15-150 Lab 5 Notes

Adapted by Amy Zhang from notes by S. Brookes, I. Voysey, and D. Licata

25 Sept 2013

Programs must be written for people to read, and only incidentally for machines to execute. — Abelson and Sussman, Structure and Interpretation of Computer Programs

Programming is about communicating with people: you, when you come back to a piece of code next year; other people on your project. Today we take the kid gloves off, and talk about a few ways to make your complex ML programs more readable:

- Use the type system to communicate. We talked about *options* as a way of doing this. Later in the semester we will use and *domain-specific datatypes*, or datatypes that we create to solve a specific problem, as a way of doing this.
- Abstract repeated patterns of code. We saw our first examples of *higher-order functions*, which are an extremely powerful way of doing this. Clean and elegant code communicates ideas faster and better.

1 Option Types

You have probably asked one of your friends "do you know what time it is?" before, and had them answer with a terse "yes." Culturally, if someone asks you "do you know what time it is?" they usually really mean "do you know what time it is, and, if you happen to know what time it is, tell me what time it is". The literalist interpretation is rude, but not technically wrong.

It's better to ask "do you know what time is it, and if so, please tell me?"—that's what "what time is it?" usually means. This way, you get the information you were after, when it's available.

Here's the analogy in code:

```
  doyouknowwhattimeitis? : person -> bool
  tellmethetime : person -> int
  whattimeisit : person -> int option
```

Options are a simple datatype defined for us in SML by something that looks like this:

```
datatype 'a option =
     NONE
   | SOME of 'a
```

Here again is a type parameterized over the type variable `'a`, so this means there is a type `T option` for every type `T`. The same constructors can be used to construct values of different types. For example

```
val x : int option = SOME 4
val y : string option = SOME "a"
```

This is because NONE and SOME are polymorphic: they work for any type 'a.

```
NONE : 'a option
SOME : 'a -> 'a option
```

1.1 Boolean Blindness

1.1.1 Asking For The Time

Don't fall prey to *boolean blindness*: boolean tests let you *look*, options let you *see*. If you write

```
case (doyouknowwhattimeitis? p) of
  true => tellmethetime p
  false => ...
```

The problem is that there's nothing about the *code* that prevents you from also saying tellmethetime in the false branch. The specification of tellmethetime might not allow it to be called—but that's in the math, which is not automatically checked.

On the other hand, if you say

```
case (whattimeisit p) of
  SOME t => ...
  NONE => ...
```

you naturally have the time in the SOME branch, and not in the NONE branch, without making any impolite demands. The structure of the code helps you keep track of what information is available—this is easier than reasoning about the meaning of boolean tests yourself in the specs. And the type system helps you get the code right.

Languages like C and Java are for commitmentophobes, in that you can only say "maybe...". The reason for this is the *null pointer*: when you say String in Java, or int* in C, what that means is "maybe there is a string at the other end of this pointer, or maybe not". You end up writing findOdd but give it the type int list -> int, and so you'd never be sure that the odd is actually there. You always have to remember to check the result without any mechanical reminder to do so, which is exactly the sort of thing that humans are excellent at forgetting.

Sir Charles Antony Richard Hoare[1]—the inventor of QuickSort, ALGOL, Hoare Logic, and countless other brilliant ideas—calls this his "billion dollar mistake". Not rigidly enforcing the distinction between things and potential things,

> "has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years".

The really beautiful thing about ML and other languages that have options it that you can use the type system to track these obligations—and at compile time, no less. If you say something has type T, you know that you definitely absolutely have a T. If you say T option, you know that maybe you have one, or maybe not—and the type system forces you to handle either possibility.

Thus, the distinction between T and T option lets you make appropriate promises, *using types to communicate.*

[1] "Tony," to his friends.

