15-150 Fall 2020

Lecture 9

Higher-order functions

Stephen Brookes

Transforming and combining data

*We focus first on **lists.***

*Ideas adapt to **trees**, etc.*

- Functions as values
- Higher-order functions
- The power of polymorphism

transforming data

- We often need to ***apply a function*** to all the items in a list.
- The built-in function map does this.
- It's ***polymorphic***
  (works uniformly…)

  map : ('a -> 'b) -> ('a list -> 'b list)

- And it's ***curried***…
  (so you can use *partial application*)

  map (**fn** x => x+1) : (int list -> int list)

map spec

map : ('a -> 'b) -> ('a list -> 'b list)

ENSURES

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

What this means...

For all $n \geq 0$, all types t_1 and t_2,
all functions f : t_1 -> t_2,
and all values x_1, ..., x_n : t_1,

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

(and this holds even if f isn't total!)

not what it means

map : ('a -> 'b) -> ('a list -> 'b list)

ENSURES

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

For all n≥0,

all functions f : 'a -> 'b,

and all values x_1, ..., x_n : 'a,

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

not what it means

map : ('a -> 'b) -> ('a list -> 'b list)

ENSURES

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

For all n≥0,
all functions f : 'a -> 'b,
and all values x_1, ..., x_n : 'a,
map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

Very few function values
have the type 'a -> 'b

not what it means

map : ('a -> 'b) -> ('a list -> 'b list)

ENSURES

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

For all n≥0,
all functions f : 'a -> 'b,
and all values x_1, ..., x_n : 'a,

map f [x_1, ..., x_n] = [f x_1, ..., f x_n]

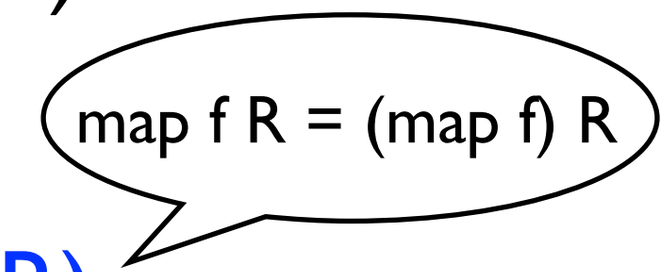
Very few function values
have the type 'a -> 'b

```
fun loop( ) = ( );
val f = (fn x => loop( ))
```

defining map

map : ('a -> 'b) -> ('a list -> 'b list)

```
fun map f [ ]      = [ ]
  |   map f (x::R) = (f x) :: (map f R)
```



correctness of map

Let f be a function.

Theorem

For all types t_1, and t_2,… *yada yada yada*

$$\text{map } f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

Proof

By induction on n.

Use the definition of map and the fact that when n>0, $[x_1, \ldots, x_n] = x_1 :: [x_2, \ldots, x_n]$.

totality

- If f : t_1 -> t_2 is total,
  so is (map f) : t_1 list -> t_2 list

- We often REQUIRE f to be total,
  to avoid dealing with non-termination

But map f [x, y] = [f x, f y] holds,
even if f or f x or f y doesn't terminate!

currying

For a function f with "multiple arguments"
there is a *corresponding* function F
of the "first" argument, that returns a
function of the "remaining" arguments…

f : int * int list -> bool list

curry ↓ ↑ *uncurry*

F : int -> (int list -> bool list)

corresponding, in that f (n, L) = (F n) L

terminology

- A *function with "multiple arguments"* is really a function with a *single* argument of a *tuple* type

$$f : t_1 * \ldots * t_k \text{ -> } t'$$

$$\text{curry}(f) : t_1 \text{ -> } (t_2 * \ldots * t_k \text{ -> } t')$$

- The "fully curried" version of f has type

$$t_1 \text{ -> } (t_2 \text{ -> } \ldots \text{ -> } (t_k \text{ -> } t')\ldots)$$

and ML abbreviates this as

$$t_1 \text{ -> } t_2 \text{ -> } \ldots \text{ -> } t_k \text{ -> } t'$$

why curry?

