15-150 Fall 2018

Lecture 10

Stephen Brookes
Transforming and combining data

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

Map-reduce is a data processing paradigm for condensing large data into aggregated results. Implemented at Google in 2004; based on ideas from functional programming.

A framework for large-scale parallel processing

We focus on lists. Ideas adapt to trees, etc.
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…)

  \[
  \text{map : } ('a \rightarrow 'b) \rightarrow ('a \text{ list} \rightarrow 'b \text{ list})
  \]

• And it’s **curried**…

  (so you can use **partial application**)

  \[
  \text{map (fn } x \rightarrow x+1) : (\text{int list} \rightarrow \text{int list})
  \]
map spec

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

ENSURES

map f [x₁, ..., xₙ] = [f x₁, ..., f xₙ]

For all n ≥ 0, all types t₁ and t₂, all total functions f : t₁ -> t₂, and all values x₁, ..., xₙ : t₁,

map f [x₁, ..., xₙ] = [f x₁, ..., f xₙ]
defining map

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

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

can also be defined as

fun map f = fn L =>
  case L of
    [ ] => [ ]
  | x::R => (f x) :: (map f R)
correctness of map

Let $f$ be a total function.

**Theorem**

For all $n \geq 0$, and all $x_1, \ldots, x_n$

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

**Proof**

By induction on $n$.

Use the definition of map and the fact that when $n > 0$, $[x_1, \ldots, x_n] = x_1 :: [x_2, \ldots, x_n]$. 
For a function $f$ with “multiple arguments” there is an equivalent function $F$ of a single argument, that returns a function of the “remaining” arguments.

$$f : \text{int} \times \text{int list} \rightarrow \text{bool list}$$

$$F : \text{int} \rightarrow (\text{int list} \rightarrow \text{bool list})$$

In that

$$f (n, L) = (F n) L$$
A function with “multiple arguments” is a function of type $t_1 \times \ldots \times t_k \rightarrow t'$

Really, this is a function with a single argument of a tuple type

The fully curried version of this function would have type $t_1 \rightarrow \ldots \rightarrow t_k \rightarrow t'$
why bother?

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

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

is more succinct than

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

Generalizes to *heavily curried* functions of “several” arguments
ML has a *streamlined* syntax for *curried* functions

```plaintext
fun merge [ ] R = R
|  merge L [ ] = L
|  merge (x::L) (y::R) = …
```

is more succinct than

```plaintext
fun merge xs = fn ys =>
  case (xs, ys) of
    ([ ], R) => R
| (L, [ ]) => L
| (x::L, y::R) => …
```
curried vs. uncurried

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

```plaintext
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!

```plaintext
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]
```
examples

**add : int -> int -> int**

