15-150 Fall 2020
Lecture 11

Using higher-order functions

Stephen Brookes

Midterm 1 exam
Tuesday October 13

last week

- higher-order functions
- type inference rules

today

- using the type rules: map and foldr
- some important properties
- using higher-order functions

type rules

There's a rule for each program construct

- Pattern p matches type t,
  and produces bindings $x_1 : t_1, \ldots, x_k : t_k$

  x::L matches int list, produces x : int, L : int list

- Expression e is well-typed, has type t

  if L : 'a list, L@L has type 'a list

- Declaration d is well-typed,
  and produces bindings $x_1 : t_1, \ldots, x_k : t_k$

  fun twice L = L@L introduces twice : 'a list -> 'a list

type rules

Each rule is syntax-directed

- For curried recursive functions with k clauses…

fun F p_1 q_1 = e_1 | F p_2 q_2 = e_2

declares F : t_1 -> t_2 -> t

the curried fun rule (k=2)

each clause must *fit*

iff, for each clause F p_i q_i = e_i

p_i matches t_1, q_i matches t_2, and they give type bindings such that, assuming F : t_1 -> t_2 -> t gives e_i : t

type of map

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

using the curried fun rule (k=2)

To show map : ('a -> 'b) -> 'a list -> 'b list
need to check that each clause *fits* this type

- First clause: LHS patterns match types ('a -> 'b), 'a list, and bind f : ('a -> 'b). RHS expression [ ] has type 'b list ✓
- Second clause: LHS patterns match types ('a -> 'b), 'a list, and bind f : ('a -> 'b), x : 'a, L : 'a list. Using these bindings, RHS expression (f x) :: map f L has type 'b list ✓

map f L is (map f) L

type of foldr

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

using the curried fun rule (k=3)

To show foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
need to check that each clause *fits* this type

- First clause: LHS patterns match types ('a * 'b -> 'b), 'b, 'a list
  and bind g : ('a * 'b -> 'b), z : 'b.
  Using these bindings, the RHS expression z has type 'b ✓

- Second: LHS patterns match types ('a * 'b -> 'b), 'b, 'a list
  and bind g : ('a * 'b -> 'b), z : 'b, x : 'a, L : 'a list.
  With these bindings, the RHS expression g(x, foldr g z L)
  has type 'b ✓

foldr g z L is ((foldr g) z) L

comments

To fill in the details behind these type derivations you use the typing rules

- Given f : ('a -> 'b), L : 'a list, and
  map : ('a -> 'b) -> 'a list -> 'b list

  we get
  (i) map f : 'a list -> 'b list by *application* rule, given types for f, map
  (ii) (map f) L : 'b list by *application* rule, given type for L

Get used to obeying the rules as you design code, and you'll have fewer type errors!

Even using the rules ***informally*** *can help… they embody* ***principles*** *of good code design*

next

- Using maps and folds

  - some useful properties

totality

- When f is total, so is map f
- When g is total and z is valuable, foldl g z and foldr g z are total

(useful to know)

(easy to prove…
using induction on lists)

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

check that LHS, RHS have type t_3 list

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

check that LHS, RHS have type t_3 list

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

(map g) ∘ (map f) = map (g ∘ f)

avoids building an extra list

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

(map g) ∘ (map f) = map (g ∘ f)

infix ∘ is ***function composition***

map/map fusion

Let $f : t_1 \rightarrow t_2$, $g : t_2 \rightarrow t_3$ be total functions

For all values $L : t_1$ list,

```
map g (map f L)
    = map (fn x => g (f x)) L
```

avoids building an extra list

(map g) ∘ (map f) = map (g ∘ f)

infix ∘ is ***function composition***

g ∘ f = **fn** x => g (f x)

map/foldr fusion

Let f : t_1 -> t_2, g : t_2 * t_3 -> t_3 be total functions

and z : t_3, L : t_1 list be values. Then

```
foldr g z (map f L)
        = foldr (fn (x, y) => g(f x, y)) z L
```

a map followed by a fold
can be replaced by
a single fold

avoids building an extra list

proof sketch

Assume f, g total, z and L suitably typed values.

