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 \texttt{msort} was easy, because of \texttt{split} and \texttt{merge}
so far

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

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

Many ML constructs use left-to-right evaluation
work rules

W(e) is the work for e

\[ W(e_1 + e_2) = W(e_1) + W(e_2) + 1 \]
\[ W(e_1 :: e_2) = W(e_1) + W(e_2) + 1 \]
\[ W(e_1 @ e_2) = W(e_1) + W(e_2) + \text{length } e_1 + 1 \]
\[ W(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) = W(e_0) + \max(W(e_1), W(e_2)) + 1 \]
\[ W((f \ x, f \ y)) = W(f \ x) + W(f \ y) + 1 \]

**sequential** evaluation:

f x, then f y, then build pair
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*

**GOVERNMENT HEALTH WARNING**
*ML constructs are inherently sequential*
span

S(e) is the span for e

S(e₁+e₂) = S(e₁) + S(e₂) + 1

S(e₁::e₂) = S(e₁) + S(e₂) + 1

for sequential e
work = span

S(e₁@e₂) = S(e₁) + S(e₂) + length e₁ + 1

S(if e₀ then e₁ else e₂) = S(e₀) + max(S(e₁),S(e₂)) + 1

S ((f x, f y)) = max (S(f x), S(f y)) + 1

W ((f x, f y)) = W(f x) + W(f y) + 1

for parallelizable e
span < work
example

fun fib (n:int) : int = 
  if n <= 1 then 1 else
  let
    val (x, y) = (fib(n-1), fib(n-2))
  in
    x + y
  end

• $W_{fib}(n) = W_{fib}(n-1) + W_{fib}(n-2) + 1$
• $S_{fib}(n) = \max(S_{fib}(n-1), S_{fib}(n-2)) + 1$

$W_{fib}(n)$ is exponential
$S_{fib}(n)$ is linear
fun W_fib n = 
  if n <= 1 then 1 else W_fib (n-1) + W_fib (n-2) + 1;

val W_fib = fn : int -> int

- [W_fib 10, W_fib 20, W_fib 30, W_fib 40];
val it = [177,21891,2692537,331160281] : int list

fun S_fib n = 
  if n <= 1 then 1 else Int.max(S_fib (n-1), S_fib (n-2)) + 1;

val S_fib = fn : int -> int

- [S_fib 10, S_fib 20, S_fib 30, S_fib 40];
val it = [10,20,30,40] : int list
caveat

• In ML pairs are not parallel, execution is sequential
  • So in ML, \( \text{fib } n \) has work = span

• The span speed-up is only achievable on parallel hardware

• Nevertheless it’s important for you to be aware of the potential for speed-up!

  Evaluating \( \text{fib } n \) on parallel hardware causes lots of duplicated effort ...
  there are faster ways to \( \text{fib} \), even sequentially!
**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*
work

$$\text{msort } L = \text{let } \text{val } (A, B) = \text{split } L \text{ in }$$
$$\text{merge } (\text{msort } A, \text{msort } B) \text{ end}$$

when length $L > 0$

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

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

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

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

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

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

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

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

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

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

\[ O(n) \subset O(n \log n) \]

mergesort is potentially worth parallelizing
summary

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

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

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

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
integer 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 definition

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 integers \( x \).

That’s enough! Why?

Contrast with structural induction for lists
# tree patterns

Empty

Node(p₁, p, p₂)

<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($p_1$, $p$, $p_2$) 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 3 and binds $x$ to 3,
A to Node(Empty,4,Empty)
B to Node(Empty,2,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
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 *induction on size* to prove correctness.
fun depth Empty = 0
  | depth (Node(t1, _, t2)) = Int.max(depth t1, depth t2) + 1

Can prove by structural induction that for all trees $t$, 
$\text{depth}(t)$ = 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 $n \geq 0$

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

\[
\text{Full}(42, 3)
\]

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

left before root before right

[2, 42, 3, 21, 7]
fun inord Empty = []
| inord (Node(t1, x, t2)) = inord t1 @ (x :: inord t2)

For all trees T,
length (inord T) = size T

For all lists L₁, L₂ of the same type
length (L₁ @ L₂) = length L₁ + length L₂
work analysis

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

• Let $W_{inord}(n)$ be the work to evaluate inord(T) when T is a full binary tree of depth n

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

• $W_{inord}(0) = 1$

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

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

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