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 different types of objects. 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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-calculus, another extension of the untyped \( \lambda \)-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 \( \lambda \)-calculus. Hence the study of System F is our first step toward the polymorphic type system of SML!

1.1 System F

Consider a \( \lambda \)-abstraction \( \lambda x. x \) of the untyped \( \lambda \)-calculus. We wish to extend the definition of the untyped \( \lambda \)-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 \( \lambda \)-calculus, we have to choose a specific type for variable \( x \), say, bool. Then the resultant \( \lambda \)-abstraction \( \lambda x. \text{bool} \cdot x \) has type \( \text{bool} \rightarrow \text{bool} \). Ideally, however, we do not want to stipulate a specific type for \( x \) because \( \lambda x. x \) is an identity function that works for any type. For example, \( \lambda x. x \) is an identity function for an integer type int, but once the type of \( \lambda x. x \) is fixed as \( \text{bool} \rightarrow \text{bool} \), we cannot use it for integers. Hence a better answer would be to bind \( x \) to an “any type” \( \alpha \) and assign type \( \alpha \rightarrow \alpha \) to \( \lambda x: \alpha. x \).

Now every variable in the untyped \( \lambda \)-calculus is assigned an “any type,” and there arises a need to distinguish between different “any types.” As an example, consider a \( \lambda \)-abstraction \( \lambda x. \lambda y. (x, y) \) where

---

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

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

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” \( \beta \) 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 \( \alpha \) as a fresh “any type,” or a fresh type variable. That is, a type abstraction \( \Lambda \alpha. e \) declares a type variable \( \alpha \) for use in expression \( e \); we may rename \( \alpha \) in a way analogous to \( \alpha \)-conversions on \( \lambda \)-abstractions. If \( e \) has type \( A \), then \( \Lambda \alpha. e \) is assigned a polymorphic type \( \forall \alpha.A \) which reads “for all \( \alpha \), \( A \)” Note that \( A \) in \( \forall \alpha.A \) may use \( \alpha \) (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 value \( \text{true} \). First we need to convert \( \Lambda \alpha. \lambda x: \alpha. x \) to an identity function \( \lambda x: \text{bool}. x \) by instantiating \( \alpha \) 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 \( \alpha \) in \( e \):

\[ (\Lambda \alpha. e) [A] \rightarrow [A/\alpha]e \]

(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 \( (\Lambda \alpha. \lambda x: \alpha. x) [\text{bool}] \) reduces to an identity function specialized for type bool, and an ordinary application \( (\Lambda \alpha. \lambda x: \alpha. x) \text{true} \) finishes the job.

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

<table>
<thead>
<tr>
<th>type</th>
<th>( A )</th>
<th>( A \rightarrow A \mid \alpha \mid \forall \alpha. A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>( e )</td>
<td>( x \mid \lambda x: A. e \mid e e \mid \Lambda \alpha. e \mid e [A] )</td>
</tr>
<tr>
<td>value</td>
<td>( v )</td>
<td>( \lambda x: A. e \mid \Lambda \alpha. e )</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{c}
Tlam \quad Tapp \\
\frac{e \rightarrow e'}{e [A] \rightarrow e' [A]} \\
\frac{(\Lambda \alpha. e) [A] \rightarrow [A/\alpha]e}{[A/\alpha]e \text{ substitutes } A \text{ for } \alpha \text{ in } e; \text{ we omit its pedantic definition here. For ordinary applications, we reuse the reductions rules for the simply typed } \lambda \text{-calculus.}}
\end{array}
\]

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 \( \forall \alpha. \alpha \rightarrow \alpha \)! 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:

\[
\begin{align*}
\text{compose} & : \forall \alpha. \forall \beta. \forall \gamma. (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma) \\
\text{compose} & = \Lambda \alpha. \Lambda \beta. \Lambda \gamma. \lambda f : \alpha \rightarrow \beta. \lambda g : \beta \rightarrow \gamma. \lambda x : \alpha. g (f x)
\end{align*}
\]

Alternatively we may require that only the first two type variables \(\alpha\) and \(\beta\) be instantiated before producing a `λ`-abstraction which returns a `λ` abstraction expecting the third type variable \(\gamma\):

\[
\begin{align*}
\text{compose} & : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma) \\
\text{compose} & = \Lambda \alpha. \Lambda \beta. \lambda f : \alpha \rightarrow \beta. \lambda g : \beta \rightarrow \gamma. \lambda x : \alpha. g (f x)
\end{align*}
\]

As for the type system, System F is not a straightforward extension of the simply typed \(\lambda\)-calculus because of the inclusion of type variables. In the simply typed \(\lambda\)-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 2. In System F, \(x : A\) may not qualify as a valid type binding if type \(A\) contains type variables. For example, without type abstractions declaring type variables \(\alpha\) and \(\beta\), we may not use \(\alpha \rightarrow \beta\) as a type and hence \(x : \alpha \rightarrow \beta\) 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 \(\alpha\) type declares \(\alpha\) as a type, or equivalently, \(\alpha\) as a valid type variable:

\[
\text{typing context} \quad \Gamma ::= \cdot | \Gamma, x : A | \Gamma, \alpha \text{ type}
\]

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

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 \(\lambda\)-calculus, and a type judgment \(\Gamma \vdash A\) type which means that \(A\) is a valid type with respect to typing context \(\Gamma\). 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:

\[
\begin{array}{c}
\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A \rightarrow B \text{ type}} \quad \text{Ty} \\
\frac{\Gamma \vdash B \text{ type}}{\Gamma \vdash A \rightarrow B \text{ type}} \quad \text{Ty} \\
\text{Var} \quad x : A \in \Gamma \\
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A. e : A \rightarrow B} \quad \text{–l} \\
\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e : A \rightarrow B} \quad \text{–E} \quad \forall I \\
\frac{\Gamma, \alpha \text{ type} \vdash e : A}{\Gamma \vdash \lambda \alpha. e : \forall \alpha A} \quad \forall \text{L} \\
\frac{\Gamma \vdash e : \forall \alpha A}{\Gamma \vdash e [A] : [A/\alpha] B} \quad \forall \text{E}
\end{array}
\]

