CSE 341:
Programming Languages

Winter 2006
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 ’a and ’b really mean)
  – Type constructors (why is int list 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.
Type Inference

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*; ignoring a few things like “dot-dot-dot patterns” programmers never need to write the types of bindings.

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

fun f x =
    let val (y,z) = x in
      (Math.abs y) + z
    end
Example 2

fun sum lst =
  case lst of
    [] => 0
    | hd::tl => hd + (sum tl)
Example 3

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 let val x = e1 in e2 end is not really sugar for (fn x => e2) e1 because the latter gives a type error if e2 contains x 0 and x "foo", even if e1 if fn y => y.
  – Don’t worry if that doesn’t make sense.
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:

\[ tv ::= 'a \mid 'b \mid \ldots \]
\[ t ::= \text{int} \mid \text{string} \mid \text{bool} \mid t_1 \rightarrow t_2 \mid \{l_1 : t_1, \ldots, l_n : t_n\} \]
\[ \mid \text{dtname} \mid tv \mid \text{forall } 'tv. \ t \]

Given an expression of type \( \text{forall } 'tv. \ t \), we can instantiate it at type \( t_2 \) to get an expression of type “\( t \) with \( 'tv \) replaced by \( t_2 \)”

Example: We can instantiate

\( \text{forall } 'a. \text{forall } 'b. \ ( 'a \times 'b) \rightarrow ( 'b \times 'a) \)

with string for \( 'a \) and \( \text{int} \rightarrow \text{int} \) for \( 'b \) to get

\( \text{string} \times (\text{int} \rightarrow \text{int}) \rightarrow ((\text{int} \rightarrow \text{int}) \times \text{string}) \)
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) -> (`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

Easy to express this restriction syntactically:

\[
tv ::= `a | `b | ...   \\
s ::= int | string | bool | t1->t2 | \{l1:t1, ..., ln:tn\}   \\
     \quad | dtname | tv   \\
t ::= s | forall tv. t
\]
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, hashtables, 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) -> ('a list) -> 'b

list is 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 non_mt_list = One of 'a
                 | More of 'a * ('a non_mt_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.

```ml
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`)