A curried function can be ***partially applied*** to a "first" argument, to get a specialized function of the "remaining" arguments

map : ('a -> 'b) -> ('a list -> 'b list)

```
- fun addtoeach x = map (fn y => x+y)
- addtoeach 42;
val it = fn - : int list -> int list
```

syntax

ML has a *streamlined* syntax for ***curried*** functions

```
fun map f [ ]  = [ ]
  |   map f (x::R) = (f x) :: map f R
```

is (arguably) more succinct than

```
fun map f =  fn  [ ] => [ ]
               |  (x::R) => (f x) :: map f R
```

Generalizes to *heavily curried* functions of "several" arguments

curried *vs.* **uncurried**

An ***uncurried*** version of map would look like this

map : ('a -> 'b) * 'a list -> 'b list

```
fun map (f, [ ])    = [ ]
  |   map (f, x::R) = (f x) :: map (f, R)
```

map cannot be used instead of map
… because the type is wrong!

map (**fn** x => 2*x) [1,2,3] = [2,4,6]

map (**fn** x => 2*x) [1,2,3] … *type error*

map **(fn** x => 2*x**,** [1,2,3]**)** = [2,4,6]

back to map

map : ('a -> 'b) -> ('a list -> 'b list)

- map is polymorphically typed
- Can be used at any *instance* of this type

map length : 'a list list -> int list

map length [[2,3],[4]] = [2, 1]

length : 'a list -> int

using map

prefs : 'a list -> 'a list list
ENSURES prefs L = a list of the
non-empty prefixes of L

prefs $[x_1, \ldots, x_n] = [[x_1], [x_1,x_2], \ldots, [x_1,\ldots,x_n]]$

prefs [ ] = [ ]

prefixes

characterized, inductively

[ ] has no (non-empty) prefixes

[x] is a prefix of x::R

x::P is a prefix of x::R
if P is a prefix of R

The (non-empty) prefixes of [1,2]
are [1] and [1,2].

prefs

```
fun prefs [ ] = [ ]
  |   prefs (x::R) = [x] :: map (fn P => x::P) (prefs R)
```

prefs $[x_1, \ldots, x_n] = [[x_1], [x_1,x_2], \ldots, [x_1,\ldots,x_n]]$

(Proof: induction on length of list.)

(For n>0, $[x_1, \ldots, x_n] = x_1 :: [x_2, \ldots, x_n]$)

exercise

```
fun preefs [ ] = [ [ ] ]
  |  preefs (x::R) = [x] :: map (fn P => x::P) (preefs R)
```

- This function looks very similar to prefs
- What is its type?
- What does it do? Prove it.

A small syntax change
can have a big effect

using map

sublists : 'a list -> 'a list list
ENSURES sublists L = a list of all *sublists* of L

ideas?

sublists

characterized, inductively

[ ] is the only sublist of [ ]

S is a sublist of x::R
if S is a sublist of R

x::S is a sublist of x::R
if S is a sublist of R

The sublists of [2,3] are [ ], [2], [3], and [2,3]

sublists

sublists : 'a list -> 'a list list
ENSURES sublists L = a list of all *sublists* of L

```
fun sublists [ ] = [ [ ] ]
  | sublists (x::R) =
      let
        val S = sublists R
      in
        S @ map (fn A => x::A) S
      end
```

sublists [2,3] = [[ ], [3], [2], [2,3]]

exercises

- Prove that for all suitably typed f and L_1, L_2

  map f (L_1@L_2) = (map f L_1) @ (map f L_2)

- Prove that for all suitably typed total functions f and lists L,

  length (map f L) = length L

  (note why we assume totality!)

- Prove that for all lists L,

  length (sublists L) = $2^{\text{length L}}$

be careful

```
fun sublists' [ ] =  [ ]
|     sublists' (x::R) =
        let
           val S = sublists' R
        in
           S @ map (fn A => x::A) S
        end
```

almost
the same as
sublists

- What is the type of this function?
- What does it do? Prove it.

sublists' [42] = ???

combining data

- Given a *collection* of data, in a ***list***
- We may want to ***combine*** the data,
  using a *binary operation* and a *base value*
- There are built-in functions for doing this…

We talk about ***lists***…
but there are
similar ways to deal with
trees, etc…

combining lists

Suppose we have a function

F : $t_1 * t_2 \rightarrow t_2$

and we want to combine the data in a list

$[x_1,\ldots,x_n]$: t_1 list

with z : t_2

to get (the value of)

