Parametric polymorphism, Records, and Subtyping

Lecture 14 Tuesday, March 10, 2022

1 Parametric polymorphism

Polymorph means "many forms". *Polymorphism* is the ability of code to be used on values of different types. For example, a polymorphic function is one that can be invoked with arguments of different types. A polymorphic datatype is one that can contain elements of different types.

Several kinds of polymorphism are commonly used in modern languages.

- *Subtype polymorphism* gives a single term many types using the subsumption rule. For example, a function with argument τ can operate on any value with a type that is a subtype of τ.
- *Ad-hoc polymorphism* usually refers to code that appears to be polymorphic to the programmer, but the actual implementation is not. A typical example is *overloading*: using the same function name for functions with different kinds of parameters. Although it looks like a polymorphic function to the code that uses it, there are actually multiple function implementations (none being polymorphic) and the compiler invokes the appropriate one. Ad-hoc polymorphism is a dispatch mechanism: the type of the arguments is used to determine (either at compile time or run time) which code to invoke.
- *Parametric polymorphism* refers to code that is written without knowledge of the actual type of the arguments; the code is parametric in the type of the parameters. Examples include polymorphic functions in ML, or generics in Java 5.

We consider parametric polymorphism in more detail. Suppose we are working in the simply-typed lambda calculus, and consider a "doubling" function for integers that takes a function f, and an integer x, applies f to x, and then applies f to the result.

$$\mathit{doubleInt} \triangleq \lambda f\!:\!\textsf{int} \to \textsf{int}.\ \lambda x\!:\!\textsf{int}.\ f\ (f\ x)$$

We could also write a double function for booleans. Or for functions over integers. Or for any other type...

$$\begin{aligned}\mathit{doubleBool} &\triangleq \lambda f\!:\!\textsf{bool} \to \textsf{bool}.\ \lambda x\!:\!\textsf{bool}.\ f\ (f\ x)\\ \mathit{doubleFn} &\triangleq \lambda f\!:\!(\textsf{int} \to \textsf{int}) \to (\textsf{int} \to \textsf{int}).\ \lambda x\!:\!\textsf{int} \to \textsf{int}.\ f\ (f\ x)\\ &\vdots\end{aligned}$$

In the simply typed lambda calculus, if we want to apply the doubling operation to different types of arguments in the same program, we need to write a new function for each type. This violates the *abstraction principle* of software engineering:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. When similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

In the doubling functions above, the varying parts are the types. We need a way to abstract out the type of the doubling operation, and later instantiate this abstract type with different concrete types.

We extend the simply-typed lambda calculus with abstraction over types, giving the *polymorphic lambda calculus*, also called *System F*.

A *type abstraction* is a new expression, written $\Lambda X.\ e$, where Λ is the upper-case form of the Greek letter lambda, and X is a *type variable*. We also introduce a new form of application, called *type application*, or *instantiation*, written $e_1\ [\tau]$.

When a type abstraction meets a type application during evaluation, we substitute the free occurrences of the type variable with the type. Note that instantiation does not require the program to keep run-time type information, or to perform type checks at run-time; it is just used as a way to statically check type safety in the presence of polymorphism.

1.1 Syntax and operational semantics

The new syntax of the language is given by the following grammar.

$$e ::= n \mid x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e\ [\tau]$$
$$v ::= n \mid \lambda x{:}\tau.\, e \mid \Lambda X.\, e$$

The evaluation rules for the polymorphic lambda calculus are the same as for the simply-typed lambda calculus, augmented with new rules for evaluating type application.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E\ [\tau]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\text{TYPE-REDUCTION}\ \frac{}{(\Lambda X.\, e)\ [\tau] \longrightarrow e\{\tau/X\}}$$

Let's consider an example. In this language, the polymorphic identity function is written as

$$ID \triangleq \Lambda X.\, \lambda x{:}X.\, x$$

We can apply the polymorphic identity function to **int**, producing the identity function on integers.

$$(\Lambda X.\, \lambda x{:}X.\, x)\ [\textbf{int}] \longrightarrow \lambda x{:}\textbf{int}.\, x$$

We can apply ID to other types as easily:

$$(\Lambda X.\, \lambda x{:}X.\, x)\ [\textbf{int} \to \textbf{int}] \longrightarrow \lambda x{:}\textbf{int} \to \textbf{int}.\, x$$

1.2 Type system

We also need to provide a type for the new type abstraction. The type of $\Lambda X.\, e$ is $\forall X.\, \tau$, where τ is the type of e, and may contain the type variable X. Intuitively, we use this notation because we can instantiate the type expression with any type for X: for any type X, expression e can have the type τ (which may mention X).

