15–150: Principles of Functional Programming

Parallelism, Cost Graphs, and Sequences

Michael Erdmann[*]
Fall 2024

1 Topics

- Functional Programming and Parallelism
- Cost Semantics
- Sequences

2 Introduction

2.1 The Big Picture

Before we start, let's think about the big picture of parallelism. Parallelism is relevant to situations where many things can be done at once—e.g., using the multiple cores in multi-processor machine, or the many machines in a cluster. Overall, the goal of parallel programming is to describe computation in a way that allows us to exploit the potential for doing work on multiple processors simultaneously. At the lowest level, this means deciding, at each step, what to do on each processor. These decisions are constrained by the data dependencies in a problem or a program. For example, evaluating `(1 + 2) * (3 + 4)` takes three units of work, one for each arithmetic operation, but you cannot do the middle multiplication until you have done the two additions. So even with three processors, you cannot perform the calculation in fewer than two time-steps. That is, the expression has work 3 but span 2.

Now, one way to do parallel programming is to say explicitly what to do on each processor at each time-step — by giving what is called a *schedule*. There are languages that let you write out a schedule explicitly, but there are disadvantages to this approach. For example, when you buy a new machine, you may need to adapt your program that was written for (say) 4 processors to the new machine's larger number of processors, maybe 16, or 64, or a million. Moreover, it is tedious and boring to think about assigning work to processors, when what you really want think about is the problem you are trying to solve. After all, we might reasonably expect the scheduling details to be something a smart compiler can best figure out, whereas it is the programmer's job to design an algorithm that best solves the overall problem.

[*]Adapted from documents by Stephen Brookes and Dan Licata.

2.2 Our Approach

The approach to parallelism that we advocate in this class (and is further developed in 15-210) is based on raising the level of abstraction at which you can think, by *separating algorithm specification (and work/span analysis) from scheduling.* You, the programmer, worry about specifying what work there is to do, and how much potential for parallelism there is (as measured by span complexity); the compiler should take care of scheduling the work onto processors.

Three points to facilitate this *separation of concerns*:

1. The code you write to implement an algorithm must not "bake in" a schedule; design your code to avoid unnecessary data dependencies.
2. You must be able to reason about the evaluation *behavior* of your code independent of the schedule.
3. You must be able to reason about the *time complexity* of your code independent of the schedule.

Functional Programming

Our central tool for avoiding schedule-baking is functional programming. In pure functional programming, evaluation has no side effects. This purity limits the dependence of one chunk of work on another to whatever is inherent to the data-flow within the program. All that matters from a prior evaluation is the value returned. When there is little or no data dependency, as often is the case with bulk operations on large chunks of data, one can do work in parallel without specifying evaluation order, and be sure to get the same results no matter what evaluation order occurs.

Functional programming therefore also allows us to reason about code behavior independent of schedule. Purely functional programs are said to exhibit *deterministic parallelism*, meaning the extensional behavior of a program does not depend on whatever scheduling decisions occurred during evaluation. This is true because pure functional programs have well-defined mathematical meanings, independent of any implementation.

Analogous statements are possible but not as straightforward when programming imperatively, where function calls might influence one another via memory updates or other side effects.

Cost Graphs

Our central tool for reasoning about time complexity, independent of the schedule, is a *cost semantics.* This involves the asymptotic work and span analyses that we have been doing all semester. These analyses let us reason abstractly, yet actually imply some concrete constraints on how well any potential scheduler might be able to divide work among processors.

As we will see, work and span analyses can be phrased in terms of *cost graphs*, which can be used to reason abstractly about the running time of a program independent of the schedule. It turns out that cost graphs are actually also useful data structures for creating schedules.

Sequences

Many modern algorithms perform *bulk operations* on potentially very large collections of data. Here higher-order functional programming creates a particularly nice opportunity for parallelism.

In many practical applications, one wants to combine large amounts of data using some binary operation (such as `+`). When such a binary operation is *associative*, the order of pairwise combinations is irrelevant. By representing the data as an abstract type with efficient bulk operations, one can write code that avoids being overly specific about evaluation order. Lists are *not* very well suited for this task, because the built-in combination operations on lists (`foldl` and `foldr`) bake in a sequential evaluation order. For trees one can define "tree folding" operations as higher-order functions in which the left and right subtrees get combined independently, and as we saw earlier the tree fold operations have better span than the list fold operations. But we can do even better than trees.