Let h be **fn** (x, y) => g(f x, y).

Prove by structural induction on L that

foldr g z (map f L) = foldr h z L.

- Base case: for L = [ ] EASY (both = z)
- Ind step: let L = x::R and assume as IH that
  foldr g z (map f R) = foldr h z R.
  Show (using defs of map, foldr and IH) that

  foldr g z (map f (x::R)) = foldr h z (x::R)

… *see next slide for more detail*

proof sketch

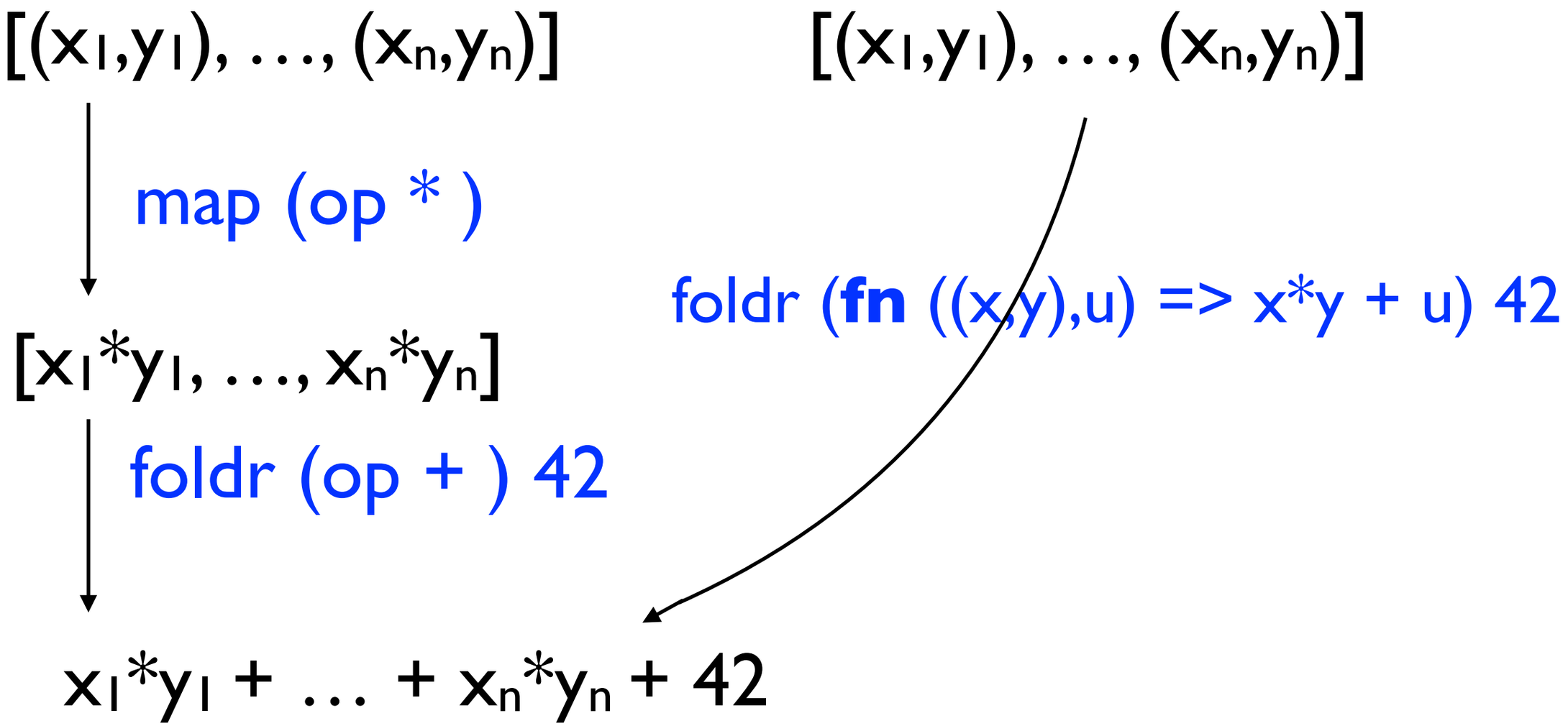
By assumptions on f and g
it follows that
map f, foldr g z and h
are all total…