$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\, \tau$$

Type checking expressions is slightly different than before. Besides the type environment Γ (which maps variables to types), we also need to keep track of the set of type variables Δ. This is to ensure that a type variable X is only used in the scope of an enclosing type abstraction $\Lambda X.\, e$. Thus, typing judgments are now of the form $\Delta, \Gamma \vdash e : \tau$, where Δ is a set of type variables, and Γ is a typing context. We also use an additional judgment $\Delta \vdash \tau\ \mathsf{ok}$ to ensure that type τ uses only type variables from the set Δ.

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}} \qquad \frac{\Delta \vdash \tau\ \mathsf{ok}}{\Delta, \Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \frac{\Delta, \Gamma, x{:}\tau \vdash e : \tau' \quad \Delta \vdash \tau\ \mathsf{ok}}{\Delta, \Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Delta \cup \{X\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda X.\, e : \forall X.\, \tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall X.\, \tau' \quad \Delta \vdash \tau\ \mathsf{ok}}{\Delta, \Gamma \vdash e\ [\tau] : \tau'\{\tau/X\}}$$

$$\frac{}{\Delta \vdash X\ \mathsf{ok}}\ X \in \Delta \qquad \frac{}{\Delta \vdash \mathbf{int}\ \mathsf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ok} \quad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \mathsf{ok}} \qquad \frac{\Delta \cup \{X\} \vdash \tau\ \mathsf{ok}}{\Delta \vdash \forall X.\, \tau\ \mathsf{ok}}$$

1.3 Examples

Let's consider the doubling operation again. We can write a polymorphic doubling operation as

$$\mathit{double} \triangleq \Lambda X.\, \lambda f : X \to X.\, \lambda x : X.\, f\ (f\ x).$$

The type of this expression is

$$\forall X.\, (X \to X) \to X \to X$$

We can instantiate this on a type, and provide arguments. For example,

$$\begin{aligned} \mathit{double}\ [\mathbf{int}]\ (\lambda n : \mathbf{int}.\, n + 1)\ 7 &\longrightarrow (\lambda f : \mathbf{int} \to \mathbf{int}.\, \lambda x : \mathbf{int}.\, f\ (f\ x))\ (\lambda n : \mathbf{int}.\, n + 1)\ 7 \\ &\longrightarrow^* 9 \end{aligned}$$

Recall that in the simply-typed lambda calculus, we had no way of typing the expression $\lambda x.\, x\ x$. In the polymorphic lambda calculus, however, we can type this expression if we give it a polymorphic type and instantiate it appropriately.

$$\vdash \quad \lambda x : \forall X.\, X \to X.\, x\ [\forall X.\, X \to X]\ x \quad : \quad (\forall X.\, X \to X) \to (\forall X.\, X \to X)$$

1.4 Erasure

The semantics of System F presented above explicitly passes type. In an implementation, one often wants to eliminate types for efficiency. The following translation "erases" the types from a System F expression.

$$\begin{aligned} \mathit{erase}(x) &= x \\ \mathit{erase}(n) &= n \\ \mathit{erase}(\lambda x : \tau.\, e) &= \lambda x.\, \mathit{erase}(e) \\ \mathit{erase}(e_1\ e_2) &= \mathit{erase}(e_1)\ \mathit{erase}(e_2) \\ \mathit{erase}(\Lambda X.\, e) &= \lambda z.\, \mathit{erase}(e) && \text{where } z \notin FV(e) \\ \mathit{erase}(e\ [\tau]) &= \mathit{erase}(e)(\lambda x.\, x) \end{aligned}$$

The following theorem states that the translation is adequate.

Theorem 1 (Adequacy)**.** *For all expressions e and e', we have $e \longrightarrow^* e'$ iff $\mathit{erase}(e) \longrightarrow^* \mathit{erase}(e')$.*

The type reconstruction problem asks whether, for a given untyped λ-calculus expression e' there exists a well-typed System F expression e such that $\mathit{erase}(e) = e'$. It was shown to be undecidable by Wells in 1994, by showing that type checking is undecidable for a variant of untyped λ-calculus without annotations. See Pierce Chapter 23 for further discussion, and restrictions of System F for which type reconstruction is decidable.

2 Records

We have previously seen binary products, i.e., pairs of values. Binary products can be generalized in a straightforward way to n-ary products, also called *tuples*. For example, $\langle 3, (), \mathsf{true}, 42\rangle$ is a 4-ary tuple containing an integer, a unit value, a boolean value, and another integer. Its type is $\mathbf{int} \times \mathbf{unit} \times \mathbf{bool} \times \mathbf{int}$.

