Lesson 17
SEQUENCES

July 20, 2023

1 - Fundamental Data Structures

We previously claimed that, in computer science, there are only a few fundamental data structures, from which everything else pretty much is derived from.

One of these items is the `list`. The other is the `tree`. Many things, such as priority queues, tries, queues, and sets can be implemented with just these two ideas.

The third is the **array**, which we have given substantially less treatment of, so far in this course. That changes today.

Def **Arrays** are a kind of mutable data structure that have a fixed number of elements, all of the same type. These elements support *indexing*, which allows access of any given element by its position in the array, in constant time.

Arrays are usually associated with lower-level programming languages, meaning that they are mutable by default – indices of the array can be rewritten and assigned, which will change future accesses.

Constant time access is cool. We can skip the mutability.

Instead of arrays, we will discuss a similar type of data structures called **sequences**.

Def **Sequences**[1] are a kind of **immutable** data structure that have a fixed size of elements, all of the same type. These elements support *indexing*, which allows access of any given element by its position in the array, in constant time.

The key is that sequences are not mutable, however! Changing an given element of a sequence entails producing a new sequence entirely. This is how we will be able to take advantage of the various benefits of arrays, without falling for the trap of mutability.

Note Other than the fact that sequences are immutable, you can think of them as being implemented as arrays.

[1] Worth noting that as far as I can tell, "sequences" to refer to essentially immutable arrays is a CMU thing. Other people don't use this terminology.

But how are sequences implemented?

The answer: **It's a secret**.

No, actually. For our purposes, we are taking advantage of the module system, and using a library which implements sequences as an abstract type, which prevents us from knowing anything about the way that it is implemented under the hood. This means we cannot pattern-match, nor can we interact with sequences in any way which is not prescribed to us by the sequences interface.

The signature of the `Seq` module is available on the next few slides (and online, at this link):

```
signature SEQUENCE =
  sig
    type 'a t
    type 'a seq = 'a t (* abstract *)

    exception Range of string


    (* Constructing a Sequence *)

    val empty : unit -> 'a seq
    val singleton : 'a -> 'a seq
    val tabulate : (int -> 'a ) -> int -> 'a seq
    val fromList : 'a list -> 'a seq

    (* ... *)
```

```
(* ... *)
    (* Deconstructing a Sequence *)

    val nth : 'a seq -> int -> 'a
    val null : 'a seq -> bool
    val length : 'a seq -> int
    val toList : 'a seq -> 'a list
    val toString : ('a -> string) -> 'a seq -> string
    val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool

    (* Simple Transformations *)

    val rev : 'a seq -> 'a seq
    val append : 'a seq * 'a seq -> 'a seq
    val flatten : 'a seq seq -> 'a seq
    val cons : 'a -> 'a seq -> 'a seq
(* ... *)
```

```
(* ... *)
  (* Combinators and Higher - Order Functions *)

  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
  val zip : ('a seq * 'b seq) -> ('a * 'b) seq
  val zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq

  (* Indexing-Related Functions *)
  val enum : 'a seq -> (int * 'a) seq
  val mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq
  val update : ('a seq * (int * 'a)) -> 'a seq
  val inject : 'a seq * (int * 'a) seq -> 'a seq

  val subseq : 'a seq -> int * int -> 'a seq
  val take : 'a seq -> int -> 'a seq
  val drop : 'a seq -> int -> 'a seq
  val split : 'a seq -> int -> 'a seq * 'a seq

  (* Sorting and Searching *)

  val sort : ('a * 'a -> order) -> 'a seq -> 'a seq
  val merge : ('a * 'a -> order) -> 'a seq * 'a seq -> 'a seq
  val search : ('a * 'a -> order) -> 'a -> 'a seq -> int option
  (* ... *)
end
```

OK, that's a lot. Moreover, there isn't any description on what each function does![2]

If you glance at the names of each function, however, it doesn't look terribly different than the list library (though a little better equipped). What gives?

Sequences don't offer us anything that can't be done with lists, in terms of their structure. We could very well be using lists instead. The difference will be in each function's **cost**, as sequences admit a different cost model.

Key In particular, **sequences are very parallelizable**.

For instance, we can execute `Seq.map f` within constant time, for a constant function `f`.

