CSE 341: Programming Languages

Dan Grossman
Winter 2008
Lecture 10—Higher-Order Functions Wrapup; Type inference; Parametric Polymorphism
One Last Closure Example

Closures are essential to elegant functional programming.

See our 15 ways of counting zeros in a list to see how currying and higher-order functions give us lots of flexibility.

- And some interesting reuse vs. straightforwardness vs. efficiency trade-offs
Now inference and type variables

• We have learned an interesting subset of ML expressions

• 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*; programmers rarely need to write bindings’ types (e.g., if using features like #1)

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.
- (One extra restriction to be discussed at the end.)

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

[Let’s walk through a few examples, doing type inference by hand.]
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.
Parametric Polymorphism

Fancy phrase for “forall types” or sometimes “generics.” In ML since mid-80s and now in Java, C#, VB, etc.

- C++ templates used similarly, but more like macros (later).

In ML, it’s like there’s an implicit “for all” at the beginning of any type with ’a, ’b, etc. Example:

\[(’a \times ’b) \to (’b \times ’a)\]

really means:

\[\text{forall } ’a. \text{forall } ’b. (’a \times ’b) \to (’b \times ’a)\]

(though forall is just for lecture purposes; it is not in ML)

We can instantiate the type variables to get a less general type. For example, with string for ’a and int→int for ’b we get:

\[(\text{string } \times (\text{int } \to \text{int})) \to ((\text{int } \to \text{int}) \times \text{string})\]
All the types

In principle, we could have a very flexible way of building types:

- **Base types** like int, string, real, ...
- **Compound types** like $t_1 * t_2$, $t_1 \rightarrow t_2$, and datatypes where $t_1$ and $t_2$ are any type
- **Polymorphic types** like forall ’a. t where ’a can appear in t.

This would let you have types like

(forall ’a. ’a -> (’a * ’a)) -> ((int * int) * (bool * bool))

Every language has limits; in ML there is no type like this, the forall is always implicit and always “all the way to the outside left”, for example this different type:

(’a -> (’a * ’a)) -> ((int * int) * (bool * bool))

(caller must pick one instantiation)
Example

This code is fine, but ML disallows it to make type inference easier.

(* function f does _not_ type-check *)
fun f pairmaker = (pairmaker 7, pairmaker true)
val x = f (fn y => (y,y))
**Versus Subtyping**

Compare

\[
\text{fun swap} \ (x,y) = (y,x) \quad (* \quad ('a \times 'b) \to ('b \times 'a) *)
\]

with

\[
\text{class Pair} \ {\text{Object} x; \text{Object} y; \ldots \ }
\]
\[
\text{Pair swap(Pair} \ pr) \ {\text{return new Pair(pr.y, pr.x); \}}
\]

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.

That’s why Java added generics...
Java Generics

class Pair<T1,T2> {
    T1 x;
    T2 y;
    Pair(T1 _x, T2 _y) { x=_x; y=_y; }
    static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> pr) {
        return new Pair<T2,T1>(pr.y,pr.x);
    }
}

This really is a step forward despite the clutter (explicit types and type definitions) versus

fun swap (x,y) = (y,x)
Containers

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

Example: ML lists

```ml
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!! *)
```

To prevent this, ML has “the value restriction”: bindings can only get polymorphic types if they are initialized with values.

Alas, that means this does not work even though it should be fine:

```ml
val pr_list = List.map (fn x => (x,x))
```

But these all work:

```ml
val pr_list = fun lst => List.map (fn x => (x,x)) lst
fun pr_list lst = List.map (fn x => (x,x)) lst
val pr_list : int list -> (int*int) list = List.map (fn x => (x,x))
```