Handout 6: Polymorphic Type Systems

1. Polymorphic functions
A function that can take arguments of different types or return results of different types, is called a polymorphic function. Several examples of polymorphic functions were mentioned in Handout 2. Their types were written with type variables $A, B, \ldots$. For example, the identity function $\lambda x. x$ is of type $A \rightarrow A$ for every type $A$. It can take arguments of any type $A$ and return results of the same type $A$. The function $\text{fst}$ is of type $A \rightarrow B \rightarrow A$ and the function $\text{snd}$ is of type $A \rightarrow B \rightarrow B$ etc.

More important polymorphic functions arise when we deal with data structures. Suppose $\text{list } A$ is the type of lists with elements of type $A$. Then a function for selecting the element at a particular position has the type $\text{list } A \times \text{int } \rightarrow A$. So, the selection function can take as argument lists of different element types $A$ and returns results of the same element type. A function for appending two lists has the type $\text{list } A \times \text{list } A \rightarrow \text{list } A$. A function to sort a list using a particular comparison function has the type $(A \times A \rightarrow \text{bool}) \rightarrow \text{list } A \rightarrow \text{list } A$. All of these are polymorphic functions.

2. Two kinds of polymorphism
Christopher Strachey, writing in 1967, distinguished between two kinds of polymorphic functions.

Ad-hoc polymorphic functions have a finite number of different types and provide separate implementations of the function for each type. Overloading is a good example of ad-hoc polymorphism. In Java, the + symbol is used for functions of type $\text{int } \times \text{int } \rightarrow \text{int}$, $\text{float } \times \text{float } \rightarrow \text{float}$, $\text{double } \times \text{double } \rightarrow \text{double}$ and $\text{String } \times \text{String } \rightarrow \text{String}$. In C++ and Haskell, the programmer can define even more versions of + for other types.

The second form of polymorphism is called parametric polymorphism. Functions exhibiting parametric polymorphism have an infinite number of types, obtained by instantiating type parameters, and use the same implementation for all types. All the functions mentioned in the preceding paragraph are parametrically polymorphic functions. Such functions are also referred to as generic functions. (This is the term used in the Java definition for the concept.)

Our main focus in this handout is on parametric polymorphism.

Polymorphic lambda calculus
A general form of lambda calculus for dealing with polymorphic functions was defined by John Reynolds (in Computer Science) and J.-Y. Girard (in Mathematical Logic) in the mid 70’s. The calculus is called the polymorphic lambda calculus or second-order lambda calculus.

3. Types of the polymorphic lambda calculus
The types of the calculus include, in addition to the types of the Simply Typed Lambda Calculus, type variables (denoted by $t, u, v, \ldots$) and universal quantification on type variables for representing the types of polymorphic functions. So, the syntax of types is given by:

$$T ::= \text{int} \mid \text{bool} \mid \text{list } T \mid t \mid T_1 \rightarrow T_2 \mid \forall t. T$$

Here are some examples of types written using this notation:

- $\text{id} : \forall t. t \rightarrow t$
- $\text{fst} : \forall t. \forall u. t \rightarrow u \rightarrow t$
- $\text{snd} : \forall t. \forall u. t \rightarrow u \rightarrow u$
- $\text{comp} : \forall t. \forall u. \forall v. (u \rightarrow v) \rightarrow (t \rightarrow u) \rightarrow (t \rightarrow v)$
- $\text{select} : \forall t. \text{list } t \rightarrow \text{int } \rightarrow t$
- $\text{append} : \forall t. \text{list } t \rightarrow \text{list } t \rightarrow \text{list } t$
- $\text{sort} : \forall t. (t \rightarrow t \rightarrow \text{bool}) \rightarrow \text{list } t \rightarrow \text{list } t$

To say that $\text{id}$ is of type $\forall t. t \rightarrow t$ is to imply that $\text{id}$ is of type $t \rightarrow t$ for all $t$. So on for all the other examples.

---

1. This language should be called an applied polymorphic lambda calculus. The calculus defined by Girard and Reynolds is the pure polymorphic lambda calculus, which does not have any base types or primitive operations.
It is important to note that $\forall t$ binds the type variable $t$ (just as $\lambda x$ binds the ordinary variable $x$). So, in a type such as $\forall u. t \rightarrow u \rightarrow u$, $u$ is a bound type variable and $t$ is a free type variable. We can substitute free type variables of a type term by other type terms. We use the notation $T[t := T']$ for the result of substituting $t$ by $T'$ in a type term $T$.