We will be discussing *sequences* frequently in the next few lectures. We will use the (math) notation `<x1,..., xn>` for a sequence of length `n` containing the items `x1` through `xn`. Our implementation of sequences includes many familiar operations, including a `map` operation. This is a higher-order function similar in spirit to `List.map`, with the following equivalence specification [1]:

```
map f <x1, x2, . . ., xn>  ≅  <f x1, f x2, . . ., f xn>
```

In terms of *evaluation*,

```
map f <x1, x2, ..., xn> ⟹ <v1,. . ., vn>
  if f x1 ⟹ v1, f x2 ⟹ v2, ..., f xn ⟹ vn.
```

This description also implicitly specifies the data dependencies, namely none:
To calculate `map f <x1, ..., xn>`, one needs to calculate `f x1`, ..., `f xn`. There are no data dependencies, assuming `f` is purely functional. Moreover, we did not specify any particular schedule or evaluation strategy. This suggests plenty of opportunity for parallel evaluation.

(There is one subtlety when the function `f` is not total. One possibility is to specify more generally an evaluation rule for a sequence `<e1, e2, ..., en>` when the constituent expressions `e1`, ..., `ek` are not necessarily valuable. One may evaluate the constituent expressions in parallel, but if one or more fails to return a value, one might choose the leftmost such behavior as the overall behavior of evaluating the sequence.)

2.3 Caveat Programmer

There is an important caveat: even with today's technology, this methodology based on separation of concerns — you design the algorithm, the compiler schedules it — is not always going to deliver good performance. It is difficult to get parallel programs to run quickly in practice, and many smart researchers are actively working on this problem. Some of the issues include: *overhead* (it takes time to distribute tasks to processors, notice when they've completed, etc.); *spatial locality* (we want to ensure that needed data can be accessed quickly); and *schedule-dependence* (the choice of schedule can sometimes make an asymptotic difference in time or space usage).

There are some implementations of functional languages that address these issues, to varying degrees of success: Manticore, MultiMLton, and PolyML are implementations of SML that have a multi-threaded run-time, so you *can* run in parallel 15-150's sequence code. NESL is a research language by Guy Blelloch (who designed 15-210), and that's where a lot of the ideas on parallelism that we discuss in this class originated; there is a real implementation of NESL and some benchmarks with actual speedups on multiple processors. GHC, a Haskell compiler, implements many of the same ideas, and you can get some real speedups there too. Scala is a hybrid functional/object-oriented language with some demonstrated potential for parallel performance.

[1]We extend the notion of extensional equivalence to sequences as follows: two sequence values are extensionally equivalent iff they have the same length and contain extensionally equivalent values at corresponding positions.

3 Cost Semantics

A cost graph is a type of *series-parallel graph.* A series-parallel graph is a directed graph with a designated source node (no edges in) and a designated sink node (no edges out), formed by two operations called *sequential* and *parallel composition.* The particular series-parallel graphs we need are of the following form (there are three graphs in this diagram):

```
                        .
.       G1             / \
         |            G1 G2
        G2             \ /
                        .
```

(Notation: We always draw a cost graph so that edges point down the page. Consequently, we omit drawing arrows on the edges.)

1. The first graph is a graph with one node (the node is both source and sink in that graph). It represents no computation.

2. The second graph is the *sequential combination* of graphs `G1` and `G2`, formed by putting an edge from the sink of `G1` to the source of `G2`; its source is the source of `G1` and its sink is the sink of `G2`. As a special case,

   ```
   .
   |
   .
   ```

   represents a single reduction or evaluation in a program.

3. The third graph is the *parallel combination* of graphs `G1` and `G2`, formed by adding a new source and sink, and adding edges from the new source to the sources of `G1` and `G2`, and from the sinks of `G1` and `G2` to the new sink.

We also use n-ary versions of sequential and parallel composition for cost graphs. We draw an n-ary parallel combination of graphs `G1 ... Gn` as

```
        .
   /    |    \
 G1    G2 ... Gn
   \    |    /
        .
```

Again, there is a new source and a new sink, with edges from the new source to the sources of the `Gi`, and edges from the sinks of the `Gi` to the new sink.

The *work* of a cost graph `G` is the number of nodes in `G`. The *span* of `G` is the number of nodes on the longest path from the source of `G` to the sink of `G`, which we may refer to as the *critical path.* We will associate a cost graph with each closed program, and define the work/span of a program to be the work/span of its cost graph.

These graphs model *fork-join parallelism*: a computation forks into various subcomputations that are run in parallel, but these come back together at a well-defined join point. These forks and joins are well-nested, in the sense that the join associated with a later fork precedes the join associated with an earlier fork.