A proof of a type judgment \(\Gamma \vdash A\) type does not use type bindings in \(\Gamma\). In the rule –l, the typing context \(\Gamma, x : A\) assumes that \(A\) is a valid type with respect to \(\Gamma\). Hence the rule –l does not need a separate premise \(\Gamma \vdash A\) type. The rule \(\forall I\), called the \(\forall\) introduction rule, introduces a polymorphic type \(\forall \alpha A\) from the judgment in the premise. The rule \(\forall E\), called the \(\forall\) elimination rule, eliminates a polymorphic type \(\forall \alpha B\) by substituting a valid type \(A\) for type variable \(\alpha\). Note that the typing rule \(\forall E\) uses a substitution of a type into another type whereas the reduction rule \(\text{Tapp}\) uses a substitution of a type into an expression.

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

May 28, 2009
As an example of a typing derivation, let us find the type of an identity function specialized for type bool; we assume that $\Gamma \vdash \text{bool}$ type holds for any typing context $\Gamma$ (see the type rule $\text{TyBool}$ below):

\[
\frac{\alpha \text{ type, } x : \alpha \vdash x : \alpha}{\text{Var}} \quad \frac{\alpha \text{ type } \vdash \lambda x : \alpha. x : \alpha \rightarrow \alpha}{\text{Lam}} \quad \frac{\vdash \lambda \alpha. \lambda x : \alpha. x : \forall \alpha. \alpha \rightarrow \alpha}{\text{Vl}} \quad \frac{\vdash (\lambda \alpha. \lambda x : \alpha. x) \, [\text{bool}] : [\text{bool/\alpha}(\alpha \rightarrow \alpha)]}{\text{TyBool}}
\]

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

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 $\text{Tapp}$, and $[e'/x]e$ for the rule $\text{App}$. We write $[A/\alpha]\Gamma$ for substituting $A$ for $\alpha$ in all type bindings in $\Gamma$.

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

**Lemma 1.2 (Type substitution into expressions).**
If $\Gamma \vdash A$ type and $\alpha$ type, $\alpha' \vdash e : B$, 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 $\Gamma'$, which may contain types involving $\alpha$. In Lemma 1.3 we have to substitute $A$ into $e$ and $B$, both of which may contain types involving $\alpha$. 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 $\lambda$-calculus. We need to extend the canonical forms lemma (Lemma 2.2) and the inversion lemma (Lemma 2.2):

**Lemma 2.2 (Canonical forms).**
If $v$ is a value of type $\forall \alpha. A$, then $v$ is a type abstraction $\lambda \alpha. e$.

**Lemma 2.2 (Inversion).** Suppose $\Gamma \vdash e : C$.
If $e = \lambda \alpha. e'$, then $C = \forall \alpha. A$ and $\Gamma, \alpha$ type $\vdash e' : A$.

**Theorem 2.2 (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 2.2 (Type preservation).** If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.

### 1.2 Type reconstruction

The type systems of the simply typed $\lambda$-calculus and System F require that all variables in $\lambda$-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 $\text{fn} \; x \mapsto \text{fn} \; y \mapsto x + y$, which is no less readable than a fully type-annotated function $\text{fn} \; x : \text{int} \mapsto \text{fn} \; y : \text{int} \mapsto x + y$. 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 $\text{fn} \; x \mapsto \text{fn} \; y \mapsto x + y$ gives enough information to decide a unique type $\text{int}$ 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 goal of type reconstruction is to convert an expression $e$ in the untyped $\lambda$-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 $\text{erase}(\cdot)$ which takes an expression in System F and erases all type annotations in it:

$$
\begin{align*}
\text{erase}(x) &= x \\
\text{erase}(\lambda x : A. e) &= \lambda x. \text{erase}(e) \\
\text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{erase}(e_2) \\
\text{erase}(\lambda \alpha. e) &= \text{erase}(e) \\
\text{erase}(e[A]) &= \text{erase}(e)
\end{align*}
$$

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 $\text{erase}(e) \mapsto^* \text{erase}(e')$ holds in the untyped $\lambda$-calculus.

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

As an example, let us consider an untyped $\lambda$-abstraction $\lambda x. x$. It is not typable in the simply typed $\lambda$-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 $\lambda x. x$.

$$
\frac{x : \forall \alpha. \alpha \to \alpha \vdash x : \forall \alpha. \alpha \to \alpha}{\text{Var}} \\
\frac{x : \forall \alpha. \alpha \to \alpha \vdash x : [\forall \alpha. \alpha \to \alpha] : (\forall \alpha. \alpha \to \alpha) \to (\forall \alpha. \alpha \to \alpha)}{(\forall \E)} \\
\frac{x : \forall \alpha. \alpha \to \alpha \vdash x : \forall \alpha. \alpha \to \alpha}{\text{Var}} \\
\frac{x : \forall \alpha. \alpha \to \alpha \vdash \lambda x : \forall \alpha. \alpha \to \alpha. x [\forall \alpha. \alpha \to \alpha] x : (\forall \alpha. \alpha \to \alpha) \to (\forall \alpha. \alpha \to \alpha)}{(\forall \I)}
$$

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

$$
\frac{\alpha \text{ type } \in x : \forall \alpha. \alpha \to \alpha \to \alpha \alpha \to \alpha \alpha \to \alpha}{\text{TyVar}} \\
\frac{x : \forall \alpha. \alpha \to \alpha \to \alpha \to \alpha \to \alpha}{\text{Ty}}
$$

(The proof does not use the type binding $x : \forall \alpha. \alpha \to \alpha$.) Hence a type reconstruction algorithm for System F, if any, would convert $\lambda x. x$ 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 $\lambda$-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 $\lambda$-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 $\lambda$-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 1 how to encode common datatypes in the untyped $\lambda$-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 into 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 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 $\alpha$ to both options:

\[
\text{bool} = \forall \alpha. \alpha \rightarrow \alpha
\]

