**Transforming and combining data**

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

*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…)
  \[ \text{map : ('a -> 'b) -> ('a list -> 'b list)} \]
• And it’s curried…
  (so you can use partial application)
  \[ \text{map (fn x => x+1) : (int list -> int list)} \]
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 a corresponding function $F$ of a single argument, that returns a function of the “remaining” arguments…

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

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

$f (n, L) = (F n) L$
• 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.

```plaintext
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.

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

is more succinct than

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

Generalizes to heavily curried functions of “several” arguments.
more syntax

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

```ml
fun merge [ ] R = R
| merge L [ ] = L
| merge (x::L) (y::R) = ...
```

is more succinct than

```ml
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

fun add (x:int) (y:int) = x+y

fun add (x:int) = fn (y:int) => x+y

val add = fn (x:int) => (fn (y:int) => x+y)

plus : int * int -> int

fun plus (x:int, y:int) = x+y

val plus = fn (x:int, y:int) => x+y
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, …, x_n] = [[x_1], [x_1,x_2], …, [x_1,…,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].
fun prefs [ ] = [ ]

\[
\begin{align*}
\text{pref} (x::R) &= [x] :: \text{map (fn P => x::P)} (\text{pref} R)
\end{align*}
\]

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

prefs [x_1, ..., x_n] = [[x_1], [x_1,x_2], ..., [x_1,...,x_n]]

(Proof: induction on length of list.)
(For n>0, [x_1, ..., x_n] = x_1 :: [x_2, ..., 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
sublists

characterized, inductively

\[
\begin{align*}
[ ] & \text{ is (the only) sublist of } [ ] \\
S & \text{ is a sublist of } x::R \\
& \quad \text{ if } S \text{ is a sublist of } R \\
x::S & \text{ is a sublist of } x::R \\
& \quad \text{ if } S \text{ is a sublist of } R
\end{align*}
\]

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' [ ] = [ ]
  | sublists' (x::R) =
  | 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? Prove it.

sublists' [42] = ???

be careful
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

\( F : t_1 \times t_2 \rightarrow t_2 \)

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)

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

\( v_0 = z \quad v_1 = F(x_n, v_0) \quad v_2 = F(x_{n-1}, v_1), \ldots \)
to calculate

\[ F(x_1, F(x_2, ..., F(x_n, z)\ldots)) \]

Will need sequential evaluation

\[
\begin{align*}
    v_0 &= z \\
    v_1 &= F(x_n, v_0) \\
    v_2 &= F(x_{n-1}, v_1) \\
    & \ldots \\
    v_n &= F(x_1, v_{n-1})
\end{align*}
\]

\( v_n \) is the value of \( F(x_1, F(x_2, ..., F(x_n, z)\ldots)) \)
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

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

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

(combines from *right to left*)
why this type?

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

- Easy to \textit{partially apply}, with a specific combining function, e.g.
  \[
  \text{foldr (op +)} : \text{int} \to \text{int list} \to \text{int}
  \]
  and then supply a base value, e.g.
  \[
  \text{foldr (op +)} 0 : \text{int list} \to \text{int}
  \]
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 [x_1,...,x_n] = F(x_1, ...F(x_n, z)...)
foldr

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

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

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

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

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

```haskell
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

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

```ml
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
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 [x₁,...,xₙ] = x₁ * (x₂ * ... (xₙ * 1.0)...)

= x₁ * x₂ * ... * xₙ
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
**flatten** : 'a list list -> 'a list

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

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

= \(L_1 @ \ldots @ L_n\)

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 \(m\).
map and foldr

Can be used separately or together…

- **map** for transforming data
- **foldr** for combining data

Let \( \text{ins} : \text{int} \times \text{int list} \rightarrow \text{int list} \) be as defined earlier.

\[
\text{foldr \ ins \ [ ]} : \text{int list} \rightarrow \text{int list}
\]

is equivalent to *insertion sort*
meme time

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

= "all your base are belong to us"
• For all suitably typed \( g, z, L_1 \) and \( L_2 \)

\[
\text{foldr } g \ z \ (L_1 @ L_2) = \text{foldr } g \ (\text{foldr } g \ z \ L_2) \ L_1
\]

NOTE how this shows the combination order used by \text{foldr}

Proof: induction on length of \( L_1 \)
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

\[
[(x_1, y_1), \ldots, (x_n, y_n)] \\
\downarrow \text{map (op * )} \\
[x_1 \times y_1, \ldots, x_n \times y_n] \\
\downarrow \text{foldr (op + ) 0} \\
x_1 \times y_1 + \ldots + x_n \times y_n
\]

\[
[(x_1, y_1), \ldots, (x_n, y_n)] \\
\downarrow \text{foldr (fn ((x,y),u) => x*y + u) 0} \\
x_1 \times y_1 + \ldots + x_n \times y_n
\]