last time

- Sorting a list of integers
- Specifications and proofs
  - *helper functions* that really help
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
so far

- **sorting** for integer lists
- **specifications** and **correctness**
- … but what about **efficiency**?

work? span?

Before Work   After Work
work

**work** is number of evaluation steps, assuming a *sequential* processor

- For *dependent* sub-expressions, **add** the work

  - *dependent*: one gets evaluated **before** the other

\[
W(e_1 + e_2) = W(e_1) + W(e_2)
\]

\[
W(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) = W(e_0) + \max(W(e_1), W(e_2))
\]
**span**

*span* is the *number of evaluation steps*, assuming *unlimited parallelism*

- For *dependent* sub-expressions, *add* the span
- For *independent* sub-expressions, *max* the span

\[
\text{span} = \max(W(f \ x), W(f \ y))
\]

\[
\text{work} = W(f \ x) + W(f \ y)
\]
work, eggs, bacon and spam

work is the number of evaluation steps, assuming sequential processing

span is the number of evaluation steps, assuming unlimited parallelism

span is always $\leq$ work

For sequential code, span = work
msort L = \textbf{let} \textbf{val} (A, B) = split L \textbf{in}
merge (msort A, msort B) \textbf{end}

when length L > 0

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

$W_{msort}(n) = W_{split}(n) + 2W_{msort}(n \ \text{div} \ 2) + W_{merge}(n)$

$W_{msort}(n) = O(n) + 2W_{msort}(n \ \text{div} \ 2)$

$W_{msort}(n) \ \text{is} \ O(n \ \text{log} \ n)$
• `msort(L)` does $O(n \log n)$ work, where $n$ is the length of $L$

• List operations are inherently *sequential*
  
  • $e_1 :: e_2$ evaluates $e_1$ first, then $e_2$
  
  • *split* and *merge* are not easily *parallelizable*

• We *could* use parallel evaluation in `msort(L)` for the recursive calls to `msort A` and `msort B`

*How would this affect runtime?*
Let $S_{msort}(n) = \text{span of sort } L \text{ when length } L = n$

$$S_{msort}(n) = S_{split}(n) + S_{msort}(n \text{ div } 2) + S_{merge}(n)$$

$$S_{msort}(n) = \Omega(n) + S_{msort}(n \text{ div } 2)$$

$S_{msort}(n)$ is $\Theta(n)$
work and span

\[ W_{\text{msort}}(n) = O(n) + 2W_{\text{msort}}(n \div 2) \]

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

\[ S_{\text{msort}}(n) = O(n) + S_{\text{msort}}(n \div 2) \]

\( S_{\text{msort}}(n) \) is \( O(n) \)

\( O(n) \subset O(n \log n) \)

mergesort
is
potentially
worth parallelizing
summary

• \texttt{msort(L)} has \(O(n \log n)\) work, \(O(n)\) span

• So the potential \textit{speed-up} factor from parallel evaluation is \(O(\log n)\)

\[\text{... \textit{in principle}, we can speed up mergesort on lists by a factor of } \log n\]

\[\text{To do any better, we need a different data structure...}\]
next

*Trees are better than lists for parallel evaluation*

• Sorting a *tree*
  • Specifications and proofs
  • Asymptotic analysis

  Insertion
  “Parallel” Mergesort
int 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
An inductive definition

A tree value is either Empty
or has the form Node(t₁, x, t₂),
where t₁ and t₂ are tree values and x is an integer.

Contrast with integer lists:

A list value is either nil
or has the form x::L,
where L is a list value and x is an integer.
structural induction

To prove: For all tree values $t$, $P(t)$ holds by structural induction on $t$

- **Base case:** Prove $P(\text{Empty})$.

- **Inductive case:**
  Assume Induction Hypothesis: $P(t_1)$ and $P(t_2)$.
  Prove $P(\text{Node}(t_1, x, t_2))$, for all integers $x$.

That’s enough! Why?