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

\[
\text{true} = \lambda \alpha.\lambda: \alpha. \lambda f : \alpha. \alpha \quad \text{false} = \lambda \alpha.\lambda: \alpha. \lambda f : \alpha. \alpha
\]

The intuition behind Church numerals is that a Church numeral $\hat{n}$ 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:

\[
\text{nat} = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)
\]

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

\[
\text{zero} = \Lambda \alpha.\lambda: \alpha. \lambda x : \alpha. x \\
\text{succ} = \lambda n: \text{nat}. \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. (n [\alpha] f) (f x)
\]

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 $A \rightarrow B \rightarrow \alpha$ as a type for a function taking two arguments of types $A$ and $B$ and returning a value of type $\alpha$, a value of type $A \times B$ contains everything necessary for applying such a function, which is expressed in the following definition of $A \times B$:

\[
A \times B = \forall \alpha. (A \rightarrow B \rightarrow \alpha) \rightarrow \alpha
\]

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

\[
\begin{align*}
\text{pair} & : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta = \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda y : \beta. \Lambda \gamma. \lambda f : \alpha \rightarrow \beta \rightarrow \gamma. f x y \\
\text{fst} & : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha = \Lambda \alpha. \Lambda \beta. \lambda p : \alpha \times \beta. p [\alpha] (\lambda x : \alpha. \lambda y : \beta. x) \\
\text{snd} & : \forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \beta = \Lambda \alpha. \Lambda \beta. \lambda p : \alpha \times \beta. p [\beta] (\lambda x : \alpha. \lambda y : \beta. y)
\end{align*}
\]

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

\[
(\cdot) : \text{unit} = \Lambda \alpha. \lambda x : \alpha. x
\]

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

\[
A+B = \forall \alpha. (A \rightarrow \alpha) \rightarrow (B \rightarrow \alpha) \rightarrow \alpha
\]

Injections and case expressions are translations of the typing rules $+I_L$, $+I_R$, and $+E$:

\[
\begin{align*}
\text{inl} & : \forall \alpha. \forall \beta. \alpha \rightarrow \alpha + \beta \\
\text{inr} & : \forall \alpha. \forall \beta. \beta \rightarrow \alpha + \beta \\
\text{case} & : \forall \alpha. \forall \beta. \forall \gamma. \alpha + \beta \rightarrow (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma
\end{align*}
\]

**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?)

---

3 We may also interpret nat as $\text{nat} = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ such that a Church numeral $\hat{n}$ takes a successor function succ of type $\alpha \rightarrow \alpha$ and a zero zero of type $\alpha$ to return succ$^0 \ 0$ of type $\alpha$. May 28, 2009
### 1.4 Predicative polymorphic $\lambda$-calculus

This section presents the predicative polymorphic $\lambda$-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 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:

- **monotype**: $A ::= A \rightarrow \alpha$
- **polytype**: $U ::= A | \forall \alpha. U$
- **expression**: $e ::= x | \lambda x : A. e | e e | \Lambda \alpha. e | e [A]$
- **value**: $v ::= \lambda x : A. e | \Lambda \alpha. e$

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, $(\forall \alpha. A \rightarrow \alpha) \rightarrow (\forall \alpha. \beta \equiv \beta)$ is a polytype whereas $\forall \alpha. (A \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$ is not. 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 $\lambda$-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 $\forall \exists$ below).

As in System F, the type system of the predicative polymorphic $\lambda$-calculus uses two forms of judgments: a typing judgment $\Gamma \vdash e : U$ and a type judgment $\Gamma \vdash A \text{ type}$. The difference is that $\Gamma \vdash A \text{ type}$ now checks if a given type is a valid monotype. That is, we do not use a type judgment $\Gamma \vdash U \text{ type}$ (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 $\forall \exists$ from System F is gone:

$\begin{align*}
\Gamma \vdash A \text{ type} & \quad \Gamma \vdash B \text{ type} \quad \text{Ty} \rightarrow \\
\Gamma \vdash A \rightarrow B & \quad \text{TyVar} \\
\Gamma \vdash \alpha \text{ type} & \quad \text{Ty} \rightarrow \\
\Gamma \vdash \alpha & \quad \text{Ty} \rightarrow \\
\Gamma, x : A & \vdash e : B \quad \text{Var} \\
\Gamma, x : A & \vdash \lambda x : A. e : A \rightarrow B \quad \text{LVar} \\
\Gamma \vdash e : A \rightarrow B & \quad \text{LVar} \\
\Gamma \vdash \alpha & \quad \text{Ty} \rightarrow \\
\Gamma \vdash \Lambda \alpha. e & \vdash \forall \alpha. U \quad \forall \exists \\
\Gamma \vdash e : \forall \alpha. U & \quad \forall \exists \\
\Gamma \vdash e [A] : [A/\alpha][U] & \quad \forall \exists
\end{align*}$

Unfortunately the use of a monotype $A$ in a $\lambda$-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 $\text{id} = \Lambda \alpha. \lambda x : \alpha. x$ to two different types, say, bool and int. In the untyped $\lambda$-calculus, we would bind a variable $f$ to an identity function and then apply $f$ twice:

$$(\lambda f. \text{pair} (f \text{ true}) (f \text{ false})) (\lambda x. x)$$

In the predicative polymorphic $\lambda$-calculus, it is impossible to reuse $\text{id}$ more than once in this way, since $f$ must be given a monotype while $\text{id}$ has a polytype.