4. Terms of the polymorphic lambda calculus
The terms of the polymorphic lambda calculus have the following context-free syntax:

$$M ::= c | x | \lambda x: T. M' | M_1 M_2 | \Lambda t. M' | M'[T]$$

In addition to the term forms of the simply typed lambda calculus, we have added:

- $\Lambda t. M'$, which builds a polymorphic function parametrized by the type parameter $t$, and
- $M'[T]$, which specializes a polymorphic function $M'$ by supplying particular type $T$ for its type parameter.

The two term forms are called type abstraction and type application.
Here are some examples of terms built using this notation:

- $\text{id} = \Lambda t. \lambda x: t. x$
- $\text{fst} = \Lambda t. \Lambda u. \lambda x: t. \lambda y: u. x$
- $\text{snd} = \Lambda t. \Lambda u. \lambda x: t. \lambda y: u. y$
- $\text{comp} = \Lambda t. \Lambda u. \Lambda v. \lambda g: u \rightarrow v. \lambda f: t \rightarrow u. \lambda x: t. g(f x)$

The identity function is defined by parametrizing it by a type variable $t$ and accepting an argument $x$ of type $t$. The result is $x$ which is again of type $t$. Hence we can see that the function is of type $\forall t. t \rightarrow t$.

The type rules of the polymorphic lambda calculus include all the rules of the simply typed lambda calculus and, in addition, the following rules for type abstraction and type application:

$$(\forall \text{Intro}) \quad \frac{\Gamma \vdash M : T \quad \text{if } t \text{ is not a free type variable in } \Gamma}{\Gamma \vdash \Lambda t. M : \forall t. T}$$

$$(\forall \text{Elim}) \quad \frac{\Gamma \vdash M : \forall t. U}{\Gamma \vdash M[T] : U[t := T]}$$

These rules capture the intuition explained above.

Here is an example type derivation for the $\text{fst}$ function:

$$\frac{x: t, y: u \vdash x : t \quad (\rightarrow \text{Intro})}{x: t \vdash \lambda y: u. x : u \rightarrow t \quad (\rightarrow \text{Intro})}$$
$$\frac{\vdash \lambda x: t. \lambda y: u. x : t \rightarrow u \rightarrow t \quad (\rightarrow \text{Intro})}{\vdash \Lambda u. \lambda x: t. \lambda y: u. x : \forall u. t \rightarrow u \rightarrow t \quad (\forall \text{Intro})}$$
$$\frac{\vdash \Lambda u. \lambda x: t. \lambda y: u. x : \forall t. \forall u. t \rightarrow u \rightarrow t \quad (\forall \text{Intro})}{\vdash \Lambda t. \Lambda u. \lambda x: t. \lambda y: u. x : \forall t. \forall u. t \rightarrow u \rightarrow t}$$

5. Constants for the polymorphic calculus
The following constants are inherited from the simply typed lambda calculus of Handout 9:

- The integer constants $\ldots, -2, -1, 0, 1, 2, \ldots$ are all of type $\text{int}$.
- The boolean constants true and false are of type $\text{bool}$.
- The arithmetic operations $+, -, *, /$ and $\text{mod}$ are of type $\text{int} \rightarrow \text{int} \rightarrow \text{int}$.
- The relational operations $=, \neq, <, \leq, >, \geq$ are of type $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$.

We generalize conditional branching and the recursion combinator to polymorphic functions:

- if of type $\forall t. \text{bool} \rightarrow t \rightarrow t \rightarrow t$, and
- $Y$ of type $\forall t. (t \rightarrow t) \rightarrow t$. 
Now we have conditionals for all types. Likewise, the use of the Y combinator was treated as a special construct in the simply typed lambda calculus. Now, it can be treated as a constant since the type system can deal with its polymorphic character.

In addition to these constants, we use the following constants for list operations:

\[
\begin{align*}
\text{nil} & : \forall t. \text{list } t \\
\text{cons} & : \forall t. t \rightarrow \text{list } t \rightarrow \text{list } t \\
\text{car} & : \forall t. \text{list } t \rightarrow t \\
\text{cdr} & : \forall t. \text{list } t \rightarrow \text{list } t \\
\text{null} & : \forall t. t \rightarrow \text{bool}
\end{align*}
\]

6. List examples

We show some examples of polymorphic list operations to illustrate the use of the polymorphic lambda calculus. These terms are a bit cumbersome to write because of the excessive need for type abstraction and type application operations. For clarity, we first show the untyped variants of these functions and then annotate them with the type information.