[2] There is at the online reference, though.

We mentioned previously that sequences will admit constant time access to each element, whereas in a list you must perform $O(i)$ work to access the ith element.

We can implement the `nth` function for lists as such:

```
fun nth ([], 0)    = raise Subscript
  | nth (x::_, 0) = x
  | nth (x::xs, n) = nth (xs, n - 1)
```

Other advantages of sequences are that length can be computed in constant time, and that they are parallel-friendly, meaning that bulk operations (like `map`, `fold`, and `filter`) can be done without having to always pay a linear cost up front.

For all those advantages, however, we have some disadvantages that come as a consequence. The most prominent one, from a programming standpoint, is that you cannot pattern match upon a sequence. This means that something as simple as the following, with lists:

```
case L of
  []     => (* 1 *)
| x::xs  => (* 2 *)
```

...requires the following, for a sequence `S`:

```
case Seq.length S of
  0 => (* 1 *)
| _ =>
  let
    val (x, xs) = (Seq.nth S 0, Seq.drop S 1)
  in
    (* 2 *)
  end
```

Tradition dictates that I mention: *this is disgusting*.[3]

[3] Time permitting, at the end of this lecture we can discuss a nicer way of doing this. The point stands.

The other disadvantage of sequences is the same as its primary benefit: *sequences are designed with parallelism in mind*. This means that bulk operations, as in, operations which require dealing with a fixed number of elements all at once, are very easy, but sequential operations are slow.

This is most salient through the fact that **cons is expensive**. For sequences, consing an element onto the sequence takes linear time in the length of the sequence, as opposed to constant time for lists.

Key Fact The root cause for this is that when you cons an element onto a sequence, you must create a brand new sequence, meaning you must copy over everything that was in the old sequence! This is an easy linear cost.[4]

[4] Usually, talk of things like "copying" and "memory" are beneath us. But this is a case where it actually matters, because it shows up in our cost bound.

When discussing sequences, we will usually textually represent them using the mathematical notation $\langle x_1, x_2, ..., x_n \rangle$, for the sequence with elements $x_1, ..., x_n$ at each index.

For instance, we might write that `Seq.map f` $\langle x_1, x_2, ...x_n \rangle$ has the cost of $\max_{x_i \in S}(W(\texttt{f xi}))$, meaning the maximum cost over any given application of `f` to an element of the sequence, and evaluates to the sequence $\langle \texttt{f x1}, \texttt{f x2}, ..., \texttt{f xn} \rangle$.

Before we can do a deeper dive into the cost of sequences, we need to come up with a conceptual model of how to think about the cost of sequence functions.

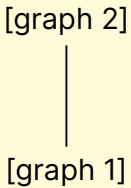
To that end, we need to discuss **cost graphs**.

2 - Cost Graphs

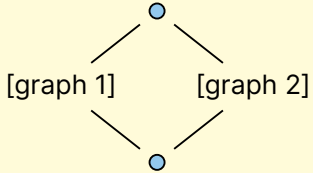
We use **cost graphs** to visually reason about the cost of a given operation of a sequence. Although we don't know specifically the underlying implementation of sequences, cost graphs still give us a way of reasoning about their cost.

Def A **cost graph** is a graph which indicates the cost of performing a certain function, but with visual indications for when parallelism can be achieved.

This is exactly the same as the idea of task dependency graphs, from the previous lecture on asymptotic analysis! We will boil in a few more sequence-specific things, however.



sequential composition of cost graphs



parallel composition of cost graphs

node denoting computation of `f`

Cost graphs are **inductively defined** by these three constructs.

Cost graphs always run from top to bottom, so while we can view them as directed, we will omit the arrows for brevity. Every graph also has a **source** and **sink**, which denote the starting and ending positions of executing that cost graph.

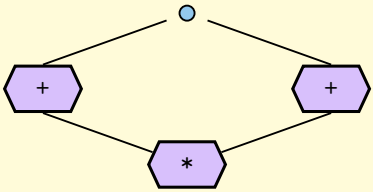
For the sequential case, the source and sink are just the source and sink of graphs 1 and 2, respectively.

For the parallel case, the source is the originating dot, which then **forks** to perform graph 1 and graph 2 in parallel, before **joining** back together.

