Theory of Higher Order Functions

Higher-Order Totality, Staging, & Combinators

15-150 M21

Lecture 0623
23 June 2021

0623.0 (more-hofs.sml)

```
fun filter p [] = []
  | filter p (x::xs) =
       if p(x)
       then x::filter p xs
       else filter p xs
```

0623.1 (more-hofs.sml)

```
val isEven = fn x => x mod 2 = 0
```

0623.2 (more-hofs.sml)

```
val [] = filter isEven []
val [] = filter isEven [3,5,7]
val [2,4] = filter isEven [2,3,4]
val [] = filter (Fn.const false) ["a","b","c"]
val ["a","b","c"] =
    filter (Fn.const true) ["a","b","c"]
```

`mappartiali : (int * 'a -> 'b option) -> 'a list -> 'b list`

REQUIRES: `g(i,x)` is valuable for `i` $\geq$ `0`

ENSURES: `mappartiali g L` evaluates to the list of all those `z` such that `g(i,x)` $\Longrightarrow$ `SOME(z)`, where `i` is the index of `x` in `L`.

0623.4 (more-hofs.sml)

```
fun half (_,x) = if isEven x
                 then SOME(x div 2)
                 else NONE
fun convert (i,x) = if i<x
                 then SOME(Int.toString x)
                 else NONE
val [1,2,3] =
  mappartiali half [1,2,3,4,5,6,7]
val ["5","9"] =
  mappartiali convert [5,0,1,9,~6,4]
```

Live Coding:

`mappartiali`

0623.3 (more-hofs.sml)

```
fun mappartiali g [] = []
  | mappartiali g (x::xs) =
      let
        fun g' (i,x') = g(i+1,x')
      in
        (case g(0,x) of
          (SOME y) =>  y::mappartiali g' xs
        | _ => mappartiali g' xs)
      end
```

0 Evaluation and Equivalence of HOFs

HOFs are trivially total

Thm. 1 `map` is total

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

```
map f ⟹ fn [] => [] | x::xs => ...
```

□

Thm. 2 `filter` is total

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

```
filter p ⟹ fn [] => [] | x::xs => ...
```

□

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`

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

Theorem:

For all types `t1`, `t2` and all total values `f : t1 -> t2`,

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

1 Staging

What's the difference?

`square : int -> int`

REQUIRES: true

ENSURES: `square x` $\cong$ `x*x`, but it takes a really long time

0623.5 (staging.sml)

```
fun ex1 x y =
  let
    val xsq = square x
  in
    xsq + y
  end
```

0623.6 (staging.sml)

```
fun ex2 x =
  let
    val xsq = square x
  in
    fn y => xsq + y
  end
```

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
    )
```

2 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]` $\cong$ `g(x1,g(...,g(xn,acc)...))`

0623.7 (more-hofs.sml)

```
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 ^) ""
```

Resource: Origami

`https://www.cs.cmu.edu/~15150/resources/handouts/origami/origami.pdf`

`https://www.cs.cmu.edu/~15150/resources/handouts/origami/origami.sml`

```
foldl (op^) "!" ["H","E","L","L","O"]
⟹ foldl (op^) "H!" ["E","L","L","O"]
⟹ foldl (op^) "EH!" ["L","L","O"]
⟹ foldl (op^) "LEH!" ["L","O"]
⟹ foldl (op^) "LLEH!" ["O"]
⟹ foldl (op^) "OLLEH!" []
⟹ "OLLEH!"
```

```
(fn L => foldr (op^) "" L) ["H","E","L","L","O"]
⟹ 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"
```

strConcat Analysis

5-minute break

3 Combinators

In mathematics and computer science, a **binary operation** is a function* (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

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

REQUIRES: true

ENSURES: `(g o f)` $\cong$ `h` such that `h(x)` $\cong$ `g(f(x))`

0623.8 (combinators.sml)

```
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])
```

Addition is a binary operation on `int`s:

- Associativity:

$$\texttt{x + (y + z)} \cong \texttt{(x + y) + z}$$

- Identity:

$$\texttt{0 + x} \cong \texttt{x} \cong \texttt{x + 0}$$

Composition is a binary operation on functions (constrained by types)

- Associativity:

$$\texttt{h o (g o f)} \cong \texttt{(h o g) o f}$$

- Identity:

$$\texttt{Fn.id o f} \cong \texttt{f} \cong \texttt{f o Fn.id}$$

The algebraic study of the composition operation is the mathematical discipline of *category theory*.

```
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
```

0623.10 (combinators.sml)

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

0623.11 (combinators.sml)

```
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 Your Understanding

Verify that this implementation of `mappartiali` matches our earlier definition

0623.12 (combinators.sml)

```
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
fun mappartiali f L =
  L |> mapi f |> filter isSome |> map valOf
```

Summary

- We can write more complex versions of familiar HOFs
- With standard HOFs like `map` or `filter`, we're often interested in a higher-order notion of totality
- In some circumstances, we want to be careful about how the computation is staged in curried functions of several arguments
- We can analyze the runtime of HOFs just like with other functions, but often must make assumptions about the runtime of the functions given as arguments
- Functions have their own "algebra" of combinators

Next Time

- Tree search and options
- Tree balancing
- Tree sorting

Thank you!