Contrast with structural induction for lists
**tree patterns**

<table>
<thead>
<tr>
<th>pattern</th>
<th>tree values that match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>an empty tree</td>
</tr>
<tr>
<td>Node(_, _, _)</td>
<td>a non-empty tree</td>
</tr>
<tr>
<td>Node(Empty, _, Empty)</td>
<td>a tree with one node</td>
</tr>
<tr>
<td>Node(_, 42, _)</td>
<td>a tree with 42 at root</td>
</tr>
</tbody>
</table>
patterns
match values

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

Node\((A, x, B)\) matches

and binds \( x \) to 3,
\( A \) to Node\((\text{Empty},4,\text{Empty})\)
\( B \) to Node\((\text{Empty},2,\text{Empty})\)
fun Leaf(x:int):tree = Node(Empty, x, Empty)

fun Full(x:int, n:int):tree = 
  if n=0 then Empty 
  else 
      let 
          val T = Full(x, n-1) 
      in 
          Node(T, x, T) 
      end
fun size Empty = 0
  | size (Node(t1, _, t2)) = size t1 + size t2 + 1

Uses tree patterns
Recursion is structural

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

the number of nodes
size matters

• Size is always non-negative

\[
\text{size}(t) \geq 0
\]

• Children have smaller size

\[
\text{size}(t_i) < \text{size}(\text{Node}(t_1, x, t_2))
\]

• Many recursive functions on trees make recursive calls on trees with smaller size.

  • Use \textit{induction on size} to prove correctness.
**depth**
(or *height*)

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

Can prove by structural induction that for all trees $t$,

$$\text{depth}(t) = \text{a non-negative integer}$$

the length of longest path from root to a leaf node
depth matters

- For all trees $t$, $\text{depth}(t) \geq 0$.
- Children have smaller depth
  \[
  \text{depth}(t_i) < \text{depth}(\text{Node}(t_1, x, t_2))
  \]
- Many recursive functions on trees make recursive calls on trees with smaller depth.
- Can use induction on depth to prove properties or analyze efficiency.
exercises

• Prove that for all $n \geq 0$

\[
\text{size}(\text{Full}(42, n)) = 2^n - 1 \\
\text{depth}(\text{Full}(42, n)) = n
\]

\[
\text{Full}(42, 3) \\
\text{size is 7} \\
\text{depth is 3}
\]
**in-order traversal**

\[ \text{inord : tree} \rightarrow \text{int list} \]

\[
\begin{align*}
\text{fun } \text{inord Empty} & = [ ] \\
\text{inord (Node(t1, x, t2))} & = \text{inord t1} \ @ \ (x :: \text{inord t2})
\end{align*}
\]

\[ \text{left before root before right} \]

inord \( t \) = the in-order traversal list for \( t \)
**inord**

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

For all trees T,

```
length (inord T) = size T
```

prove by structural induction on T

For all lists L₁, L₂ of the same type

```
length (L₁ @ L₂) = length L₁ + length L₂
```

prove by structural induction on L₁
work analysis

fun inord Empty = []
|   inord (Node(t1, x, t2)) = inord t1 @ (x :: inord t2)

• Let $W_{\text{inord}}(n)$ be the work to evaluate $\text{inord}(T)$ when $T$ is a *full binary tree* of depth $n$
  \[
  \text{depth}(T) = n, \text{size}(T) = 2^n - 1
  \]
  \[
  W_{\text{inord}}(0) = 1
  \]
  \[
  W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n), \text{ for } n > 0
  \]
  \[
  W_{\text{inord}}(n) \text{ is } O(n2^n)
  \]

if $T = \text{Node}(A, x, B)$ is full and depth$(T) = n$, then
  \[
  \text{size}(A) = \text{size}(B) = 2^{n-1} - 1
  \]

work for $L_1@L_2$ is $O(\text{length } L_1)$
faster inord

inorder : tree * int list -> int list

fun inorder (Empty, L) = L
| inorder (Node(t1, x, t2), L) = inorder (t1, x :: inorder (t2, L))

