Parametric polymorphism, Records, and Subtyping

CS 1520 (Spring 2025)

Harvard University

Thursday, March 13, 2025

Today, we will learn about

- Parametric polymorphism
- Records
- Subtyping: Covariant, Contravariant, Invariant

Polymorphism

Polymorphism

- *Polymorph* means “many forms”.
- *Polymorphism* is the ability of code to be used on values of different types.
- E.g. 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.

Polymorphism used in modern languages

Polymorphism used in modern languages: Subtype polymorphism

- Gives a single term many types using the subsumption rule.
- E.g. a function with argument τ can operate on any value with a type that is a subtype of τ.

Polymorphism used in modern languages: Ad-hoc polymorphism

- The code 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.

Polymorphism used in modern languages: Ad-hoc polymorphism

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.

Polymorphism used in modern languages: 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.

Suppose we are working in the simply-typed lambda calculus, and consider a "doubling" function for integers

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

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

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

$$\mathit{doubleBool} \triangleq \lambda f : \mathbf{bool} \to \mathbf{bool}.\ \lambda x : \mathbf{bool}.\ f\ (f\ x)$$

$$\mathit{doubleFn} \triangleq \lambda f : (\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int}).\ \lambda x : \mathbf{int} \to \mathbf{int}.\ f\ (f\ x)$$

$$\vdots$$

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

In the simply typed lambda calculus, we need to write a new function for each type.

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

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.*

Parametric polymorphism: System F

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

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 $e_1\ [\tau]$.

System F

- When a type abstraction meets a type application during evaluation, we substitute the free occurrences of the type variable with the type.
- 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.

System F: Syntax

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

System F: Operational Semantics

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

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-reduction}\ \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\textsc{Type-reduction}\ \frac{}{(\Lambda X.\, e)\ [\tau] \longrightarrow e\{\tau/X\}}$$

System F: Example

In this language, the polymorphic identity function is written as

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

System F:

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

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

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

We can apply *ID* to other types as easily:

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

System F: Type system

The type of $\Lambda X.\, e$ is $\forall X.\, \tau$, where τ is the type of e, and may contain the type variable X. We use this notation because 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

- 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 $\Delta \vdash \tau$ ok to ensure that type τ uses only type variables from the set Δ.

Type checking expressions

$$\frac{}{\Delta, \Gamma \vdash n : \mathbf{int}} \qquad \frac{\Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash x : \tau}\, \Gamma(x) = \tau$$

$$\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'}$$

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

$$\frac{\Delta, \Gamma \vdash e : \forall X.\, \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash e\ [\tau] : \tau'\{\tau/X\}}$$

Type checking expressions

$$\frac{}{\Delta \vdash X \text{ ok}}\, X \in \Delta$$

$$\frac{}{\Delta \vdash \textbf{int} \text{ ok}}$$

$$\frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \to \tau_2 \text{ ok}}$$

$$\frac{\Delta \cup \{X\} \vdash \tau \text{ ok}}{\Delta \vdash \forall X.\, \tau \text{ ok}}$$

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

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

Example:

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

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

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

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

Example: $\lambda x.\, x\ x$

In the simply-typed lambda calculus, we had no way of typing the expression $\lambda x.\, x\ x$.

Example: $\lambda x.\, x\ x$

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:

$$\begin{aligned} \vdash \quad & \lambda x{:}\forall X.\ X \to X.\, x\ [\forall X.\ X \to X]\ x \\ : \quad & (\forall X.\ X \to X) \to (\forall X.\ X \to X) \end{aligned}$$

Records

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, $< 3, (), \mathsf{true}, 42 >$ 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

- *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 $\{\mathsf{foo} : \mathbf{int}, \mathsf{bar} : \mathbf{bool}\}$.

Records

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

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

Records: Evaluation contexts

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

Records: Evaluation rules

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

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

Records: Typing rules

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

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

Records: Typing rules

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

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

- The order of labels is important:
  $\{\mathsf{lat} = -40, \mathsf{long} = 175\}$ has type $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$,
  while $\{\mathsf{long} = 175, \mathsf{lat} = -40\}$ has type
  $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$.

- We will consider weakening this restriction in the next section.

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

- If τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$), 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.

The principle of subtyping

This is also referred to as the "subsumption typing rule" and can be expressed in a typing rule

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

Subtyping

The subtype relation is both reflexive and transitive.

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

Subtyping for records

Let's define the type **Point** to be the record type $\{\mathsf{x} : \mathbf{int}, \mathsf{y} : \mathbf{int}\}$, that contains two fields x and y, both integers. That is:

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

Subtyping for records

Let's 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.

Subtyping for records

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

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

- 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 $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$?
- This is the case so long as the fields of records are immutable.

Subtyping for records

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

Subtyping for products

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

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

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?

Subtyping for functions

Consider the following expression:

$$G \triangleq \lambda f : \tau_1' \to \tau_2'.\, \lambda x : \tau_1'.\, f\ x.$$

This function has type

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

Subtyping for functions:

$$G \triangleq \lambda f : \tau_1' \to \tau_2'.\, \lambda x : \tau_1'.\, f\ x.$$

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

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.