1.2 How not to program with lists

This whole discussion about options may seem obvious, but functional languages like Lisp and Scheme are based on the `doyouknow/tellme` style. For example, the interface to lists is

```
  nil? -- check if a list is empty
  first (car) -- get the first element of a list, or error if it's empty
  rest (cdr) -- get the tail of a list, or error if it's empty
```

and you write code that looks like

```
(if (nil? l) <case for empty> <case for cons, using (first l) and (rest l)>)
```

Of course you can also accidentally call `first` and `rest` in the `<case for empty>`, but they will fail. On the other hand, if you write something like the ML-style

```
case l of
    [] => ...
  | x :: xs => ...
```

then the code naturally lets you see what you looked at—by following the structure of the type, you completely avoid this class of bugs and have code that makes sense at a glance every time.

2 Functions as Arguments

Next, we're going to talk about the thing that makes functional programming *functional*:

> Functions are values that can be passed to, and returned from, other functions.

This extremely powerful idea allows us to write higher-order functions that neatly abstract away repeated patterns.

2.1 Example: `map`

Recall the following two functions, `doubleList` and `quadrupleList`:

```
val double = (fn x => 2 * x)
val quadruple = (fn x => double(double x))

fun doubleList ([ ] : int list) : int list = [ ]
  | doubleList (x::xs : int list) : int list =
     (double x)::(doubleList xs)

fun quadrupleList ([ ] : int list) : int list = [ ]
  | quadrupleList (x :: xs: int list) : int list =
     (quadruple x)::(quadrupleList xs)
```

They both perform the some fixed transformation uniformly to each element of a list, but they do different things—doubling and quadrupling every element in a list respectively. We can write one function that expresses the pattern that is common to both of them.

```
fun mapList (f : 'a -> 'b, [] : 'a list) : 'b list = []
  | mapList (f           , x::xs       )           = f x :: (map (f,xs))
```

The idea with `map` is that it takes a function `f : 'a -> 'b` as an argument, which represents what you are supposed to do to each element of the list.

`'a -> 'b` is a type and can be used just like any other type in ML. For example, a function like `map` can take an argument of type `int-> int`; and a function can return a function as a result.

For example, we can recover `doubleList` like this:

```
fun doubleList (l : int list) : int list = mapList (double, l)
```

2.2 Anonymous functions

Another way to instantiate `map` is with an anonymous function, which is written `fn x => 2 * x`:

```
fun doubleList (l:int list) : int list = mapList ((fn x => 2 * x), l)
```

`doubleList` can be defined anonymously too:

```
val doubleList : int list -> int list = fn l => mapList (fn x => 2 * x, l)
```

It's important to note that the value of `fn l => mapList (fn x => 2*x, l)` is `fn l => mapList (fn x => 2*x, l)`. Just like on homework 2, you don't evaluate the body until you apply the function, because how could you? You must first know what value the variable `l` stands for before you can do something with the function. Values of function types are tools for transforming data in the exact same sense that values of the natural number type are tools for counting things.

3 Currying

Moses Schönfinkel was a Russian logician and mathematician working on something called combinatory logic in the early twentieth century. In his work, he was formalizing a language that included the idea of functions, and found that it was easier to prove things about his language if he assumed every function took exactly one argument and there were no pairs. The observation that made this sound was that you can emulate a function that takes multiple arguments all at once with a function that takes the first argument and produces a function expecting the next argument.

Haskell Curry was an American logician and mathematician working on something called combinatory logic in the early twentieth century. He made a similar observation and, unfortunately for Schönfinkel, his name is a lot easier to spell, write, and pronounce, so this observation is called *Currying* not *Schönfinkelization.*

3.1 Example: `map`

This observation can be carried out in SML just as well. For example, we can rewrite `mapList` as a curried function. In lab, we called this function `mapList'`, but it actually exists as a built in function `List.map`, or just `map`:

```
fun map (f : 'a -> 'b) : 'a list -> 'b list =
    fn l => case l of
              [] => []
            | x::xs => f x :: map f xs
```

