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,
 msort(L) = a <-sorted permutation of L.

lemmas

For all L:int list, if length(L)>1

then 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,

merge(A, B)= a sorted permutation of A@B

proof outline

> **Theorem**
> 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.

inductive step

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

- Let length(L) > 1. Then

  msort L = merge(msort A, msort B)

  where (A, B) = split L

  - msort A and msort B are sorted lists (why?)
  - merge(msort A, msort B) = a sorted list (why?)
  - merge(msort A, 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,
  msort(L) = a <-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

- split [x] = ([x], [ ])
- msort [x] =>* (fn ... => ...) (msort [x], msort [ ])

infinite computation

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

For all L:int list,
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,
merge(L, R) = 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
```

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

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

$W_{split}(n) = c_0$ for $n=0, 1$
$W_{split}(n) = c_1 + W_{split}(n-2)$ for $n>1$
for some constants c_0, c_1

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);
```

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

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

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

$$W_{msort}(n) = W_{split}(n) + 2W_{msort}(n \text{ div } 2) + W_{merge}(n)$$
$$\leq cn + 2W_{msort}(n \text{ div } 2) \qquad \text{for } n>1$$

for some constant c

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

exercise

- Give a recurrence relation for $W_{ins}(n)$, the work for 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 $W_{isort}(n)$, the worst-case work for 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 $\text{Node}(t_1, x, t_2)$, 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

Node(p_1, p, p_2) *matches* t iff

t is 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 = Empty.
  Show P(Empty) holds.
- **Inductive case**: For t = Node(t_1, x, t_2).
  Induction hypothesis: P(t_1) and P(t_2) hold.
  Show that P(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(t_1, v, t_2))
calls size(t_1) and size(t_2)

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,
depth(t) = a non-negative integer

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

depth matters

- For all trees t, $depth(t) \geq 0$.
- If $t = Node(t_1, x, t_2)$, $depth(t_1) < depth(t)$ and $depth(t_2) < depth(t)$.
- 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

trav : tree -> int list

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

For all trees t,
trav(t) returns a list of the integers in t

in-order traversal

sorted trees

- Empty is sorted
- 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
trav(t) is a sorted list

insertion

ins : int * int list -> int list

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

as before

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

Insertion

Ins : int * tree -> 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,
Ins(x,t) = a sorted tree t′ such that
trav(t′) is a perm of x::trav(t)