… so we can use
equational reasoning
based on the definitions of
map and foldr and h

```
foldr g z (map f (x::R))
= foldr g z (f x :: map f R)          def of map
= g(f x, foldr g z (map f R))         def of foldr*
= h(x, foldr g z (map f R))           def of h
= h(x, foldr h z R)                   IH for R
= foldr h z (x::R)                    def of foldr
```

*The foldr equation
foldr g z (y::ys) = g(y, foldr g z ys)
holds for all values g, z, y and ys.
It also holds when y and ys are valuable,
e.g. y = f x and ys = map f R.

a map/foldr example



foldr *vs* foldl

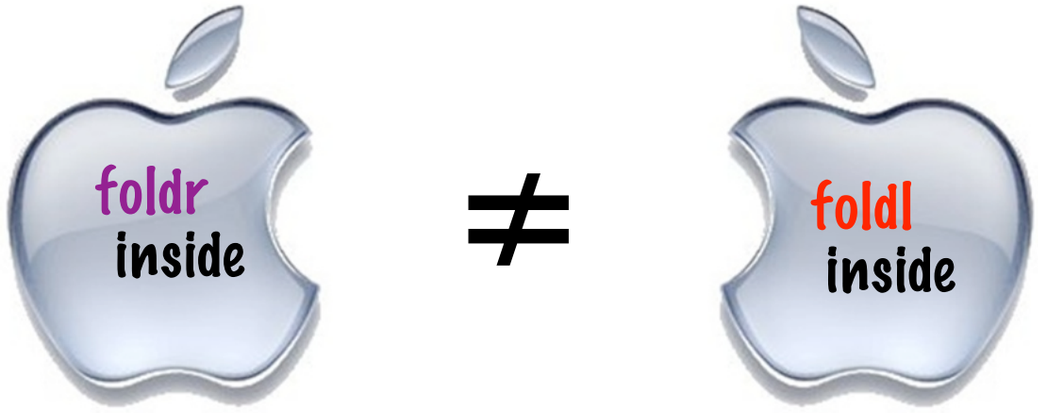
```
fun foldr g z [ ] = z
 |     foldr g z (x::L) = g(x, foldr g z L)

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

For all suitably typed values g, z and L

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

Proof: by induction on length of L



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 ?

when does foldr g = foldl g?

- Clearly foldr g z [ ] = foldl g z [ ] = z
- Also foldr g z [x] = foldl g z [x] = g(x, z)
- To get foldr g z [x, y] = foldl g z [x, y]
  we need g to satisfy

$$g(x, g(y, z)) = g(y, g(x, z))$$

Turns out, this is enough!

foldr g *vs* foldl g

Let g be a function such that
for all (suitably typed) values x, y and u

$$g(x, g(y, u)) = g(y, g(x, u)).$$

Then for all L and z

$$\text{foldr } g \; z \; L = \text{foldl } g \; z \; L$$

foldr g *vs* foldl g

Let g be a function such that
for all (suitably typed) values x, y and u

g(x, g(y, u)) = g(y, g(x, u).

Then for all L and z

foldr g z L = foldl g z L

(op +) and (op *) have this property
(op @) and (op ^) do not

foldr g *vs* foldl g

Let g be a function such that
for all (suitably typed) values x, y and u

$$g(x, g(y, u)) = g(y, g(x, u)).$$

Then for all L and z

$$\text{foldr } g \; z \; L = \text{foldl } g \; z \; L$$

foldr g *vs* foldl g

Let g be a function such that
for all (suitably typed) values x, y and u

$$g(x, g(y, u)) = g(y, g(x, u)).$$

Then for all L and z

$$\text{foldr } g \; z \; L = \text{foldl } g \; z \; L$$

foldr g = foldl g

proof sketch

Assume $g(x, g(y, u)) = g(y, g(x, u))$ for all x, y, u

- Let L be $[y_1, \ldots, y_n]$, and define sequences

$u_0 = g(x, z)$ $\quad$ $v_0 = z$

$u_1 = g(y_1, u_0)$ $\quad$ $v_1 = g(y_1, v_0)$

… $\quad$ …

$u_n = g(y_n, u_{n-1})$ $\quad$ $v_n = g(y_n, v_{n-1})$

Prove that for $0 \le i \le n$, $u_i = g(x, v_i)$.

- By def, u_n = foldl g z (x::L), v_n = foldl g z L
  so we get foldl g z (x::L) = g(x, foldl g z L).

- Use this result to show foldl g z = foldr g z

details

- That was only a sketch!
- First use induction on i
- Then use induction on R to show that (assuming g has the given property) for all lists R and values z,
