Parallel Algorithms

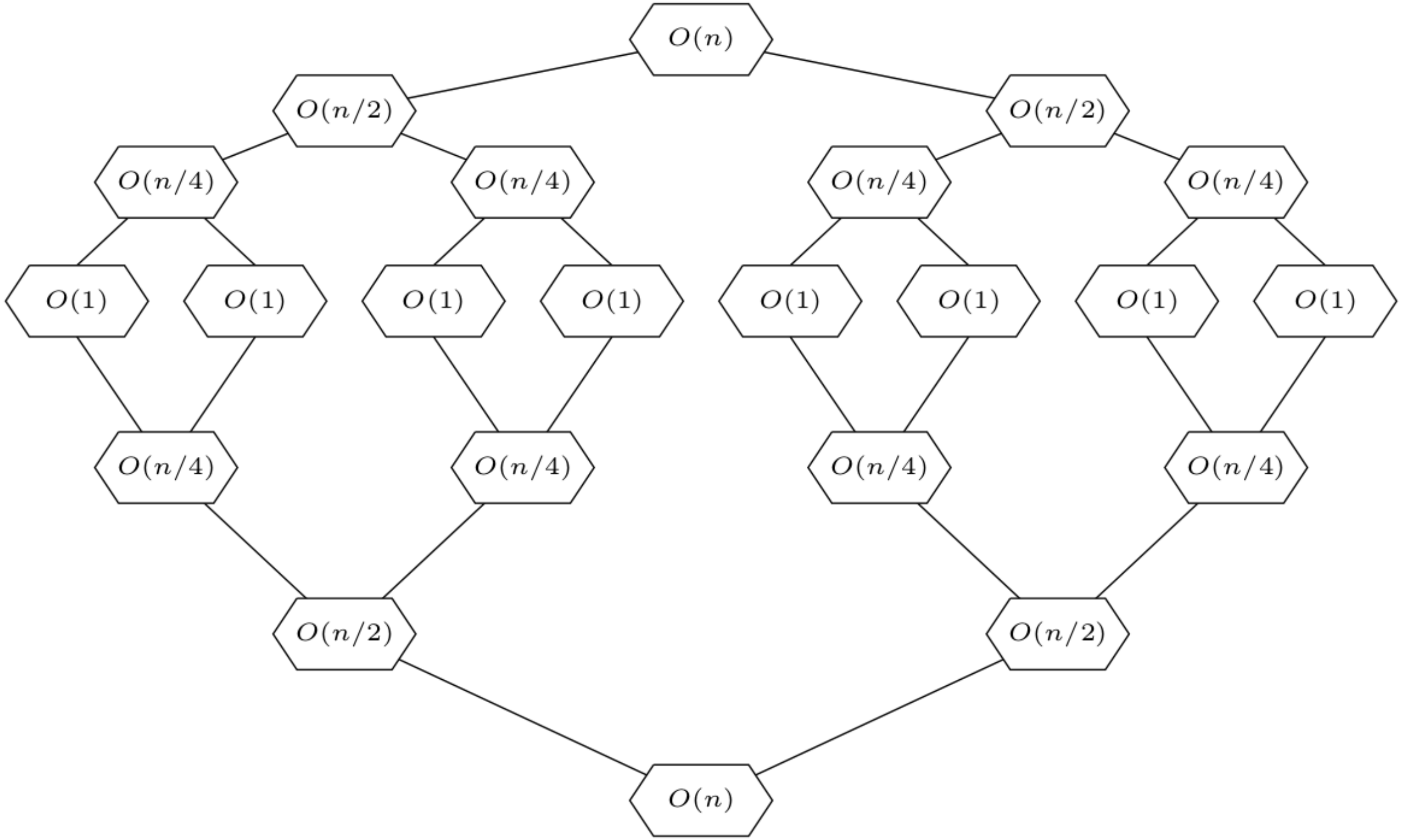
15-150 M21

Lecture 0721
21 July 2021

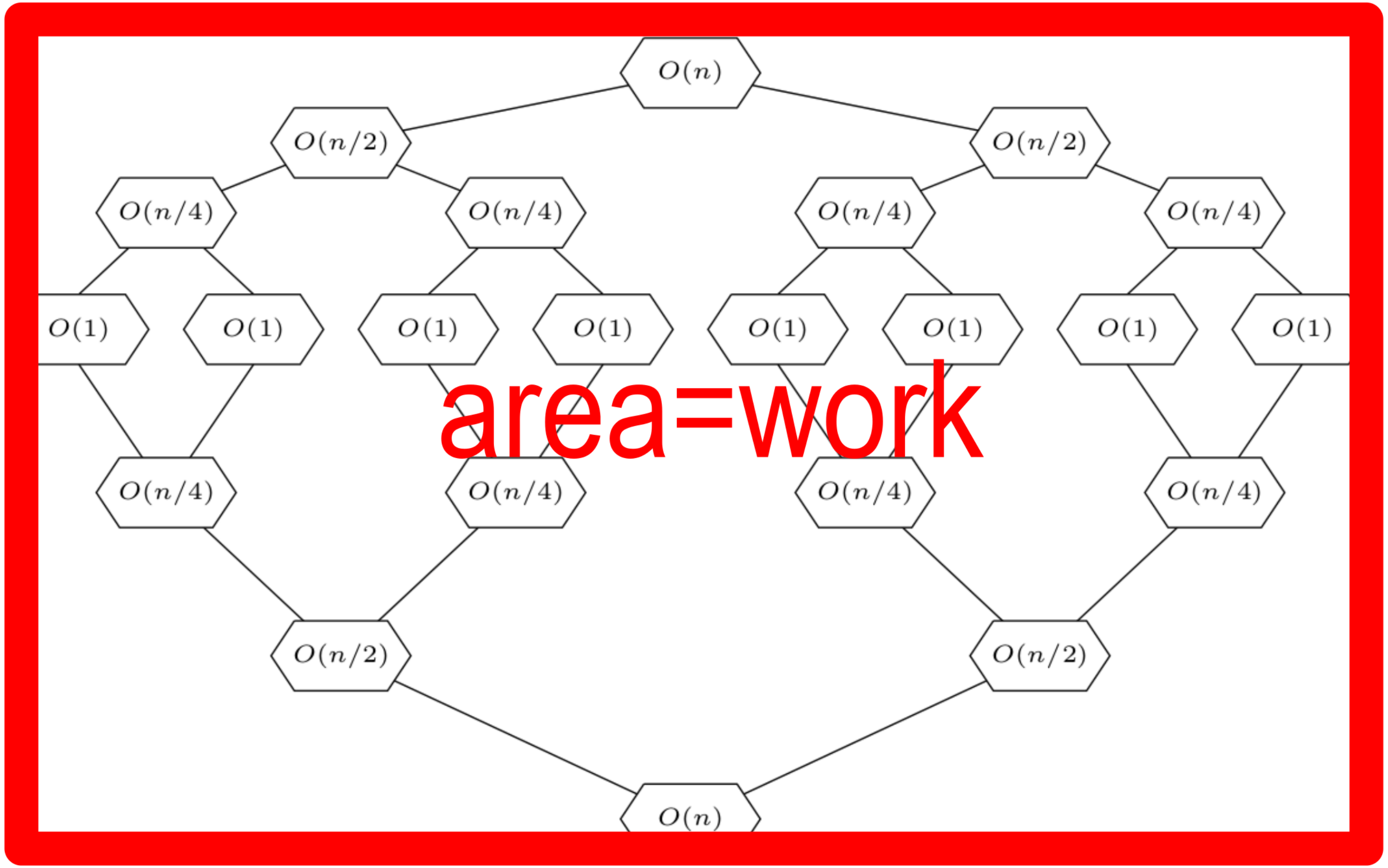
Review: Sequences

- We have a signature `SEQUENCE` and *some* structure `Seq` opaquely ascribing to it, satisfying the specs given in the sequence reference document
- `SEQUENCE` contains an abstract type `'a seq`, representing **sequences**
- Sequences combine the best of both lists and trees: they are linearly-ordered and easily indexed (like lists) but highly parallel (like trees)