The operation for selecting \(i\)’th element of a list can be defined in the untyped lambda calculus as follows:

\[
\text{sel} = \text{Y } \lambda f. \lambda l. \lambda i. \quad \text{if } (i = 0) (\text{car } l) \\
(\text{f } (\text{cdr } l) \ (i - 1))
\]

The corresponding definition in the polymorphic lambda calculus is:

\[
\text{sel} = \Lambda t. \text{Y } [\text{list } t \rightarrow \text{int } \rightarrow t] \lambda f: \text{list } t \rightarrow \text{int } \rightarrow t. \\
\lambda l: \text{list } t. \lambda i: \text{int}. \quad \text{if } [l] (i = 0) (\text{car } [l] l) \\
(\text{f } (\text{cdr } [l] l) \ (i - 1))
\]

Note that we first use a type abstraction to make \(\text{sel}\) a polymorphic function. The Y combinator must be specialized to the type of the function being defined, if the constant must be specialized to the result type, and the car and cdr functions must be specialized with the element type of the lists as well.

Similarly, consider the untyped version of the append function:

\[
\text{append} = \text{Y } \lambda f. \lambda l. \lambda m. \quad \text{if } (\text{null } l) m \\
(\text{cons } (\text{car } l) \ (\text{f } (\text{cdr } l) \ m))
\]

The corresponding definition in the polymorphic lambda calculus is:

\[
\text{append} = \Lambda t. \text{Y } [\text{list } t \rightarrow \text{list } t \rightarrow \text{list } t] \lambda f: \text{list } t \rightarrow \text{list } t \rightarrow \text{list } t. \\
\lambda l: \text{list } t. \lambda m: \text{list } t. \quad \text{if } [l] (\text{null } [l] l) m \\
(\text{cons } [l] (\text{car } [l] l) (\text{f } (\text{cdr } [l] l) \ m))
\]

ML Polymorphism

Robin Milner designed the programming language ML to serve as the meta-language for his work on theorem-proving programs. For the type system of ML, he used an interesting variant of the polymorphic lambda calculus which places a few simple restrictions on the type system so that the type annotations of the functions can be automatically inferred. This variant is dubbed ML Polymorphism or the Hindley-Milner type system and became highly popular in the subsequent programming language designs, including Haskell and Java.

7. Types of ML

ML stratifies its types into two layers. The bottom layer consists of simple types with type variables. The upper layer consists of type schemes, which are true polymorphic types. Here is the syntax:

\[
\begin{align*}
\text{(Simple types)} & \quad T ::= \text{int} \ | \ \text{bool} \ | \ T \ | \ T_1 \rightarrow T_2 \\
\text{(Type schemes)} & \quad S ::= T \ | \ \forall t. S
\end{align*}
\]

Note that every simple type can be regarded as a type scheme. But a nontrivial type scheme with universal quantifiers is not a simple type.

The stratification places two restrictions. First, the polymorphic types do not participate in function types or other structured types. So, the arguments and results of functions and the elements of data structures can only be of simple types. They cannot be polymorphic themselves. Second, the type variables only range over simple types. We cannot instantiate a type variable \(t\) to another polymorphic type. Both of these restrictions have been found to be only mild restrictions in practical programming.
8. Terms of ML

The terms of ML include the usual terms of the simply typed lambda calculus, but with the addition of the let construct. Normally, we treat let as syntactic sugar for the application of a lambda function. However, in ML, let is treated specially, as a conduit for polymorphism. So, ML’s let cannot be desugared into application of lambda functions.

Here are the type rules of ML:

\[
\begin{align*}
\text{(Const)} & \quad \Gamma \vdash c : S \\
\text{(Variable)} & \quad \Gamma \vdash x : S \\
\text{(--Intro)} & \quad \Gamma, x : T \vdash M : T' \\
\quad & \quad \Gamma \vdash \lambda x. M : T \rightarrow T' \\
\text{(--Elim)} & \quad \Gamma \vdash M : T_1 \rightarrow T_2 \quad \Gamma \vdash N : T_1 \\
\quad & \quad \Gamma \vdash M \; N : T_2 \\
\text{(\forall Intro)} & \quad \Gamma \vdash M : S \\
\quad & \quad \Gamma \vdash M : \forall t. S \\
\text{(\forall Elim)} & \quad \Gamma \vdash M : S[t := T] \\
\text{(LET)} & \quad \Gamma \vdash \text{let } x = M \text{ in } N : T
\end{align*}
\]

These rules have subtle interactions. So, let us explain. The type rules for abstraction and application work with only simple types. However, the rule for let uses type schemes for the let-bound identifier. How do we go from simple types to type schemes and vice versa? Via the \forall Intro and \forall Elim rules. These rules work “silently” in that there is nothing in the term to indicate that these rules should be applied. However, in practice the let rule controls their usage. We use \forall Intro to obtain the type of the let-bound identifier and, then, we use \forall Elim to eliminate all the quantifiers when the identifier is used in the let body.

