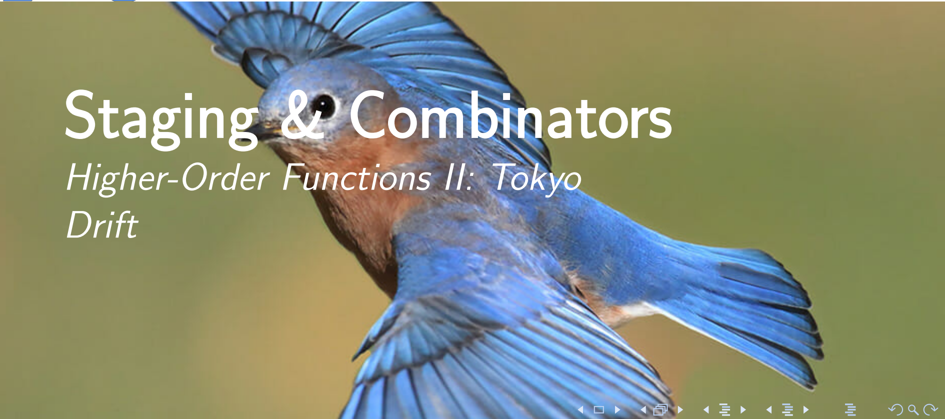
Lecture 11
Principles of Functional Programming
Summer 2020



Staging & Combinators

Higher-Order Functions II: Tokyo Drift

Section 1

Evaluation and Equivalence of HOFs

HOFs are trivially total

Thm. 1 `map` is total

Proof For any value `f : t1 -> t2`,

$$\texttt{map f} \Longrightarrow \texttt{fn [] => [] | x::xs => ...}$$

□

Thm. 2 `filter` is total

Proof For any value `p : t -> bool`,

$$\texttt{filter p} \Longrightarrow \texttt{fn [] => [] | x::xs => ...}$$

□

Higher-Order Totality?

A more interesting claim:

Thm. 3 For any types `t1`, `t2` and any total `f : t1 -> t2`, `map f` is total.

Proof By structural induction on `L : t1 list`

BC `L=[]`

$$\texttt{map f []} \Longrightarrow \texttt{[]}$$

IS `L=x::xs` for some `x:t1` and some `xs:t1 list`

IH `map f xs` $\hookrightarrow$ `vs` for some value `vs:t2 list`

$$\begin{aligned} & \texttt{map f (x::xs)} & \\ & \Longrightarrow \texttt{(f x)::map f xs} & \text{(defn }\texttt{map}\text{)} \\ & \Longrightarrow \texttt{(f x)::vs} & \boxed{\text{IH}} \\ & \Longrightarrow \texttt{v::vs} & \text{(}\texttt{f}\text{ is total)} \end{aligned}$$

for some value `v : t2`.

Thm. 4 For all total values `f : t1 -> t2`,

$$\texttt{len o (map f)} \cong \texttt{len}$$

Proof It suffices to show that for all values `L : t1 list`,

$$\texttt{(len o (map f)) L} \cong \texttt{len L}$$

where the right-hand side, by defn of `o`, is equivalent to `len(map f L)`. We prove this by structural induction on `L`.

BC `L=[]`

$$\begin{aligned} &\texttt{len(map f [])} \\ \Longrightarrow\ &\texttt{len []} && \text{(defn of } \texttt{map}\text{)} \end{aligned}$$

Thm. 4 For all total values `f : t1 -> t2`,

$$\texttt{len o (map f)} \cong \texttt{len}$$

Proof (continued)

IS `L=x::xs` for some `x:t1` and some `xs:t1 list`

IH $\texttt{len(map f xs)} \cong \texttt{len xs}$

$$\begin{aligned}
&\texttt{len(map f (x::xs))} && \\
&\cong \texttt{len((f x)::map f xs)} && \text{(defn of \texttt{map})} \\
&\cong \texttt{1 + len(map f xs)} && \text{(totality of \texttt{f}, Thm. 3)} \\
&\cong \texttt{1 + len xs} && \text{IH} \\
&\cong \texttt{len(x::xs)} && \text{(defn of \texttt{len})}
\end{aligned}$$

Section 2

Staging

What's the difference?

11.0

```
(* square : int -> int
 * ENSURES: square n ==> n * n, but it takes a long time *)

(* ex1,ex2 : int -> int -> int
 * REQUIRES: x>=0
 * ENSURES: ex1 x y == (x*x)+y
 *             ex2 x y == (x*x)+y
 *)
fun ex1 x y =
  let
    val xsq = square x
  in
    xsq + y
  end
fun ex2 x =
  let
```

Staging

Staging is deliberately structuring a curried function to perform computations once certain arguments are obtained.

```
fun foo x =
  let
    val v1 = horribleComputation x
  in
    (fn y =>
      let
        val v2 = otherHorribleComp(v1,y)
      in
        fn z => z + v1 + v2
      end
    )
  end
```

