last time

- Sorting a list of integers
  - insertion sort
  - merge sort
- Specifications and proofs
  - helper functions that really help
back to 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
split and merge

For all lists $L$ with length $n > 1$

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

where $A$ and $B$ have length $\approx n \div 2$

For all sorted lists $A$ and $B$,

$\text{merge}(A, B) =$ a sorted permutation of $A @ B$
**Theorem**
For all L:int list,
msort(L) = a <-sorted permutation of L.

**Proof:** 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

• Let length(L) > 1. Then

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

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

• \text{msort } A \text{ and } \text{msort } B \text{ are sorted lists (why?)}
• \text{merge}(\text{msort } A, \text{msort } B) = \text{a sorted list (why?)}
• \text{merge}(\text{msort } A, \text{msort } B) = \text{a perm of } L \text{ (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 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 \([x] \Rightarrow^* (\text{fn} \ldots \Rightarrow \ldots) (\text{msort} \ [x], \text{msort} \ [\ ])\)

\[\text{infinite computation}\]

What happens if we try to \textbf{prove} that

For all \(L:\text{int list}\),

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

• Every function needs a spec
• Every spec needs a proof
• Recursive functions need inductive proofs
  • 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 proof for msort relied only on the specification proven for split (and the specification proven for merge)

We can replace split by any function that satisfies this specification, and the msort proof will still be valid!

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;
• 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

- **sorting** for integer lists
- **specifications** and **correctness**
- … but what about **efficiency**?
An estimate of the *number of evaluation steps*, on a *sequential* processor.

- basic ops count as *one* unit of work
- *add* the work for sub-expressions
- some ops, like @, incur an extra cost
work rules

$W(e)$ is the work for $e$

$W(e_1 + e_2) = W(e_1) + W(e_2) + 1$

$W(e_1 \odot e_2) = W(e_1) + W(e_2) + \text{length } e_1 + 1$

$W(\text{let val } x = e_1 \text{ in } e_2 \text{ end}) = W(e_1) + W(e_2) + 1$

\hspace{1cm} data dependency: value of $e_1$ used by $e_2$

$W(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) = W(e_0) + \max(W(e_1), W(e_2)) + 1$

\hspace{1cm} control dependency: value of $e_0$ determines branch

$W(e_1, e_2) = W(e_1) + W(e_2) + 1$
work to split

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

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

\[
W_{\text{split}}(n) = 1 \quad \text{for } n=0, 1 \\
W_{\text{split}}(n) = 1 + W_{\text{split}}(n-2) \quad \text{for } n>1
\]

$W_{\text{split}}(n)$ is $O(n)$
Let \( W_{\text{merge}}(n) \) = work of \( \text{merge}(A, B) \)
when \( A, B \) are values and
\[
\text{length}(A) + \text{length}(B) = n
\]

\( W_{\text{merge}}(n) \) is \( \Theta(n) \)
work to msort

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 \( L \) is a list value of length \( n \)

When length(L) = \( n > 1 \)
and \((A, B)\) is the value of split \( L \),
A and \( B \) have length \( \approx n \text{ div } 2 \)
work to msort

\[
\text{fun } \text{msort } [ ] = [ ] \mid \text{msort } [x] = [x] \\
\mid \text{msort } L = \text{let } \text{val } (A, B) = \text{split } L \text{ in } \\
\quad \text{merge } (\text{msort } A, \text{msort } B) \text{end}
\]

\[
W_{\text{msort}}(n) = 1 \quad \text{for } n=0,1
\]

\[
W_{\text{msort}}(n) = W_{\text{split}}(n) + 2W_{\text{msort}}(n \div 2) + W_{\text{merge}}(n) + 1
\]

\[
= O(n) + 2W_{\text{msort}}(n \div 2) \quad \text{for } n>1
\]

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

An estimate of the number of evaluation steps, assuming unlimited parallelism

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

independent:
  no data or control dependency, so can be evaluated in parallel
span rules

