CSE 341, Spring 2008, Lecture 10 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This lecture covers 3 separate topics:

1. We complete our discussion of *higher-order functions* with one final example.
2. We discuss ML-style *type inference* to understand what type inference for a statically typed language is and how ML's algorithm to infer types is actually fairly straightforward.
3. We begin discussing *modules*, focusing first on *namespace management* and *separate type-checking*.

Higher-Order Functions Wrapup Example

Higher-order functions provide lots of conciseness, flexibility, and code reuse when used well. There are also many different ways to do the same thing, and which is "best" is somewhat a matter of taste (which approach is most readable, easiest to get correct, etc.) and efficiency. Consider the simple example of counting how many 0s are in an `int list`. Before learning higher-order functions, we might have written:

```
fun count_zeroes_1 lst =
    case lst of
        [] => 0
      | first::rest => (if first=0 then 1 else 0) + count_zeroes_1 rest
```

Using a couple higher-order functions defined in the `List` library we can instead think of counting 0s as having two parts: get rid of non-zeros and see how many elements remain:

```
fun count_zeroes_2 lst =
    List.length (List.filter (fn x => x=0) lst)
```

We could then recognize that this approach is essentially function composition with partial application of the curried filter function:

```
val count_zeroes_3 = List.length o (List.filter (fn x => x=0))
```

And if we *really* wanted to use lots of currying, we could recognize that if `=` were itself a curried function, we could partially apply it to 0 to get a function that compares numbers to 0. Now, `=` is not a curried function, but we can make it one like this:

- Use the ML keyword `op` for turning an infix function into a regular function. For example, `op+` is a function of type `int*int->int` and `op=` is a function of type `''a*''a->bool`.
- Use a helper function to convert a function that takes a pair to a function that takes two curried arguments.

```
fun curry f x y = f (x, y)
val count_zeroes_3' = List.length o (List.filter ((curry (op=)) 0))
```

These "filter then count" approaches seem fine especially when you are trying to be a productive programmer and counting 0s is hardly where you should spend a lot of human time. But they are inefficient since they create a whole list (the list holding the 0s) just to see how long the list is. Our initial approach did not have this inefficiency, nor does a solution using `foldl`:

```
fun count_zeroes_4 lst =
    List.foldl (fn (x,y) => (if x=0 then 1 else 0) + y) 0 lst
```

We can recognize that this can be simplified using partial application of the curried `foldl` function (much like `if e then true else false` can be simplified to `e`):

```
val count_zeroes_5 =
    List.foldl (fn (hd,acc) => (if hd=0 then 1 else 0) + acc) 0
```

While these uses of `foldl` work great, if we plan to write lots of functions that count list elements, we might instead write our own higher-order function that makes such counting easier:

```
fun count_list1 f lst =
    case lst of
        [] => 0
      | hd::tl => (if f hd then 1 else 0) + count_list1 f tl
```

Here we have abstracted out whether an element should be counted, letting callers pass in a function `f` that takes a list element and returns a `bool`. Here are two ways to pass in functions that see if an element is 0:

```
val count_zeroes_6 = count_list1 (fn x => x=0)
val count_zeroes_6' = count_list1 (curry (op=) 0)
```

While `count_list1` is a bit more convenient for callers than `foldl` — assuming that callers want to do counting — we could *implment* the count-list function in terms of fold:

```
fun count_list2 f =
    List.foldl (fn (hd,acc) => (if f hd then 1 else 0) + acc) 0
```

This implementation decision is irrelevant to callers; they use `count_list2` exactly like `count_list1`:

```
val count_zeroes_7 = count_list2 (fn x => x=0)
val count_zeroes_7' = count_list2 (curry (op=) 0)
```

Type Inference

While we have been using ML type inference for a couple weeks, we have not studied it carefully. Let's first carefully define what type inference *is* and then see via several examples how ML type inference works.

Java and ML are *statically typed* languages, meaning every binding has a type that is determined "at compile-time" i.e., before any part of the program is run. The type-checker is a compile-time procedure that either accepts or rejects a program. By contrast, Scheme and Ruby are dynamically-typed languages; the type of a binding is not determined ahead of time and computations like binding 42 to `x` and then treating `x` as a string result in run-time errors. We will spend a later lecture comparing the advantages and disadvantages of static versus dynamic typing.