This is a function that after taking the first argument produces a function expecting the rest. This gives `map : ('a -> 'b) -> ('a list -> 'b list)`. By convention, the type constructor `->` associates to the right, so we could have also just said

```
                     map :  ('a -> 'b) -> 'a list -> 'b list
```

How does this new version of `map` get used? Here's a small evaluation trace to demonstrate what happens (with some parentheses added for clarity):

```
    map (fn x => x + 1)
=> fn l => case l
             of [] => []
              | x::xs => ( (fn x => x + 1) x)::(map (fn x => x + 1) xs)
```

And that's it! Why do we stop? We've reached a value. Since there are no arguments available, there's no more work to do. What we get is a function that's waiting for a list is ready to add one to every element; we never wrote this function in our source code, it gets created from a *partial application* of `map`.

If instead we'd given `map` both of its arguments, we'd get something that looked like this:

```
    (map (fn x => x + 1)) [1,2]
=> (fn l => case l
             of [] => []
              | x::xs => ((fn x => x + 1) x)::(map (fn x => x + 1) xs)) [1,2]
=> case [1,2] of [] => []
              | x::xs => ((fn x => x + 1) x)::(map (fn x => x + 1) xs))
=> ((fn x => x + 1) 1)::(map (fn x => x + 1) [2]))
=> (1 + 1)::(map (fn x => x + 1) [2]))
=> 2::(map (fn x => x + 1) [2]))
=> 2::(case [2] of [] => []
                | x::xs => ((fn x => x + 1) x)::(map (fn x => x + 1) xs))
=> 2::((fn x => x + 1) 2)::(map (fn x => x + 1) []))
=> 2::(2+1::(map (fn x => x + 1) []))
=> 2::(3::(map (fn x => x + 1) []))
=> 2::(3::(case [] of [] => []
                   | x::xs => ((fn x => x + 1) x)::(map (fn x => x + 1) xs)))
=> 2::3::[]
```

Note that function application, dual to the type constructor `->`, associates to the left, so we could have also just written `map (fn x => x + 1) [1,2]`.

Instead of taking two arguments, a function and a list, this version of `map` takes one argument (a function), and returns a function that takes the list and computes the result. Note that function application left-associates, so `f x y` gets parsed as `(f x) y`. Thus, a curried function can be applied to multiple arguments just by listing them in sequence, as in `map f xs` above. This is really two function applications: one of `map` to `f`, producing `map f : 'a list -> 'b list`; and one of `map f` to `xs`.

3.2 Syntactic Support

Because currying turns out to be so useful and omnipresent, there is special syntax for curried functions, where you write multiple arguments to a function in sequence, with spaces between them:

```
fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map f xs
```

This means exactly the same as the explicit `fn` binding in the body, as above.

3.2.1 Another way to write map

Many students in lab wrote `map` this way. Although it is correct, the way it is written is not quite as clear as the previous versions. However, a good exercise is to type annotate the various expressions in the program and to figure out why it is equivalent to map as we wrote it previously.

```
fun map (f : 'a -> 'b ) : 'a list -> 'b list =
    let
      fun mapf ([]:'a list) : 'b list = []
        | mapf (x::xs)                = f x :: (mapf xs)
    in
      mapf
    end
```

3.2.2 Exercises

- What is the type of `map double`?
- What is the type of `map Int.toString`?
- What is the type of `map Int.toString [1,2,3]`?

3.3 Okay, But Why Bother?

Thus far, currying may seem under-motivated: is it just a trivial syntactic transformation? It's clear that we can do this, but why do we want to? We will eventually discuss several advantages of curried functions, but here's a big-picture.

- It is easy to *partially apply* a curried function, e.g. you can write `map f`, rather than `fn l => map (f,l)`, when you want the function that maps `f` across a list. This will come up when you want to pass `map f` as an argument to another higher-order function, or when you want to compose it with other functions, as we will see below. This is a difference in readability, but not functionality.

