15-150 Fall 2013
Lecture 11

Stephen Brookes

last time

- higher-order functions
- maps and folds on lists and trees

today

- A **case study**
- Putting ideas to work
- Developing an *abstract* and *general* solution to a *family* of problems
- The benefits of ***polymorphic types*** and ***higher-order functions***
- Currying and partial evaluation

general sorting

An abstract formulation

- A type of *data*, with a *comparison* function
- Sorting *lists* and *trees* of *data*

data

- A *type* equipped with a *comparison* function
  - satisfying the *usual* properties
- Common instances include

type	***comparison***	***ML***
int	*usual*	*compare*
int * int	*lexicographic*	*lex (compare, compare)*
string	*dictionary*	*String.compare*

wikipedia says

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 other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; also ... for canonicalizing data and producing human-readable output.

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

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"

for int

compare : int * int -> order

```
fun compare(x:int, y:int):order =
  if x<y then LESS else
  if y<x then GREATER else EQUAL
```

This is a comparison function!

compare(2,3) = LESS
compare(2,2) = EQUAL

for int * int

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

fun leftcompare((x_1, y_1), (x_2, y_2)) = compare(x_1, x_2)

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

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

These are comparison functions.

for int * int

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

```
fun leftcompare((x1, y1), (x2, y2)) = compare(x1, x2)
```

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

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

These are comparison functions.

lexcompare((2,3),(3,2)) = LESS

lexcompare((2,3),(2,0)) = GREATER

upside down

```
flip: ('a * 'a -> order) -> ('a * 'a -> order)
fun flip cmp (x, y) = cmp (y, x)
```

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

(flip compare) (2,3) = GREATER

lex

```
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 $t_1 * t_2$.

lex

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(compare, compare)
: (int * int) * (int * int) => order

listlex

Define a function

listlex : ('a * 'a -> order) -> 'a list * 'a list -> order

such that

when cmp **is a comparison for** t,

listlex cmp **is a comparison for** t list

Hint:

```
listlex cmp ([ ], [ ]) = EQUAL
listlex cmp ([ ], y::R) = LESS
listlex cmp (x::L, [ ]) = GREATER
listlex cmp (x::L, y::R) = cmp(x,y)              if cmp(x,y)<>EQUAL
listlex cmp (x::L, y::R) = listlex cmp (L, R) if cmp(x,y)=EQUAL.
```

less and **lesseq**

```
less   : ('a * 'a -> order) -> ('a * 'a -> bool)
lesseq : ('a * 'a -> order) -> ('a * 'a -> bool)
```

```
fun less cmp (x, y) = (cmp(x, y) = LESS)

fun lesseq cmp (x, y) = (cmp(x, y) <> GREATER)
```

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**

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,
ins cmp (x, L) = a cmp-sorted permutation of x::L.

reflection

Why did we choose the type

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

instead of

(’a * ’a -> order) ***** (’a * ’a list) -> ’a list ?

currying functions

- A "curried" function $F : t_1 \to t_2 \to t$ can be "partially applied" to an argument of type t_1, producing a function of type $t_2 \to t$

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

ins compare : int * int list -> int list

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

- There is also an "uncurried" version of F, a function G of type $t_1 * t_2 \to t$ such that for all $x{:}t_1$, $y{:}t_2$, $G(x,y) = (F\ x)\ y$

curry recipe

We chose

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

curry recipe

We chose

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

Why not

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

curry recipe

We chose

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

Why not

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

Why not

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

note

It's obvious from the function definition that

if cmp(x,y)=EQUAL, then

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

(Remember this!)

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 permutation of L
& isortr cmp L = a cmp-sorted permutation of L

examples

isortl 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)]

connection

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

  = isort : int list -> int list

  (as defined previously)

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

question

- For ***integer*** data with the *usual* comparison, isortl compare = isortr compare.
- Is it true that for *all* types and *all* comparisons cmp, isortl cmp = isortr cmp **?**

“algebraic” specs

If g is total, then for all z and $[x_1, \ldots, x_n]$,

$$\text{foldr } g\ z\ [x_1, \ldots, x_n] = g(x_1, g(x_2, \ldots g(x_n, z)\ldots))$$

$$\text{foldl } g\ z\ [x_1, \ldots, x_n] = g(x_n, g(x_{n-1}, \ldots g(x_1, z)\ldots))$$

Proof?

Induction on n

$$\underbrace{[x_1, \ldots, x_n]}_{\text{length } n} = x_1 :: \underbrace{[x_2, \ldots, x_n]}_{\text{length } n-1}$$

so...

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

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

inserts "equal" items in the opposite order

stability

A sorting function is ***stable*** iff
it preserves the relative ordering of items
for which the comparison result is EQUAL.

The “algebraic” specs imply that
isortr cmp is stable but isortl cmp is not.

counterexample

fun leftcompare((x,y), (x’,y’)) = 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

more generally

- If all items in L are cmp-equal

isortl cmp L = rev L

isortr cmp L = L

reflection

- isortl and isortr are not equal
- What's special about compare:int*int -> order that makes isortl compare = isortr compare?

reflection

- isortl and isortr are not equal
- What's special about compare:int*int -> order that makes isortl compare = isortr compare?

For all x,y:int
compare(x,y)=EQUAL iff x=y

stability

as an equational property

curried!

```
(* same : ('a * 'a -> order) -> 'a -> 'a -> bool *)
fun same cmp x y = (cmp(x,y) = EQUAL);

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

A function s : ('a * 'a -> order) -> 'a list -> 'a list is **STABLE**
iff for all comparisons cmp, and all x and L,
filter (same cmp x) L = filter (same cmp x) (s cmp L).

partially applied

partially applied

on reflection

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

We can rewrite to make insertion a ***local*** function...

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

```
fun isortr cmp L =
 let
   fun ins cmp (x, [ ]) = [x]
    |     ins cmp (x, y::R) = (case cmp(x, y) of
                                GREATER => y::ins cmp(x, R)
                              | _              => x::y::R)
 in
   foldr (ins cmp) [ ] L
 end;
```

on reflection

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

We can rewrite to make cmp local...

```
fun isortr cmp L =
 let
   fun ins (x, [ ]) = [x]
    |    ins (x, y::R) = (case cmp(x, y) of
                            GREATER => y::ins(x, R)
                           | _          => x::y::R)
 in
   foldr ins [ ] L
 end;
```

ins : 'a * 'a list -> 'a list

going further

- Easy to generalize *mergesort* and *quicksort*
- Easy to generalize from *lists* to *trees*

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

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

benefits

of using a higher-order function

- One ***polymorphic*** *sorting* function s
- Can be used with different *types* and *comparisons*

```
s compare
              : int list -> int list
```

```
s (lex(compare,compare))
              : (int*int) list -> (int*int) list
```

```
s (listlex compare)
              : int list list -> int list list
```

benefits

of polymorphism

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