Unlike Java, ML is *implicitly typed*, meaning programmers rarely need to write down the types of bindings. This is often convenient (though some disagree as to whether it makes code easier or harder to read), but in no way changes the fact that ML is statically typed. Rather, the type-checker has to be more sophisticated because it must *infer* (i.e., figure out) what the *type annotations* "would have been" had the programmers written all of them. In principle, type inference and type checking could be separate steps (the inferencer could do its thing and the checker could see if the result should type-check), but in practice they are often merged into "the type-checker". Note that a correct type-inferencer must find a solution to what all the types should be whenever such a solution exists, else it must reject the program.

Whether type inference for a particular programming language is easy, hard, or impossible (in the halting-problem sense of CSE322) is often hard to determine. It is *not* proportional to how permissive the type system is. For example, the "extreme" type systems that "accept everything" and "accept nothing" are both very easy to do inference for.

ML was rather cleverly designed so that type inference is a straightforward algorithm. We will demonstrate that algorithm with a few examples; writing down the whole thing in code is not difficult but we will choose not to do so. ML type inference ends up intertwined with parametric polymorphism — when the inferencer determines a function's argument or result "could be anything" the resulting type uses `'a`, `'b`, etc. — but inference and polymorphism are separate concepts: a language could have one or the other. For example, Java has generics but no inference. We will study parametric polymorphism more carefully in a couple lectures.

Here is an overview of how ML type inference works (examples to follow):

- It determines the types of bindings in order, using the types of earlier bindings to infer the types of later ones. This is why you cannot use later bindings in a file. (When you need to, you use mutual recursion and type inference determines the types of all the mutually recursive bindings together.)
- For each `val` or `fun` binding, it analyzes the binding to determine necessary facts about its type. For example, if we see the expression `x+1`, we conclude that `x` must have type `int`. We gather similar facts for function calls, pattern-matches, etc.
- Afterward, use *type variables* (e.g., `'a`) for any unconstrained types in function arguments or results.
- (There is one extra restriction to be discussed in lecture 12.)

The amazing fact about the ML type system is that "going in order" this way never causes us to unnecessarily reject a program that could type-check nor do we ever accept a program we should not. So explicit type annotations really are optional (unless you use features like `#1`).

As a first example, consider inferring the type for this function:

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

Here is how we can infer the type:

- Looking at the first line, `f` must have type `T1->T_2` for some types `T1` and `T2` and in the function body `f` has this type and `x` has type `T1`.
- Looking at the `val`-binding, `x` must be a pair type (else the pattern-match makes no sense), so in fact `T1=T3*T4` for some `T3` and `T4`, and `y` has type `T3` and `z` has type `T4`.
- Looking at the addition expression, we know from the context that `abs` has type `int->int`, so `y` has type `T3` means `T3=int`. Similarly, since `abs y` has type `int`, the other argument to `+` must have type `int`, so `z` having type `T4` means `T4=int`.
- Since the type of the addition expression is `int`, the type of the let-expression is `int`. And since the type of the let-expression is `int`, the return type of `f` is `int`, i.e., `T2=int`.

Putting all these constraints together, `T1=int*int` (since `T1=T3*T4`) and `T2=int`, so `f` has type `int*int->int`.

Note that humans doing type inference "in their head" often take shortcuts just like humans doing long division in their head, but the point is there is an algorithm that methodically goes through the code gathering constraints and putting them together to get the answer.

Next example:

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

- From the first line, there exists types `T1` and `T2` such that `sum` has type `T1->T2` and `lst` has type `T1`.
- Looking at the case-expression, `lst` must have a type that is compatible with all of the patterns. Looking at the patterns, both of them match any list, since they are built from list constructors (in the `hd::tl` case the subpatterns match anything of any type). So since `lst` has type `T1`, in fact `T1=T3 list` from some type `T3`.
- Looking at the right-hand sides of the case branches, we know they must have the same type as each other and this type is `T2`. Since 0 has type `int`, `T2=int`.

- Looking at the second case branch, we type-check it in a context where `hd` and `tl` are available. Since we are matching the pattern `hd::tl` against a `T3 list`, it must be that `hd` has type `T3` and `tl` has type `T3 list`. Now looking at the right-hand side, we add `hd`, so in fact `T3=int`. Moreover, the recursive call type-checks because `tl` has type `T3 list` and `T3 list=T1` and `sum` has type `T1->T2`. Finally, since `T2=int`, adding `sum tl` type-checks. Notice that before we got to `sum tl` we had already inferred everything, but we still have to check that types are used consistently and reject otherwise (e.g., if we had written `sum hd`, that cannot type-check).

Putting everything together, we get `sum` has type `int list -> int`.