  foldl g z R = foldr g z R

"foldl g z = foldr g z"

comments

- We just showed that foldr g = foldl g if g : t_1 * t_2 -> t_2 satisfies

$$g(x, g(y, u)) = g(y, g(x, u))$$

for all values x, y : t_1 and u : t_2

- Special case: this holds when t_2 = t_1 and g is ***associative*** and ***commutative***

$$\begin{aligned} g(x, g(y, u)) &= g(g(x, y),u) && (g \textit{ associative}) \\ &= g(g(y, x),u) && (g \textit{ commutative}) \\ &= g(y, g(x, u)) && (g \textit{ associative}) \end{aligned}$$

more comments

- foldr g z = foldl g z
  if g : t_1 * t_2 -> t_2 satisfies

  g(x, g(y, z)) = g(y, g(x, z))

  for all values x, y : t_1

- Special case: this holds when t_2 = t_1 and
  g is *commutative, and* z *is an* ***identity*** *for* g

g(x, g(y, z)) = g(x, y) *(identity:* g(y, z)=y*)*

= g(y, x) *(commutative)*

= g(y, g(x, z)) *(identity:* g(x, z)=x*)*

next

polymorphic sorting

- A **case study**
- Developing an *abstract* and *general* solution to a *family* of problems
- The benefits of ***polymorphic types*** and ***higher-order functions***
- Advantages of tasteful ***currying***

examples

type	*comparison*	*examples*
int	*usual*	3 < 4
int * int	*usual lexicographic*	(2,4) < (3,2) < (3,3)
string	*dictionary*	“car” < “card” < “kardashian”

obviously in ML
we can’t use
the same symbol <
for all of these!

WIKIPEDIA

A **sorting algorithm** ... puts elements of a list in a certain **order**.
The most-used **orders** are numerical ... and lexicographical.

Efficient sorting is important …

for optimizing search and merge algorithms

and for producing human-readable output.

Q: What's an ***order***?

What properties do we need?

comparisons

A ***comparison*** for type t is a *total* function

cmp : t * t -> order

such that

cmp(x,y) = LESS iff cmp(y,x) = GREATER

cmp(x,y) = EQUAL iff cmp(y,x) = EQUAL

and

cmp(x,y)=LESS & cmp(y,z) <> GREATER implies cmp(x,z)=LESS
cmp(x,y)=GREATER & cmp(y,z) <> LESS implies cmp(x,z)=GREATER
cmp(x,y)=EQUAL & cmp(y,z)=EQUAL implies cmp(x,z)=EQUAL

"the obvious properties"

integers

Int.compare : int * int -> order

For all values x,y:int

Int.compare(x,y) = LESS if x<y

= EQUAL if x=y

= GREATER if x>y

This is a comparison function!

Int.compare

Int.compare : int * int -> order

is a total function, and

x<y iff y>x

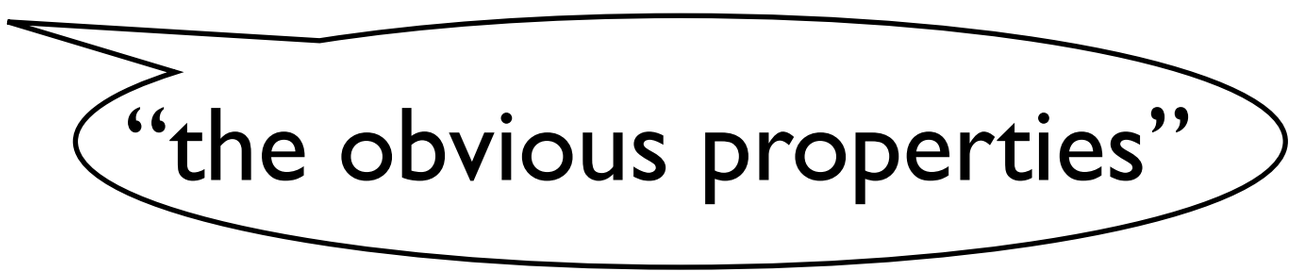
x=y iff y=x

and

x<y & y≤z implies x<z

x>y & y≥z implies x>z

x=y & y=z implies x=z



comparisons

- There are MANY
- Not just the "usual" one on int
- We can *define* specific ones
- And we can define functions for *building* new comparisons from old, e.g. *lexicographic* on pairs, on lists, …

pairs of integers

lexcompare : (int * int) * (int * int) -> order