Let us illustrate the operation of the rules via an example term:

\[
\begin{align*}
\text{let } id & = \lambda x. x \\
\text{in } id \; id
\end{align*}
\]

This term cannot be type checked in the simply typed lambda calculus. We have argued that the self-application of any term, \( M \; M \), cannot be type checked in the simply typed lambda calculus. However, in ML, the term type checks fine. We first obtain the type \( t \rightarrow t \) for \( \lambda x. x \). Then we infer the type scheme \( \forall t. t \rightarrow t \) for the let-bound identifier \( id \). We can instantiate \( t \) to \( u \) and obtain the type \( id : u \rightarrow u \). We can also instantiate \( t \) to \( u \rightarrow u \) to obtain the type \( id : (u \rightarrow u) \rightarrow (u \rightarrow u) \). So, we can infer that \( id \; id \) is of type \( u \rightarrow u \). Here is the full type derivation:

\[
\begin{align*}
\Gamma \vdash x : t & \quad \text{Variable} \\
\Gamma \vdash \lambda x. x : t \rightarrow t & \quad \text{\forall Intro} \\
\Gamma \vdash \lambda x. x : \forall t. t \rightarrow t & \quad \text{\forall Elim} \\
\Gamma \vdash id : \forall t. t \rightarrow t & \quad \text{\forall Elim} \\
\Gamma \vdash id : u \rightarrow u & \quad \text{\forall Elim} \\
\Gamma \vdash id \; id : u \rightarrow u & \quad \text{LET}
\end{align*}
\]

where \( \Gamma \) is the typing context \( id : \forall t. t \rightarrow t \).

Similarly the term

\[
\begin{align*}
\text{let } twice & = \lambda f. \lambda x. f \; (f \; x) \\
\text{in } twice \; twice
\end{align*}
\]

can be type checked in ML. We first infer the type of the let-bound identifier \( twice \) to be \( \forall t. (t \rightarrow t) \rightarrow (t \rightarrow t) \). Then the two occurrences of \( twice \) in the let body can be given the types

\[
\begin{align*}
((u \rightarrow u) \rightarrow (u \rightarrow u)) \rightarrow ((u \rightarrow u) \rightarrow (u \rightarrow u)) \\
((u \rightarrow u) \rightarrow (u \rightarrow u))
\end{align*}
\]
9. Type inference

Type checking using the above type rules of ML is not straightforward. Note that the types of lambda-bound variables are not declared. The type system is expected to “guess” their types. Similarly, the $\forall$Elim rule requires the type checker to “guess” how to instantiate the type variables. Carrying out this form of “guessing” is called type inference. In addition to designing the type system for ML, Milner devised an algorithm for type inference for the type system.

The general idea is simple. Whenever any type needs to be “guessed” create a fresh type variable and use it in place the type. When terms are type checked, the use of the application rule places constraints on the types involved. Recall the type checking rule for applications:

\[
\text{An application term } (M \, T \, N \, T') \text{ is type correct only if } T \text{ is a function type of the form } T_1 \to T_2, \text{ and the type of the argument } T' \text{ is equal to the argument type of the function, i.e., } T_1 = T'. \text{ In that case, } \\
\text{the application term is annotated with the type } T_2. \text{ If the rules are not satisfied, then the term is not type correct.}
\]

During type inference, we may not have concrete types for $T$ and $T'$; they could just be type variables. So, we adapt it to the following type inference rule:

To type check an application term $(M \, T \, N \, T')$, postulate two fresh type variables $t_1$ and $t_2$ and place the following constraints:

\[
T = t_1 \to t_2 \quad \text{and} \quad T = t_1.
\]

While type checking a term, we accumulate a number of equational constraints of this form. When a term is used as the definition of a let-bound identifier, we solve all the accumulated constraints to find the most general solution of the equations. The type of the term using the most general solution is called the principal type of the term.

