

15-150 Fall 2013

Lecture 7

Stephen Brookes

Announcements

Read the notes!

Homework 3 due...

NO CHEATING

last time

- We implemented *insertion sort* and *mergesort* for *integer lists*
- We proved **correctness** of *insertion sort*
- We proved some specs for ***split*** and ***merge***
- How about ***mergesort***?
- What about ***efficiency***?

mergesort

`msort : int list -> int list`

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

For all L :int list,
 $\text{msort}(L)$ = a <-sorted permutation of L .

lemmas

For all L :int list, if $\text{length}(L) > 1$

then $\text{split}(L) = (A, B)$

where A and B have *shorter length* than L

and $A @ B$ is a permutation of L

For all sorted lists A and B ,

$\text{merge}(A, B) =$ a sorted permutation of $A @ B$

proof outline

Theorem

For all L :int list,

$\text{msort}(L)$ = a \leftarrow -sorted permutation of L .

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

inductive step

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

- Let $\text{length}(L) > 1$. Then

$\text{msort } L = \text{merge}(\text{msort } A, \text{msort } B)$

where $(A, B) = \text{split } L$

- $\text{msort } A$ and $\text{msort } B$ are sorted lists (why?)
- $\text{merge}(\text{msort } A, \text{msort } B)$ = a sorted list (why?)
- $\text{merge}(\text{msort } A, \text{msort } B)$ = a perm of L (why?)

correct!

`msort : int list -> int list`

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

For all L :int list,
 $\text{msort}(L)$ = a \leftarrow -sorted permutation of L .

a variation

`msort : int list -> int list`

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

a variation

`msort : int list -> int list`

```
fun msort [ ] = [ ]
```

```
  | msort L = let
```

```
      val (A, B) = split L
```

```
  in
```

```
      merge (msort A, msort B)
```

```
  end;
```

loops forever
on non-empty lists

the problem

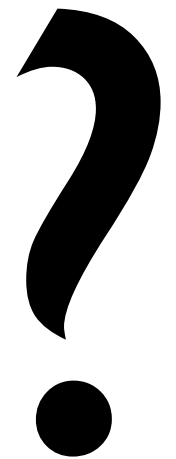
- $\text{split } [x] = ([x], [])$
- $\text{msort } [x] \Rightarrow^* (\text{fn } \dots \Rightarrow \dots) (\text{msort } [x], \text{msort } [])$

infinite computation

What happens if we
try to **prove** that

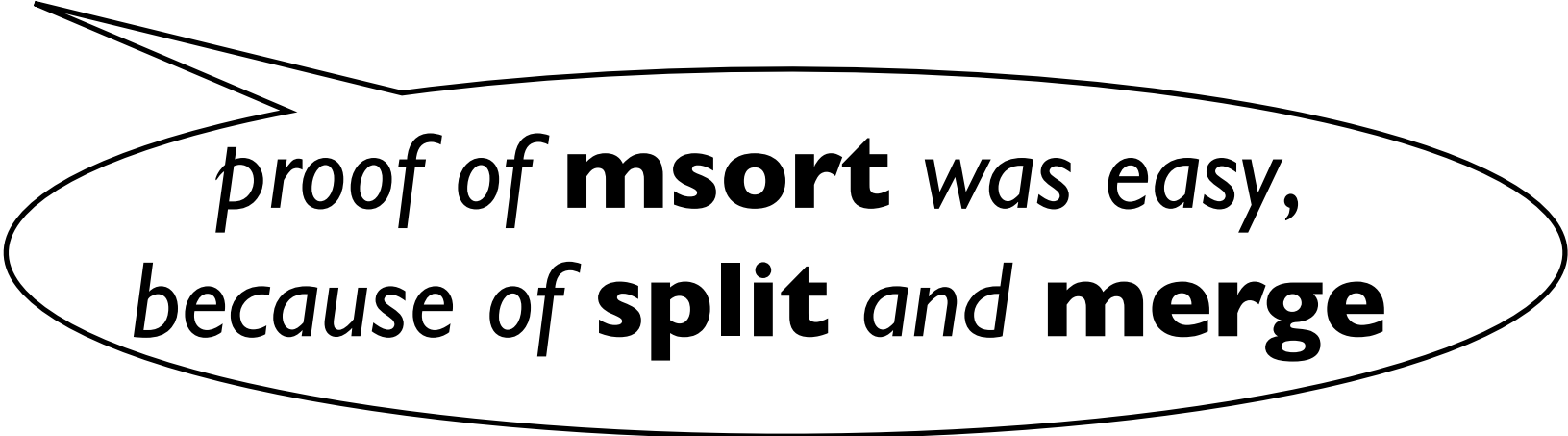
For all L :int list,

$\text{msort}(L)$ = a <-sorted permutation of L .



principles

- Every function needs a spec
- Every spec needs a proof
- Recursive functions need inductive proofs
 - Learn to pick an appropriate method...
 - Choose helper functions wisely!



