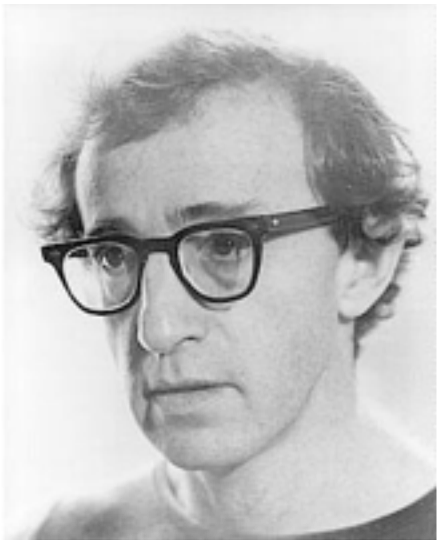
15-150 Fall 2013
Lecture 6

Stephen Brookes

Most of the time I don't have much **fun**.
The rest of the time I don't have any **fun** at all.



Announcements

Read the notes!

Homework 3 out...

NO CHEATING

today

- Sorting an integer list
    - Specifications and proofs
    - Asymptotic analysis

Insertion sort

Mergesort

comparison

compare : int * int -> order

```
type order = LESS | EQUAL | GREATER;

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

compare(x,y) = LESS	if x<y
compare(x,y) = EQUAL	if x=y
compare(x,y) = GREATER	if x>y

comparison

- $\leq$ is a ***linear ordering*** on values of type int

  For all values a,b,c :int

  If $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry)
  If $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
  Either $a \leq b$ or $b \leq a$ (totality)

- $<$ is defined by

  For all values a,b:int
  $a < b$ if and only if (a $\leq b$ and $a \neq b$)

  and satisfies

  For all values a,b:int

  $a < b$ or $b < a$ or $a = b$ (trichotomy)

sorted

sorted : int list -> bool

A list of integers is <-sorted

if each item in the list is ≤ all items that occur later.

```
fun sorted [ ] = true
 |   sorted [x] = true
 |   sorted (x::y::L) =
     (compare(x,y) <> GREATER) andalso sorted(y::L);
```

For all L : int list,
sorted(L) = true   if L is <-sorted
= false   otherwise

specs and code

- We use **sorted** only in *specifications*.
- Our sorting functions won't *use* it.
- But you *could* use it for testing...

We will say
"L is sorted"
when sorted(L)=true

insertion sort

- **Insertion sort** is a simple sorting algorithm that builds the sorted list recursively, one item at a time.
- If list is empty, do nothing.
- Otherwise, each recursive call inserts an item from the input list into its correct position in the already-sorted list obtained so far.

insertion

ins : int * int list -> int list

(* REQUIRES L is a sorted list *)
(* ENSURES ins(x, L) = a sorted perm of x::L *)

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

For all sorted integer lists L,
ins(x, L) = a sorted permutation of x::L.

proof outline

> **Theorem**
> For all sorted lists L,
> ins(x, L) = a sorted permutation of x::L.

- **Proof**: By induction on length of L.
- **Base case**: When L has length 0, L must be [ ].
  [ ] is <-sorted. Show ins(x, [ ]) = a sorted perm of x::[ ].
- **Inductive case**: Let k>0 and assume IH:

  For all sorted lists A of length < k,
  ins(x, A) = a sorted perm of x::A.

  - Let L be a sorted list of length k.
    Pick y and R such that L=y::R. So length(R) < k.
  - R is a sorted list with length < k, and y ≤ all of R
  - By IH, ins(x, R) = a sorted perm of x::R
  - Show: ins(x, y::R) = a sorted perm of x::(y::R)

sketch

```
ins (x, y::R) =  case compare(x, y) of
                   GREATER => y::ins(x, R)
                 |    _    => x::y::R;
```

- R is sorted and y ≤ all of R.
- By IH, ins(x, R) = a sorted perm of x::R
  - If x>y we have ins(x, y::R) = y::ins(x,R)
    This list is *sorted* because...
    This list is a *perm* of x::y::R because...
  - If x≤y we have ins(x, y::R) = x::y::R
    This list is *sorted* because...
    This list is a *perm* of x::y::R because...
- In all cases, ins(x, y::R) = a sorted perm of x::y::L

insertion sort

isort : int list -> int list

(* REQUIRES true *)
(* ENSURES isort(L) = a sorted perm of L *)

```
fun isort [ ] = [ ]
|   isort (x::L) = ins (x, isort L)
```

For all values L: int list,
isort L = a <-sorted permutation of L.

For all integer lists L,
isort L = a <-sorted permutation of L.

proof outline

For all L: int list,
isort L = a <-sorted permutation of L.

- **Proof**: By induction on length of L.
- **Base case**: for L = [ ].
  Show that isort [ ] = a sorted perm of [ ].
- **Inductive case**: for L = y::R.
  IH: isort R = a sorted perm of R.
  Show: isort(y::R) = a sorted perm of y::R.

Use the *proven* spec for ins!

variation

isort′ : int list -> int list

```
fun isort′ [ ] = [ ]
 |    isort′ [x] = [x]
 |    isort′ (x::L) = ins (x, isort′ L);
```

clause for isort′ [x]
is redundant!

equivalent

- isort and isort′ are extensionally equivalent.

  For all L : int list, isort L = isort′ L.

- Proof?

mergesort

- A recursive ***divide-and-conquer*** algorithm
- If list has length 0 or 1: do nothing.
- If list has length 2 or more:

  split the list into two smaller lists,
  mergesort these lists,
  merge the results

implementation

- We'll use helper functions to do the splitting and merging

