CSE 341: Programming Languages

Autumn 2005

Lecture 9 — ADTs, Callbacks, and Currying

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")

Provide an ADT

A record of functions is much like an object.

Free variables are prviate variables.

Our "set" example is fancy stuff, but you should be able to understand it.

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

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.

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

Mutable State in ML (Used in Callback Example)

Style guideline: only use mutable state in ML if it is truly necessary. (It won't necessary on any 341 homework or exam, unless explicitly stated.)

```
(* create a reference to something *)
val r1 = ref 3;

(* get the value (dereference) *)
val k = !r1;

(* change the value being referred to *)
r1 := 4;   (* Note fix from printed copy of the slides! *)
```

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 f l =
        case l of
          [] => []
        | (f,a)::tl =>
           if (f i) then a::(f tl) else inner tl
    in f (!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: client 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" after the logician Haskell Curry. It is used in some cases in ML; in other languages such as Miranda and Haskell, it is used for virtually all functions.

Example:

```
fun add x y = x+y;
val addone = add 1;
```

What is the type of `add`? Of `addone`?

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 has the potential problem that the function writer chooses which *partial application* is most convenient, rather than the user.

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
```

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.