Currying and Partial Application

and other tasty closure recipes

More idioms for closures

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

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 o

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

Right to left.

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

(F#, Microsoft's ML flavor, defines this by default)
Currying

• 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

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

• Calling \((\text{sorted3} \ 7)\) returns a closure with:
  – Code \(\text{fn} \ y \Rightarrow \text{fn} \ z \Rightarrow z >= y \ \text{andalso} \ y >= x\)
  – Environment binds \(x\) to 7

• Calling \(\text{that}\) closure on 9 returns a closure with:
  – Code \(\text{fn} \ z \Rightarrow z >= y \ \text{andalso} \ y >= x\)
  – Environment binds \(x\) to 7, \(y\) to 9

• Calling \(\text{that}\) closure on 11 returns true

Function application is left-associative

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

\(e1 \ e2 \ e3 \ e4\)

means

\(((e1 \ e2) \ e3) \ e4\)

```plaintext
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 sugar (again)

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

```plaintext
fun f p1 p2 p3 ... = e
```

desugars to

```plaintext
fun f p1 = fn p2 => fn p3 => ... => e
```

```plaintext
fun sorted3 x y z = z >= y andalso y >= x
```

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

\[
\begin{align*}
\text{fun sorted3 } x \ y \ z &= z \geq y \text{ andalso } y \geq x \\
\text{val t1 = sorted3 } 7 \ 9 \ 11
\end{align*}
\]

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

\[
\begin{align*}
\text{val sorted3 } &= \text{fn } x \Rightarrow \text{fn } y \Rightarrow \text{fn } z \Rightarrow \\
&\quad z \geq y \text{ andalso } y \geq x \\
\text{val t1 } &= ((\text{sorted3} \ 7) \ 9) \ 11
\end{align*}
\]

Function application is left-associative.

Types are right-associative:

\[
\begin{align*}
\text{sorted3} &\colon \text{int } \rightarrow \text{int } \rightarrow \text{int } \rightarrow \text{bool} \\
\text{means} &\quad \text{sorted3} : \text{int } \rightarrow (\text{int } \rightarrow (\text{int } \rightarrow \text{bool}))
\end{align*}
\]

Curried foldl

\[
\begin{align*}
\text{fun foldl } f \ acc \ xs &= \\
&\quad \text{case } xs \ of \\
&\quad \quad [] \Rightarrow acc \\
&\quad \quad x :: xs' \Rightarrow \text{foldl } f (f(acc,x)) \ xs'
\end{align*}
\]

\[
\begin{align*}
\text{fun sum } xs &= \text{foldl } (\text{fn } (x,y) \Rightarrow x+y) \ 0 \ xs
\end{align*}
\]

Partial Application

\[
\begin{align*}
\text{fun foldl } f \ acc \ xs &= \\
&\quad \text{case } xs \ of \\
&\quad \quad [] \Rightarrow acc \\
&\quad \quad x :: xs' \Rightarrow \text{foldl } f (f(acc,x)) \ xs'
\end{align*}
\]

\[
\begin{align*}
\text{fun sum_inferior } xs &= \text{foldl } (\text{fn } (x,y) \Rightarrow x+y) \ 0 \ xs \\
\text{val sum} &= \text{foldl } (\text{fn } (x,y) \Rightarrow x+y) \ 0
\end{align*}
\]

\[
\begin{align*}
\text{foldl } (\text{fn } (x,y) \Rightarrow x+y) \ 0 \text{ evaluates to a closure that, when called with a list } xs, \text{ evaluates the case-expression with:} \\
&\quad f \text{ bound to the result of } \text{foldl } (\text{fn } (x,y) \Rightarrow x+y) \\
&\quad acc \text{ bound to 0}
\end{align*}
\]

Unnecessary function wrapping

\[
\begin{align*}
\text{fun f } x &= g \ x \text{ (* bad *)} \\
\text{val f } &= g \text{ (* good *)}
\end{align*}
\]

\[
\begin{align*}
\text{fun sum_inferior } xs &= \text{fold } (\text{fn } (x,y) \Rightarrow x+y) \ 0 \ xs \\
\text{val sum} &= \text{fold } (\text{fn } (x,y) \Rightarrow x+y) \ 0
\end{align*}
\]

\[
\begin{align*}
\text{fold } (\text{fn } (x,y) \Rightarrow x+y) \ 0 \text{ evaluates to a closure that, when called with a list } xs, \text{ evaluates the case-expression with:} \\
&\quad f \text{ bound to the result of } \text{fold } (\text{fn } (x,y) \Rightarrow x+y) \\
&\quad acc \text{ bound to 0}
\end{align*}
\]

Treat infix operator as normal function.
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 😞

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