CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 9— Callbacks, Currying, and Mutual Recursion

Key idioms with closures

- Create similar functions
- Pass functions with private data to iterators (map, *fold*, ...)
- Combine functions
- Provide an ADT (see section)

- As a *callback* without the "wrong side" specifying the environment.
- Partially apply functions ("currying")

Callbacks

A common idiom: Library takes a function to apply later, when an *event* occurs. Examples:

- When a key is pressed, a mouse moved, etc.
- When a packet arrives from the network

The function may be a filter ("I want the packet") or return a result ("draw a line"), etc.

Library may accept multiple callbacks. Different callbacks may need different private state with different types.

Fortunately, the type of a function does not depend on the type of free variables.

Callback example (with mutable state!)

Library interface:

```
datatype action = ...
fun register_callback : ((int -> bool),action) -> unit
```

Library implementation (mutation, but hidden from clients)

```
val cbs : (int -> bool) list ref = ref []
fun register_callback f = cbs := f::(!cbs)
fun on_event i =
   let fun inner l =
        case l of
          [] => []
        | (f,a)::tl =>
           if (f i) then a::(inner tl) else inner tl
    in inner (!cbs) end
```

Example continued

Clients (kind of pseudocode):

```
register_callback ((fn i => true),   Log ...)
register_callback ((fn i => i = 80), Http_get ...)
val lst = countup(1,10)
fun in_lst j =
  case lst of [] => false | hd::tl => hd=j orelse in_lst tl
register_callback (in_lst, Other ...)
```

Key point: clients functions can use client-defined data, without library knowing anything about that data

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" because of someone named Curry.

Example:

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

More currying idioms

Currying is particularly convenient when creating similar functions with iterators:

```
fun fold_old (f,acc,l) =
  case l of
    []     => acc
  | hd::tl => fold_old (f, f(acc,hd), tl)
fun fold_new f = fn acc => fn l =>
  case l of
    []     => acc
  | hd::tl => fold_new f (f(acc,hd)) tl
fun sum1 l = fold_old ((fn (x,y) => x+y), 0, l)
val sum2 = fold_new (fn (x,y) => x+y) 0
```

There's even convenient syntax: `fun fold_new f acc l = ...`

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
```

Digression: There's something really, really intriguing about the types of `curry` and `uncurry` if you pronounce `->` as *implies* and `*` as *and*.

Function-Call Efficiency

First: Function calls take constant ($O(1)$) time, so until you're using the right algorithms and have a critical *bottleneck*, forget about it.

That said, ML's "all functions take one argument" can be inefficient in general:

- Create a new n-tuple
- Create a new function closure (2 homeworks from now)

In practice, implementations *optimize* common cases. In some implementations, n-tuples are faster (avoid building the tuple). In others, currying is faster (avoid building intermediate closures).

In the < 1 percent of code where detailed efficiency matters, you program against an implementation. Bad programmers worry about this stuff at the wrong stage and for the wrong code.

Mutual Recursion

We haven't yet seen how multiple functions can recursively call each other? (Why can't we do this with what we have?)

ML uses the keyword `and` to provide different *scope* rules. Example:

```
fun even i = if i=0 then true  else odd (i-1)
and odd  i = if i=0 then false else even (i-1)
```

Roughly extends the binding form for functions from `fun f1 x1 = e1` to `fun f1 x1 = e1 and f2 x2 = e2 and ... and fn xn = en`.

Actually, you can have `val` bindings too, but bindings being defined are in scope only inside function bodies. (Why?)

Syntax gotcha: I always forget you write `and fi xi = ei`, not `and fun fi xi = ei`.

Mutual Recursion Idioms

1. Encode a state machine (see `product_sign` example)
   - Sometimes easier to understand than explicit state values.
2. Process mutually recursive types, example:

```
datatype webtext = Empty
                 | Link of webpage * string * webtext
                 | Word of string * webtext
and webpage = Found of string * webtext
            | Unfound of string
```

   A function "crawl for word" is inherently mutually recursive. (You could make a datatype for "webtext or webpage", but that's ugly.)

   Problem: the Web has *cycles*, which (sigh) is a common need for mutation in ML.

   Unproblem: When crawling, we don't want cycles (use Unfound if we have seen the page before).