May 28, 2009

7
that it never stands alone as a first-class object and must be applied to a polymorphic expression
by the rule

\[
(\lambda f : \forall \alpha. \alpha \rightarrow \alpha. \ (f \ [\text{true}] \ [\text{false}]), \ f \ 0 \ (or \ f \ \text{true}) \ in \ the \ body \ fails \ to \ typecheck:
\]

\[
(\lambda f : \text{bool} \rightarrow \text{bool}. \ (f \ 	ext{true}, \ f \ 0)) \ (\text{id} \ [\text{bool}]) \quad \text{ill-typed}
\]

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 \textit{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 \textit{let-binding}, for declaring variables of polytypes. A \textit{let-binding} \( x : U = e \ in \ e' \) binds \( x \) to a polymorphic expression \( e \) of type \( U \) and allows multiple occurrences of \( x \) in \( e' \). With a \textit{let-binding}, we can apply a polymorphic identity function to two different (mono)types \text{bool} and \text{int} as follows:

\[
\text{let } f : \forall \alpha. \alpha \rightarrow \alpha = \Lambda \alpha. \ \lambda x : \alpha. \ x \ in \ (f \ [\text{true}], \ f \ [\text{int}])
\]

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:

| expression | \( e := \cdots \mid \text{let } x : U = e \ in \ e \) |
| typing context | \( \Gamma ::= \cdot \mid \Gamma, x : U \mid \Gamma, \alpha \text{ type} \) |
| \( x : U \in \Gamma \) | \( \Gamma \vdash e : U \mid \Gamma, x : U \vdash e' : A \) |
| \( \Gamma \vdash \text{let } x : U = e \ in \ e' : A \) | Let |

The reduction of a \textit{let-binding} \( x : U = e \ in \ e' \) proceeds by substituting \( e \) for \( x \) in \( e' \):

\[
\text{let } x : U = e \ in \ e' \rightarrow [e/x]e'
\]

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

Although \( x : U = e \ in \ e' \) 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 by the rule \text{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 expression polymorphically, \textit{we must bind it to a variable using a let-binding instead of a λ-abstraction}.

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:

\[
\text{monotype } \ A ::= U \rightarrow U \mid \alpha \quad \leftrightarrow \quad \text{type } U ::= U \rightarrow U \mid \alpha \mid \forall \alpha. U
\]

We may construe a \textit{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, which 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 that type variables are instantiated explicitly by type annotations. 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 $\lambda$-abstractions. That is, polymorphic types arise “implicitly” from lack of type annotations in $\lambda$-abstractions.

As an example, consider an identity function $\lambda x.x$. It can be assigned such types as $\text{bool}\rightarrow\text{bool}$, $\text{int}\rightarrow\text{int}$, $(\text{int}\rightarrow\text{int})\rightarrow(\text{int}\rightarrow\text{int})$, 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 $\alpha$ 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 $\text{bool}\rightarrow\text{bool}$ and $\text{int}\rightarrow\text{int}$. The type reconstruction algorithm of SML infers a unique principal type for every well-typed expression. Below we discuss the type system of SML and defer details of the type reconstruction algorithm to Section 1.8.

In essence, the type system of SML uses let-polymorphism without type annotations (in $\lambda$-abstractions and let-bindings), type abstractions, and type applications:

\[
\text{monotype} \quad A \quad ::= \quad A \rightarrow A \mid \alpha
\]

\[
\text{polytype} \quad U \quad ::= \quad A \mid \forall\alpha.U
\]

\[
\text{expression} \quad e \quad ::= \quad x \mid \lambda x.e \mid e e \mid \text{let} \ x = e \ \text{in} \ e
\]

\[
\text{value} \quad v \quad ::= \quad \lambda x.e
\]

\[
\text{typing context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : U \mid \Gamma, \alpha \text{ type}
\]

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 $e$ by the following erasure function:

\[
\text{erase}(x) = x
\]

\[
\text{erase}(\lambda x:A. e) = \lambda x. \text{erase}(e)
\]

\[
\text{erase}(e_1 e_2) = \text{erase}(e_1) \text{ erase}(e_2)
\]

\[
\text{erase}(\lambda \alpha.e) = \text{erase}(e)
\]

\[
\text{erase}(e[\alpha]) = \text{erase}(e)
\]

\[
\text{erase(let } x:U = e \text{ in } e') = \text{let } x = \text{erase}(e) \text{ in } \text{erase}(e')
\]

That is, if $\Gamma \triangleright e : U$ holds, $e$ has its counterpart in let-polymorphism in Section 1.8. The rules for the typing judgment $\Gamma \triangleright e : U$ are given as follows:

\[
\frac{x : U \in \Gamma \quad \text{Var}}{\Gamma \vdash x : U} \quad \frac{\Gamma, x : A \triangleright e : B}{\Gamma \vdash \lambda x.e : A \rightarrow B} \quad \frac{\Gamma \triangleright e : B}{\Gamma \vdash e : B} \quad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e \\ e' : B} \quad \frac{\Gamma \triangleright e : U \quad \Gamma, x : U \triangleright e' : A}{\Gamma \vdash \text{let } x = e \text{ in } e' : A} \quad \frac{\Gamma, \alpha \text{ type } \triangleright e : U}{\Gamma \vdash \forall \alpha.U} \quad \frac{\Gamma \vdash e : \forall \alpha.U \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash e : [A/\alpha]U}
\]

Not that unlike in the predicative polymorphic $\lambda$-calculus, the rule $\rightarrow I$ allows us to assign any monotype $A$ to variable $x$ as long as expression $e$ is assigned a valid monotype $B$. Hence, for example, the same $\lambda$-abstraction $\lambda x.x$ can now be assigned different monotypes such as $\text{bool}\rightarrow\text{bool}$, $\text{int}\rightarrow\text{int}$, and $\alpha\rightarrow\alpha$. The rules Gen and Spec correspond to the rules $\forall I$ and $\forall E$ in the predicative polymorphic $\lambda$-calculus (but not in System F because $A$ in the rule Spec is required to be a monotype).

