CSE 341:
Programming Languages

Spring 2007
Lecture 12 — Type Inference, Parametric Polymorphism, Type Constructors

CSE 341 Spring 2007, Lecture 12 1
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 \textit{statically typed}; every binding has one type, determined during type-checking (compile-time).

ML is \textit{implicitly typed}; programmers rarely 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
    (Real.abs y) + z
    end
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 the following aren’t really equivalent:
  – let val x = e1 in e2 end
  – (fn 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
  – (fn x => (x 0, x "foo")) (fn y => y)

The latter gives a type error ...
Parametric polymorphism

Fancy words for “forall-types”. Late add-on to Java, C#, VB, etc. Sometimes called generics. A bit like C++ templates

In principle, just two new kinds of types:

\[ tv ::= 'a | 'b | ... \]
\[ t ::= \text{int} | \text{string} | \text{bool} | t1\rightarrow t2 | \{l1: t1, \ldots, ln: tn\} \]
\[
\quad | \text{dtname} | tv | \text{forall 'tv. t}
\]

Given an expression of type \texttt{forall 'tv. t}, we can \textit{instantiate} it at type \texttt{t2} to get an expression of type “t with ‘tv replaced by t2”

Example: We can instantiate
\[ \text{forall 'a. forall 'b. ('a * 'b) -> ('b * 'a)} \]
with string for ‘a and int\rightarrow int for ‘b to get
\[ \text{(string * (int\rightarrow int)) -> ((int\rightarrow int) * string)} \]
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
int -> (forall ’a. ’a -> int)
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

::  : (’a * (’a list)) -> ’a list (* infix is syntax *)
map : ((’a -> ’b) * (’a list)) -> ’b list
sum : int list -> int
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.

Again, with original Java containers, you just had list of Object & a lot of casts...
User-defined type constructors

Language-design: don't provide a fixed set of a useful thing.
Let programmers declare type constructors.

Examples:

```haskell
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!