The single node is itself both source and sink.

Cost graphs can be mixed and matched and put together! This means that although forking has a source and sink which are just before and after the two graphs in between, it can be sequentially merged with other graphs.

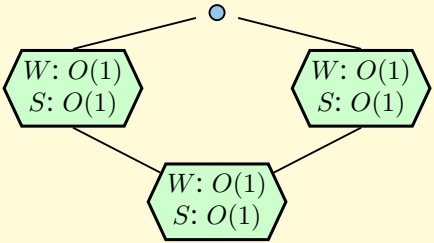
So for instance, we might say that the cost graph of the computation `(1 + 2) * (3 + 4)` has cost graph:



This comes from **forking** to perform each addition in parallel (in constant work and span), before **joining** to then perform the multiplication, also in constant work and span. Note that each purple node here has a constant cost.

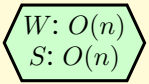
Once we know the cost of a particular computation node, we might also replace it with a work/span node (or **cost node**), like in the resulting graph:

Now, we can easily see that the work of this graph is just a constant $O(1)$, and the span is also.

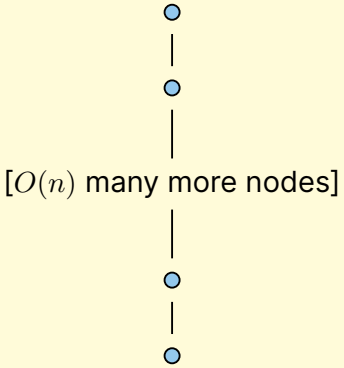


As before, the work of the entire cost graph is simply the work of each node, summed up. The span of a cost graph is the greatest cost among paths from the source to the sink.

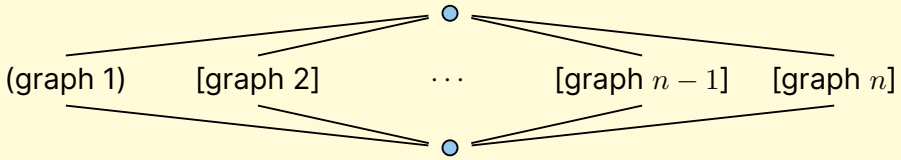
Actually, we don't necessarily need labeled cost nodes, as in the last slide, as we can rederive it from just the parallel and sequential composition cases. For instance, we could rederive this node:



as the cost graph on the right:

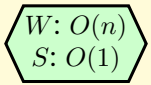


We will also use the following shorthand for many simultaneously forking paths:



This could be desugared as just many parallel composed graphs on top of each other, but nobody has time for that.

When all of the subgraphs are themselves constant cost nodes, this ends up being the same as the cost node:



That's pretty much all you need to know about cost graphs.

It's worth noting that all of the ideas present in this section are strictly conceptually interesting, and the actual encoding doesn't matter. You will not be tested on your ability to faithfully render a cost graph exactly, so long as you are approximately correct. Don't worry too much about the smaller details.

Now, we can move on to discussion of sequence functions in general.

3 - Sequence Functions

We don't have nearly enough time to go through every sequence function, so we won't. Fortunately, many functions within the sequence library are themselves derivable in terms of others, meaning that we only need to discuss a few fundamental functions to be able to understand the whole library.

For our purposes, the interesting functions to note will be `Seq.tabulate`, `Seq.map`, `Seq.filter`, and `Seq.reduce`. We will also have some brief notes on some other sequence functions, and their cost.

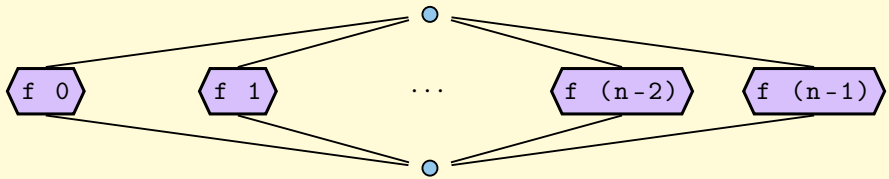
Note We will occasionally make reference to values as though we are inside of the `Seq` structure. That means we would ordinarily write `Seq.map` or `'a Seq.seq`, but for brevity I will write `map` and `'a seq`.