Section 3

Runtime Analysis of HOFs

Combine all the elements of a list

`foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`
REQUIRES: g is total
ENSURES: `foldr g acc [x1,...,xn]` ≅ `g(x1,g(...,g(xn,acc)...))`

11.1

```
fun foldr g acc [] = acc
  | foldr g acc (x::xs) = g(x,foldr g acc xs)

val sum = foldr (op +) 0
val prod = foldr (op * ) 1
val strConcat = foldr (op ^) ""
val listConcat = foldr (op @) []
```

```
foldr (op^) "!" ["H","E","L","L","O"]
 ⟹ "H"^(foldr (op^) "!" ["E","L","L","O"])
 ⟹ "H"^("E"^(foldr (op^) "!" ["L","L","O"]))
 ⟹ "H"^("E"^("L"^(foldr (op^) "!" ["L","O"])))
 ⟹ "H"^("E"^("L"^("L"^(foldr (op^) "!" ["O"]))))
 ⟹ "H"^("E"^("L"^("L"^("O"^(foldr (op^) "!" []))))
 ⟹ "H"^("E"^("L"^("L"^("O"^"!"))))
 ⟹ "HELLO!"
```

Analysis of `strConcat`

11.1

```
fun foldr g acc [] = acc
  | foldr g acc (x::xs) = g(x,foldr g acc xs)

val sum = foldr (op +) 0
val prod = foldr (op * ) 1
val strConcat = foldr (op ^) ""
val listConcat = foldr (op @) []
```

0 Notion of size: length of input list

1 Recurrence:

$$W_{\texttt{sc}}(0) = k_0$$
$$W_{\texttt{sc}}(n) = k_1 + W_{\texttt{sc}}(n-1)$$

2..4

5 $W_{\texttt{sc}}(n)$ is $O(n)$

Analysis of `listConcat`

11.1

```
fun foldr g acc [] = acc
  | foldr g acc (x::xs) = g(x,foldr g acc xs)

val sum = foldr (op +) 0
val prod = foldr (op * ) 1
val strConcat = foldr (op ^) ""
val listConcat = foldr (op @) []
```

0 Size of input: input contains n lists, each of length at most m

1 Recurrence:

$$\begin{aligned} W_{\texttt{lc}}(0, m) &= k_0 \\ W_{\texttt{lc}}(n, m) &= k_1 + W_{\texttt{lc}}(n-1, m) + k_3 m \end{aligned}$$

2..4

5 $W_{\texttt{lc}}(n, m)$ is $O(nm)$

Section 4

Combinators

Binary Operations

In mathematics and computer science, a **binary operation** is a function[1] (often written infixed) which takes two "things" of the same "kind" and "combines" them into another thing of that "kind".

Mathematical Examples:

- $+$ is a binary operation on complex numbers
- $\cup$ is a binary operation on sets
- $\times$ is a binary operation on 3-dimensional vectors

SML examples

- `div` is a (partial) binary operation on `int`s
- "Tupling" or "pairing" is a binary operation on expressions: if `e1` and `e2` are expressions, `(e1,e2)` is an expression
- Composition is a binary operation on functions

---

[1] Or function-like thing

Stick two functions together

`(op o) : ('b -> 'c) * ('a -> 'b) ->('a -> 'c)`
REQUIRES: true
ENSURES: `(g o f)` ≅ `h` such that `h(x)` ≅ `g(f(x))` for all suitably-typed `x`

11.2

```
infix o
fun (g o f) x = g(f(x))
(* OR: fun (g o f) = fn x => g(f(x)) *)

val collapse : int list -> string
    = concat o (map Int.toString)
```

Using o

11.3

```
fun zip([],_)=[]
  | zip(_,[])=[]
  | zip(x::xs,y::ys) = (x,y) :: zip(xs,ys)

val dotProd = (foldr op+ 0) o (map op* ) o zip

(*   (1*4) + (2*5) + (3*6)  *)
val 32 = dotProd([1,2,3],[4,5,6])
val 32 = dotProd([1,2,3],[4,5,6,7])
```

Some other binary ops

11.7

```
infix &&& ***
fun f &&& g = fn x => (f x, g x)
fun f *** g = fn (x,y) => (f x,g y)

fun listToString toStr L =
   "[" ^
   (String.concatWith "," (map toStr L)) ^
   "]"
val strAndLen =
  (listToString Int.toString) &&& List.length
val format =
  (fn (s,l) =>
    "The list " ^ s ^ " has length " ^ (Int.toString l)
  ) o strAndLen
```

Function Application Pipe

11.4

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

11.5

```
fun dotProd' (L1,L2) =

    (L1,L2)               (* int list * int list *)
  |>  zip                 (* (int * int) list    *)
  |>  map op*             (* int list            *)
  |>  foldr (op+) 0       (* int                 *)

val 32 = dotProd' ([1,2,3],[4,5,6])
```

Check for understanding: `mappartial`

11.6

```
fun isSome NONE = false
  | isSome _ = true

fun valOf NONE = raise Option
  | valOf (SOME x) = x

fun mappartial f L =
  L |> map f |> filter isSome |> 
    map valOf
```

Thank you!