Cost graph analysis

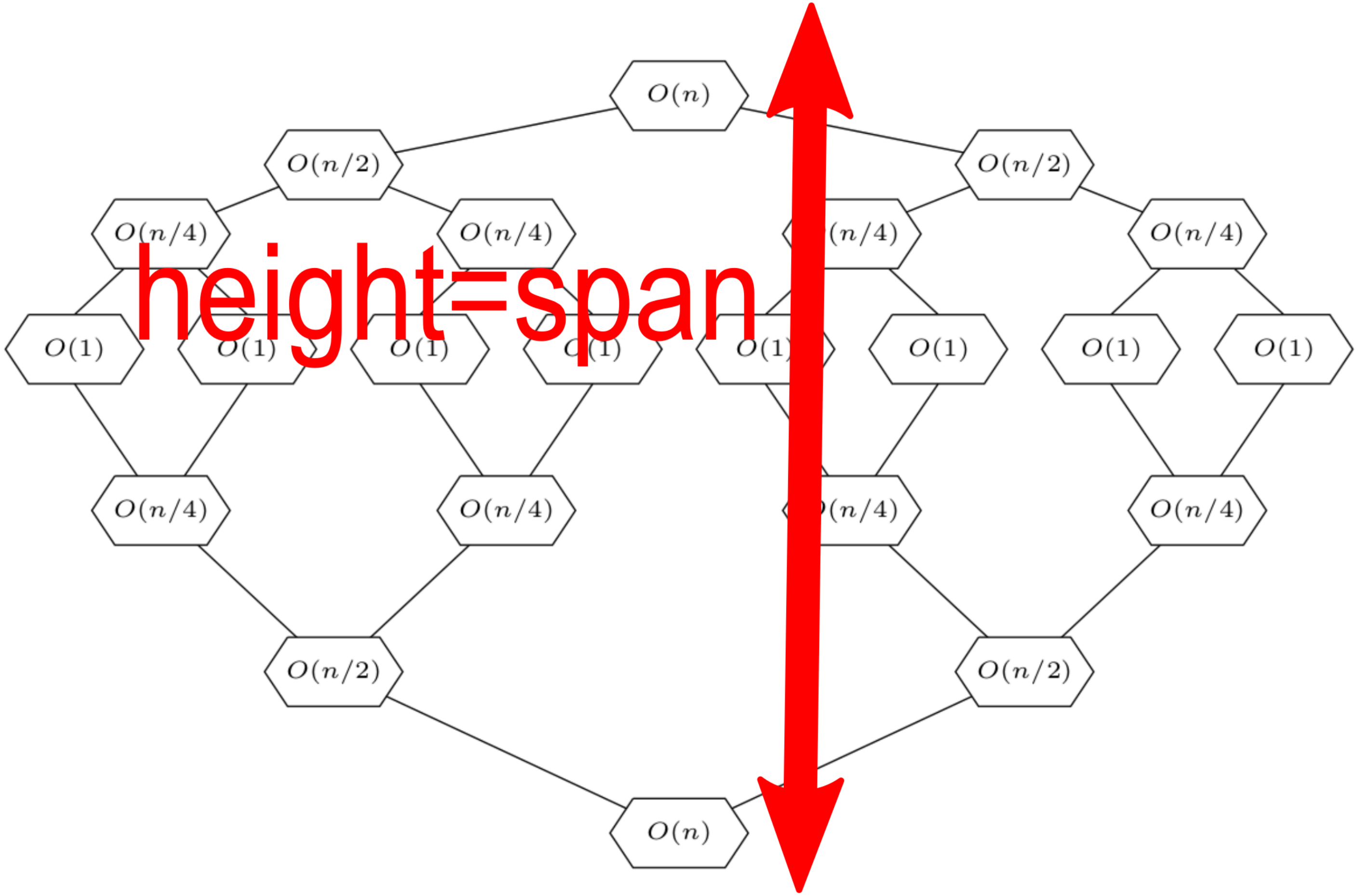


Cost graph analysis



area=work

Cost graph analysis



0 Spec/Code/Cost Graph/Big-O

`interleave : 'a seq * 'a seq -> 'a seq`

REQUIRES: true

ENSURES: `interleave(S1,S2)` $\Longrightarrow$ `S` where $|\texttt{S}| = 2 \cdot \min(|\texttt{S1}|, |\texttt{S2}|)$ and `S` consists of alternating elements of `S1` and `S2`.

