15-150

Principles of Functional Programming

Lecture 11

February 20, 2025

Michael Erdmann

Combinators, Staging, and Natural Folding

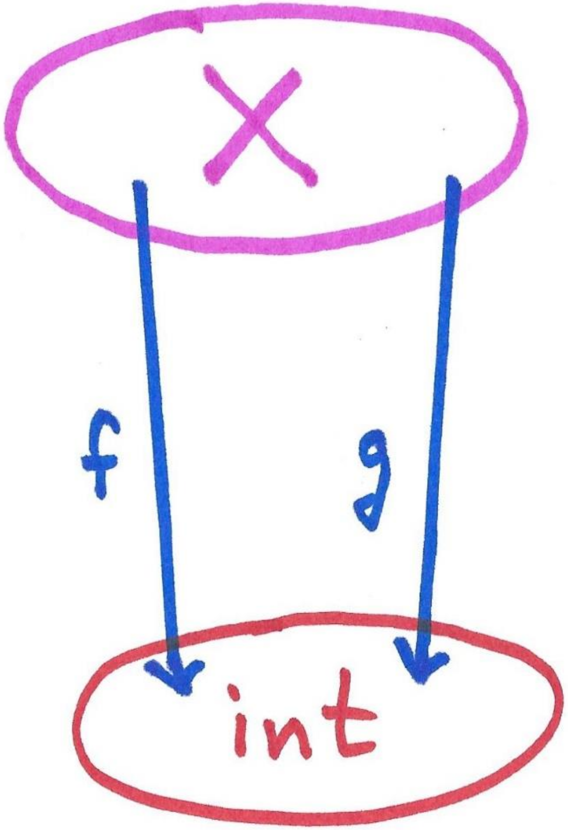
Combinators

E.g., composition

$f \circ g$

Combinators are functions that combine small pieces of code into larger pieces of code.

We will view combinators as higher order functions that expect functions and return functions.



operations on
integer-valued
functions

++, **, MIN

+, *, Int.min, ...

operations on
integers

In math, one may write the sum of two integer-valued functions in a point-free way:

$$f + g.$$

If someone asks "What does that mean?", we would explain using a point-specific equation:

$$(f \underset{\substack{\uparrow \\ \text{combinator}}}{+} g)(x) = f(x) \underset{\substack{\uparrow \\ \text{integer addition}}}{+} g(x).$$

In SML, we will define combinators in code using this pointwise principle, then use the combinators for point-free programming.

<u>infixr</u> ++ (* declares ++ to be an infix right-associative operator *)

<u>fun</u> (f ++ g) x = f(x) + g(x)

Alternatively, we could first declare

<u>fun</u> ++ (f, g) x = f(x) + g(x)

and subsequently write <u>infixr</u> ++.

Other forms of declaration are possible, e.g.,

<u>fun</u> ++ (f, g) = <u>fn</u> x ⇒ f(x) + g(x).

What is the type of ++? i.e., of (op ++)

('a → int) * ('a → int) → 'a → int

fun square x = x * x

fun double x = 2 * x

We can combine these function values:

val quadratic = square ++ double

Observe: quadratic $\cong$ **fn** x => x * x + 2 * x

i.e., quadratic represents the function $x^2 + 2x$.