Theorem

For all trees T, integer lists L,
inorder (T, L) = (inord T) @ L

The work for inorder(T, L), when T is a full tree of depth n, is O(2^n)
fun all (p : int -> bool, T : tree) : bool =
case T of
  Empty => true
| Node(A, x, B) =>
  (p x) andalso all (p, A) andalso all (p, B)

REQUIRES p is total

ENSURES all (p, T) = true iff
  every integer in T satisfies p
Empty is a sorted tree

Node(t₁, x, t₂) is a sorted tree iff

- every integer in t₁ is \( \leq x \),
- every integer in t₂ is \( \geq x \),
- and t₁, t₂ are sorted trees

Theorem

t is a sorted tree iff

\[ \text{inord}(t) \text{ is a sorted list} \]
fun is_sorted (T : tree) : bool = case T of
  Empty => true
| Node(A, x, B) =>
      all (fn y => y <= x, A) andalso
      all (fn y => y >= x, B) andalso
      is_sorted A andalso is_sorted B

is_sorted T = true iff
T is a sorted tree
balanced trees

• Empty is size-balanced
• Node(A, x, B) is size-balanced iff
  \[|\text{size}(A) - \text{size}(B)| \leq 1\]
  and A, B are size-balanced

• Empty is depth-balanced
• Node(A, x, B) is depth-balanced iff
  \[|\text{depth}(A) - \text{depth}(B)| \leq 1\]
  and A, B are depth-balanced
fun takedrop (0, L) = ([ ], L)
| takedrop (n, x::L) = let
| | val (A, B) = takedrop (n-1, L)
| | in
| | (x::A, B)
| end

fun list2tree [ ] = Empty
| list2tree [x] = Node(Empty, x, Empty)
| list2tree L = let
| | val n = length L
| | val (A, x::B) = takedrop (n div 2, L)
| | in
| | Node(list2tree A, x, list2tree B)
| end
specs

takedrop : int * int list -> int list * int list
REQUIRES 0 <= n <= length L
ENSURES takedrop (n, L) = (A, B) such that
L = A@B and length A = n

list2tree : int list -> tree
ENSURES
list2tree L = a size-balanced tree T
containing the integers from L

list2tree : int list -> tree
ENSURES
list2tree L = a size-balanced tree T
such that inord(T) = L
trees >> lists?

- Representing a collection of integers as a (balanced) tree may yield a parallel speed-up
- Using a sorted (and balanced) tree may even support faster sequential code
- Using lists, even sorted lists, only allows sequential code, and precludes parallelism
- Badly balanced trees are no better than lists!
fun mem (x, []) = false
| mem (x, y::L) = (x = y) orelse mem (x, L)

REQUIRES  true
ENSURES   mem (x, L) = true iff x is in L

W_{\text{mem}}(x, L) is O(length L)

Worst case: when x is not in L
or x is last element of L

S_{\text{mem}}(x, L) is also O(length L)
fun mem (x, [ ] ) = false \\
| mem (x, y::L) = case Int.compare(x, y) of \\
| LESS => false \\
| EQUAL => true \\
| GREATER => mem (x, L)

REQUIRES L is a sorted list 
ENSURES mem (x, L) = true iff x is in L

\[ W_{\text{mem}}(x, L) \text{ is } O(\text{length } L) \]

Worst case: when x is > all of L…

\[ S_{\text{mem}}(x, L) \text{ is also } O(\text{length } L) \]
fun mem (x, Empty) = false
| mem (x, Node(A, y, B)) = 
  (x = y) orelse mem (x, A) orelse mem (x, B)

REQUIRES  T is a tree
ENSURES   mem (x, T) = true iff x is in T

$W_{\text{mem}}(x, T)$ is $O(\text{size } T)$
Worst case: when $x$ is not in $T$
  or $x$ is inorder-last element of $T$

$S_{\text{mem}}(x, T)$ is also $O(\text{size } T)$
fun mem (x, Empty) = false
| mem (x, Node(A, y, B)) =
  (x = y) orelse
  let
    val (a, b) = (mem (x, A), mem (x, B))
  in
    a orelse b
  end

\[ W_{\text{mem}}(x, T) \text{ is } O(\text{size } T) \]
\[ S_{\text{mem}}(x, T) \text{ is } O(\text{depth } T) \]
fun mem (x, Empty) = false
| mem (x, Node(A, y, B)) = case Int.compare(x, y) of
| LESS => mem(x, A)
| EQUAL => true
| GREATER => mem (x, B)

REQUIRES T is a sorted tree
ENSURES mem (x, T) = true iff x is in T

\[ W_{mem}(x, T) \text{ is } O(\text{depth } T) \]
\[ S_{mem}(x, T) \text{ is } O(\text{depth } T) \]
trees >> lists?