```
fun lexcompare((x1, y1), (x2, y2)) =
      case Int.compare(x1, x2) of
           LESS => LESS
         | GREATER => GREATER
         |     EQUAL => Int.compare(y1, y2)
```

This is a comparison function

flipping

flip: ('a * 'a -> order) -> ('a * 'a -> order)

```
fun flip cmp (x, y) = cmp (y, x)
```

(flip Int.compare) (2,3) = GREATER

If cmp is a comparison,
so is flip(cmp).

pairing comparisons

lex : ('a * 'a -> order) * ('b * 'b -> order) -> (('a * 'b) * ('a * 'b) -> order)

```
fun lex (cmp1, cmp2) ((x1, y1), (x2, y2)) =
    case cmp1(x1, x2) of
      LESS    => LESS
    | GREATER => GREATER
    | EQUAL   => cmp2(y1, y2)
```

If cmp_1 is a comparison for t_1
and cmp_2 is a comparison for t_2
then lex(cmp_1, cmp_2) is a comparison for $t_1 * t_2$

```
lexcompare = lex(Int.compare, Int.compare)
  : (int * int) * (int * int) -> order
```

sorted

- The concept of a *sorted* collection of data makes sense for any *type* equipped with a *comparison function*
- For Int.compare this is same as "sorted w.r.t. <"
- We can *adapt* sorted, isort, msort, etc…
  by going *polymorphic* and adding a
  comparison function as extra (first) argument…

```
sorted : ('a * 'a -> order) -> 'a list -> bool
   ins : ('a * 'a -> order) -> 'a * 'a list -> 'a list
 isort : ('a * 'a -> order) -> 'a list -> 'a list
```

sorted

sorted : ('a * 'a -> order) -> 'a list -> bool

```
fun sorted cmp [ ] = true
 |    sorted cmp [x] = true
 |    sorted cmp (x::y::L) =
          case cmp(x, y) of
            GREATER => false
 |    _           => sorted cmp (y::L)
```

L is cmp-sorted iff
sorted cmp L = **true**

Every element is cmp-≤ all later elements

sorted

- When cmp : t * t -> order is a *comparison for type* t, for all list values L : t list we get

sorted cmp L = true

if and only if

every item in L is cmp-≤ all later items

- We say "L is cmp-sorted"

sorted

- When cmp : t * t -> order is a *comparison for type* t, for all list values L : t list we get

sorted cmp L = true

if and only if

every item in L is cmp-≤ all later items

- We say "L is cmp-sorted"

x::R is cmp-sorted

if and only if

x is cmp-≤ all of R & R is cmp-sorted

insertion

ins : (’a * ’a -> order) -> (’a * ’a list) -> ’a list