In the rule Gen (for generalizing a type), expression $e$ in the conclusion plays the role of a type abstraction. That is, we can think of $e$ in the conclusion as $\text{erase}(\lambda \alpha.e)$. As an example, let us assign a polytype to the polymorphic identity function $\lambda x.x$:

\[
\Gamma \triangleright \lambda x.x : ?
\]

May 28, 2009
Intuitively \( \lambda x. x \) has type \( \alpha \to \alpha \) for an “any type” \( \alpha \), so we first assign a monotype \( \alpha \to \alpha \) under the assumption that \( \alpha \) is a valid type variable:

\[
\Gamma, \alpha \Rightarrow x : \alpha \vdash x : \alpha \\
\Gamma, \alpha \Rightarrow \lambda x. x : \alpha \to \alpha \rightarrow \neg I
\]

Note that \( \lambda x. x \) has not been assigned a polytype yet. Also note that \( \Gamma \Rightarrow \lambda x. x : \alpha \to \alpha \) cannot be a valid typing derivation because \( \alpha \) is a fresh type variable which is not declared in \( \Gamma \). Assigning a polytype \( \forall \alpha. \alpha \to \alpha \) to \( \lambda x. x \) is accomplished by the rule Gen:

\[
\begin{align*}
\Gamma, \alpha \Rightarrow x : \alpha & \vdash x : \alpha \\
\Gamma, \alpha \Rightarrow \lambda x. x : \alpha & \to \alpha \rightarrow \neg I \\
\Gamma & \Rightarrow \lambda x. x : \forall \alpha. \alpha \to \alpha \\
\end{align*}
\]

As an example of using two type variables, we assign a polytype \( \forall \alpha. \forall \beta. \alpha \to (\alpha \times \beta) \) to \( \lambda x. \lambda y. (x, y) \) as follows (where we assume that product types are available):

\[
\begin{align*}
\Gamma, \alpha \Rightarrow \beta & \vdash x : \alpha, y : \beta \vdash x : \alpha \\
\Gamma, \alpha \Rightarrow \beta & \vdash x : \alpha, y : \beta \vdash \beta \rightarrow (\alpha \times \beta) \\
\Gamma & \Rightarrow \lambda x. \lambda y. (x, y) : \forall \alpha. \forall \beta. \alpha \to (\alpha \times \beta) \\
\end{align*}
\]

Generalizing the example, we can assign a polytype to an expression \( e \) in two steps. First we introduce as many fresh type variables as necessary to assign a monotype \( A \) to \( e \). Then we keep applying the rule Gen to convert, or generalize, \( A \) to a polytype \( U \). If \( A \) uses fresh type variables \( \alpha_1, \alpha_2, \ldots, \alpha_n \), then \( U \) is given as \( \forall \alpha_1. \forall \alpha_2. \ldots. \forall \alpha_n. A \):

\[
\begin{align*}
\Gamma, \alpha_1 \Rightarrow \alpha_2 & \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow e : A \\
\Gamma, \alpha_1 \Rightarrow \alpha_2 & \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow e : \forall \alpha_n. A \\
\ldots \\
\Gamma, \alpha_1 \Rightarrow \alpha_2 & \Rightarrow \ldots \Rightarrow \forall \alpha_n \Rightarrow e : \forall \alpha_n. A \\
\Gamma & \Rightarrow e : \forall \alpha_1. \forall \alpha_2. \ldots. \forall \alpha_n. A \\
\end{align*}
\]

In the rule Spec (for specializing a type), expression \( e \) in the conclusion plays the role of a type application. That is, we can think of \( e \) in the conclusion as \( \text{erase}(e \, [\mid A]) \). Thus, by applying the rule Spec repeatedly, we can convert, or specialize, any polytype into a monotype.

A typical use of the rule Spec is to specialize the polytype of a variable introduced in a let-binding (in which case expression \( e \) in the rule Spec is a variable). Specifically a let-binding let \( x = e \) in \( e' \) binds variable \( x \) to a polymorphic expression \( e \) and uses \( x \) monomorphically within \( e' \) after specializing the type of \( x \) to monotypes by the rule Spec. For example, the following typing derivation for let \( f = \lambda x. x \) in \( (f \, \text{true}, f \, 0) \) applies the rule Spec to variable \( f \) twice, where we abbreviate \( \Gamma, f : \forall \alpha. \alpha \rightarrow \alpha \) as \( \Gamma' \):

\[
\begin{align*}
\Gamma' & \Rightarrow f : \forall \alpha. \alpha \to \alpha \\
\Gamma' & \Rightarrow f : \text{bool} \rightarrow \text{bool} \\
\Gamma' & \Rightarrow (f \, \text{true}, f \, 0) : \text{bool} \times \text{int} \\
\Gamma & \Rightarrow \text{let } f = \lambda x. x \text{ in } (f \, \text{true}, f \, 0) : \text{bool} \times \text{int} \\
\end{align*}
\]

Note that in typechecking \( (f \, \text{true}, f \, 0) \), it is mandatory to specialize the type of \( f \) to a monotype \( \text{bool} \rightarrow \text{bool} \) or \( \text{int} \rightarrow \text{int} \), since an application \( f \, e \) typechecks by the rule \( \rightarrow \text{E} \) only if \( f \) is assigned a monotype.

May 28, 2009
If expression e in the rule Spec is not a variable, the typing derivation of the premise $\Gamma \vdash e : \forall \alpha. U$ must end with an application of the rule Gen or another application of the rule Spec. In such a case, we can eventually locate an application of the rule Gen that is immediately followed by an application of the rule Spec:

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

Here we introduce a type variable $\alpha$ only to instantiate it to a concrete monotype $A$ immediately, which implies that such a typing derivation is redundant and can be removed. For example, when typing $(\lambda x. x) \text{true}$, there is no need to take a detour by first assigning a polytype $\forall \alpha. \alpha \rightarrow \alpha$ to $\lambda x. x$ and then instantiating $\alpha$ to bool. Instead it suffices to assign a monotype bool $\rightarrow$ bool directly because $\lambda x. x$ is eventually applied to an argument of type bool:

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

This observation suggests that it is unnecessary to specialize the type of an expression that is not a variable — we only need to apply the rule Spec to polymorphic variables introduced in let-bindings.

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 $\Gamma \vdash \text{erase}(e) : U$.