Records are a generalization of tuples. We annotate each field of record with a *label*, drawn from some set of labels $\mathcal{L}$. For example, $\{\mathsf{foo} = 32, \mathsf{bar} = \mathsf{true}\}$ is a record value with an integer field labeled foo and a boolean field labeled bar. The type of the record value is written $\{\mathsf{foo} : \mathbf{int}, \mathsf{bar} : \mathbf{bool}\}$.

We extend the syntax, operational semantics, and typing rules of the call-by-value lambda calculus to support records.

$$
\begin{aligned}
& l \in \mathcal{L} \\
& e ::= \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\
& v ::= \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
& \tau ::= \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{aligned}
$$

We add new evaluation contexts to evaluate the fields of records.

$$E ::= \cdots \mid \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \mid E.l$$

We also add a rule to access the field of a record.

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \longrightarrow v_i}$$

Finally, we add new typing rules for records. Note that the order of labels is important: the type of the record value $\{\mathsf{lat} = -40, \mathsf{long} = 175\}$ is $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$, which is different from $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$, the type of the record value $\{\mathsf{long} = 175, \mathsf{lat} = -40\}$. In many languages with records, the order of the labels is not important; indeed, we will consider weakening this restriction in the next section.

$$\frac{\forall i \in 1..n. \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

3 Subtyping

Subtyping is a key feature of object-oriented languages. Subtyping was first introduced in SIMULA, invented by Norwegian researchers Dahl and Nygaard, and considered the first object-oriented programming language.

The principle of subtyping is as follows. If τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$, and also sometimes as $\tau_1 \leq: \tau_2$), then a program can use a value of type τ_1 whenever it would use a value of type τ_2. If $\tau_1 \leq \tau_2$, then τ_1 is sometimes referred to as the subtype, and τ_2 as the supertype.

We can express the principle of subtyping in a typing rule, often referred to as the "subsumption typing rule" (since the supertype subsumes the subtype).

$$\textsc{Subsumption}\ \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

The subsumption rule says that if e is of type τ, and τ is a subtype of τ', then e is also of type τ'.

Recall that we provided an intuition for a type as a set of computational entities that share some common property. Type τ is a subtype of type τ' if every computational entity in the set for τ can be regarded as a computational entity in the set for τ'.

So what types are in a subtype relation? We will define inference rules and axioms for the subtype relation $\leq$.

The subtype relation is both reflexive and transitive. These properties both seem reasonable if we think of subtyping as a subset relation. We add inference rules that express this.

$$\frac{}{\tau \leq \tau} \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

3.1 Subtyping for records

Consider records and record types. A record consists of a set of labeled fields. Its type includes the types of the fields in the record. Let's define the type **Point** to be the record type $\{\mathsf{x}:\textbf{int}, \mathsf{y}:\textbf{int}\}$, that contains two fields x and y, both integers. That is:

$$\textbf{Point} = \{\mathsf{x}:\textbf{int}, \mathsf{y}:\textbf{int}\}.$$

Lets also define

$$\textbf{Point3D} = \{\mathsf{x}:\textbf{int}, \mathsf{y}:\textbf{int}, \mathsf{z}:\textbf{int}\}$$

as the type of a record with three integer fields x, y and z.

Because **Point3D** contains all of the fields of **Point**, and those have the same type as in **Point**, it makes sense to say that **Point3D** is a subtype of **Point**: **Point3D** $\leq$ **Point**.

Think about any code that used a value of type **Point**. This code could access the fields x and y, and that's pretty much all it could do with a value of type **Point**. A value of type **Point3D** has these same fields, x and y, and so any piece of code that used a value of type **Point** could instead use a value of type **Point3D**.

We can write a subtyping rule for records.

$$\frac{}{\{l_1:\tau_1, \ldots, l_{n+k}:\tau_{n+k}\} \leq \{l_1:\tau_1, \ldots, l_n:\tau_n\}}\, k \geq 0$$

But why not let the corresponding fields be in a subtyping relation? For example, if $\tau_1 \leq \tau_2$ and $\tau_3 \leq \tau_4$, then is $\{\mathsf{foo} : \tau_1, \mathsf{bar} : \tau_3\}$ a subtype of $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$? Turns out that this is the case so long as the fields of records are immutable. More on this when we consider subtyping for references.

Also, we could relax the requirement that the order of fields must be the same. The following is a more permissive subtyping rule for records.

$$\frac{\forall i \in 1..n.\ \exists j \in 1..m. \quad l'_i = l_j \ \wedge \ \tau_j \leq \tau'_i}{\{l_1:\tau_1, \ldots, l_m:\tau_m\} \leq \{l'_1:\tau'_1, \ldots, l'_n:\tau'_n\}}$$

3.2 Subtyping for products

Like records, we can allow the elements of a product to be in a subtyping relation.

$$\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2}$$

3.3 Subtyping for functions

Consider two function types $\tau_1 \to \tau_2$ and $\tau'_1 \to \tau'_2$. What are the subtyping relations between τ_1,, τ_2, τ'_1,, and τ'_2 that should be satisfied in order for $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$ to hold?