$F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$: t_2

to calculate

$$F(x_1, F(x_2, ..., F(x_n, z)...))$$

Will need sequential evaluation

$$v_0 = z$$
$$v_1 = F(x_n, v_0)$$
$$v_2 = F(x_{n-1}, v_1)$$
$$\dots$$
$$v_n = F(x_1, v_{n-1})$$

v_n is the value of

$$F(x_1, F(x_2, ..., F(x_n, z)...))$$

examples

- ***add*** a list of integers
- ***multiply*** a list of reals
- ***least*** integer in a non-empty list
- ***flatten*** a list of lists into a single list

In each case,
combine a list of data
using a *binary operation* and a *base value*

a solution

A ***polymorphic*** function

foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

such that

For all types t_1, t_2, all $n \geq 0$, and all values

$F : t_1 * t_2 \text{ -> } t_2$, $[x_1,...,x_n] : t_1$ list, $z : t_2$,

foldr F z $[x_1,...,x_n]$ = $F(x_1, \ldots F(x_n, z)...)$

(combines from *right* to *left*)

why this type?

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

- Easy to *partially apply*, with a specific combining function, e.g.

```
foldr (op +) :  int -> int list -> int
```

and then supply a base value, e.g.

```
foldr (op +) 0 :  int list -> int
```

defining foldr

```
fun foldr F z [ ]     = z
|     foldr F z (x::L) = F(x, foldr F z L)
```

foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

REQUIRES true

ENSURES

$$\text{foldr } F\ z\ [x_1,\ldots,x_n] = F(x_1, \ldots F(x_n, z)\ldots)$$

NOTE: usually we assume F is total but the equation holds always

Use induction to prove correct

sum : int list -> int

ENSURES sum L = the sum of the integers in L

sum : int list -> int

ENSURES sum L = the sum of the integers in L

```
fun sum L = foldr (op +) 0 L
```

sum : int list -> int

ENSURES sum L = the sum of the integers in L

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

sum : int list -> int

ENSURES sum L = the sum of the integers in L

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

foldr (op +) 0 $[x_1,...,x_n]$ = $x_1 + (x_2 + ... (x_n + 0)...)$

sum : int list -> int

ENSURES sum L = the sum of the integers in L

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

foldr (op +) 0 $[x_1,\ldots,x_n]$ = $x_1 + (x_2 + \ldots (x_n + 0)\ldots)$

= $x_1 + x_2 + \ldots + x_n$

sum : int list -> int

ENSURES sum L = the sum of the integers in L

```
fun sum L = foldr (op +) 0 L

val sum = foldr (op +) 0
```

foldr (op +) 0 $[x_1,...,x_n]$ = $x_1 + (x_2 + ... (x_n + 0)...)$

= $x_1 + x_2 + ... + x_n$

foldr (op +) 42 $[x_1,...,x_n]$ = $x_1 + x_2 + ... + x_n + 42$

prod : real list -> real

ENSURES prod L = the product of the reals in L

```
fun prod L = foldr (op * ) 1.0 L

val prod = foldr (op * ) 1.0
```

foldr (op *) 1.0 $[x_1,...,x_n]$ = x_1 * (x_2 * ... (x_n * 1.0)...)

= x_1 * x_2 * ... * x_n

least : int list -> int

REQUIRES L is a non-empty list of integers

ENSURES least L = smallest element of L

```
fun least (x::R) = foldr Int.min x R
```

Warning: non-exhaustive patterns

least [2.4, 3.9, ~22.8] = ~22.8

Int.min : int * int -> int

flatten : 'a list list -> 'a list

```
fun flatten Ls = foldr (op @) [ ] Ls

val flatten = foldr (op @) [ ]
```

flatten $[L_1, \ldots, L_n] = L_1 \mathbin{@} (L_2 \mathbin{@} \ldots \mathbin{@} (L_n \mathbin{@} [\,]) \ldots)$

$= L_1 \mathbin{@} \ldots \mathbin{@} L_n$

flatten [[1,2], [ ], [3,4]] = [1,2,3,4]

Estimate the work to evaluate
flatten $[L_1, \ldots, L_n]$ when each L_i has length m

flatten analysis

- Let W(n, m) = work for flatten Ls
  when Ls is a list of n lists, each of length m
- flatten (L::Ls) = (op @)(L, flatten Ls)