*proof of **msort** was easy,
because of **split** and **merge***

choose wisely

- Use *helpful* specs
- **merge** also satisfies other specs, e.g.

For all integer lists L and R,
 $\text{merge}(L, R) = \text{a perm of } L @ R.$

Every program has (at least) two purposes:
The one for which it was written
and another for which it wasn't.



the joy of specs

- The **proof** for `msort` relied only on the ***specification*** proven for `split` (and the specification proven for `merge`)
- In the definition of `msort` we can *replace* `split` by *any* function that satisfies this *specification*, and the proof will still be valid, for the new version of `msort`

example

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

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

example

- `split` and `split'` are not *extensionally equivalent*, but they both satisfy the *specification* used in the correctness proof
- ... so `msort` and `msort'` are both correct

so far

- We've implemented insertion sort and mergesort in ML, *correctly*
- What about efficiency?

split work

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

$W_{\text{split}}(n)$ is $O(n)$

Let $W_{\text{split}}(n)$ = work of `split(L)` when `length(L)=n`

$$\begin{aligned} W_{\text{split}}(n) &= c_0 && \text{for } n=0, 1 \\ W_{\text{split}}(n) &= c_1 + W_{\text{split}}(n-2) && \text{for } n>1 \\ &\text{for some constants } c_0, c_1 \end{aligned}$$

merge work

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);

$W_{\text{merge}}(n)$ is $O(n)$

Let $W_{\text{merge}}(n)$ = work of merge(A,B)
when $\text{length}(A) + \text{length}(B) = n$

msort work

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

Let $W_{\text{msort}}(n)$ = work of $\text{msort}(L)$ when $\text{length}(L)=n$

$$W_{\text{msort}}(0) = 1 \qquad W_{\text{msort}}(1) = 1$$

$$\begin{aligned} W_{\text{msort}}(n) &= W_{\text{split}}(n) + 2W_{\text{msort}}(n \text{ div } 2) + W_{\text{merge}}(n) \\ &\leq cn + 2W_{\text{msort}}(n \text{ div } 2) \qquad \text{for } n > 1 \\ &\qquad \text{for some constant } c \end{aligned}$$



$W_{\text{msort}}(n)$ is $O(n \log n)$

exercise

- Give a recurrence relation for $W_{\text{ins}}(n)$, the work for $\text{ins}(x, L)$ when L has length n , making the worst-case assumption that x is greater than every item in L .
- Then give a recurrence relation for $W_{\text{isort}}(n)$, the worst-case work for $\text{isort}(L)$ when L has length n .
- Solve, and classify using big-O notation.
- Which lists incur worst-case behavior?

assessment

- `msort(L)` on lists does $O(n \log n)$ work, where n is `length(L)`
- Lists are built from `[]` and `::` so are inherently *sequential* data structures
- Not easy to redesign `msort` to exploit parallel evaluation

next

- Sorting an integer ***tree***
 - Specifications and proofs
 - Asymptotic analysis

Insertion

“Parallel” Mergesort

trees

datatype tree = Empty | Node of tree * int * tree;

- A user-defined type named **tree**
- With constructors **Empty** and **Node**

Empty : tree

Node : tree * int * tree -> tree

tree values

- Every tree value is either **Empty** or has the form **Node(t_1 , x , t_2)**, where t_1 and t_2 are tree values and x is an integer.

Contrast with integer lists:

Every list value is either **nil** or has the form **$x::L$** , where **L** is a list value and **x** is an integer.

tree patterns

Empty

Node(p_1 , p , p_2)

- Empty empty tree
- Node($_$, $_$, $_$) non-empty tree
- Node(Empty, $_$, Empty) tree with one node
- Node($_$, 42, $_$) tree with 42 at root

tree patterns

Empty matches t iff t is *Empty*

$\text{Node}(p_1, p, p_2)$ matches t iff

t is $\text{Node}(t_1, v, t_2)$ such that

p_1 matches t_1 , p matches v , p_2 matches t_2

(and combines all the bindings
when the match succeeds)

structural induction

for trees

- **To prove:** “For all trees t , $P(t)$ holds”
- **Base case:** For $t = \text{Empty}$.
Show $P(\text{Empty})$ holds.
- **Inductive case:** For $t = \text{Node}(t_1, x, t_2)$.
Induction hypothesis: $P(t_1)$ and $P(t_2)$ hold.
Show that $P(\text{Node}(t_1, x, t_2))$ holds.