```
fun ins cmp (x, [ ]) = [x]
 |  ins cmp (x, y::L) =
     case cmp(x, y) of
          GREATER => y :: ins cmp (x, L)
        |     _           => x :: (y::L)
```

If cmp is a comparison and L is cmp-sorted, then
ins cmp (x, L) = a cmp-sorted permutation of x::L.

note

If

cmp(x,y) = EQUAL,

then

ins cmp (x, y::L) = x::y::L

(Remember this!)

insertion

(uncurried version)

ins : ('a * 'a -> order) ✶ ('a * 'a list) -> 'a list

```
fun ins (cmp, (x, [ ])) = [x]
| ins (cmp, (x, y::L)) =
    case cmp(x, y) of
        GREATER => y :: ins (cmp, (x, L))
      | _       => x :: (y::L)
```

why curry?

We prefer the *curried* insertion function

ins : (’a * ’a -> order) **->** (’a * ’a list) -> ’a list

to the *uncurried* version

ins : (’a * ’a -> order) ***** (’a * ’a list) -> ’a list

The curried function can be partially applied...

ins Int.compare : int * int list -> int list

ins String.compare : string * string list -> string list

isortl and isortr

isortl, isortr : ('a * 'a -> order) -> 'a list -> 'a list

```
fun isortl cmp L = foldl (ins cmp) [ ] L
fun isortr cmp L = foldr (ins cmp) [ ] L
```

If cmp is a comparison, then
for all lists L,
isortl cmp L = a cmp-sorted perm of L
& isortr cmp L = a cmp-sorted perm of L

connection

- Int.compare : int * int -> order (usual <)
- isortl Int.compare = isortr Int.compare

= isort : int list -> int list
(as defined previously)

Follows from the (proven) specs,
since an integer list
has only ONE <-sorted permutation

examples

isortl Int.compare [3,1,2,1] = [1,1,2,3]

isortr lexcompare [(1,2),(2,2),(1,1),(2,1)]

= [(1,1),(1,2),(2,1),(2,2)]

```
isortl String.compare [“all”, “your”, “base”, “are”, “belong”, “to”, “us”]