`tabulate : (int -> 'a) -> int -> 'a seq`
REQUIRES: For all $0 \leq$ `i` $<$ `n`, `f i` is valuable.
ENSURES: `tabulate f n` evaluates to $\langle$`f 0, f 1, ..., f (n - 1)`$\rangle$

This function actually exists for lists too, but it's rarely used, and not very parallelizable.

This specification means that, for instance, `Seq.tabulate (fn x => x) n` is equivalent to the sequence $\langle$`0, 1, ..., n - 1`$\rangle$.

Cost graph:



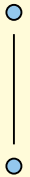
Recall that this cost graph indicates that each of the calls to `f` can be parallelized! This means that, for instance, given a constant time function `f`, `Seq.tabulate f n` is $O(n)$ work and $O(1)$ span.

This is much improved over the $O(n)$ span for the list equivalent. The main difference is that for lists, the list must be made by consing on elements repeatedly. For sequences, the entire sequence can be created at once, with each part independent, without any sequential operations at all.

As promised before, one of the most important functions for sequences will be `nth`, which allows constant time access to any given element of the sequence.

```
nth : 'a seq -> int -> 'a
```
REQUIRES: `true`
ENSURES: `nth S i` evaluates to the `i`th element of `S`. If `i` is negative or greater than or equal to `length S`, then `Range` is raised.

The cost graph just looks like:



It has exactly one edge, denoting that there is only one operation, and it runs in $O(1)$ work and span.

Similarly, sequences promise a length function in $O(1)$ work and span as well. This is achieved by just storing the length of the sequence within the sequence itself.

```
length : 'a seq -> int
```
REQUIRES: `true`
ENSURES: `length` $\langle \mathtt{x}_0, \mathtt{x}_1, ..., \mathtt{x}_{n-1} \rangle \cong \mathtt{n}$

The cost graph looks the same as `nth`:

```
map : ('a -> 'b) -> 'a seq -> 'b seq
```
REQUIRES: `f` x_i is valuable for all elements x_i in the input sequence
ENSURES: `map f` $\langle x_0, x_1, ..., x_{n-1} \rangle \cong \langle$ `f` x_0, `f` x_1, ..., `f` $x_{n-1} \rangle$

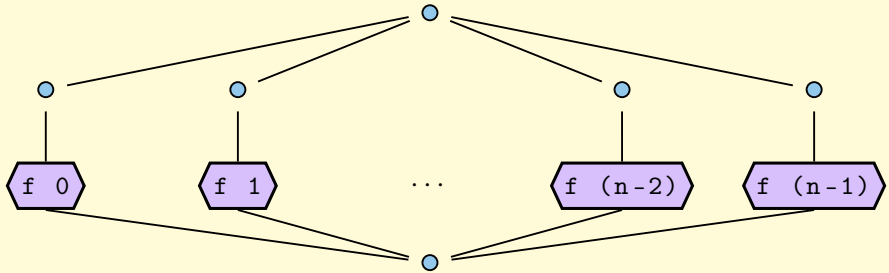
`map` will end up having an identical cost profile to `tabulate`, but this ends up being because we don't need special case it in terms of our definitions, because we can derive it in terms of the functions we've already seen!

```
fun map f S = tabulate (fn i => f (nth S i)) S
```

Namely, `tabulate` and `nth`.

How do we analyze the cost of `map`? Well, intuitively it should be the same, but what do we say about the cost graph?

The `f` function in the expression `tabulate (fn i => f (nth S i)) S` is just the lambda `fn i => f (nth S i)`, which is the sequential combination of the call to `nth`, and the call to `f`. This means that our cost graph actually looks like:[5]



[5] Actually, I have simplified notation a bit. There should technically be another edge above each purple node, but I have just contracted a constant number of edges into one.

This produces the same cost, because a constant addition to each path does not alter the asymptotic complexity of the function.

The key conceptual understanding here is that the way to derive this cost graph, is simply to compose the cost graphs of `nth` and the original `f`, and then substitute it into the cost graph of `tabulate`!

These kinds of nested computations are common with sequences, and it's important to be able to accurately derive their cost.

The most fundamental operator we would like to analyze is that of folding on sequences.