$S(e)$ is the span for $e$

\[ S(e_1 + e_2) = \max(S(e_1), S(e_2)) + 1 \]

\[ S(e_1@e_2) = \max(S(e_1), S(e_2)) + \text{length } e_1 + 1 \]

\[ S(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) = S(e_0) + \max(S(e_1), S(e_2)) + 1 \]

\[ \text{control dependency: test before branch} \]

\[ S(\text{let val } x = e_0 \text{ in } e_1 \text{ end}) = S(e_0) + S(e_1) + 1 \]

\[ \text{data dependency: value of } e_0 \text{ used by } e_1 \]

\[ S(e_1, e_2) = \max(S(e_1), S(e_2)) + 1 \]
work 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 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**

• But we **could** use parallel evaluation for the calls to msort A and msort B

**How would this affect runtime?**
span

```ml
msort L = let val (A, B) = split L in
    merge (msort A, msort B) end
```

Let $S_{msort}(n) = \text{span of } msort \ L \text{ when length } L = n$

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

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

$S_{msort}(n)$ is $O(n)$
summary

- \texttt{msort(L)} has $O(n \log n)$ work, $O(n)$ span
- So the potential \textit{speed-up} factor from parallel evaluation is $O(\log n)$

… \textit{in principle}, we can \textit{speed up} \texttt{mergesort} on lists \textit{by a factor of} $\log n$

To do any better, we need a \textit{different data structure}…
Trees are better than lists for parallel evaluation

- Sorting a tree
  - Specifications and proofs
  - Asymptotic analysis

Insertion
“Parallel” Mergesort
trees

datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

• A user-defined type constructor - tree
• With values Empty and Node

Empty : 'a tree
Node : 'a tree * 'a * 'a tree -> 'a tree

A value of type int tree is a binary tree with integers at its nodes
A tree value is either \texttt{Empty} or has the form \texttt{Node(t}_1, \ x, \ t_2), where \ t_1 \ and \ t_2 \ are tree values and \ x \ is a value.

**Contrast with lists:**

A list value is either \texttt{nil} or has the form \texttt{x::L}, where \texttt{L} is a list value and \texttt{x} is a value.
To define a function $F$ on all trees:

- **Base clause:**
  
  Define $F(\text{Empty})$

- **Inductive clause:**
  
  Define $F(\text{Node}(t_1, x, t_2))$
  using $x$ and $F(t_1)$ and $F(t_2)$.

That’s enough! Why?

Contrast with structural definition for lists
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 values \( x \).

That’s enough! Why?

Contrast with structural induction for lists
## Tree Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matching Values</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(p₁, p, p₂) matches t iff
t is Node(t₁, v, t₂) such that
p₁ matches t₁, p matches v, p₂ matches t₂

and combines all the bindings

Node(A, x, B) matches 3

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

fun Full(x:int, n:int): 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 \( t \),
\[
    \text{size}(t) = \text{a non-negative integer}
\]

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)

```haskell
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 \( \text{depth} \) to prove properties or analyze efficiency.
exercises

• Prove that for all $n \geq 0$

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

Full(42, 3)

size is 7

depth is 3
in-order traversal

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

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,

\[ \text{length (inord T)} = \text{size T} \]

For all lists \( L_1, L_2 \) of the same type

\[ \text{length (} L_1 \text{ @ } L_2 \text{)} = \text{length } L_1 + \text{length } L_2 \]
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 $n$

  depth(T) = n, size(T) = $2^{n-1}$

• $W_{\text{inord}}(0) = 1$

• $W_{\text{inord}}(n) = 2W_{\text{inord}}(n-1) + O(2^n)$, for $n>0$

$W_{\text{inord}}(n)$ is $O(n2^n)$

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

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

inorder : 'a tree * 'a list -> 'a list

fun inorder (Empty, L) = L

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

Theorem

For all trees $T$ and lists $L$,
inorder $(T, L) = (\text{inord } T) \ @ \ L$

The work for inorder$(T, L)$,
when $T$ is a full tree of depth $n$,
is $O(2^n)$