10. Example of principal type calculation

We illustrate the process for the function term $\lambda f. \lambda x. f \, (f \, x)$.

- Assume a type $t_1$ for $f$.
- Assume a type $t_2$ for $x$.
- Type checking $f \, x$ places the constraints $t_1 = t_3 \to t_4$ and $t_2 = t_3$. The type of $f \, x$ is now $t_4$.
- Type checking $f \, (f \, x)$ places the constraints $t_1 = t_5 \to t_6$ and $t_4 = t_5$. The type of the term is $t_6$.
- The type of term $\lambda f. \lambda x. f \, (f \, x)$ is now $t_1 \to t_2 \to t_6$.

The accumulated equations are:

\[
\begin{align*}
  t_1 &= t_3 \to t_4 \\
  t_2 &= t_3 \\
  t_1 &= t_5 \to t_6 \\
  t_4 &= t_5
\end{align*}
\]

Substituting $t_3 \to t_4$ for $t_1$ in the remaining equations, we get:

\[
\begin{align*}
  t_1 &= t_3 \to t_4 \\
  t_2 &= t_3 \\
  t_3 \to t_4 &= t_5 \to t_6 \\
  t_4 &= t_5
\end{align*}
\]

The third equation can be decomposed:

\[
\begin{align*}
  t_1 &= t_3 \to t_4 \\
  t_2 &= t_3 \\
  t_3 &= t_5 \\
  t_4 &= t_6 \\
  t_4 &= t_5
\end{align*}
\]

Substituting equals for equals and eliminating unnecessary variables gives the equations:

\[
\begin{align*}
  t_1 &= t_6 \to t_6 \\
  t_2 &= t_6
\end{align*}
\]

Thus we obtain that the principal type of $\lambda f. \lambda x. f \, (f \, x)$ is $(t_6 \to t_6) \to t_6 \to t_6$.

Next, consider the term

\[
\text{let } \textit{twice} = \lambda f. \lambda x. f \, (f \, x) \\
\text{in } \textit{twice}
\]
To obtain the type of the let-bound identifier `twice`, we introduce quantifiers for all the free type variables in the type of $\lambda f. \lambda x. f (f x)$. We can also rename $t_0$ to $t$ in the process. So, the type of `twice` is $\forall t. (t \to t) \to t \to t$.

To type check `twice twice`, we associate each occurrence of `twice` with a generic instance of its inferred type, i.e., we replace its quantified type variables with fresh variables. Let the two generic instances be $(u_1 \to u_1) \to u_1 \to u_1$ and $(u_2 \to u_2) \to u_2 \to u_2$. Using the above algorithm again to type check `twice twice`, we obtain the equations:

$$(u_1 \to u_1) \to u_1 \to u_1 = t_1 \to t_2$$
$$(u_2 \to u_2) \to u_2 \to u_2 = t_1$$

The type of the term is $t_2$. Simplifying the equations, we obtain:

$t_1 = (u_2 \to u_2) \to u_2 \to u_2$
$t_2 = (u_2 \to u_2) \to u_2 \to u_2$
$u_1 = u_2$

So, the principal type of the term is $(u \to u) \to u \to u$.

11. Polymorphism in Java

Polymorphism in Java (called “generics”) is similar to that of ML, but it also borrows the notation of the polymorphic lambda calculus for specifying type parameters. Like ML, the type variables in Java can only range over simple types. The Java type checker can also do a significant amount of type inference. However, since Java requires all parameter types to be declared, notation is needed for declaring type parameters. Instead of lambda abstraction, parameter declarations are included in the procedure headers. Type parameters are enclosed in angle brackets `<>` instead of square brackets (since the latter are reserved for array subscripting). The use of instance methods reduces the amount of type annotations necessary. To illustrate the correspondence, we show the append function of paragraph 6 in Java:

```java
static <t> list<t> append(list<t> l, list<t> m) {
    if (l.null())
        return m;
    else
        return <t>cons(l.car(), <t>append(l.cdr(), m))
}
```

The first occurrence of `<t>` in the header corresponds to the type abstraction $\Lambda t$. The type parameter $t$ is then used freely in declaring the types of the parameters. We have turned `null`, `car` and `cdr` into instance methods. So, they do not need type parameters. However, `cons` still needs the type parameter if it is treated as a static method, as does the recursive call to `append`. However, the Java type checker can infer the type parameter in both of these cases (using an ML-like type checking engine).