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

### 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 mutable references and input/output. To see the problem, consider the following expression where we assume constructs for integers, booleans, and mutable references:

```plaintext
let x = ref (\lambda y. y) in
let y = x := \lambda y. y + 1 in
(\lambda x) true
```

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

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

By the rule Spec, then, we can assign either ref $\ (\text{int} \rightarrow \text{int})$ or ref $\ (\text{bool} \rightarrow \text{bool})$ to variable $x$. Now both expressions $x := \lambda y. y + 1$ and $(\lambda x) true$ are well-typed, but the reduction of $(\lambda x) true$ 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 imposes a requirement, called value restriction, that expression e in the rule Gen be a syntactic value:

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

The idea is to exploit the fact that computational effects cannot interfere with (polymorphic) values, whose evaluation terminates immediately. Now, for example, ref $\ (\lambda y. y)$ cannot be assigned a polytype because ref $\ (\lambda y. y)$ is not a value and thus its type cannot be generalized by the rule Gen.
As a consequence of value restriction, variable x in a let-binding let \( x = e \) in \( e' \) can be assigned a polytype only if expression \( e \) is a value. If \( e \) is not a value, \( x \) must be used monomorphically within expression \( e' \), even if \( e \) itself does not specify a unique monotype. This means that we may have to analyze \( e' \) in order to decide the monotype to be assigned to \( x \). As an example, consider the following expression:

\[
\text{let } x = (\lambda y. y) (\lambda z. z) \text{ in } \text{true}
\]

As \((\lambda y. y) (\lambda z. z)\) is not a value, variable \( x \) must be assigned a monotype. \((\lambda y. y) (\lambda z. z)\), however, does not specify a unique monotype for \( x \); it only specifies that the type of \( x \) must be of the form \( A \to A \) for some monotype \( A \). Fortunately the application \( x \) true fixes such a monotype \( A \) as bool and \( x \) is assigned a unique monotype bool \( \to \) bool. The following expression, in contrast, is ill-typed because variable \( x \) is used polymorphically:

\[
\text{let } x = (\lambda y. y) (\lambda z. z) \text{ in } (x \text{ true}, x 1)
\]

The problem here is that \( x \) needs to be assigned two monotypes bool \( \to \) bool and int \( \to \) int simultaneously, which is clearly out of the question.

### 1.8 Type reconstruction algorithm

This section presents a type reconstruction algorithm for the type system with implicit polymorphism in Section 1.6. Given an untyped expression \( e \), the goal is to infer a polytype \( U \) such that \( \cdot \triangleright e : U \) holds. In addition to being a valid type for \( e \), \( U \) also needs to be the most general type for \( e \) in the sense that every valid type for \( e \) can be obtained a special case of \( U \) by instantiating some type variables in \( U \). Given \( \lambda x. x \) as input, for example, the algorithm returns the most general polytype \( \forall \alpha. \alpha \to \alpha \) instead of a specific monotype such as bool \( \to \) bool.

Typically the algorithm creates (perhaps a lot of) temporary type variables before finding the most general type of a given expression. We design the algorithm in such a way that all these temporary type variables are valid (simply because there is no reason to create invalid ones). As a result, we no longer need a type declaration \( \alpha \) type in the rule Gen (because \( \alpha \) is assumed to be a valid type variable) and a type judgment \( \Gamma \vdash A \) type in the rule Spec (because \( A \) is a valid type if all type variables in it are valid). Accordingly a typing context now consists only of type bindings:

\[
\text{typing context } \quad \Gamma ::= \cdot | \Gamma, x : U
\]

With the assumption that all type variables are valid, the rules Gen and Spec are revised as follows:

\[
\begin{array}{c}
\text{Gen} \quad \Gamma \triangleright e : U \quad \alpha \notin ftv(\Gamma) \quad \Gamma \triangleright e : \forall \alpha.U \\
\text{Spec} \quad \Gamma \triangleright e : \forall \alpha.U \quad \Gamma \triangleright e : [A/\alpha]U
\end{array}
\]

Here \( ftv(\Gamma) \) denotes the set of free type variables in \( \Gamma \); \( ftv(U) \) denotes the set of free type variables in \( U \):

\[
\begin{aligned}
ftv(\cdot) & = \emptyset \\
ftv(\Gamma, x : U) & = ftv(\Gamma) \cup ftv(U) \\
ftv(A \to B) & = ftv(A) \cup ftv(B) \\
ftv(\forall \alpha.U) & = ftv(U) - \{\alpha\}
\end{aligned}
\]

In the rule Gen, the condition \( \alpha \notin ftv(\Gamma) \) checks that \( \alpha \) is a fresh type variable. If \( \Gamma \) contains a type binding \( x : U \) where \( \alpha \) is already in use as a free type variable in \( U \), \( \alpha \) cannot be regarded as a fresh type variable and generalizing \( U \) to \( \forall \alpha.U \) is not justified. In the following example, \( \alpha \to \beta \) may generalize to \( \forall \alpha. \alpha \to \beta \), assigning the desired polytype to the polymorphic identity function \( \lambda x. x \), because \( \alpha \) is not in use in the empty typing context:

\[
\begin{array}{c}
\text{Var} \quad x : \alpha \triangleright x : \alpha \\
\text{Gen} \quad \cdot \triangleright \lambda x. x : \alpha \to \alpha \\
\end{array}
\]
If $\alpha$ is already in use as a free type variable, however, such a generalization results in assigning a wrong type to $\lambda x. x$:

\[
\begin{align*}
  x : \alpha, y : \alpha & \vdash x : \alpha & \Var \\
  x : \alpha & \vdash \lambda y. x : \alpha & \lambda \Gen \\
  x : \alpha & \vdash \lambda y. x : \forall \alpha. \alpha & \lambda \Gen
\end{align*}
\]

Here variable $y$ is unrelated to variable $x$, yet is assigned the same type in the premise of the rule $\lambda \Gen$. A correct typing derivation assigns a fresh type variable to $y$ to reflect the fact that $x$ and $y$ are unrelated:

\[
\begin{align*}
  x : \alpha, y : \beta & \vdash x : \alpha & \Var \\
  x : \alpha & \vdash \lambda y. x : \beta & \lambda \Gen \\
  x : \alpha & \vdash \lambda y. x : \forall \beta, \beta & \lambda \Gen
\end{align*}
\]