Subtyping for functions:

$$G \triangleq \lambda f : \tau_1' \to \tau_2'.\ \lambda x : \tau_1'.\ f\ x.$$

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

$$h : \tau_1 \to \tau_2$$

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$$

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'.

Subtyping for functions:

$$G \triangleq \lambda f : \tau_1' \to \tau_2'.\, \lambda x : \tau_1'.\, f\ x.$$

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

$$h : \tau_1 \to \tau_2$$

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$$

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 $\tau_1 \leq \tau_1'$?)

Subtyping for functions:

$$G \triangleq \lambda f : \tau_1' \to \tau_2'.\, \lambda x : \tau_1'.\, f\ x.$$

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

$$h : \tau_1 \to \tau_2$$

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$$

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 $\tau_2 \leq \tau_2'$.

Subtyping for functions

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

Subtyping for functions

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

In this case subtyping for the function type is *covariant* in the result type, and *contravariant* in the argument type.

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**?

Subtyping for locations

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

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

Subtyping for locations

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

The program R has the type $\tau'\ \mathbf{ref} \to \tau'$. 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 $\tau\ \mathbf{ref}$).

Subtyping for locations

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

Since R is meant to return a result of type τ' **ref**, we thus want to have $\tau \leq \tau'$.

Subtyping for locations:

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** $\to \tau' \to \tau'$.

Subtyping for locations:

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

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 $\tau' \leq \tau$.

Subtyping for locations

But this suggests that subtyping for reference types is contravariant!

Subtyping for locations: Invariant subtyping

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.

Subtyping for locations: Invariant subtyping

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.

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

Invariant subtyping

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

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?

Invariant subtyping

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

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 $\{\text{foo} : \textbf{int}, \text{bar} : \textbf{int}\}$ and $\{\text{bar} : \textbf{int}, \text{foo} : \textbf{int}\}$.

Invariant vs Covariant subtyping: Java

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

Invariant vs Covariant subtyping: Java

Suppose that we have two classes Person and Student such that Student extends Person (that is, Student is a subtype of Person).

Invariant vs Covariant subtyping: Java

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") };
```

Invariant vs Covariant subtyping: Java

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];
```

Invariant vs Covariant subtyping: Java

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

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

Invariant vs Covariant subtyping: Java

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

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.

Back to System F

Metatheory

- Safety: Language is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Surprising — we saw self-application!
- Parametricity, a.k.a. theorems for free
  - Example: If $\vdash e : \forall X.\ \forall Y.\ (X \times Y) \to (Y \times X)$, then e is equivalent to $\Lambda X.\, \Lambda Y.\, \lambda x{:}(X \times Y).\, (\#2\ x, \#1\ x)$.
    Every term with this type is the swap function!!

    Intuition: e has no way to make an X or a Y and it cannot tell what X or Y are or raise an exception or diverge...
- Erasure: Types do not affect run-time behavior

Erasure

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

Adequacy

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

Type Reconstruction

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.

Connection to reality

System F has been one of the most important theoretical PL models since the 1970s and inspires languages like ML.

But you have seen ML polymorphism and it looks different. In fact, it is an implicitly typed restriction of System F.

These two qualifications ((1) implicit, (2) restriction) are deeply related.

Restrictions

- All types have the form $\forall X_1, \ldots, X_n.\tau$ where $n \geq 0$ and τ has no $\forall$. (Prenex-quantification; no first-class polymorphism.)
- Only let (rec) variables (e.g., `x` in `let x = e1 in e2`) can have polymorphic types. So $n = 0$ for function arguments, pattern variables, etc. (Let-bound polymorphism)
  - So cannot (always) desugar let to λ in ML
- In `let rec f x = e1 in e2`, the variable `f` can have type $\forall X_1, \ldots, X_n.\tau_1 \to \tau_2$ only if every use of `f` in `e1` instantiates each X_i with X_i. (No polymorphic recursion)

Restrictions (continued)

- Let variables can be polymorphic only if `e1` is a "syntactic value"
  - A variable, constant, function definition, ...
  - Called the "value restriction" (relaxed partially in OCaml)

Why?

ML-style polymorphism can seem weird after you have seen System F. And the restrictions do come up in practice, though tolerable.

- Type inference for System F (given untyped e, is there a System F term e' such that $\mathit{erase}(e') = e$) is undecidable (1995)
- Type inference for ML with polymorphic recursion is undecidable (1992)
- Type inference for ML is decidable and efficient in practice, though pathological programs of size $O(n)$ and run-time $O(n)$ can have types of size $O(2^{2^n})$

Why? (continued)

- The type inference algorithm is *unsound* in the presence of ML-style mutation, but value-restriction restores soundness
  - Based on *unification*

Recovering lost ground?

Extensions to the ML type system to be closer to System F:

- Usually require some type annotations
- Are judged by:
  - Soundness: Do programs still not get stuck?
  - Conservatism: Do all (or most) old ML programs still type-check?
  - Power: Does it accept many more useful programs?
  - Convenience: Are many new types still inferred?