For example, the expression

```
(1 + 2)
```

has cost graph

```
    .
   / \
  .   .
   \ /
    .
    |
    .
```

This says that the summands `1` and `2` are evaluated in parallel; because `1` and `2` are already values, the middle graphs have only one node. After the parallel combination, there is one step for doing the addition. For simplicity of drawing graphs, we may merge the evaluation that occurs after a join into the join itself, so we may simply write the previous graph as

```
    .
   / \
  .   .
   \ /
    .
```

This is consistent with viewing fork as splitting input into subproblems and join as merging results from those subproblems, much as with divide-and-conquer.

The work of the graph above is 4 and the span is 3.

(Note: These numbers may differ by a constant factor or term from those we defined earlier in the course, because we are counting slightly differently, due to the presence of fork and join nodes. Asymptotically, there will not be a difference.)

We can link cost graphs with our earlier discussions of work and span for expressions, as follows. Recall that we add the work and add the span for code fragments that need to be executed in sequential order; just as the number of nodes in the sequential composition or the parallel composition of `G1` and `G2` is the sum of the numbers of nodes in the two subgraphs. And for independent code fragments we combine spans with max rather than +, just as the span of the parallel composition of two graphs corresponds to the maximum of the spans of the individual graphs.

3.1 Brent's Principle

Cost graphs can be used to reason abstractly about the time complexity of a program, independent of the schedule. For example, below, we associate a cost graph with each operation on sequences. One can reason about the work and span of code that uses these operations to manipulate sequences, via these graphs, without worrying about any particular schedule.

You may well ask: what do work and span predict about the *actual* running time of your code? The work predicts the running-time when evaluated sequentially, using a single processor; the span predicts the running time if you have "infinitely many" processors and the scheduler always uses parallel evaluation for independent computations. Of course in practice you are unlikely to have infinitely many processors (it is even unlikely that you will always have "enough" processors). What can we say, based on work and span, about what happens when you evaluate SML code using the 4 processors in your laptop, or the 1000 in your cluster, relying on whatever scheduling strategy is implemented on these machines? The answer is that there is a provable bound on the best running time that can be achieved, asymptotically:

> **Brent's Theorem**
> An expression with work w and span s can be evaluated on a p-processor machine in time $\Omega(\max(w/p, s))$.

[The notation Ω is similar to big-O, except that Ω means an asymptotic lower bound whereas big-O means an asymptotic upper bound.]

The intuition behind Brent's Theorem is that the best (most efficient) way to employ p processors is to try dividing the total work w up into chunks of size p (there are w/p such chunks) and do these chunks one after another; but the data dependencies may prevent this from being feasible, so one can never do better than the span s.

For example, if you have 10 units of work to do and span 5, you can achieve the span on 2 processors. If you have 15 units of work (and still span 5), it will take at least 8 steps on 2 processors. If you increase the number of processors to 3, then you can achieve the span 5. If you have 5 units of work to do (still with span 5), then having 2 processors doesn't help: you still need 5 steps. (In this example, we pretended that Brent's Theorem asserted a direct equality rather than a big-O class, for simplicity.)

Brent's Theorem can also tell us useful information about the potential for speed-up when multiple processors are available. If your code has work w and span s, calculate the smallest integer p such that $w/p \leq s$. Given this many processors, you can evaluate your code in the best possible time. Having even more processors yields no further improvement. (Of course these are all *asymptotic* estimates, and in practice constant factors do matter. So you may not see real speedups consistent with these numbers.)

Brent's theorem should hold for any language one designs; if not, the cost semantics are wrong. Thus, we will sometimes refer to *Brent's Principle*: a language should be designed so that Brent's Principle is in fact a theorem.

3.2 Scheduling

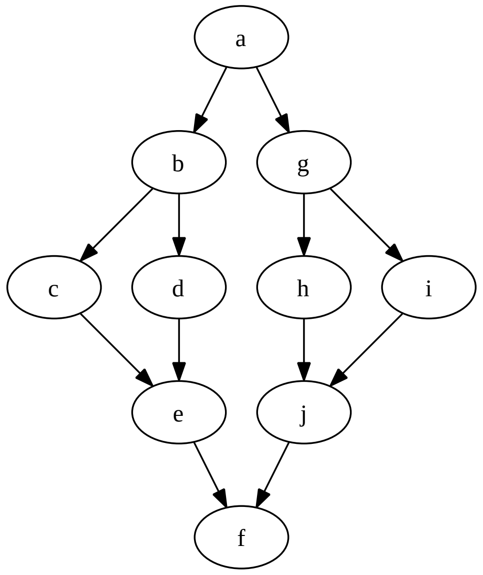
Cost graphs are also a helpful data structure for scheduling work onto processors. Let's take a look at how a compiler might do this.