As an example of applying the rule $\Spec$, here is a typing derivation assigning a monotype $\text{bool} \to \text{bool}$ to $\lambda x. x$ by instantiating a type variable:

\[
\begin{align*}
  & \vdash \lambda x. x : \forall \alpha. \alpha & \Var \\
  & \vdash \lambda x. x : \text{bool} \to \text{bool} & \lambda \Spec \\
\end{align*}
\]

For the above example, the rule $\Spec$ is unnecessary because we can directly assign $\text{bool}$ to variable $x$:

\[
\begin{align*}
  x : \text{bool} & \vdash x : \text{bool} & \Var \\
  & \vdash \lambda x. x : \text{bool} \to \text{bool} & \lambda \Gen
\end{align*}
\]

As we have seen in Section 1.6, however, the rule $\Spec$ is indispensable for specializing the type of a variable introduced in a $\text{let}$-binding. In the following example, the same type variable $\alpha$ in $\forall \alpha. \alpha \to \alpha$ is instantiated to two different types $\beta$ and $\beta'$ by the rule $\Spec$:

\[
\begin{align*}
  f : \forall \alpha. \alpha & \vdash f : \forall \alpha. \alpha & \Var \\
  f : \forall \alpha. \alpha & \vdash f : (\beta \to \beta) & \Spec \\
  f : \forall \alpha. \alpha & \vdash f : (\beta' \to \beta') & \Spec \\
  f : \forall \alpha. \alpha & \vdash f : \beta' & \Var
\end{align*}
\]

Exercise 1.12. What is wrong with the following typing derivation?

\[
\begin{align*}
  x : \forall \alpha. \alpha & \vdash x : \forall \alpha. \alpha & \Var \\
  x : \forall \alpha. \alpha & \vdash (x : \forall \alpha. \alpha) & \Spec \\
  x : \forall \alpha. \alpha & \vdash x : \forall \alpha. \alpha & \Var \\
  & \vdash \lambda x. x : \forall \alpha. \alpha & \lambda \Gen
\end{align*}
\]

The type reconstruction algorithm, called $W$, takes a typing context $\Gamma$ and an expression $e$ as input, and returns a pair of a type substitution $S$ and a monotype $A$ as output:

\[
W(\Gamma, e) = (S, A)
\]

A type substitution is a mapping from type variables to monotypes. Note that it is not a mapping to polytypes because type variables range only over monotypes:

\[
\text{type substitution} \quad S ::= \text{id} | \{A/\alpha\} | S \circ S
\]

id is an identity type substitution which changes no type variable. $\{A/\alpha\}$ is a singleton type substitution which maps $\alpha$ to $A$. $S_1 \circ S_2$ is a composition of $S_1$ and $S_2$ which applies first $S_2$ and then $S_1$. id is the identity for the composition operator $\circ$, i.e., $\text{id} \circ S = S \circ \text{id} = S$. As $\circ$ is associative, we write $S_1 \circ S_2 \circ S_3$ for $S_1 \circ (S_2 \circ S_3) = (S_1 \circ S_2) \circ S_3$.

An application of a type substitution to a polytype $U$ is formally defined as follows:
Theorem 1.13 (Soundness of \( W \)). Given a typing context \( \Gamma \) and an expression \( e \), the algorithm analyzes \( e \) to build a type substitution \( S \) mapping free type variables in \( \Gamma \) so that \( e \) typechecks with a monotype \( A \).

An invariant is that \( S \) has no effect on \( A \), i.e., \( S \cdot A = A \), since \( A \) obtained after applying \( S \) to free type variables in \( \Gamma \). Here are a few examples:

- \( W(x : \alpha, x + 0) = ([\text{int}/\alpha], \text{int}) \) where we assume a base type \( \text{int} \).
- \( W(\lambda x. x + 0) = ([\text{int}/\alpha], \text{int} \rightarrow \text{int}) \) where we assume a base type \( \text{int} \).
- \( W(\lambda x. x) = (\text{id}, \alpha \rightarrow \alpha) \)

Exercise 1.14. What is the result of \( W(y : \beta, (\lambda x. x) y) \) if the algorithm \( W \) creates a temporary type variable \( \alpha \) for variable \( x \)? Is the result unique?

Figure 1.1 shows the pseudocode of the algorithm \( W \). We write \( \vec{\alpha} \) for a sequence of distinct type variables \( \alpha_1, \alpha_2, \ldots, \alpha_n \). Then \( \forall \vec{\alpha} . A \) stands for \( \forall \alpha_1, \forall \alpha_2, \ldots, \forall \alpha_n . A \), and \( \{ \vec{\beta} / \vec{\alpha} \} \) stands for \( \{ \beta_1/\alpha_1 \} \circ \cdots \circ \{ \beta_n/\alpha_n \} \). We write \( \Gamma + x : U \) for \( \Gamma - \{ x : U' \}, x : U \) if \( x : U' \in \Gamma \), and for \( \Gamma, x : U \) if \( \Gamma \) contains no type binding for variable \( x \).

The first case \( W(\Gamma, x) \) summarizes the result of applying the rule \( \text{Spec} \) to \( \forall \vec{\alpha} . A \) as many times as the length of \( \vec{\alpha} \). Note that \([A/\alpha] U\) is written as \( \{A/\alpha\} \cdot U\) in the following typing derivation.

\[
\begin{align*}
\forall \alpha_1, \forall \alpha_2, \ldots, \forall \alpha_n . A &\in \Gamma \\
\Gamma \vdash x : \forall \alpha_1, \forall \alpha_2, \ldots, \forall \alpha_n . A &\quad \text{Var} \\
\Gamma \vdash x : \{\beta_1/\alpha_1\} \cdot \forall \alpha_2, \ldots, \forall \alpha_n . A &\quad \text{Spec} \\
\Gamma \vdash x : \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\} \cdot \forall \alpha_3, \ldots, \forall \alpha_n . A &\quad \text{Spec} \\
&\vdots \\
\Gamma \vdash x : \{\beta_n/\alpha_n\} \circ \cdots \circ \{\beta_2/\alpha_2\} \circ \{\beta_1/\alpha_1\} \cdot A &\quad \text{Spec}
\end{align*}
\]
The second case $\mathcal{W}(\Gamma, \lambda x. e)$ creates a fresh type variable $\alpha$ to be assigned to variable $x$.