0721.0 (seqFns.sml)

```
fun interleave (S1,S2) =
  let
    val n =  Int.min(Seq.length S1,
                     Seq.length S2)

    fun select 0 i = Seq.nth S1 i
      | select _ i = Seq.nth S2 i
  in
    Seq.tabulate
      (fn i => select (i mod 2) (i div 2))
      (n*2)
  end
```

Key Skill:

Annotating with cost bounds

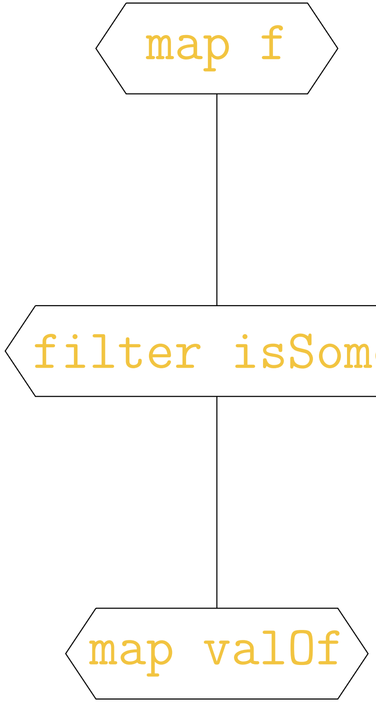
Demonstration:

Cost Graph Analysis

Recall: mappartial

0721.1 (seqFns.sml)

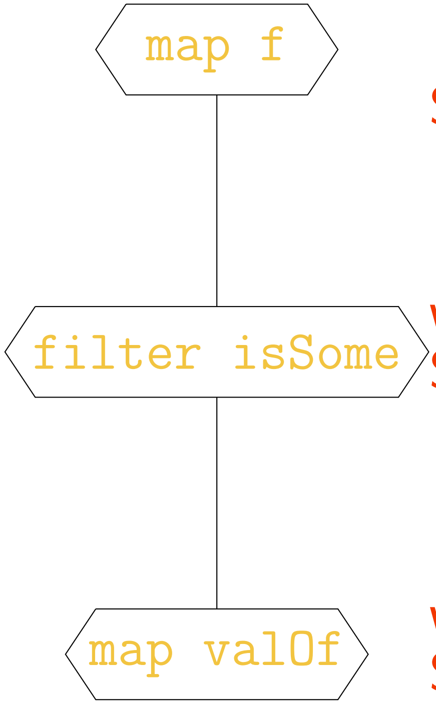
```
infix |>
fun x |> f = f x

fun mappartial f S =
      S |> (Seq.map f)
        |> (Seq.filter Option.isSome)
        |> (Seq.map Option.valOf)
```