We are familiar with left folds and right folds for lists, and for sequences it turns out we are going to have a natural equivalent. There will be some differences, however, to take advantage of the parallel nature of sequences.

The standard specification of `foldl` is that `foldl f acc [x1, ..., xn]` is equivalent to `f (xn, ..., f (x2, f (x1, acc)) ... )`. `foldl` achieves this by literally computing that expression in sequence[6], but this is a naturally $O(n)$ work and span operation! We can't possibly do any better than sequentially march along, because there's a giant data dependency, as each computation depends on its inner consistuents.

[6] It's funny how "in sequence" ends up being the worst case for sequences. It's even funnier how a data structure specifically built for parallelism ended up named after the word "sequential".

But how will we go about producing this fold for sequences, in a way that takes advantage of parallelism? We can't avoid this data dependency.

Indeed, we aren't going to be able to avoid the fact that each outer `f` depends on the inner ones, but we can change the problem statement slightly.

By analogy, let's consider an operation like addition. If we were summing all of the elements of a list, must this take $O(n)$ span?

The answer: **No, it does not**!

Suppose we were summing a list of elements, and that list is `[1, 2, 3, 4, 5, 6, 7, 8]`. There's a closed form, but pretend we weren't actually privy to the contents of the list.
The naive way to do this is to march from left to right and compute the sum:

$$\begin{aligned}
&1+2+3+4+5+6+7+8\\
&=3+3+4+5+6+7+8\\
&=6+4+5+6+7+8\\
&=10+5+6+7+8\\
&=15+6+7+8\\
&=21+7+8\\
&=28+8\\
&=36
\end{aligned}$$

How exhausting.

But, let's give the perspective a switch. What if, instead, we computed the sum of the following:

$$((1+2)+(3+4))+((5+6)+(7+8))$$

It's the same, but now *everything* is different. The different components of the addition can be done in parallel.

$$\begin{aligned}&((1+2)+(3+4))+((5+6)+(7+8))\\&=(3+7)+(11+15)\\&=10+26\\&=36\end{aligned}$$

Now three steps, when previously it took seven. All because of a simple reparenthesization.

What gives? The key distinction is that, for lists, the trace on the previous slide is *still* the best you can do, in parallel, because lists are sequential. With sequences, we are going to exploit its parallel properties to have the freedom to "reparenthesize" in this manner.

Is this always possible, though? Consider if we were simply folding from left to right with the subtraction operator, on a smaller list of `[1, 2, 3, 4]`.
Then, `foldl` should produce:

```
foldl (op-) 0 [1, 2, 3, 4]
≅ foldl (op-) (0 - 1) [2, 3, 4]
≅ foldl (op-) (~1) [2, 3, 4]
≅ foldl (op-) (~1 - 2) [3, 4]
≅ foldl (op-) (~3) [3, 4]
≅ foldl (op-) (~3 - 3) [4]
≅ foldl (op-) (~6) [4]
≅ foldl (op-) (~6 - 4) []
≅ foldl (op-) (~10) []
≅ ~10
```

Let's try the same trick:

$$\begin{aligned}&(1-2)-(3-4)\\&=(-1)-(-1)\\&=0\end{aligned}$$

...What just happened?

Key Addition is not the same as subtraction.

More particularly, addition is different than subtraction via a mathematical property that allows it to employ this "reparenthesization" and still be equivalent. This property is **associativity**.

Def We say a binary operation $\odot$ of type `t * t -> t` is **associative** if, for all `x, y, z` of type `t`, we have that `x` $\odot$ (`y` $\odot$ `z`) is equivalent to (`x` $\odot$ `y`) $\odot$ `z`.

You can also think of it as, in a long chain of binary operations, if it is associative, then you can place parentheses wherever you want, and *reassociate* the operations, and still get the same thing out. They must remain in the same order, however.

Addition is thus associative, since it doesn't matter which side of the addition you do first. For subtraction, it does, so we can't use this tactic.

We will thus restrict our attention solely to those operations which are associative, for the purpose of our folding function. Since it's not really a "fold" in the same sense as we saw for lists, we will call it another name: **reduce**.