The case $\mathcal{W}(\Gamma, e_1 e_2)$ uses an auxiliary function $\text{Unify}(E)$ where $E$ is a set of type equations between monotypes:

$$\text{Unify}(\cdot) = \text{id}$$

$$\text{Unify}(E, \alpha = A) = \text{Unify}(E, A = \alpha) = \begin{cases} \text{Unify}(E) & \text{if } \alpha = A \\ \text{fail} & \text{else if } \alpha \in \text{ftv}(A) \\ \text{Unify}(\{A/\alpha\} \cdot E) \circ \{A/\alpha\} & \text{else} \end{cases}$$

$$\text{Unify}(E, A_1 \rightarrow A_2 = B_1 \rightarrow B_2) = \text{Unify}(E, A_1 = B_1, A_2 = B_2)$$

Figure 1.2: Algorithm $\text{Unify}$

The specification of the function $\text{Unify}$ is stated as follows:

**Proposition 1.15.** If $\text{Unify}(A_1 = A'_1, \cdots, A_n = A'_n) = S$, then $S \cdot A_i = S \cdot A'_i$ for $i = 1, \cdots, n$.

Here are a few examples of $\text{Unify}(E)$ where we assume a base type int:

(1) $\text{Unify}(\alpha = \text{int} \rightarrow \alpha) = \text{fail}$
(2) $\text{Unify}(\alpha = \alpha \rightarrow \alpha) = \text{fail}$
(3) $\text{Unify}(\alpha \rightarrow \alpha = \text{int} \rightarrow \text{int}) = \{\text{int}/\alpha\}$
(4) $\text{Unify}(\alpha \rightarrow \beta = \beta \rightarrow \alpha) = \{\beta/\alpha\}$
(5) $\text{Unify}(\alpha \rightarrow \beta = \beta \rightarrow \alpha) = \{\beta/\alpha\}$
(6) $\text{Unify}(\alpha \rightarrow \beta = \beta \rightarrow \alpha, \alpha = \text{int}) = \{\beta/\alpha\}$

In cases (1) and (2), the unification fails because both int$\rightarrow$ and $\alpha \rightarrow \alpha$ contain $\alpha$ as a free type variable, but are strictly larger than $\alpha$. In case (5), either $\{\beta/\alpha\}$ or $\{\alpha/\beta\}$ successfully unifies $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$.

Case (6) uses an additional assumption $\text{Unify}(E, \text{int} = \text{int}) = \text{Unify}(E)$:

$$\text{Unify}(\alpha \rightarrow \beta = \beta \rightarrow \alpha, \alpha = \text{int}) = \text{Unify}(\{\beta/\alpha\} \cdot (\alpha \rightarrow \beta = \beta \rightarrow \alpha), \{\beta/\alpha\})$$

May 28, 2009 15
The case $\mathcal{W}(\Gamma, e) = (e_1)$ uses another auxiliary function $Gen_\Gamma(A)$ which generalizes monotype $A$ to a polytype after taking into account free type variables in typing context $\Gamma$:

$$Gen_\Gamma(A) = \forall \alpha_1.\forall \alpha_2.\cdots \forall \alpha_n. A$$

where $\alpha_i \notin ftv(\Gamma)$ and $\alpha_i \in ftv(A)$ for $i = 1, \ldots, n$.

That is, if $\alpha \in ftv(A)$ is in $ftv(\Gamma)$, $\alpha \in A$ is not interpreted as “any type” with respect to $\Gamma$. Note that $Gen_\Gamma(A) = \forall \alpha_1.\forall \alpha_2.\cdots \forall \alpha_n. A$ is equivalent to applying the rule $Gen$ exactly $n$ times as follows:

$$\frac{\Gamma \vdash e : A \quad \alpha_n \notin ftv(\Gamma)}{\Gamma \vdash e : \forall \alpha_n. A}$$

$$\frac{\Gamma \vdash e : \forall \alpha_n. A \quad \alpha_{n-1} \notin ftv(\Gamma)}{\Gamma \vdash e : \forall \alpha_{n-1}. A}$$

$$\vdots$$

$$\frac{\Gamma \vdash e : \forall \alpha_1.\forall \alpha_2.\cdots \forall \alpha_n. A}{Gen}$$

Here are a few examples of $Gen_\Gamma(A)$:

$$Gen.(\alpha \rightarrow \alpha) = \forall \alpha. \alpha \rightarrow \alpha$$

$$Gen_{x;\alpha}(\alpha \rightarrow \alpha) = \alpha \rightarrow \alpha$$

$$Gen_{x;\alpha}(\alpha \rightarrow \beta) = \forall \beta. \alpha \rightarrow \beta$$

$$Gen_{x;\alpha,y;\beta}(\alpha \rightarrow \beta) = \alpha \rightarrow \beta$$

Given an expression $e$, the algorithm $\mathcal{W}$ returns a monotype $A$ which may contain free type variables. If we wish to obtain the most general polytype for $e$, it suffices to generalize $A$ with respect to the given typing context. Specifically, if $\mathcal{W}(\Gamma, e) = (S, A)$ holds, Theorem 1.13 justifies $S \cdot \Gamma \vdash e : A$, which in turn justifies $S \cdot \Gamma \vdash e : Gen_S(\Gamma, A)$. Hence we may take $Gen_{S, \Gamma}(A)$ as the most general type for $e$ under typing context $\Gamma$, although we do not formally prove this property (called the completeness of $\mathcal{W}$) here.