CSE341, Fall 2011, Lecture 10 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture has a few loosely related topics:

- We extend the use of type variables from functions to datatypes.
- We butt heads with the Value Restriction, a necessary wart in SML due to odd interactions between polymorphic datatypes and mutation. (Mutation is introduced in the materials for the previous lecture.)
- 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.

Polymorphic Datatypes

Parametric polymorphism becomes particularly useful for defined functions over containers (lists, arrays, sets, hashtables, etc.) that have elements of the same type. As we have studied, not all functions over lists are polymorphic, but many are, including the constructors:

```
val [] : 'a list
val :: : 'a * ('a list) -> 'a list (* infix is syntax *)
val map : ('a -> 'b) * ('a list) -> 'b list
val fold : ('a * 'b -> 'b) -> 'a list -> 'b
```

What exactly, then is `list`? It is *not* a type; you cannot say a function has type `list->int`, for example. It is a *type-constructor*, something that makes a type out of another type. So `int list` is a type, `'a list` is a type, `(int->int) list` is a type, etc., and so there exist types like `(int->int) list -> int` and so on. Remember for the examples above there is an implicit "for all" on the outside left. For example, the type of `[]` is "an alpha list for all types alpha."

We can define our own type-constructors in ML. It would be a poor language design to have the only type-constructors be built-in features like lists. In general, if a feature is useful for built-in features, it is useful for user-defined things too. To define a type-constructor in ML, we just use a `datatype` binding. We explicitly give one or more type-constructor arguments that we can then use in the types of the constructors. Here are two examples:

```
datatype 'a non_mt_list = One of 'a
                        | More of 'a * ('a non_mt_list)
datatype ('a,'b) mytree =
   Leaf of 'a
 | Node of 'b * ('a,'b) mytree * ('a,'b) mytree
```

The first one is a type for lists that have at least one argument. The second is for trees where leaves have one type of data and internal nodes have a second type of data (in a particular tree the two types might or might not be the same). Notice `mytree` is not a type, something like `(int,string) mytree` is a type. Fortunately, type inference can still figure out all the types for us, since `Leaf` and `Node` are just constructors with polymorphic types — `Leaf` has type `'a -> ('a,'b) mytree` and `Node` has type `'b * ('a,'b) mytree * ('a,'b) mytree -> ('a,'b) mytree`. Here are 4 expressions and the inferred types:

```
Node("hi",Leaf 17,Leaf 4)     (* (string,int) mytree *)
Node(14,Leaf "hi",Leaf "mom") (* (int,string) mytree *)
Node(14, Leaf 4, Leaf 5)      (* (int,int) mytree *)
(* Node("hi",Leaf 17,Leaf true) *) (* doesn't type-check *)
```

Now that we can define our own type-constructors, there really was no need for ML to build in support for lists at all. Yes, the syntax of writing `::` infix (between its two arguments or subpatterns) and the syntactic sugar of `[e1,e2,...,en]` are nice, but other than that we could just have defined

```
datatype 'a list = Empty | Cons of 'a * 'a list
```

and had everything we needed.

The Value Restriction

Unfortunately, this is not quite the end of the story on ML parametric polymorphism. Without one additional restriction, ML's mutable references — which are sometimes useful, but we have used only in our callback example when studying higher-order functions — can make the type system *unsound*. An unsound type system does not actually prevent what it claims to prevent, such as treating an `int` as a function or enforcing module signatures. This is an example of a program that demonstrates the problem:

```
val x = ref []       (* 'a list ref *)
val _ = x := ["hi"]  (* instantiate 'a with string *)
val _ = (hd(!x)) + 7 (* instantiate 'a with int -- bad!! *)
```

Straightforward use of the rules for type inference as discussed below would accept this program even though we should not. To prevent this, ML will reject the first line because of something called "the value restriction". The value restriction requires any expression that is given a polymorphic type to be a variable or a value (including function definitions, constructors, etc.). Because `ref []` is not a value, we can give it type `(int list) ref` or `(string list) ref` but not `('a list) ref`. But type inference cannot figure out what non-polymorphic type to give `x` in our example, so either the programmer must supply an explicit type or the binding is rejected by the type-checker. While it's not at all obvious that this simple restriction makes the whole type system sound, it turns out to be enough.

The value restriction does sometimes get in your way even when you are not using mutation (since the type-checker does not *know* you are not using mutation). For example, this completely harmless code is rejected:

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

As cool as partial application is, if the result would have a polymorphic type, we cannot bind it to a variable due to the value restriction. We can either give an explicit non-polymorphic type or we can use an extra function wrapper so that the expression we are using is already a value. (Recall functions *are* values.) So any of these three approaches work fine:

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

You do not need to know anything about the value restriction really except how to work around it when it comes up.

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, Racket 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 CSE311) 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.
- (Enforce the value restriction — only variables and values can have polymorphic types, as discussed above.)

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->T2` 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).

For a final example, let's consider a broken version of `sum` to see how inference will behave:

```
fun broken_sum lst =
   case lst of
     [] => 0
   | x::xs => x + (broken_sum x)
```

- From the first line, there exists types `T1` and `T2` such that `broken_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 `x::xs` 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 `x` and `xs` are available. Since we are matching the pattern `x::xs` against a `T3 list`, it must be that `x` has type `T3` and `xs` has type `T3 list`. Now looking at the right-hand side, we add `x`, so in fact `T3=int`.
- Now, we diverge from the correct `sum` implementation. The recursive call applies `broken_sum` to `x`, so `x` must have the same type as `broken_sum`'s parameter, or in other words, `T1=T3`. However, we know that `T1=T3 list`, so this new constraint `T1=T3` actually generates a contradiction: `T3=T3 list`, meaning that somewhere we're using `x` (which has type `T3`) in two contexts that expect different types. If we want to be more concrete, we can use our knowledge that `T3=int` to rewrite this as `int=int list`. Looking at the definition of `broken_sum` it should be obvious that this is exactly the problem: we tried to use `x` as an `int` and as an `int list`.

When your ML program does not type-check, the type-checker reports the expression where it discovered a contradiction and what types were involved in that contradiction. While sometimes this information is helpful, other times the actual problem is with a different expression, but the type-checker did not reach a contradiction until later.

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.