- Currying you write programs that you wouldn't otherwise be able to write. Roughly, the idea is that sometimes you can do meaningful work without knowing all of the arguments. Currying gives you a way to say "if you give me just the first argument, I'll consume it entirely and produce a function that's waiting for the next".

  This is called *staged computation.* You can stage a function in more or less meaningful ways, but when it's done carefully it can have a profound effect on the performance of your program. It lets you insert real work between the arguments to the function, controlling what happens when. We will discuss this more later in the semester.

4 Higher Order Functions Are Patterns of Recursion

We've been talking a lot about patterns of recursion or templates for recursive code—most notably structural recursion, where you get one recursive call on each substructure given by the inductive definition of the type. So far we've been pretty lax about what that actually means.

One use of higher-order function is to make that discussion exactly precise. Think about `map` on lists: it totally encapsulates the idea of applying a transformation uniformly to a list of input. It would be very hard to have a more concise description of that process without losing precision in the description. The code that isn't purely the structure of that idea has been factored out into the argument `f`.

We will explore more common pattern of recursion on lists, trees, and other interesting datastructures over the remainder of the semester.

4.0.1 `op` Keyword

There's a little bit of syntactic support here, too. Specifically, SML has a lot of infix operators around, and the ability to create new ones, so there's also a way to indicate that you'd like to use something that's normally infix in a prefix way. That tool is `op`. For example, `op+` is equivalent to `fn (x : int, y : int) => x + y`.

4.1 What is in scope? (Closures)

A somewhat tricky, but very very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope. For example, lets use `map` to instantiate a function `raiseBy`:

```
fun raiseBy (l , c) = map (fn x => x + c) l
```

The function `fn x => x + c` adds `c` to its argument, where `c` bound as the argument to `raiseBy`. For example, in

```
    raiseBy ( [1,2,3] , 2)
=> map (fn x => x + 2 , [1,2,3])
=> (fn x => x + 2) 1 :: map (fn x => x + 2 , [2,3])
=> 1 + 2 :: map (fn x => x + 2 , [2,3])
=> 3 :: map (fn x => x + 2 , [2,3])
=> 3 :: (fn x => x + 2) 2 :: map (fn x => x + 2 , [3])
 = 3 :: 4 :: map (fn x => x + 2 , [3])
 = 3 :: 4 :: (fn x => x + 2) 3 :: map (fn x => x + 2 , [])
 = 3 :: 4 :: 5 :: map (fn x => x + 2 , [])
 = 3 :: 4 :: 5 :: []
```

the `c` gets specialized to `2`. If you keep stepping, the function `fn x => x + 2` gets applied to each element of `[1,2,3]`. The important fact, which takes some getting used to, is that the function `fn x => x + 2` is *dynamically generated*: at run-time, we make up new functions, which do not appear anywhere in the program's source code!

Some of the TAs may have shown you this puzzle—it demonstrates the same principle, but a little more simply:
What is the value of `z` in the following expresion?

```
val z = let val x = 3
            val f = fn y => y + x
            val x = 5
        in
          f 10
        end
```

Well, you know how to evaluate `let`: you evaluate the declarations in order, substituting as you go. So, you get

```
let val x = 3
    val f = fn y => y + 3
    val x = 5
 in
    f 10
 end
```

The fact that `x` is shadowed below is irrelevant; the result is `13`. This is one of the reasons why we've been teaching you the substitution model of evaluation all semester; it explains tricky puzzles like this in a natural way.

5 Testing and Equality

It is impossible for sml to test for equality of functions. You would have to check that the they *agree on all arguments*, and there are infintely many. (Some of you are familiar with the haling problem; if ML could do this then you'd solve the halting problem.)

This is why we write proofs of our programs. Because we can write a correct specification for particular programs, and *can* prove that they satisfy that specification.

Some amount of testing is still useful for these functions. `lab05-sol.sml` contains some good examples of how to test your code.