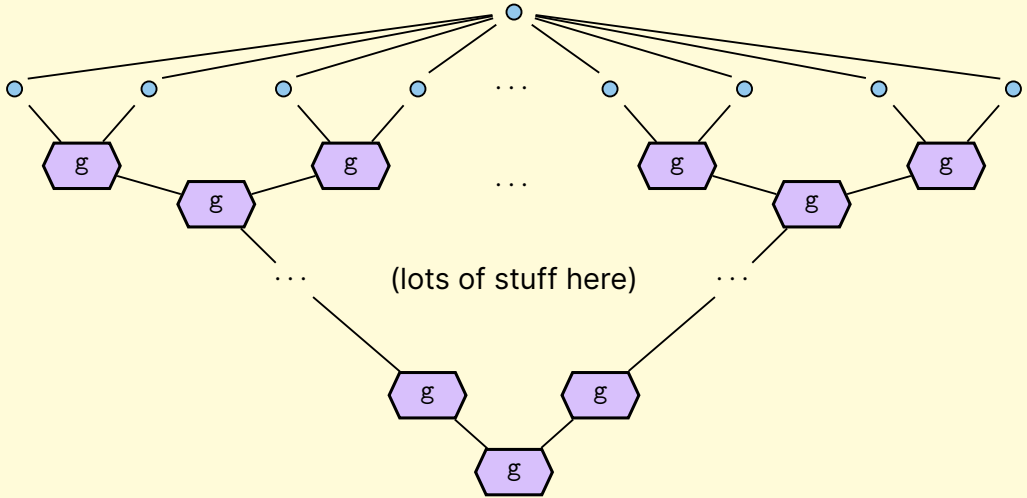
Here's our spec for `reduce`:

```
reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
```
REQUIRES: `g` is both total and associative
ENSURES: `reduce g z S` is equivalent to `foldr g z L`, where `L` is the corresponding list to the sequence `S`

It's implemented spiritually in the same way as the algoritm we just described, for fast computation of the sum of a list. We pair up the elements if we can, and apply the binary operation to them in parallel. The base case of `z` is also placed at the very end, at the right of the sequence.

We then repeat this process until we get a final value.

What's our cost graph look like? It actually looks similar to the computation trace:



The cost graph might look scary, but it's really just the same idea as the divide-and-conquer algorithm we saw for addition! All we do is divide into our pairs, and then recursively continue doing that until completion.

This means that the height of the cost graph is $O(\log n)$, in the length of the sequence, and the number of nodes is linear. Beware that the actual work and span of the `reduce` call might be worse, though, depending on if `g` is constant time or not.

For a constant function `g` (like addition), however, we have that `reduce g z S` is $O(n)$ work and $O(\log n)$ span. Not bad!

We love filtering, so we wanted one for sequences, too:

```
filter : ('a -> bool) -> 'a seq -> 'a seq
```
REQUIRES: `p` is total
ENSURES: `filter p S` is equivalent to the same sequence as `S`, but with all the elements that do not satisfy `p` removed

How is it implemented, though? Naively, we might think that we could just apply the function `p` to each element, and then fold over the sequence to collect the results.

Folding is a sequential concept, though! This would incur an $O(n)$ span bound, because we couldn't do it in parallel. We can do better than that.

We actually can't draw a cost graph here, because the implementation of `filter` is kind of subtle. The cost of filter is, given a constant-time predicate `p`, $O(n)$ work and $O(\log n)$ span.

You can think of it as being something like the following pseudocode:

- map each element to `NONE` or `SOME` depending on if it satisfies `p`
- `reduce` the sequence by combining the remaining elements into sub-sequences, joining them with `Seq.append`

This won't have the right work bound, but it's the right intuition for why the span is logarithmic. Further treatment is reserved for 15-210.

There are many more sequence functions, but we will not give a rigorous treatment of them here. Their descriptions and cost bounds can be found at the 150 sequences reference.

These are the fundamental ones that will get us going with programming with sequences. We will find that many problems having to do with dealing with bulk data become much more performant due to our use of sequences.

4 - And Then Sum

For a simple example on sequences, see the problem of summing all the entries in a 2D matrix, modelled by two sequences.