```
split : int list -> int list * int list
merge : int list * int list -> int list
```

split

split : int list -> int list * int list

```
(* REQUIRES true                                           *)
(* ENSURES split(L) = a pair of lists (A, B)                *)
(*    such that length(A) and length(B) differ by at most 1, *)
(*     and A@B is a permutation of L.                        *)
```

length(A)≈length(B)

```
fun split [ ]  = ([ ],  [ ])
  |   split [x] = ([x], [ ])
  |   split (x::y::L) =
        let val (A, B) = split L in (x::A, y::B) end
```

proof outline

For all L:int list,
split(L) = a pair of lists (A, B) such that
length(A) ≈ length(B) and A@B is a permutation of L.

- **Proof**: by (strong) induction on *length* of L
- **Base cases**: L = [ ], [x]

  (i) Show that split [ ] = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of [ ].

  (ii) Show that split [x] = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of [x].

Key facts

split [ ] = ([ ], [ ])
[ ]@[ ] = [ ]

split [x] = ([x], [ ])
[x]@[ ] = [x]

- **Inductive case**: L=x::y::R.

  Induction Hypothesis: split(R) = a pair (A′, B′) such that length(A′)≈length(B′) & A′@B′ is a perm of R.

  (iii) Show that split(x::y::R) = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of x::y::R.

Key facts split (x::y::R) = (x::A′, y::B′)

length(x::A′) ≈ length(y::B′) (x::A′)@(y::B′) is a perm of x::y::R

comments

- We used *strong* induction on length of L rather than simple induction.
- Reason: split(x::y::R) calls split(R) and length of R is *two less* than length of x::y::R.

notes

- If length(L)>1 and split(L) = (A, B), then A and B have *smaller* length than L.
- This follows from the spec, using some fairly obvious facts:

A@B is a perm of L, so
length(A)+length(B)=length(L)

length(A) & length(B) differ by 0 or 1

if n>1 and n odd, (n div 2)+1 < n
if n>1 and n even, n div 2 < n

merge

merge : int list * int list -> int list

```
(* REQUIRES  A and B are <-sorted lists                    *)
(* ENSURES merge(A, B) = a <-sorted perm of A@B            *)

fun merge (A, [ ])  = A
 |   merge ([ ], B) = B
 |  merge (x::A, y::B) = case compare(x, y) of
                         LESS    => x :: merge(A, y::B)
                     |   EQUAL   => x::y::merge(A, B)
                     | GREATER   => y :: merge(x::A, B)
```

proof outline

For all <-sorted lists A and B,
merge(A, B) = a <-sorted permutation of A@B.

- **Method**: *strong* induction on length(A)*length(B).
- **Base cases**: (A, [ ]) and ([ ], B).
  (i) Show: if A is <-sorted, merge(A,[ ]) = a <-sorted perm of A@[ ].
  (ii) Show: if B is <-sorted, merge([ ],B) = a <-sorted perm of [ ]@B.
- **Inductive case**: (x::A, y::B).
  Induction Hypothesis: for all *smaller* (A', B'), if A' & B' are <-sorted, merge(A', B') = a <-sorted perm of A'@B'.
  Show: if x::A and y::B are <-sorted,
  merge(x::A, y::B) = a <-sorted perm of (x::A)@(y::B).

comments

```
fun merge (A, [ ])  = A
 |   merge ([ ], B) = B
 |   merge (x::A, y::B) = case compare(x, y) of
                              LESS   => x :: merge(A, y::B)
                         |    EQUAL  => x::y::merge(A, B)
                         | GREATER   => y :: merge(x::A, B)
```

Does clause order matter? **NO**

Patterns are { Exhaustive; Overlap of first two clauses is harmless

Each yields merge([ ], [ ]) = [ ]

Could use *nested* **if-then-else** instead of **case**.

But we need a 3-way branch, so **case** is *better style*.

so far

- We *defined* split and merge
- We *proved* they meet their specs
- Now let's use them to implement the **mergesort** algorithm...

mergesort

msort : int list -> int list

```
(* REQUIRES  true                                   *)
(* ENSURES   msort(L) = a <-sorted perm of L        *)

fun msort [ ] = [ ]
  |  msort [x] = [x]
  |  msort L =
        let
          val (A, B) = split L
        in
          merge (msort A, msort B)
        end
```

mergesort

msort : int list -> int list

(* REQUIRES true *)
(* ENSURES msort(L) = a <-sorted perm of L *)

```
fun msort [ ] = [ ]
  | msort [x] = [x]
  | msort L = let
                val (A, B) = split L
                val A′ = msort A
                val B′ = msort B
              in
                merge (A′, B′)
              end
```

an alternative version

proof outline

For all L:int list,
  msort(L) = a <-sorted permutation of L.

- **Method**: by strong induction on *length* of L
- **Base cases**: L = [ ], L = [x]
  (i) Show msort [ ] = a sorted perm of [ ]
  (ii) Show msort [x] = a sorted perm of [x]
- **Inductive case**: length(L)>1.
  Inductive hypothesis: for all *shorter* lists R,
  msort R = a sorted perm of R.
  Show msort L = a sorted perm of L.