CSE 341: Programming Languages

Hal Perkins
Spring 2011
Lecture 10—Higher-Order Functions Wrapup; Type inference; Namespace Management
One Last Closure Example

Closures are essential to elegant functional programming.

See our 35\textsuperscript{a} 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

\textsuperscript{a}Well, only about 10 or so...
Now inference

• 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)

• Type inference and parametric polymorphism are separate concepts that end up intertwined in ML
  – A different language could have one or the other
  – Focus on inference today; type variables on Friday
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 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; reject if and 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 in lecture 12.)

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.
  – (Other languages could just make programmer write them.)
Structure basics

Large programs benefit from more structure than a list of bindings.

Syntax: structure Name = struct bindings end

If \( x \) is a variable, exception, type, constructor, etc. defined in \( \text{Name} \), the rest of the program refers to it via \( \text{Name}.x \)

(You can also do open \( \text{Name} \), which is often bad style, but convenient when testing.)

So far, this is just namespace management, which is important for large programs, but not very interesting.
Signature basics

(For those interested in learning more, we’re doing only opaque signatures on structure definitions.)

A signature signature BLAH = sig ... end is like a type for a structure.

• Describes what types a structure provides.

• Describes what values a structure provides (and their types).

That way, a module client can be type-checked without consulting the module implementation (just the signature).

Now the interesting part: A signature can promise less, i.e., hide things about the implementation.

• This is the essence of abstraction, one of the (possibly the) most important concepts in computer science.