`sumMatrix : int seq seq -> int`
REQUIRES: `true`
ENSURES: `sumMatrix S` evaluates to the sum of all the elements in the 2D matrix `S`

We can write some terse code as follows:

```
fun sum S = Seq.reduce (op+) 0 S

fun sumMatrix S =
  Seq.map sum S
  |> sum
```

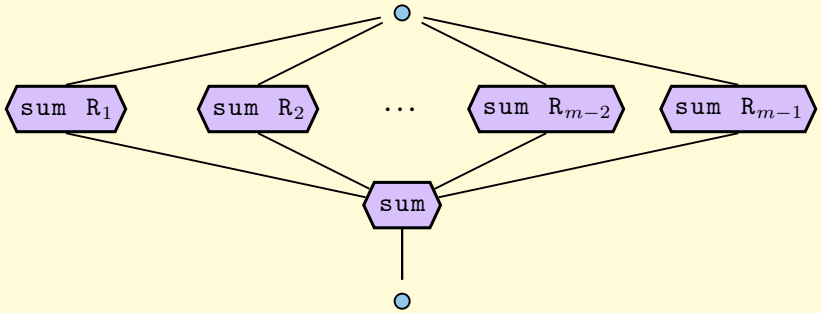
This comprises of the logic to take a 2D matrix and sum across its rows using the faster `reduce`:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \longmapsto^* \begin{bmatrix} (0+1)+(2+3) \\ (4+5)+(6+7) \\ (8+9)+(10+11) \end{bmatrix} \longmapsto^* \begin{bmatrix} 6 \\ 12 \\ 28 \end{bmatrix}$$

...and then to sum across each row's sum, using the same method:

$$(6+12)+28$$

The cost graph looks like so:



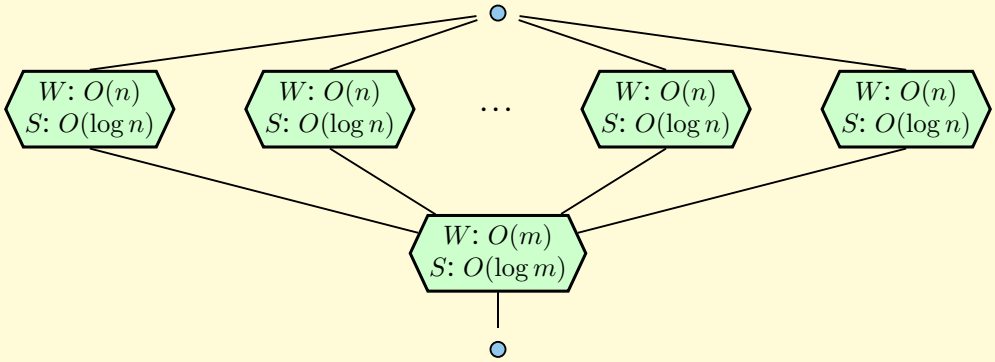
Here, we have m calls in the middle, for an $m \times n$ matrix.

What's the cost?

Well, not all of the `sum`s in this graph are made equal. In particular, each of the inner calls to `sum` $\texttt{R}_i$ are on the ith row, which is a sequence n elements long, so that is a cost of $O(n)$ work and $O(\log n)$ span.

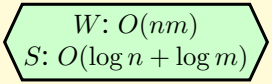
Then, we know that the sequence passed to the final `sum` must be of the same length, which is the number of rows m. So that is a cost of $O(m)$ work and $O(\log m)$ span.

So now we can get our cost graph, but with cost nodes instead of computation nodes:



From this, it should be clear that the work (area) of the graph is $m * O(n) + O(m)$, or $O(nm)$, and the span (longest path) is $O(\log n + \log m)$.

So we finally simplify, and obtain the cost node which is simply our final result.



That's all there is to it! We could then iterate this process, and substitute this cost node somewhere else as well, if this `sumMatrix` function were to be involved in another computation somewhere.

Sequences are useful for a couple of reasons. They:

- are an excellent use of abstraction to not think of lower-level details of a given implementation
- allow us to solve problems that would normally be solved by arrays, without needing to compromise our immutability
- allow us to think about span in a more nuanced way, with parallel-friendly operations

They aren't as nice to work with as lists, which are the bread and butter of a functional programmer's toolkit, but they are still quite usable and quite convenient.

Thank you!