W(0, m) = 1
W(n, m) = O(m) + W(n-1, m)
for n>0

work for L@-

W(n, m) is O(mn)

map and foldr

Can be used separately or together…

- map for transforming data
- foldr for combining data

Let ins : int * int list -> int list
be as defined earlier.

foldr ins [ ] : int list -> int list

is equivalent to *insertion sort*

map and foldr

Can be used separately or together…

- map for transforming data
- foldr for combining data

```
val sum = foldr (op +) 0

fun count L = sum (map sum L)
```

map and foldr

Can be used separately or together…

- map for transforming data
- foldr for combining data

```
val sum = foldr (op +) 0

fun count L = sum (map sum L)
```

Exercise: Find an optimal group size for covid testing.

A group size n such that cost(N, n, p) is smallest.

covid testing

The Detection of Defective Members of Large Populations
Robert Dorfman
The Annals of Mathematical Statistics, Vol. 14, No. 4

```
fun better ((i, m1:int), (j, m2:int)) = case Int.compare(m1, m2) of
                                          LESS    => (i, m1)
                                        | EQUAL   => (Int.min(i, j), m1)
                                        | GREATER => (j, m2)

val (first :: rest) = (map (fn i => (i, cost(4000,i,1))) (upto 1 100))

val optimal = foldr better first rest;

val optimal = (11,782) : int * int
```

cost(N, n, p) =
#tests needed for
population N
with group size n,
& prevalence p

Understand
why this works!

covid testing

The Detection of Defective Members of Large Populations
Robert Dorfman
The Annals of Mathematical Statistics, Vol. 14, No. 4

```
fun better ((i, m1:int), (j, m2:int)) = case Int.compare(m1, m2) of
                                          LESS    => (i, m1)
                                        | EQUAL   => (Int.min(i, j), m1)
                                        | GREATER => (j, m2)

val (first :: rest) = (map (fn i => (i, cost(4000,i,1))) (upto 1 100))

val optimal = foldr better first rest;

val optimal = (11,782) : int * int
```

cost(N, n, p) =
#tests needed for
population N
with group size n,
& prevalence p

For N = 4000, p = 1%,
an optimal group size
is n = 11
and this would require
782 tests

Understand
why this works!

foldr and @

- For all suitably typed g, z, L_1 and L_2

  foldr g z (L_1@L_2)

  = foldr g (foldr g z L_2) L_1

Proof: induction on length of L_1

NOTE how this shows
the *combination order* used by foldr

sketch

$$\text{foldr } g\ z\ ([\,]@L_2) = \text{foldr } g\ z\ L_2$$

because $[\,]@L_2 = L_2$

$$= \text{foldr } g\ (\text{foldr } g\ z\ [\,])\ L_2$$

by def of foldr

$$\text{foldr } g\ z\ ((x::L)@L_2) = \text{foldr } g\ z\ (x::(L@L_2))$$

because $(x::L)@L_2 = x::(L@L_2)$

$$= g(x, \text{foldr } g\ z\ (L@L_2))$$

by def of foldr

$$= g(x, \text{foldr } g\ (\text{foldr } g\ z\ L_2)\ L)$$

by IH for L

$$= \text{foldr } g\ (\text{foldr } g\ z\ L_2)(x::L)$$

by def of foldr

another way to fold

- A ***polymorphic*** function

foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

such that

for all types t_1, t_2,
all $n \geq 0$, and all values
$F: t_1 * t_2 \rightarrow t_2$, $[x_1,...,x_n] : t_1$ list, $z : t_2$,

foldl F z $[x_1,...,x_n]$ = $F(x_n, F(x_{n-1}, ..., F(x_1, z)...))$

(combines from *left* to *right*)

