Curried functions and other tasty closure recipes

Function composition

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

Closure “remembers” f and g

```
: ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

REPL prints something equivalent

ML standard library provides infix operator •

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

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

Right to left.

More idioms for closures

- Function composition
- Currying and partial application
- Callbacks (e.g., in reactive programming)
- Functions as data representation (later)

Pipelines (left-to-right composition)

“Pipelines” of functions are common in functional programming.

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

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

(FR, Microsoft’s ML flavor defines this by default)
Currying

• 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 logician Haskell Curry

Example

```
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

• Calling (sorted3 7) returns a closure with:
  • Code fn y => fn z => z >= y andalso y >= x
  • Environment maps x to 7

• Calling that closure on 9 returns a closure with:
  • Code fn z => z >= y andalso y >= x
  • Environment maps x to 7, y to 9

• Calling that closure on 11 returns true

Function application is left-associative

```
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

```
e1 e2 e3 e4
means
  ((e1 e2) e3) e4
val t1 = sorted3 7 9 11
```

Callers can just think
“multi-argument function with spaces instead of a tuple expression”
Does not interchange with tupled version.

Function definitions are sugared (again)

```
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

```
fun f p1 p2 p3 ... = e
means
  fun f p1 = fn p2 => fn p3 => ...
val t1 = sorted3 x y z
```

Callees can just think
“multi-argument function with spaces instead of a tuple pattern”
Does not interchange with tupled version.
Final version

As elegant syntactic sugar (fewer characters than tupling) for:

```haskell
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

Function application is left-associative.

Types are right-associative:

```haskell
sorted3 : int -> int -> bool
means sorted3 : int -> (int -> bool)
```

Partial Application ("too few arguments")

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

Curried fold

A more useful example and a call to it
Will improve call next

```haskell
fun fold f acc xs =
  case xs of
    [] => acc
    | x::xs' => fold f (f(x,acc)) xs'
fun sum xs = fold (fn (x,y) => x+y) 0 xs
```

Unnecessary function wrapping

```haskell
fun f x = g x (* bad *)
val f = g (* good *)

fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs
   (* bad *)
val sum = fold (fn (x,y) => x+y) 0
   (* good *)
val sum = fold (op+) 0

(* best? *)
val sum = fold (op+) 0
```

Partial Application ("too few arguments")

```haskell
fun fold f acc xs =
  case xs of
    [] => acc
    | x::xs' => fold f (f(acc,x)) xs'
fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs
val sum = fold (fn (x,y) => x+y) 0
```

Partial Application ("too few arguments")

```haskell
  fold (fn (x,y) => x+y) 0
```
Iterators and partial application

fun exists predicate xs = case xs of
  [] => false
| x::xs' => predicate x
  orelse exists predicate xs'
val no = exists (fn x => x=7) [4,11,23]
val hasZero = exists (fn x => x=0)

For this reason, ML library functions of this form are usually curried
• List.map, List.filter, List.foldl, ...

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 code for workarounds
• Can discuss a bit more when discussing type inference

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 is 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

Efficiency

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

• Both constant-time
  • Don’t program against an implementation until it matters!
• For the small (zero?) part where efficiency matters:
  • SML/NJ compiles tuples more efficiently
  • Many other implementations do better with currying (OCaml, FR, Haskell GHC)
    • So currying is the “normalizing” and programmers read t1 -> t2 -> t3 -> t4 as
      3-argument function that also allows partial application
More idioms

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

ML has (separate) mutation

• Mutable data structures are okay in some situations
  • When “update to state of world” is appropriate model
  • But want most language constructs truly immutable

• ML does this with a separate construct: references
  • Introducing now because will use them for next closure idiom

• Do not use references on your homework
  • You need practice with mutation-free programming
  • They will lead to less elegant solutions

References

• 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

```haskell
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
• Contents of the reference may change via :=
• There may be aliases to the reference, which matter a lot
• References are first-class values
• Like a one-field mutable object, so := and ! don’t specify the field
Callback idiom

Library takes function to apply later, when an event occurs.

Library interface:

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

Other 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
- Function's type does not include the types of bindings in its environment!
- (OOP: objects + private fields used similarly, e.g., Java Swing event-listeners)
- See also JavaScript callbacks, events

Library implementation

Mutate state not absolutely necessary, but is reasonably appropriate.

```haskell
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

Closure’s environment captures any necessary context, possibly including mutable state for “remembering” history.

```haskell
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
  timesPressed := (!timesPressed) + 1)
fun printIfPressed i =
onKeyEvent (fn j =>
  if i=j
  then print ("pressed " ^ Int.toString i)
  else ()
)
fun makeCounterCallback k = let count = ref 0 in
  onKeyEvent (fn i => if i=k
    then count := !count + 1
    else ()
  )
  count
end
```