Work: $O(n)$
Span $O(\log n)$

Work: $O(n)$
Span $O(1)$

Demonstration:
HOF cost graph analysis



`map f`

Work: $\sum_{i=0}^{n-1} W_{\texttt{f(S[i])}}$

Span: $\max_{i=0}^{n-1} S_{\texttt{f(S[i])}}$

`filter isSome`

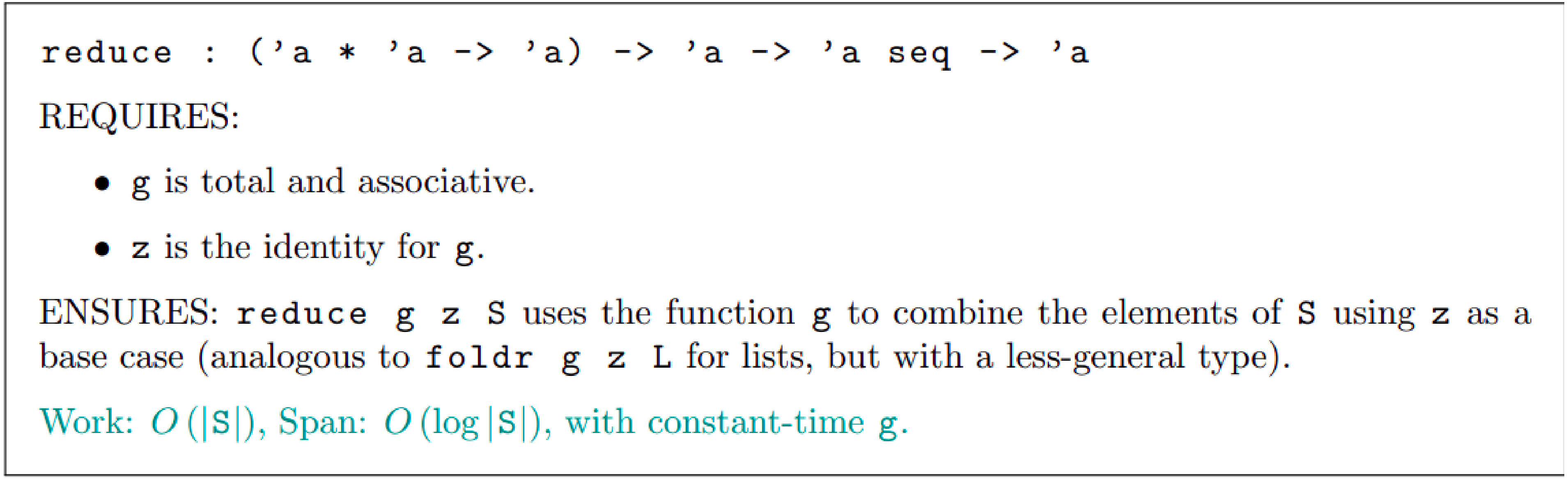
Work: $O(n)$
Span $O(\log n)$

`map valOf`

Work: $O(n)$
Span $O(1)$

5-minute break

1 Reduce



`reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a`

REQUIRES:

- `g` is total and associative.
- `z` is the identity for `g`.

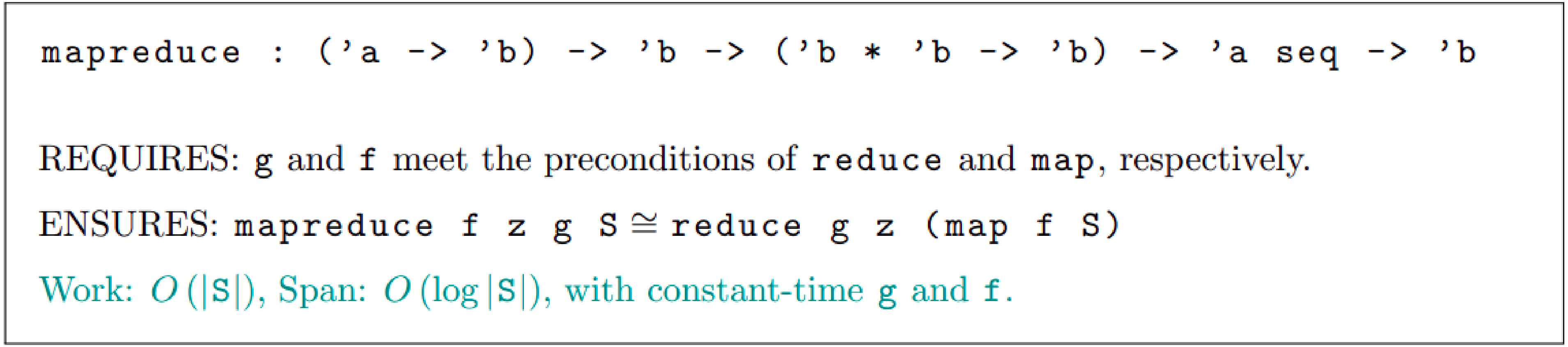
ENSURES: `reduce g z S` uses the function `g` to combine the elements of `S` using `z` as a base case (analogous to `foldr g z L` for lists, but with a less-general type).

Work: $O(|S|)$, Span: $O(\log|S|)$, with constant-time `g`.

