CSE341: Programming Languages

Lecture 9
Function-Closure Idioms

Dan Grossman

Fall 2011

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions

Combine functions

Canonical example is function composition:

```
fun compose (g,h) = fn x => g (h x)
```

- Creates a closure that "remembers" what **g** and **h** are bound to
- Type **('b -> 'c) * ('a -> 'b) -> ('a -> 'c)** but the REPL prints something *equivalent*

- ML standard library provides this as infix operator **o**
- Example (third version best):

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt(abs i))
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

Left-to-right or right-to-left

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

As in math, function composition is “right to left”

- “take absolute value, convert to real, and take square root”
- “square root of the conversion to real of absolute value”

“Pipelines” of functions are common in functional programming and many programmers prefer left-to-right

- Can define our own infix operator
- This one is very popular (and predefined) in F#

```
infix |>
fun x |> f = f x

fun sqrt_of_abs i =
   i |> abs |> Real.fromInt |> Math.sqrt
```

Another example

- "Backup function"

```
fun backup1 (f,g) =
     fn x => case f x of
                NONE => g x
              | SOME y => y
```

- As is often the case with higher-order functions, the types hint at what the function does

```
('a -> 'b option) * ('a -> 'b) -> 'a -> 'b
```

- More examples later to "curry" and "uncurry" functions

Currying and Partial Application

- Recall every ML function takes exactly one argument
- Previously encoded *n* arguments via one *n*-tuple
- Another way: Take one argument and return a function that takes another argument and…
  - Called "currying" after famous logician Haskell Curry
- Example, with full and partial application:
  - Notice relies on lexical scope

```
val sorted3 = fn x => fn y => fn z =>
                      z >= y andalso y >= x

val true_ans = ((sorted3 7) 9) 11

val is_non_negative = (sorted3 0) 0
```

Syntactic sugar

Currying is much prettier than we have indicated so far

- Can write **`e1 e2 e3 e4`** in place of **`((e1 e2) e3) e4`**
- Can write **`fun f x y z = e`** in place of

  `fun f x = fn y => fn z => e`

```
fun sorted3 x y z = z >= y andalso y >= x
val true_ans = sorted3 7 9 11
val is_non_negative = sorted3 0 0
```

Result is a little shorter and prettier than the tupled version:

```
fun sorted3 (x,y,z) = z >= y andalso y >= x
val true_ans = sorted3(7,9,11)
fun is_non_negative x = sorted3(0,0,x)
```

Return to the fold ☺

In addition to being sufficient multi-argument functions and pretty, currying is useful because partial application is convenient

Example: Often use higher-order functions to create other functions

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

fun sum_ok xs = fold (fn (x,y) => x+y) 0 xs

val sum_cool = fold (fn (x,y) => x+y) 0
```

The library's way

- So the SML standard library is fond of currying iterators
  - See types for **`List.map`**, **`List.filter`**, **`List.foldl`**, etc.
  - So calling them as though arguments are tupled won't work

- Another example is **`List.exists`**:

```
fun exists predicate xs =
    case xs of
      []      => false
    | x::xs' => predicate xs
                orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val has_seven = exists (fn x => x=7)
```

Another example

Currying and partial application can be convenient even without higher-order functions

```
fun zip xs ys =
   case (xs,ys) of
        ([],[]) => []
      | (x::xs',y::ys') => (x,y)::(zip xs' ys')
      | _ => raise Empty

fun range i j =
    if i>j then [] else i :: range (i+1) j

val countup = range 1 (* partial application *)

fun add_number xs = zip (countup (length xs)) xs
```

More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function's arguments are in the wrong order for the partial application you want?

Naturally, it's easy to write higher-order wrapper functions

- And their types are neat logical formulas

```
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
```

The Value Restriction Appears ☹

If you use partial application to create a polymorphic function, it may not work due to the value restriction

- Warning about "type vars not generalized"
  - And won't let you call the function
- This should surprise you; you did nothing wrong ☺ but you still must change your code
- See the written lecture summary about how to work around this wart (and ignore the issue until it arises)
- The wart is there for good reasons, related to mutation and not breaking the type system
- More in the lecture on type inference

Efficiency

So which is faster: tupling or currying multiple-arguments?

- They are both constant-time operations, so it doesn't matter in most of your code – "plenty fast"
  - Don't program against an *implementation* until it matters!

- For the small (zero?) part where efficiency matters:
  - It turns out SML NJ compiles tuples more efficiently
  - But many other functional-language implementations do better with currying (OCaml, F#, Haskell)
    - So currying is the "normal thing" and programmers read **`t1 -> t2 -> t3 -> t4`** as a 3-argument function

Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- Different callbacks may need different private data with different types
- Fortunately, a function's type does not include the types of bindings in its environment
- (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

- We really do want the "callbacks registered" and "events that have been delivered" to *change* due to function calls

For the reasons we have discussed, ML variables really are immutable, but there are mutable references (use sparingly)

- New types: **`t ref`** where **`t`** is a type
- New expressions:
  - **`ref e`** to create a reference with initial contents e
  - **`e1 := e2`** to update contents
  - **`!e`** to retrieve contents (not negation)

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check)
```

- A variable bound to a reference (e.g., **`x`**) is still immutable: it will always refer to the same reference
- But the contents of the reference may change via **`:=`**
- And there may be aliases to the reference, which matter a lot
- Reference are first-class values
- Like a one-field mutable object, so **`:=`** and **`!`** don't specify the field

Example call-back library

Library maintains mutable state for “what callbacks are there” and provides a function for accepting new ones

- A real library would support removing them, etc.
- In example, callbacks have type **`int->unit`** (executed for side-effect)

So the entire public library interface would be the function for registering new callbacks:

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

Library implementation

```
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =  cbs := f :: (!cbs)

fun onEvent i =
    let fun loop fs =
            case fs of
              []     => ()
            | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

Clients

Can only register an **`int -> unit`**, so if any other data is needed, must be in closure's environment

- And if need to "remember" something, need mutable state

Examples:

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
   onKeyEvent (fn _ =>
       if i=j
       then print ("pressed " ^ Int.toString i)
       else ())
```

Implementing an ADT

As our last pattern, closures can implement abstract datatypes

- Can put multiple functions in a record
- They can share the same private data
- Private data can be mutable or immutable (latter preferred?)
- Feels quite a bit like objects, emphasizing that OOP and functional programming have similarities

See lec9.sml for an implementation of immutable integer sets with operations *insert*, *member*, and *size*

The actual code is advanced/clever/tricky, but has no new features

- Combines lexical scope, datatypes, records, closures, etc.
- Client use is not so tricky