Contrast with
structural induction for *lists*

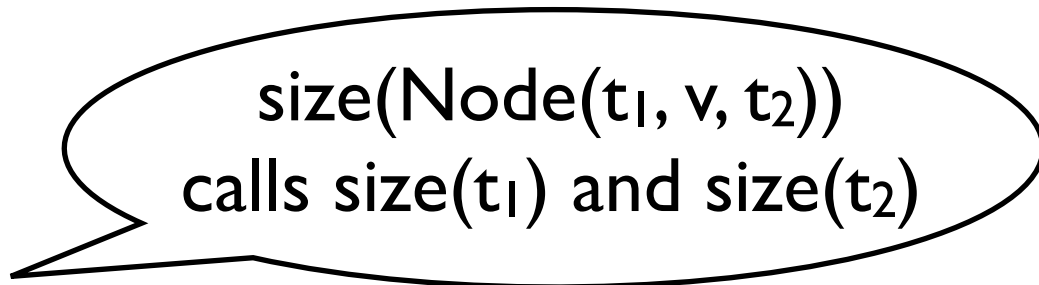
size

fun size Empty = 0

| size (Node(t1, _, t2)) = size t1 + size t2 + 1;

Uses tree patterns

Recursion is *structural*



size(Node(t₁, v, t₂))
calls size(t₁) and size(t₂)

Can prove by structural induction
that for all trees t,
size(t) = a non-negative integer



the number of nodes in t

size matters

- For all trees t , $\text{size}(t) \geq 0$.
- If $t = \text{Node}(t_1, x, t_2)$,
 $\text{size}(t_1) < \text{size}(t)$ and $\text{size}(t_2) < \text{size}(t)$.
- Many recursive functions on trees make recursive calls on trees with smaller size.
- Can often use induction on *size* to prove properties or analyze efficiency.

depth

(or *height*)

```
fun depth Empty = 0
|   depth (Node(t1, _, t2)) =
      max(depth t1, depth t2) + 1;
```

Can prove by structural induction
that for all trees t ,
 $\text{depth}(t)$ = a non-negative integer



the length of longest path
from root of t to a leaf

depth matters

- For all trees t , $\text{depth}(t) \geq 0$.
- If $t = \text{Node}(t_1, x, t_2)$,
 $\text{depth}(t_1) < \text{depth}(t)$ and $\text{depth}(t_2) < \text{depth}(t)$.
- Many recursive functions on trees make recursive calls on trees with smaller depth.
- Can often use induction on *depth* to prove properties or analyze efficiency.

traversal

$\text{trav} : \text{tree} \rightarrow \text{int list}$

```
fun trav Empty = [ ]  
  | trav (Node(t1, x, t2)) = trav t1 @ (x :: trav t2);
```

For all trees t ,
 $\text{trav}(t)$ returns a list of the integers in t



in-order traversal

sorted trees

- Empty is sorted
- $\text{Node}(t_1, x, t_2)$ is sorted iff

every integer in t_1 is $\leq x$

and

every integer in t_2 is $\geq x$

and

t_1 and t_2 are sorted

t is sorted
iff

$\text{trav}(t)$ is a sorted list

insertion

$\text{ins} : \text{int} * \text{int list} \rightarrow \text{int list}$

fun ins (x, []) = [x]

| ins (x, y::L) =

as before

case compare(x, y) **of**

GREATER \Rightarrow y::ins(x, L)

| _ \Rightarrow x::y::L;

For all sorted integer lists L,

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

Insertion

$\text{Ins} : \text{int} * \text{tree} \rightarrow \text{tree}$

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
|   Ins (x, Node(t1, y, t2)) =
    case compare(x, y) of
        GREATER => Node(t1, y, Ins(x, t2))
    |   _       => Node(Ins(x, t1), y, t2);
```

For all sorted integer trees t ,

$\text{Ins}(x, t)$ = a sorted tree t' such that

$\text{trav}(t')$ is a perm of $x::\text{trav}(t)$