0721.2 (seqFns.sml)

```
(* O(|S|) work, O(log |S|) span *)
val sum = Seq.reduce op+ 0
```

Map and reduce, together at last!



`mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b`

REQUIRES: `g` and `f` meet the preconditions of `reduce` and `map`, respectively.

ENSURES: `mapreduce f z g S` ≅ `reduce g z (map f S)`

Work: $O(|S|)$, Span: $O(\log|S|)$, with constant-time `g` and `f`.

0721.3 (seqFns.sml)

```
(* O(|S|) work, O(log |S|) span *)
val sumNonneg =
  Seq.mapreduce (fn x => Int.max(x,0)) 0 op+
```

Divide-and-Conquer

To analyze `reduce` and `mapreduce` with non-constant-time `g`, we need to know more about how they're implemented.

You may assume `reduce` is implemented according to the following **divide-and-conquer algorithm**: to calculate `reduce g z S`,

1. Split `S` into two halves, `S1` and `S2` (i.e. `Seq.append(S1,S2)` $\cong$ `S`)
2. Recursively evaluate `reduce g z S1` and `reduce g z S2` to values `v1` and `v2`, respectively
3. Calculate `g(v1,v2)`

With base cases:

- `reduce g z` $\langle\rangle \Longrightarrow$ `z`
- `reduce g z` $\langle$`x`$\rangle \Longrightarrow$ `g(x,z)`

Divide-and-Conquer

To analyze `reduce` and `mapreduce` with non-constant-time `g`, we need to know more about how they're implemented.

You may assume `mapreduce` is implemented according to the following **divide-and-conquer algorithm**: to calculate `mapreduce f z g S`,

1. Split `S` into two halves, `S1` and `S2` (i.e. `Seq.append(S1,S2)` $\cong$ `S`)
2. Recursively evaluate `mapreduce f z g S1` and `mapreduce f z g S2` to values `v1` and `v2`, respectively
3. Calculate `g(v1,v2)`

With base cases:

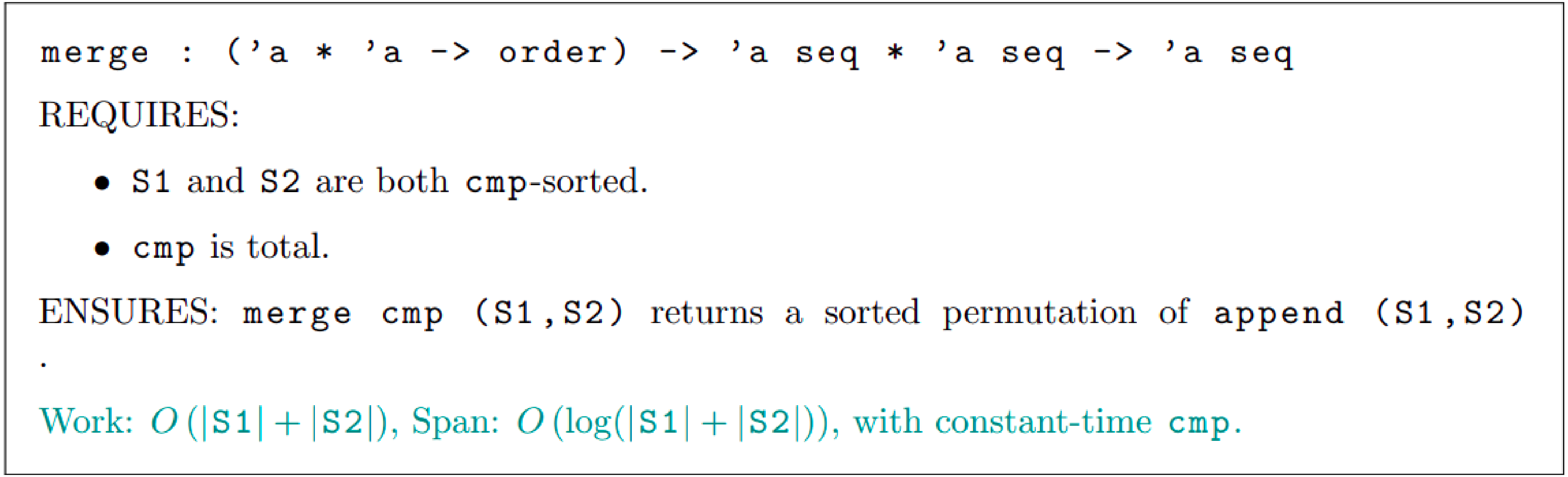
- `mapreduce f z g` $\langle\rangle \Longrightarrow$ `z`
- `mapreduce f z g` $\langle$`x`$\rangle \Longrightarrow$ `g(f x,z)`