Consider the following expression:

$$G \triangleq \lambda f:\tau'_1 \to \tau'_2.\, \lambda x:\tau'_1.\, f\ x.$$

This function has type

$$(\tau'_1 \to \tau'_2) \to \tau'_1 \to \tau'_2.$$

Now suppose we had a function $h:\tau_1 \to \tau_2$ such that $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$. By the subtyping principle, we should be able to give h as an argument to G, and G should work fine. Suppose that v is a value of type τ'_1.

Then $G\ h\ v$ will evaluate to $h\ v$, meaning that h will be passed a value of type τ_1. Since h has type $\tau_1 \to \tau_2$, it must be the case that $\tau_1' \leq \tau_1$. (What could go wrong if $\tau_1 \leq \tau_1'$?)

Furthermore, the result type of $G\ h\ v$ should be of type τ_2' according to the type of G, but $h\ v$ will produce a value of type τ_2, as indicated by the type of h. So it must be the case that $\tau_2 \leq \tau_2'$.

Putting these two pieces together, we get the typing rule for function types.

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Note that the subtyping relation between the argument and result types in the premise are in different directions! The subtype relation for the result type is in the same direction as for the conclusion (primed version is the supertype, non-primed version is the subtype); it is in the opposite direction for the argument type. We say that subtyping for the function type is *covariant* in the result type, and *contravariant* in the argument type.

3.4 Subtyping for locations

Suppose we have a location l of type τ **ref**, and a location l' of type τ' **ref**. What should the relationship be between τ and τ' in order to have τ **ref** $\leq \tau'$ **ref**?

Let's consider the following program R, that takes a location x of type τ' **ref** and reads from it.

$$R \triangleq \lambda x{:}\tau'\ \textbf{ref}.\ !x$$

The program R has the type τ' **ref** $\to \tau'$. Suppose we gave R the location l as an argument. Then $R\ l$ will look up the value stored in l, and return a result of type τ (since l is type τ **ref**. Since R is meant to return a result of type τ' **ref**, we thus want to have $\tau \leq \tau'$.

So this suggests that subtyping for reference types is covariant.

But consider the following program W, that takes a location x of type τ' **ref**, a value y of type τ', and writes y to the location.

$$W \triangleq \lambda x{:}\tau'\ \textbf{ref}.\ \lambda y{:}\tau'.\ x := y$$

This program has type τ' **ref** $\to \tau' \to \tau'$.

Suppose we have a value v of type τ', and consider the expression $W\ l\ v$. This will evaluate to $l := v$, and since l has type τ **ref**, it must be the case that v has type τ, and so $\tau' \leq \tau$. But this suggests that subtyping for reference types is contravariant!

In fact, subtyping for reference types must be *invariant*: reference type τ **ref** is a subtype of τ' **ref** if and only if $\tau \leq \tau'$ and $\tau' \leq \tau$. Indeed, to be sound, subtyping for any mutable location must be invariant.

$$\frac{\tau \leq \tau' \qquad \tau' \leq \tau}{\tau\ \textbf{ref} \leq \tau'\ \textbf{ref}}$$

(In the premises for the rule above, why isn't $\tau \leq \tau'$ and $\tau' \leq \tau$ equivalent to τ and τ' being exactly the same? To see why not, consider the record types $\{\mathsf{foo}{:}\textbf{int}, \mathsf{bar}{:}\textbf{int}\}$ and $\{\mathsf{bar}{:}\textbf{int}, \mathsf{foo}{:}\textbf{int}\}$.)

Interestingly, in the Java programming language, arrays are mutable locations but have covariant subtyping!

Suppose that we have two classes Person and Student such that Student extends Person (that is, Student is a subtype of Person). The following Java code is accepted, since an array of Student is a subtype of an array of Person, according to Java's covariant subtyping for arrays.

```
Person[] arr = new Student[] { new Student("Alice") };
```

This is fine as long as we only read from arr. The following code executes without any problems, since arr[0] is a Student which is a subtype of Person.

```
Person p = arr[0];
```

However, the following code, which attempts to update the array, has some issues.

$$\mathsf{arr}[0] = \mathsf{new\ Person}(\text{“Bob”});$$

Even though the assignment is well-typed, it attempts to assign an object of type Person into an array of Students! In Java, this produces an ArrayStoreException, indicating that the assignment to the array failed.