val it = [“all”, “are”, “base”, “belong”, “to”, “us”, “your”] : string list
```

- For ***integer*** lists we get
  isortl Int.compare = isortr Int.compare.

What does this mean?

- Is it true that for *all* types of data
  and *all* comparisons cmp we get
  isortl cmp = isortr cmp

?

not always same

Let i = ins cmp.

isortr cmp $[x_1, ..., x_n] = i(x_1, i(x_2, ... i(x_n,[\ ])...))$

inserts "equal" items in
the same order as they occur in the list

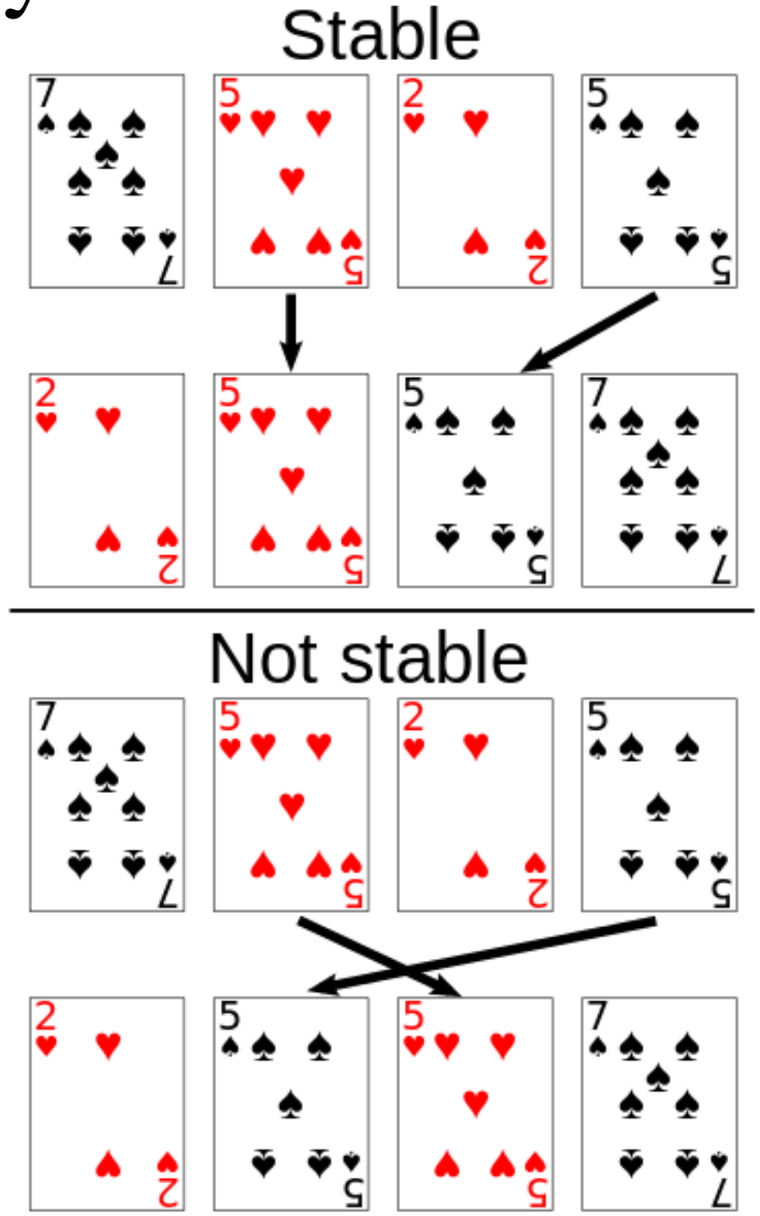
isortl cmp $[x_1, ..., x_n] = i(x_n, i(x_{n-1}, ... i(x_1,[\ ])...))$

inserts "equal" items in
the opposite order

For some comparisons,
"equal" items may be *different*

stability

A sorting function is ***stable*** if and only if it preserves the relative ordering of "equal" items.



isortr cmp is stable
isortl cmp is not

counterexample

fun leftcompare ((x,y), (x',y')) = Int.compare(x, x')

leftcompare is a comparison for int * bool

isortl leftcompare [(1,true),(1,false)] = [(1,false),(1,true)]

isortr leftcompare [(1,true),(1,false)] = [(1,true),(1,false)]

isortl ≠ isortr

special case

- If all items in L are cmp-equal

isortl cmp L = rev L

isortr cmp L = L

reflection

isortl and isortr are *not* extensionally equal

- Q: What's special about Int.compare that makes isortl Int.compare = isortr Int.compare?

- A: Int.compare(x, y) = EQUAL
  if and only if
  x = y

For Int.compare,
"equal" items are really equal

going further

- Easy to generalize *msort*
- Easy to generalize from *lists* to *trees*

msort : (’a * ’a -> order) -> (’a list -> ’a list)

Msort : (’a * ’a -> order) -> (’a tree -> ’a tree)

cmp-sorted trees

Let cmp be a comparison for type t.
Consider values of type t tree.

- Empty is a cmp-sorted tree
- Node(A, x, B) is a cmp-sorted tree iff

  (i) Everything in A is cmp-≤ to x

  (ii) Everything in B is cmp-≥ to x

  (iii) A and B are cmp-sorted trees

benefits

of using a higher-order function

A ***polymorphic*** *sorting* function

can be used with different *types* and *comparisons*

- ***One*** type, ***many*** instances
- ***One*** specification, ***many*** special cases
- ***One*** function, ***many*** uses
- ***One*** correctness proof, ***many*** consequences