Visual Aid:

`reduce` and `mapreduce` cost graphs

We’ve seen a divide-and-conquer algorithm…

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



`merge : ('a * 'a -> order) -> 'a seq * 'a seq -> 'a seq`

REQUIRES:

- `S1` and `S2` are both `cmp`-sorted.
- `cmp` is total.

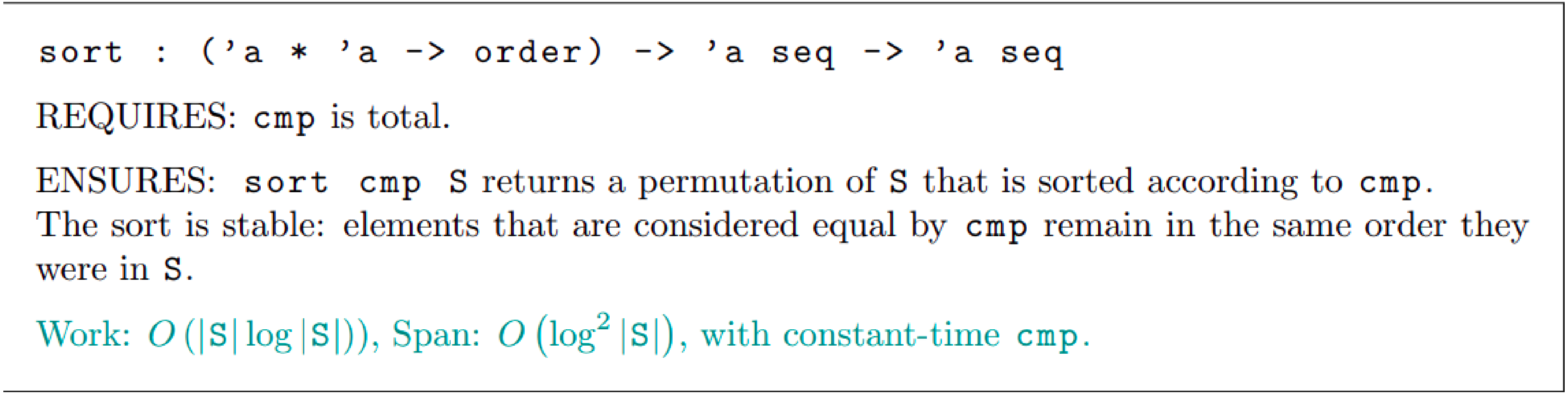
ENSURES: `merge cmp (S1,S2)` returns a sorted permutation of `append (S1,S2)`
.

Work: $O\left(|\texttt{S1}| + |\texttt{S2}|\right)$, Span: $O\left(\log(|\texttt{S1}| + |\texttt{S2}|)\right)$, with constant-time `cmp`.

```
fun msort cmp =
  mapreduce singleton (empty()) (merge cmp)
```

Demonstration:

`msort` Cost Graph Analysis



`sort : ('a * 'a -> order) -> 'a seq -> 'a seq`

REQUIRES: `cmp` is total.

ENSURES: `sort cmp S` returns a permutation of `S` that is sorted according to `cmp`. The sort is stable: elements that are considered equal by `cmp` remain in the same order they were in `S`.

Work: $O\left(|\texttt{S}| \log |\texttt{S}|\right)$, Span: $O\left(\log^2 |\texttt{S}|\right)$, with constant-time `cmp`.

Summary

- We can analyze sequence functions using cost graphs
- including HOFs with non-constant-time arguments
- We can use the divide-and-conquer nature of `reduce` to understand its asymptotic complexity

Next Time

- Fun and games (using sequences!)

Thank you!