- Representing a collection of integers as a *balanced* tree may yield a *parallel* speed-up.
- Using a *sorted* (and *balanced*) tree may even support faster *sequential* code.
- Using lists, even sorted lists, only allows *sequential* code, and precludes parallelism.
- Badly balanced trees are no better than lists!
sorting a tree

- If the tree is Empty, do nothing
- Otherwise
  (recursively) sort the two children, then
  merge the sorted children, then
  insert the root value

We’ll design helpers to insert and merge

merge will also need a helper to split a tree in two
inserting in a tree

Ins : int * tree -> tree
REQUIRES  t is a sorted tree
ENSURES  Ins(x,t) is a sorted tree consisting of x and all of t

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)

(contrast with list insertion)
inserting in a list

\[
\text{ins} : \text{int} \times \text{int list} \to \text{int list}
\]

\[
\text{fun} \ \text{ins} \ (x, \ [ \ ]) = [x] \\
| \ \text{ins} \ (x, \ y::L) = \\
\text{case} \ \text{compare}(x, \ y) \ \text{of} \\
\text{GREATER} \Rightarrow y::\text{ins}(x, \ L) \\
\text{_<_>} \Rightarrow x::y::L
\]

For all sorted integer lists L,
\[
\text{ins}(x, \ L) = \text{a sorted permutation of } x::L.
\]
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)

\[
\begin{array}{c}
\text{Ins}(4, 3) \\
\text{Node(1, 6, Node(2, 5, Empty))}
\end{array}
\]

\[
\begin{array}{c}
\text{Ins}(4, 6) \\
\text{Node(1, Node(2, 5, Empty), Empty)}
\end{array}
\]

\[
\begin{array}{c}
\text{Ins}(4, 5) \\
\text{Node(1, Node(2, Node(6, Empty), Empty), Empty)}
\end{array}
\]
value equations

\[
\begin{align*}
\text{Ins}(x, \text{Empty}) &= \text{Node}(\text{Empty}, x, \text{Empty}) \\
\text{Ins}(x, \text{Node}(t_1, y, t_2)) &= \text{Node}(t_1, y, \text{Ins}(x, t_2)) \quad \text{if } x > y \\
&= \text{Node}(\text{Ins}(x, t_1), y, t_2) \quad \text{if } x \leq y
\end{align*}
\]

These equations hold, for all integer values \(x, y\) and all tree values \(t_1, t_2\) by definition of Ins.

\[
\text{Ins}(4, 3) = \begin{array}{c}
1 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
1 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
1 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
3 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
1 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
6 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
2 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
5 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
4 \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
T \text{ sorted} \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
= \begin{array}{c}
\text{Ins}(4, T) \text{ sorted} \\
\_ \\
\_ \\
\_ \\
\_ \\
\_ \\
. \\
. \\
. \\
. \\
. \\
\end{array}
\]
merging trees

Merge : tree * tree -> tree

REQUIRES $t_1$ and $t_2$ are sorted trees

ENSURES Merge($t_1$, $t_2$) = a sorted tree
consisting of the items of $t_1$ and $t_2$

$$\text{Merge (Node}(L_1,x,R_1), t_2) = ???$$

We could split $t_2$ into two subtrees $(L_2, R_2)$,
then do Node(Merge($L_1$, $L_2$), $x$, Merge($R_1$, $R_2$))

But we need to stay sorted and not lose data…

… so our split should use $x$ and
build $(L_2, R_2)$ so that $L_2 \leq x \leq R_2$ …
splitting a tree