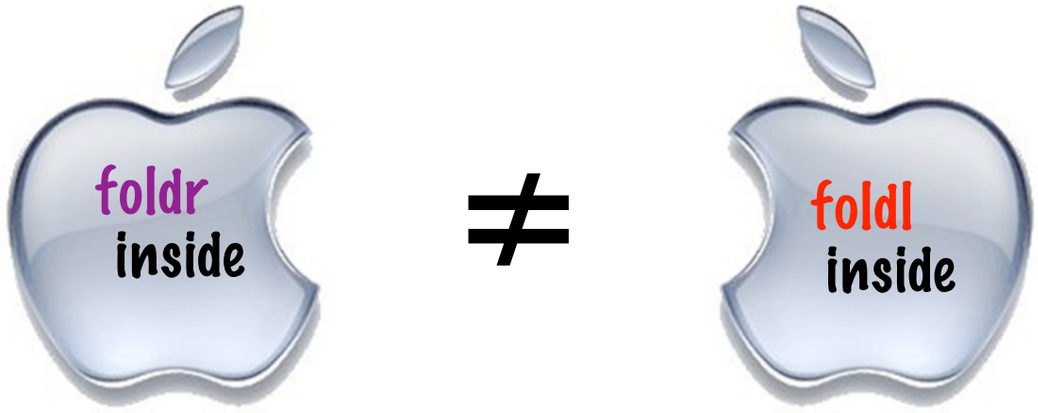
foldl

```
fun foldl F z [ ] = z
  |  foldl F z (x::L) = foldl F (F(x, z)) L
```

foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

foldl F z $[x_1,...,x_n]$ = $F(x_n, F(x_{n-1},..., F(x_1,z)...))$

(combines from *left* to *right*)



foldr (op @) [ ] [[1,2], [ ], [3,4]] = [1,2,3,4]

foldl (op @) [ ] [[1,2], [ ], [3,4]] = [3,4,1,2]

In general,
when is
foldr g = foldl g ?

We'll return to this question later.

foldl and @

- For all suitably typed g, z, L_1 and L_2

foldl g z (L_1@L_2)

= foldl g (foldl g z L_1) L_2

Proof: by induction on length of L_1

NOTE how this tells us
the *combination order* used by foldl

Contrast with

foldr g z (L_1@L_2)

= foldr g (foldr g z L_2) L_1

foldr *vs* foldl

For all g, z and L

foldr g z L = foldl g z (rev L)

Proof: by induction on length of L

(or use fold/append properties)

folds and invariance

Let $g : t_1 * t_2 \to t_2$ and $p : t_2 \to bool$
be total functions

- Say g ***preserves*** p if for all $z : t_2$ and $x : t_1$,
  $p(z)$ ***implies*** $p(g(x,z))$

Invariance Theorem
If g ***preserves*** p,
then for all $z : t_2$ and $L : t_1$ list,
$p(z)$ ***implies*** $p(foldr\ g\ z\ L)$

(also for foldl)

example

- ins : int * int list -> int list
  preserves sorted : int list -> bool
- So foldr ins [ ] L = a sorted list

(this will be *useful*, so remember)

summary

- Higher-order functions

```
map : ('a -> 'b) -> 'a list -> 'b list
foldr, foldl : ('a * 'b -> 'b) -> 'b ->  'a list -> 'b
```

- Polymorphic types imply ***versatility***
- Useful for many purposes

```
map (fn x => x+2)
foldr (fn (x, y) => x+y) 0
```

and so on

- Similarly for *trees*

map : ('a -> 'b) -> 'a tree -> 'b tree

```
fun map f Empty = Empty
|     map f (Node(A, x, B)) = Node(map f A, (f x), map f B)
```

foldr, foldl : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b

```
fun foldr g z Empty = z
|      foldr g z (Node(A, x, B)) = ???
```

reduce : ('a * 'a -> 'a) -> 'a -> 'a tree -> 'a

exploration

Try defining some functions for tree folding.

There are several ways to do it.

- In particular, define
  treefoldr : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b
  to correlates with foldr & inorder traversal, in that

  treefoldr g z T = foldr g z (inord T)

Define treefoldr using *structural induction* on *trees*!

Don't use foldr or inord!

reduce : ('a * 'a -> 'a) -> 'a -> 'a tree -> 'a

```
fun reduce g z Empty = z
 |     reduce g z (Node(A, x, B)) =
    let
       val (a, b) = (reduce g z A, reduce g z B)
    in
       g(x, g(a, b))
    end
```

- Why not reduce : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b ?
- What should we REQUIRE of g and z
  to ENSURE that reduce g z T does something sensible?