CSE 341: Programming Languages

Spring 2006

Lecture 9 — Closures, Map, Fold, Curry

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?)

Another example of dynamic scope: shell commands and shell scripts (environment variables)

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
fun f n =
   if n=0
   then []
   else (addn n)::(f (n-1))
```

Partial application (“currying”)

Recall every function in ML takes exactly one argument.

Previously, we simulated multiple arguments by using one n-tuple argument.

Another way: take one argument and return a function that takes another argument and ...

This is called “currying” after its inventor, Haskell Curry

Example:

```
fun inorder3 x = fn y => fn z =>
           z >= y andalso y >= x
((inorder3 4) 5) 6
inorder3 4 5 6
val is_pos = inorder3 0 0
```

A currying shortcut

We've generally seen curried functions written like this:

```
fun inorder3 x = fn y => fn z =>
           z >= y andalso y >= x
```

But there's a much more convenient syntax:

```
fun inorder3 x y z =
           z >= y andalso y >= x
```

More currying idioms

Currying is particularly convenient when creating similar functions with map or fold:

```
val sum = foldl (op +) 0;
val product = foldl (op *) 1;
fun mymap f [] = []
  | mymap f (x::xs) = f x :: mymap f xs;
val k = mymap (fn x => x+1) [1,2,3];
```

Currying vs. Pairs

Currying is elegant, but a bit backward: the function writer chooses which *partial application* is most convenient.

Of course, it's easy to write wrapper functions:

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Private data, for map/fold

Previously we saw `map`. This `fold` function is even more useful:

```
fun fold f acc l =
  case l of
    []     => acc
  | x::xs => fold f (f(x,acc)) xs
```

Example uses (without using private data):

```
fun f1 l = fold (fn (x,y) => x+y) 0 l
fun f2 l = fold (fn (x,y) => y andalso x >= 0) true l
```

Example use (with private data):

```
fun f3 (l,lo,hi) =
  fold (fn (x,y) =>
          if x >= lo andalso x <= hi then y+1 else y)
       0 l
```

More on fold and private data

Another more general example:

```
fun f4 g l = fold (fn (y,l2) => (g y)::l2) [] l
```

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
```