SplitAt : int * tree -> tree * tree

REQUIRES t is a sorted tree

ENSURES SplitAt(x, t) =
   a pair of sorted trees (u₁, u₂) such that
      u₁ ≤ x ≤ u₂ and u₁, u₂ is a perm of t

Not completely specific, but that’s OKAY!
**SplitAt**

\[
\text{SplitAt} : \text{int} * \text{tree} -> \text{tree} * \text{tree}
\]

If \( t \) is sorted,

\[
\text{SplitAt}(x, t) = \text{a pair of trees } (u_1, u_2) \text{ such that }
\]

\[
\text{every integer in } u_1 \text{ is } \leq x,
\]

\[
\text{every integer in } u_2 \text{ is } \geq x,
\]

\[
\text{and } u_1, u_2 \text{ is a perm of } t.
\]

**Any ideas??**
Plan

Define SplitAt(t) using *structural recursion*

- SplitAt(x, Node(t₁, y, t₂)) should
  - *compare* x and y
  - call SplitAt(x, -) on a *subtree*
  - build the result
fun SplitAt(x, Empty) = (Empty, Empty)

| SplitAt(x, Node(t1, y, t2)) = |
| \_ case compare(y, x) of |
| \_ GREATER => |
| \_ let val (l1, r1) = SplitAt(x, t1) in (l1, Node(r1, y, t2)) end |
| \_ _ => |
| \_ let val (l2, r2) = SplitAt(x, t2) in (Node(t1, y, l2), r2) end |

SplitAt : int * tree -> tree * tree

REQUIRES t is a sorted tree
ENSURES SplitAt(x, t) = a pair of sorted trees (u₁, u₂) such that u₁ ≤ x ≤ u₂ and u₁, u₂ is a perm of t
**Merge**

Merge : tree * tree -> tree

REQUIRES $t_1$ and $t_2$ are sorted trees

ENSURES Merge($t_1$, $t_2$) = a sorted tree consisting of the items of $t_1$ and $t_2$

```plaintext
fun Merge (Empty, t2) = t2

let
    val (l2, r2) = SplitAt(x, t2)
  in
    Node(Merge(l1, l2), x, Merge(r1, r2))
  end

(as we promised!)
```
Merge

Merge : tree * tree -> tree

REQUIRES  

ENSURES  

fun Merge (Empty, t2) = t2

let
    val (l2, r2) = SplitAt(x, t2)
in
    Node(Merge(l1, l2), x, Merge(r1, r2))
end
**depth lemma**

For all trees $t$ and integers $x$,

\[
\text{depth}(\text{Ins}(x, t)) \leq \text{depth } t + 1
\]

For all trees $t$, if $\text{SplitAt}(y, t) = (t_1, t_2)$, then

\[
\text{depth}(t_1) \leq \text{depth } t \land \text{depth}(t_2) \leq \text{depth } t
\]

For all trees $t_1$ and $t_2$,

\[
\text{depth} (\text{Merge}(t_1, t_2)) \leq \text{depth } t_1 + \text{depth } t_2
\]

(no, we won’t prove this!)
fun Msort Empty = Empty
   | Msort (Node(t1, x, t2)) =
         Ins (x, Merge(Msort t1, Msort t2))
Correct?

• **Q:** How to *prove* that **Msort** is correct?
  **A:** Use structural induction.

• First prove that the *helper functions* **Merge**, **SplitAt**, **Ins** are correct. Again use structural induction.

• The helper specs were carefully chosen to make the proof of **Msort** straightforward.
  (An easy structural induction, using the proven facts about helpers.)
Mergesort

Msort : tree -> tree

REQUIRES true
ENSURES Msort(t) = a sorted tree consisting of the items of t

fun Msort Empty = Empty
|
| Msort (Node(t1, x, t2)) =
| Ins (x, Merge(Msort t1, Msort t2))
example

val A = list2tree [4,1,2]
val B = list2tree [3,5,0]
val T = Node(A, 42, B)
val S = Msort T