CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 9— More function-closure idioms

Key idioms with closures

- Create similar functions
- Combine functions
- Pass functions with private data to iterators (map, *fold*, ...)

- Provide an abstract data type (ADT)
- As a *callback* without the "wrong side" specifying the environment.
- Partially apply functions ("currying")

Provide an ADT

A record of functions is much like an object.

Free variables are much like private, immutable fields.

Our "set" example is fancy stuff, but you should be able to understand it even if you wouldn't have thought to do it.

```
datatype set = S of {add:int -> set, member:int -> bool}
val empty_set = S {add=fn,member=fn} : set
```

Callbacks

A common idiom: Library takes a function to apply later, when an *event* occurs (e.g., Java Swing library). 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"), do something ("draw a line"), etc.

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

Fortunately, a function's type doesn't depend on the type of free variables.

Note: This is why Java has anonymous inner classes (for "event listeners").

Callback example (with mutable state!)

Library interface:

```
val onKeyEvent : (int -> unit) -> unit
```

Library implementation (mutation, but hidden from clients)

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f::(!cbs)
fun on_event i =
   let fun loop l =
        case l of
          [] => []
        | f::tl => f i; loop tl
    in loop (!cbs) end
```

Example continued

Clients (kind of pseudocode):

```
onKeyEvent (fn i => write_to_log(Int.toString i
                                 ^ " got pressed\n"));

val f4_key = 75; (* no idea what it really is *)
onKeyEvent (fn i => if i=f4_key then minimizeWindow() else ());

fun prohibit_keys lst =
  onKeyEvent (fn i => if (List.exists (fn j => j=i) lst)
                      then exitProgram()
                      else ());
prohibit_keys [13, 42, 99];
```

Key point: clients functions can use client-defined data of any type

Note: `List.exists` is a library function using currying...

Partial application ("currying")

Recall every function in ML takes exactly one argument.

Previously, we encoded n 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 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_non_negative = 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 = ...`

Back to List.exists

```
fun exists predicate lst =
    case lst of
      [] => false
    | hd::tl => predicate hd orelse exists predicate tl
```

Example clients:

```
fun has_seventeen1 lst = exists (fn x => x=17) lst
val has_seventeen2 = exists (fn x => x=17)
fun make_has_n n = exists (fn x => x=n)
val has_seventeen3 = make_has_n 17
```

Currying vs. Tuples

Currying is more elegant than tuples, but still 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
```

If you look at the types of these functions, you can get a very good idea of what they do.

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

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.