction

The type system presented in the previous section is sound only if it does not interact with computational effects such as references. Since SML allows computational effects in any expression, it uses a restricted version of the type system.

To see the problem with the rule Gen, consider the following example where we assume constructs for integers, booleans, and references:

$$\begin{array}{l} \mathsf{let}\ x = \mathsf{ref}\ (\lambda y.\, y)\ \mathsf{in} \\ \mathsf{let}\ \_ = x := \lambda y.\, y + 1\ \mathsf{in} \\ (!x)\ \mathsf{true} \end{array}$$

We can assign a polytype $\forall \alpha . \mathsf{ref}\ (\alpha \rightarrow \alpha)$ to $\mathsf{ref}\ (\lambda y.\, y)$ as follows (where we ignore store typing contexts):

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma, \alpha \text{ type}, y : \alpha \triangleright y : \alpha}\ \text{Var}}{\Gamma, \alpha \text{ type} \triangleright \lambda y.\, y : \alpha \rightarrow \alpha}\ \rightarrow\mathsf{I}}{\Gamma, \alpha \text{ type} \triangleright \mathsf{ref}\ (\lambda y.\, y) : \mathsf{ref}\ (\alpha \rightarrow \alpha)}\ \text{Ref}}{\Gamma \triangleright \mathsf{ref}\ (\lambda y.\, y) : \forall \alpha . \mathsf{ref}\ (\alpha \rightarrow \alpha)}\ \text{Gen}$$

By the rule Spec, then, we can assign either $\mathsf{ref}\ (\mathsf{int} \rightarrow \mathsf{int})$ or $\mathsf{ref}\ (\mathsf{bool} \rightarrow \mathsf{bool})$ to variable x. Now both expressions $x := \lambda y.\, y + 1$ and $(!x)\ \mathsf{true}$ are well-typed, but the reduction of $(!x)\ \mathsf{true}$ must not succeed because it ends up adding a boolean truth true and an integer 1!

In order to avoid the problem arising from the interaction between polymorphism and computational effects, the type system of SML requires that expression e in the rule Gen be a *syntactic value*:

$$\frac{\Gamma, \alpha \text{ type} \triangleright v : U}{\Gamma \triangleright v : \forall \alpha . U} \text{Gen}$$

The idea is to exploit the fact that computational effects cannot interfere with the evaluation of a value, which terminates immediately anyway.

In the case that expression e in a let-binding $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is a non-value with a polytype U, we may use variable x in e' only monomorphically. For example, the first expression below typechecks while the second expression does not:

$$\begin{array}{l} \mathsf{let}\ x = (\lambda y.\, y)\ (\lambda z.\, z)\ \mathsf{in}\ x\ \mathsf{true} \\ \mathsf{let}\ x = (\lambda y.\, y)\ (\lambda z.\, z)\ \mathsf{in}\ (x\ \mathsf{true}, x\ 1) \end{array}$$