Our remaining examples will infer polymorphic types. All we do is follow the same procedure we did above, but when we are done we will have some parts of the function's type that are still *unconstrained*. For each `Ti` that "can be anything" we use a type variable (`'a`, `'b`, etc.).

```
fun length lst =
   case lst of
     [] => 0
   | hd::tl => 1 + (length tl)
```

Type inference proceeds much like with `sum`: We end up determining

- `length` has type `T1->T2`
- `lst` has type `T1`
- `T1=T3 list` (due to the pattern-match)
- `T2=int` because 0 can be the result of a call to `length`.
- `hd` has type `T3` and `tl` has type `T3 list`
- The recursive call `length tl` type-checks because `tl` has type `T3 list`, which is `T1`, the argument type of `length`. And we can add the result because `T2=int`.

So we have all the same constraints as for `sum`, *except* we do not have `T3=int`. In fact, `T3` can be anything and `length` will type-check. So type inference recognizes that when it is all done, it has `length` with type `T3 list -> int` and `T3` can be anything. So we end up with the type `'a -> int`, as expected. Again the rule is simple: for each `Ti` in the final result that can't be constrained, we use a type variable.

Final example:

```
fun compose (f,g) = fn x => f (g x)
```

- Since the argument to `compose` must be a pair (from the pattern used for its argument), `compose` has type `T1*T2->T3`, `f` has type `T1` and `g` has type `T2`.
- Since `compose` returns a function, `T3` is some `T4->T5` where in that function's body, `x` has type `T4`.
- So `g` must have type `T4->T6` for some `T6`, i.e., `T2=T4->T6`.
- And `f` must have type `T6->T7` for some `T7`, i.e., `T1=T6->T7`.
- But the result of `f` is the result of the function returned by `compose`, so `T7=T5` and so `T1=T6->T5`.

Putting together `T1=T6->T5` and `T2=T4->T6` and `T3=T4->T5` we have a type for `compose` of `(T6->T5)*(T4->T6) -> (T4->T5)`. There is nothing else to constrain the types `T4`, `T5`, and `T6`, so we replace them consistently to end up with `('a->'b)*('c->'a) -> ('c->'b)` as expected (and the last set of parentheses are optional, but that is just syntax).

Now that we have seen how ML type inference works, we can make two interesting observations:

- Inference would be more difficult if ML had subtyping (e.g., if every triple could also be a pair) because we would not be able to conclude things like, "`T3=T1*T2`" since the *equals* would be overly restrictive.

- Inference would be more difficult if ML did *not* have parametric polymorphism since we would have to pick some type for functions like `length` and `compose` and that could depend on how they are used.

Modules for Namespace Management

To learn the basics of ML, pattern-matching, and functional programming, we have written fairly small programs that are just a sequence of bindings. For larger programs, it definitely helps to have more structure. In ML, we can use *structures* to define *modules* that together are a collection of bindings. At its simplest, you can just write `structure Name = struct` ***`bindings`*** `end` where `Name` is the name of your structure (you can pick anything; capitalization is a convention) and ***`bindings`*** is any list of bindings, containing values, functions, exceptions, datatypes, and types. Inside of the structure you can use earlier bindings just like we have been doing "at top-level" (i.e., outside of any module). Outside of the structure, you can refer to a binding b in `Name` by writing `Name.`b. This is exactly what we have been doing to use functions like `List.foldl`; now you know how to define your own structures.

Used like this, structures are providing just *namespace management*, a way to avoid different bindings in different parts of the program from shadowing each other. That is very useful, but not particularly interesting. Much more interesting is giving structures *signatures*, which are types for modules and let us provide strict *interfaces* that code outside the module must obey. The next lecture will describe several ways to do this; here we just show one way to write down an explicit signature for a module. Here is an example signature definition and structure definition that says the structure must have the signature (i.e., type) `MATHLIB`:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x =
    if x=0
    then 1
    else x * fact (x - 1)

val half_pi = Math.pi / 2.0

fun doubler y = y + y
end
```

Because of the `:> MATHLIB`, the structure `MyMathLib` will typecheck only if it actually provides everything the signature `MATHLIB` claims it does and with the right types. Signatures can also contain datatype, exception, and type bindings. Because we check the signature when we compile `MyMathLib`, we can use this information when we check any code that uses `MyMathLib`. In other words, we can just check clients *assuming* that the signature is correct.

What we will do in the next lecture is consider signatures for structures that *hide* things about the implementation. This abstraction is an essential tool in software engineering because it lets us change the implementation without clients being able to tell thanks to the signature.