CSE 341: Programming Languages

Autumn 2005

Lecture 8 — Function Closures

Today

- Continue examples of functions taking and returning other functions
- Discuss *free variables* in function bodies
- In general, discuss environments and lexical scope
- See key idioms using first-class functions

Environments (Key concept!)

We evaluate expressions in an evironment, and function bodies in an environment extended to map arguments to values.

But which one? The environment in which the function was defined!

An equivalent description:

- Functions are values, but they're not just code.
- `fun f p = e` and `fn p => e` evaluate to values with two parts (a "pair"): the code and the current environment
- Function application evaluates the "pair"'s function body in the "pair"'s environment (extended)
- This "pair" is called a *(function) closure*.

There are *lots* of good reasons for this semantics.

For hw, exams, and competent programming, you must "get this"!

Example 1

```
val x = 1;
fun f y = x + y;
val x = 2;
val y = 3;
f (x+y);
```

Example 2

```
val x = 1;
fun f y = let val x = 2 in fn z => x + y + z end;
val x = 3;
val g = f 4;
val y = 5;
g 100;
```

Example 3

```
fun f g = let val x = 3 in g 2 end;
val x = 4;
fun h y = x + y;
f h;
```

Scope

A key language concept: how are user-defined things *resolved*?

We have seen that ML has *lexically scoped* variables?

Another (more-antiquated-for-variables, sometimes-useful) approach is *dynamic scope*

Example of dynamic scope: Exception handlers (where does `raise` transfer control?)

The more restrictive "no free variables" makes important idioms impossible.

Why lexical scope?

1. Functions can be reasoned about (defined, type-checked, etc.) where defined
2. Function meaning not related to choice of variable names
3. "Closing over" local variables creates private data; function definer *knows* function users do not depend on it

Example:

```
fun add_2x x = fn z => z + x + x

fun add_2x x = let val y = x + x in fn z => z + y end
```

Key idioms with closures

- Create similar functions
- Pass functions with private data to iterators (map, *fold*, ...)
- Combine functions
- Provide an ADT
- As a *callback* without the "wrong side" specifying the environment.
- Partially apply functions ("currying")

Create similar functions

```
val addn = fn n => fn m => n+m

val increment = addn 1

val add_two = addn 2
```

map and fold

Earlier we saw `map`. A slightly more complex but also very useful function is `fold`:

```
fun fold (f,acc,[]) = acc
| fold (f,acc,x::xs) = fold (f, f(acc,x), xs)
```

Example uses (without using private data):

```
fun sum s = fold ((fn (x,y) => x+y), 0, s)
fun product s = fold ((fn (x,y) => x*y), 1, s)
fun and_list s = fold ((fn (x,y) => x andalso y), true, s)
fun or_list s = fold ((fn (x,y) => x orelse y), false, s)

(* interesting definition of member - not so practical
   in ML though - works better in Miranda or Haskell *)
fun member (x,list) = or_list (map ((fn y => x=y), list))
```

More on fold

Another more general example:

A fold function over a data structure is much like a *visitor pattern* in OOP.

We define fold once and do not restrict the type of the function passed to fold or the environment in which it is defined.

In general, libraries should not unnecessarily restrict clients.

Combine functions

```
fun f1 (g,h) = fn x => g (h x)
fun f2 (g,h) = fn x =>
                case g x of NONE => h x | SOME y => y
```