quadratic (3) $\hookrightarrow$ 15

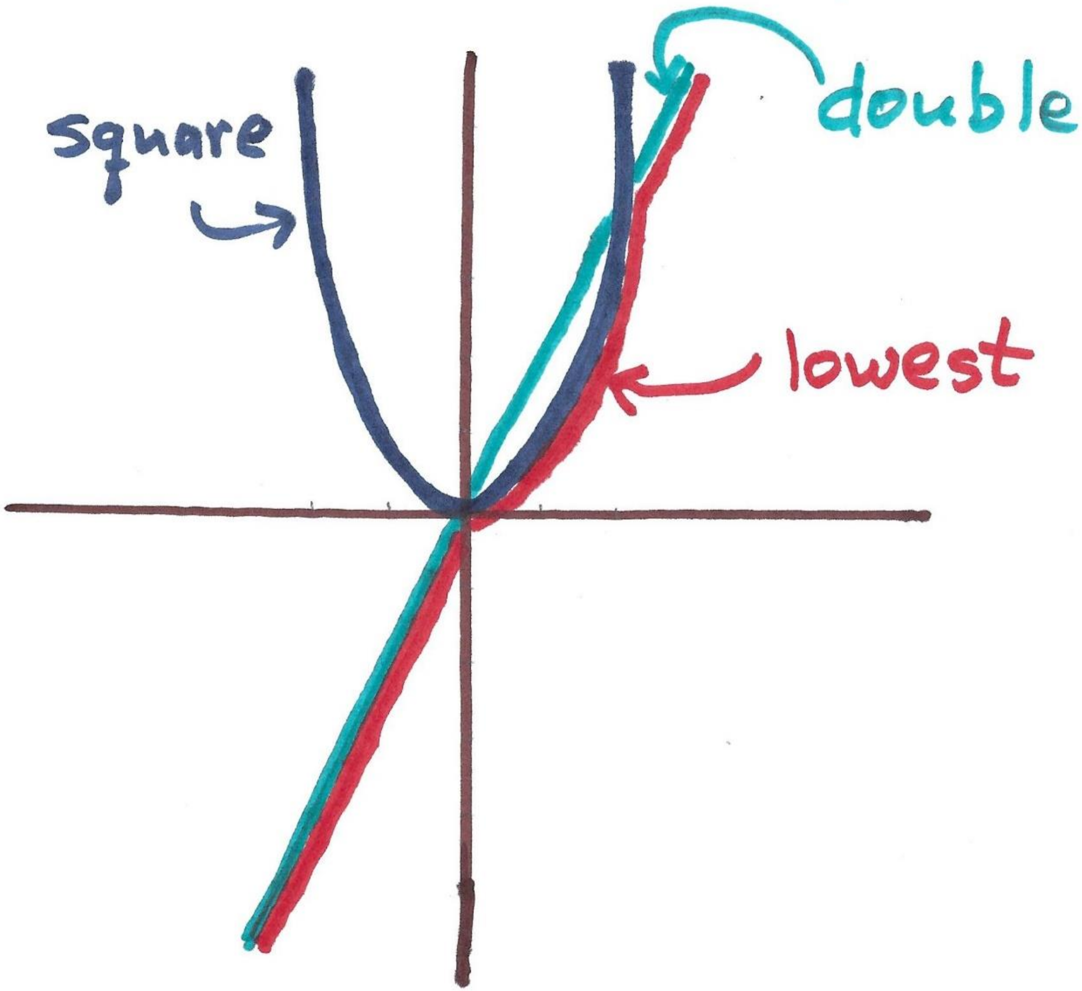
```
infixr **
fun (f ** g) x = f(x) * g(x)
fun MIN (f, g) x = Int.min(f(x), g(x))
```

```
(op **) : ('a -> int) * ('a -> int) -> 'a -> int

MIN : ('a -> int) * ('a -> int) -> ('a -> int)
```

(Two different ways of writing the same type, by right-associativity of arrows in function types.)

val lowest = MIN(square, double)



Staging

Staging is a coding technique that has a function perform useful work prior to receiving all its arguments.

```
fun f (x:int, y:int): int =
    let
        val z:int = horriblecomputation(x)
    in
        z + y
    end
```

Suppose the horrible computation takes 10 months. (Suppose addition takes a picosec.)

Then each of these expression takes at least 10 months to evaluate:

f (5, 2)
f (5, 3)

If only we could recall horriblecomputation (5) !

What is the type of f ?

int * int $\rightarrow$ int

Maybe currying would help.

Let's define a curried version of f.

```
fun g (x: int) (y: int) : int =
    let
        val z : int = horriblecomputation(x)
    in z + y
    end
```

Now g : int → int → int , so we can define

```
val g5 : int -> int = g(5)
```

and then evaluate

```
g5 2    (* instead of f(5,2) *)
g5 3    (* instead of f(5,3) *)
```

How long does each of the three lines above take?

It helps to remember that the declaration of g created the binding

$$[\ \underline{\text{fn}}\ x \Rightarrow \underline{\text{fn}}\ y \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}}\,/\,g\ ].$$

In declaring $\underline{\text{val}}\ g5 = g(5)$, one evaluates

$$g(5)$$

$$\Rightarrow (\underline{\text{fn}}\ x \Rightarrow \underline{\text{fn}}\ y \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}})(5)$$

$$\Rightarrow \underbrace{[\,5/x\,]\ \underbrace{\underline{\text{fn}}\ y \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}}}_{\substack{\text{This is a lambda expression} \\ \text{(it is } \underline{\text{not}} \text{ applied to any argument here)}}}}_{\text{This closure is the value returned by } g(5).}$$

This value is returned nearly instantaneously.
The horrible computation has not yet happened.

We now have the binding

$$\left[\ \boxed{\begin{array}{l} [5/x] \\ \underline{\text{fn}}\ y \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}} \end{array}}\ \Big/\ g5\ \right]$$

Evaluating $\quad g5\ \ 2$

$$\Longrightarrow [5/x,\ 2/y]\ \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}}$$

This step takes 10 months

$$\Longrightarrow [5/x,\ 2/y,\ \text{some integer}/z]\ z+y$$

$$\Longrightarrow \text{some other integer}$$

Similarly, evaluation of $g5\ 3$ takes 10 months.