```ml
fun add x y = x+y
```

```ml
fun add x = fn y => x+y
```

```ml
val add = fn x => (fn y => x+y)
```

**plus : int * int -> int**

```ml
fun plus (x, y) = x+y
```

```ml
val plus = fn (x, y) => x+y
```
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 prefixes of [1,2] are [1], and [1,2]
fun prefs [ ] = [ ]
  |  prefs (x::R) = [x] :: map (fn P => x::P) (prefs R)

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

prefs [x₁, ..., xₙ] = [[x₁], [x₁,x₂], ..., [x₁,...,xₙ]]
exercise

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

• This function looks very similar to prefs
• What is its type?
• What does it do?

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
          map (fn A => x::A) S @ S
      end

sublists [2,3]  = [[2,3], [2], [3], [ ]]
sublists [1,2,3] = [[1,2,3], [1,2], [1,3], [1], [2,3], [2], [3], [ ]]
exercises

• Prove that for all suitably typed $f$ and $L_1, L_2$
  $$\text{map } f \ (L_1 \@ L_2) = (\text{map } f \ L_1) \@ (\text{map } f \ L_2)$$

• Prove that for all suitably typed total functions $f$ and lists $L$,
  $$\text{length } (\text{map } f \ L) = \text{length } L$$

• Prove that for all lists $L$,
  $$\text{length } (\text{sublists } L) = 2^{\text{length } L}$$
fun sublists' [ ] = [ ]

let
  val S = sublists' R
in
  map (fn A => x::A) S @ S
end

• What is the type of this function?
• What does it do?

sublists' [42] = ???

almost the same as sublists
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…
Suppose we have a function
\[ F : t_1 \times t_2 \rightarrow t_2 \]
and we want to combine the data in list
\[ [x_1, \ldots, x_n] : t_1 \text{ list} \]
with \[ z : t_2 \]
to get (the value of)
\[ F(x_1, F(x_2, \ldots, F(x_n, z)\ldots)) : t_2 \]
examples

• **add** a list of integers
• **multiply** a list of reals
• **largest** 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

\[
\text{foldr} : (\text{a} \times \text{b} \to \text{b}) \to \text{b} \to \text{a list} \to \text{b}
\]

such that

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

\( F : t_1 \times t_2 \to t_2, \ [x_1, \ldots, x_n] : t_1 \ \text{list}, \ z : t_2 \),

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

(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
foldr F z [x₁,...,xₙ] = F(x₁, ...F(xₙ, z)...)
**foldr**

\[\textbf{fun} \quad \text{foldr} \ F \ z \ [\ ] \ = \ z \]
\[\mid \quad \text{foldr} \ F \ z \ (x::L) \ = \ F(x, \text{foldr} \ F \ z \ L)\]

For all \( n \geq 0 \), for all values \( F, z, x_1, \ldots, x_n \)

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

Proof: use induction on \( n \)
base case

\[
\textbf{fun}\ \text{foldr}\ F\ z\ [\ ]\ =\ z \\
|\ \text{foldr}\ F\ z\ (x::L)\ =\ F(x,\ \text{foldr}\ F\ z\ L)
\]

For n=0, we must show that for all values F and z,

\[
\text{foldr}\ F\ z\ [\ ]\ =\ z
\]

This is immediate, by definition of foldr
inductive step

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

Let n>0 and assume result for n-1, i.e.

For all values $F, z, y_1, \ldots, y_{n-1}$
\[
\text{foldr } F \ z \ [y_1, \ldots, y_{n-1}] = F(y_1, \ldots, F(y_{n-1}, z)\ldots)
\]

Let $F, z, x_1, \ldots, x_n$ be values…

Must show that
\[
\text{foldr } F \ z \ [x_1, \ldots, x_n] = F(x_1, \ldots, F(x_n, z)\ldots)
\]
fun foldr F z [ ] = z
| foldr F z (x::L) = F(x, foldr F z L)

[x₁,...,xₙ] = x₁ :: [x₂,...,xₙ]

foldr F z [x₁,...,xₙ] = F(x₁, foldr F z [x₂,...,xₙ]) by def foldr
= F(x₁, F(x₂,..., F(xₙ, z)…)) by IH
= F(x₁,... F(xₙ, z)...)

(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 [x₁,...,xₙ] = x₁ + (x₂ + ... (xₙ + 0)...) = x₁ + x₂ + ... + xₙ

foldr (op +) 42 [x₁,...,xₙ] = x₁ + x₂ + ... + xₙ + 42
\textbf{prod} : real list -> real

ENSURES \( \text{prod} \ L = \text{the product of the reals in } L \)

\begin{align*}
\textbf{fun} \ \text{prod} \ L &= \text{foldr} \ (\text{op} \ * \ ) \ 1.0 \ L \\
\textbf{val} \ \text{prod} &= \text{foldr} \ (\text{op} \ * \ ) \ 1.0
\end{align*}

\[ \text{foldr} \ (\text{op} \ * \ ) \ 1.0 \ [x_1, \ldots, x_n] = x_1 \ast (x_2 \ast \ldots \ast (x_n \ast 1.0)\ldots) \]
\[ = x_1 \ast x_2 \ast \ldots \ast x_n \]
largest : real list -> real

REQUIRES  L is a non-empty list of reals
ENSURES  largest L = largest element of L

fun largest (x::R) = foldr Real.max x R

Warning: non-exhaustive patterns

largest [2.4, 3.9, ~22.8] = 3.9

Real.max : real * real -> real
\textbf{flatten} : \texttt{’a list list} -> \texttt{’a list}

\texttt{fun flatten Ls = foldr (op @) [ ] Ls}

\texttt{val flatten = foldr (op @) [ ]}

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

\[
= L_1 @ \ldots @ L_n
\]

\[
\text{flatten } [[1,2], [ ], [3,4]] = [1,2,3,4]
\]

\begin{itemize}
\item Estimate the work to evaluate \texttt{flatten } [L_1,\ldots,L_n] \texttt{ when each } L_i \texttt{ has length } m
\end{itemize}
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*
meme time

foldr (op ^) "are belong to us"
["all ","your ","base "]

=  "all your base are belong to us"
foldr and @

- For all suitably typed g, z, L₁ and L₂

\[
\text{foldr } g \; z \; (L₁ \circ L₂) \\
= \text{foldr } g \; (\text{foldr } g \; z \; L₂) \; L₁
\]

NOTE how this shows the combination order used by foldr

Proof: induction on length of L₁
map/foldr fusion

- For all suitably typed f, g, z and L

\[
\text{foldr } g \ z \ (\text{map } f \ L) \\
= \text{foldr } (\text{fn } (x, y) \Rightarrow g(f \ x, y)) \ z \ L
\]
map/foldr fusion

\[(x_1, y_1), \ldots, (x_n, y_n)\] → map (op \* ) → foldr (op + ) 0 → \[x_1 y_1, \ldots, x_n y_n\] → \[x_1 y_1 + \ldots + x_n y_n\] → foldr (fn ((x,y),u) => x*y + u) 0 → \[x_1 y_1 + \ldots + x_n y_n\] → [x_1, y_1), \ldots, (x_n, y_n)\]