CSE 341: Programming Languages

Autumn 2005
Lecture 11 — Type Inference, Parametric Polymorphism, Type Constructors

Today

- We have learned an interesting subset of the ML expression language
- But we have been really informal about some aspects of the type system:
  - Type inference (what types do bindings implicitly have)
  - Type variables (what do \( \tau_a \) and \( \tau_b \) really mean)
  - Type constructors (why is \( \text{int}\text{ list}\text{ a type but not list} \))
- Note: Type inference and parametric polymorphism are separate concepts that end up intertwined in ML. A different language could have one or the other.

Types - Basic Concepts

Some languages are untyped or dynamically typed.

ML is *statically typed*: every binding has one type, determined during type-checking (compile-time).

ML is *implicitly typed*: programmers rarely need to write the types of bindings.

ML is *type safe*: a value of one type cannot be misused as being a value of another type.

Java, Scheme, and Smalltalk are also type safe

Examples of languages that aren’t type safe: C, FORTRAN

What about MiniML?

Type Inference

The type-inference question: Given a program without explicit types, produce types for all bindings such that the program type-checks, or reject (only) if it is impossible.

Whether type inference is easy, hard, or impossible depends on details of the type system: Making it more or less powerful (i.e., more programs typecheck) may make inference easier or harder.
ML Type Inference

- Determine types of bindings in order (earlier first) (except for mutual recursion)
- For each `val` or `fun` binding, analyze the binding to determine necessary facts about its type.
- Afterward, use type variables (e.g., `'a`) for any unconstrained types in function arguments or results.
- Some extra details for type variables and references we’ll mention later.

Amazing fact: For the ML type system, “going in order” this way never causes unnecessary rejection.

Example 1

```haskell
fun f x = 
  let val (y, z) = x in 
   (Real.abs y) + z 
  end
```

Example 2

```haskell
fun sum lst = 
  case lst of 
    [] => 0 
  | hd::tl => hd + (sum tl)
```

Example 3

```haskell
fun compose (f, g, x) = f (g x)
```
Comments on ML type inference

- If we had subtyping, the "equality constraints" we generated would be unnecessarily restrictive.
- If we did not have type variables, we would not be able to give a type to compose until we saw how it was used.
  - But type variables are useful regardless of inference.
- Inference is why the following aren't really equivalent:
  - let val x = e1 in e2 end
  - (∀x. e2) e1

E.g., let's try e2 = (x 0, x "foo") and something simple for e1 like fn y => y:

  - let val x = (fn y => y) in (x 0, x "foo") end
  - (∀x. (fn y => y)) (fn y => y)

The latter gives a type error ...

ML-style polymorphism

The ML type system is actually more restrictive:

- "forall" must appear "all the way on the outside-left"
- So it's implicit; no way to write the words "for all"

Example: (∀a. ∀b. (a * b) -> (b * a)) means
forall a. forall b. (a * b) -> (b * a)

Non-example: There's no way to have a type like
(forall a. a -> int) -> int

Parametric polymorphism

Fancy words for "forall-types". Coming to next version of Java, C#, VB, etc. Sometimes called generics. A bit like C++ templates if C++ didn't have operator-overloading.

In principle, just two new kinds of types:

\[
\begin{align*}
  tv & ::= \ 'a \mid \ 'b \mid \ldots \\
  t & ::= \text{int} \mid \text{string} \mid \text{bool} \mid \text{ti} \rightarrow \text{t2} \mid \{\text{li} : \text{t1}, \ldots, \text{ln} : \text{tn}\} \\
  & \quad \mid \text{dtname} \mid tv \mid \text{forall } tv. t
\end{align*}
\]

Given an expression of type forall tv. t, we can instantiate it at type t2 to get an expression of type "t with 'tv replaced by t2"

Example: We can instantiate

forall 'a. forall 'b. ('a * 'b) -> ('b * 'a)

with string for 'a and int->int for 'b to get

(string * (int->int)) -> ((int->int) * string)

Versus Subtyping

Compare

fun swap (x,y) = (y,x) (* ('a * 'b) -> ('b * 'a) *)

with

class Pair { Object x; Object y; ... }

Pair swap(Pair pr) { return new Pair(pr.y, pr.x); }

ML wins in two ways (for this example):

- Caller instantiates types, so doesn't need to cast result
- Callee cannot return a pair of any two objects.
Containers

Parametric polymorphism (forall types) are also the right thing for containers (lists, sets, hashables, etc.) where elements have the same type.

Example: ML lists

val :: : ('a * ('a list)) -> 'a list (* infix is syntax *)
val map : (('a -> 'b) * ('a list)) -> 'b list
val sum : int list -> int
val fold : ('a * 'b -> 'b) -> 'b -> ('a list) -> 'b

List is a type constructor, not a type; if t is a type, then t list is a type.

User-defined type constructors

Language-design: don’t provide a fixed set of a useful thing.

Let programmers declare type constructors.

Examples:

datatype 'a nonMt_list = One of 'a
        | More of 'a * ('a nonMt_list)
datatype 'a rope = Empty
        | Cons of 'a * ('a rope)
        | Rope of ('a rope) * ('a rope)

You can have multiple type-parameters (not shown here).

And now, finally, everything about lists is syntactic sugar!

One last thing – not on the test

Polymorphism and mutation can be a dangerous combination.

val x = ref [] (* 'a list ref *)
val _ = x := ["hi"] (* instantiate 'a with string *)
val _ = (hd(!x)) + 7 (* instantiate 'a with int -- bad!! *)

Roughly, ML ensures the t in t ref has no new type variables.

But they do it with a non-obvious way: function applications (such as ref []) cannot get polymorphic types; user specifies (e.g.,
int list ref)