We now have the binding

$$\left[\boxed{\begin{array}{l}[5/x] \\ \underline{\text{fn}}\ y \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}}\end{array}} \Big/ g5\right]$$

Evaluating $g5\ 2$

This step takes 10 months

$\Longrightarrow [5/x, 2/y]\ \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ z+y\ \underline{\text{end}}$

$\Longrightarrow [5/x, 2/y, \text{some integer}/z]\ z+y$

$\Longrightarrow$ some other integer

Similarly, evaluation of $g5\ 3$ takes 10 months.

Defining g in place of f has *not* yet helped!

Recall the lambda expression for g :

fn x ⇒ **fn** y ⇒ **let** **val** z = hc(x) **in** z+y **end**

Let us move this computation to here.

Such a rearrangement is what we mean by staging.

We can do that since the computation does not depend on y .

```
fun h (x:int) : int -> int =
    let
        val z : int = horriblecomputation(x)
    in
        fn (y:int) => z + y
    end
```

What is the type of h? int → int → int

How long does it take to evaluate each of these three lines?

```
val h5 : int -> int = h(5)
h5 2
h5 3
```

The declaration of h created the binding

$$[\,\underline{\text{fn}}\ x \Rightarrow \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ \underline{\text{fn}}\ y \Rightarrow z+y\ \underline{\text{end}} \,/\, h\,].$$

So, in declaring $\underline{\text{val}}\ h5 = h(5)$, one evaluates

$$h(5)$$

$$\Longrightarrow [5/x]\ \underline{\text{let}}\ \underline{\text{val}}\ z = hc(x)\ \underline{\text{in}}\ \underline{\text{fn}}\ y \Rightarrow z+y\ \underline{\text{end}}$$

$$\Longrightarrow [5/x, \text{some integer}/z]\ \underline{\text{fn}}\ y \Rightarrow z+y$$

This is a lambda expression (not applied to an argument).

This closure is the value returned by h(5).

This step takes 10 months.

We now have the binding

$$\left[\ \boxed{\begin{array}{c}[\,5/x,\ \text{some integer}/z\,]\\[6pt] \underline{\text{fn}}\ y \Rightarrow z+y\end{array}}\ \Big/\ h5\ \right]$$

Evaluating $h5\ 2$

$$\left.\begin{array}{l}\Rightarrow [\,5/x,\ \text{some integer}/z,\ 2/y\,]\ z+y\\ \Rightarrow \text{some other integer}\end{array}\right.$$

These steps occur quickly.

So, having staged the horrible computation to occur right after argument x is available, evaluation of $h5\ 2$ is fast. Similarly for $h5\ 3$.

Summary

f(5,2)	>10 months
f(5, 3)	>10 months
<u>val</u> g5 = g(5)	Fast
g5 2	>10 months
g5 3	> 10 months
<u>val</u> h5 = h(5)	> 10 months
h5 2	Fast
h5 3	Fast

Mapping & Folding

We can define natural mapping and folding functions over datatypes more general than lists.

map: replace constituent values

fold: replace constructors with functions (n-ary constructors become n-ary functions)

$\boxed{\text{catamorphism}}$

[More specialized folds are sometimes possible.]

```
datatype 'a tree = Empty
                 | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)

fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) =
      Node (tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)

fun tfold f z Empty = z
  | tfold f z (Node (l,x,r)) =
      f (tfold f z l, x, tfold f z r)
```

Examples

```
val stringify = tmap Int.toString
val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

Observe:

stringify : int tree → string tree

treesum : int tree → int

```
datatype 'a leafy = Leaf of 'a
                  | Node of 'a leafy * 'a leafy

(* lmap : ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f (Leaf x) = Leaf (f x)
  | lmap f (Node(l,r)) =
      Node (lmap f l, lmap f r)

(* lfold : ('b * 'b -> 'b) -> ('a -> 'b) -> 'a leafy -> 'b *)
fun lfold f g (Leaf x) = g(x)
  | lfold f g (Node(l,r)) =
      f (lfold f g l, lfold f g r)
```

Examples

```
val lstringify = lmap Int.toString
val leafysum   = lfold (op +) (fn x => x)
```

Observe

lstringify : int leafy → string leafy

leafysum : int leafy → int

Idea applies as well to nonrecursive datatypes.

```
datatype 'a option = NONE | SOME of 'a

(* opmap : ('a -> 'b) -> 'a option -> 'b option *)

fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold : ('a -> 'b) -> 'b -> 'a option -> 'b *)

fun opfold f z NONE = z
  | opfold f z (SOME x) = f x
```

Examples

```
val ostringify = opmap Int.toString
val osum       = opfold (fn x => x) 0
```

Observe

```
ostringify : int option -> string option
osum       : int option -> int
```

Question

Is the natural fold for lists foldr or foldl ?

That is all.

Have a good weekend.

See you Tuesday, when we will talk about continuations.