A schedule can be generated by *pebbling* a cost graph. To schedule a graph onto p processors, you play a pebbling game with p pebbles. The rules of the game are: You can pick up a pebble from a graph node or from your hand and place it on any graph node all of whose predecessors have been *visited* but which itself has not yet been visited. When you place a pebble on a node, you mark that node as visited. (You can also put the pebble back in your hand, if you don't want it on the graph anymore.)

A *schedule* is supposed to say what piece of work each processor does in each time-step. To generate a schedule, we divide the pebbling up into steps. In any step, you can place p (or fewer) pebbles onto the graph. The nodes with pebbles on them represent the units of work to be completed by the p processors during that time-step. The restriction on placing pebbles only on nodes whose predecessors have been visited ensures that data dependencies are respected. The nodes whose predecessors have been visited, but have not themselves been visited, form what is sometimes called the *frontier*, meaning the currently available work.

Consider the following cost graph:

(This graph might represent evaluation of the expression `(1 + 2) * (3 + 4)`, as can be seen by generalizing the example of page 5.)



A 2-pebbling with pebbles X and O might start out like this:

Step	**X**	**O**
1	a	
2	b	g
3	c	

In the first step, we can only place a pebble on the source (all of its predecessors have been pebbled, trivially, because it has no predecessors), because no other node in the graph is available. In this step, one processor is idle because there is not enough available work to be done. In the second step, we can place pebbles on both of the next two nodes and did so, meaning that both processors had work to do in that time-step. In the third step, we can place our two pebbles on any two of the four nodes at the next level. We chose, however, to place only a single pebble, on one of those nodes. So this is again a step in which a processor is idle, though this time by choice; there was work that could have been done by the other processor. Why did we choose to place only a single pebble? Simply to indicate that it is possible to do so. A *greedy schedule* assigns as many processors as possible work at each time step. Sometimes, a non-greedy scheduler might actually be more efficient overall.

For a fixed number of processors p we can identify two particularly natural scheduling algorithms, known as $pDFS$ (p depth-first search) and $pBFS$ (p breadth-first search). A DFS schedule prefers the left-most bottom-most available nodes, whereas a BFS schedule prefers higher nodes (but then tackles them left-to-right). At each step, one can place as many pebbles (a maximum of p) as one can, subject to availability of work.

Here is a schedule for the cost graph above, using 2DFS (giving preference to processor X when only one unit of work is available):

Step	**X**	**O**
1	a	
2	b	g
3	c	d
4	e	h
5	i	
6	j	
7	f	

In step 4, we prefer node *e* to node *i* because *e* is lower (bottom-most). In the remaining steps there is only one node available to work on.

Here's a schedule for the same graph, using 2BFS:

Step	**X**	**O**
1	a	
2	b	g
3	c	d
4	h	i
5	e	j
6	f	

This time, in step 4 we prefer *i* to *e* because it is higher. Consequently, in step 6, two units of work are available to do, so we can finish in 6 steps instead of 7.

Thus: different scheduling algorithms can give different overall run-time. Additionally, they differ in how many times a processor stops working on one computation and switches to working on an unrelated computation. For example, in the 2BFS schedule, step 4, both processors "jump" over to the other side of the graph. This can be bad for spatial reasons (cache performance may be worse) but the details are subtle. And actually in this kind of small example the differences aren't really likely to be noticeable. Of course, this may not be the case in more realistic examples.

4 Sequences

To introduce sequences, here is a signature containing some of their primary operations:

```
signature SEQUENCE =
sig
   type 'a seq   (* abstract *)
   val empty : unit -> 'a seq
   exception Range of string
   val tabulate : (int -> 'a) -> int -> 'a seq
   val length : 'a seq -> int
   val nth : 'a seq -> int -> 'a
   val map : ('a -> 'b) -> 'a seq -> 'b seq
   val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
   val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
   val filter : ('a -> bool) -> 'a seq -> 'a seq
end
```

Let's assume we have an implementation of this signature, i.e., a structure `Seq:SEQUENCE`. For any type `t`, the type `t Seq.seq` has values that represent sequences of values of type `t`. Sequences are *parallel collections*: ordered collections of values, with parallelism-friendly operations on them. Don't think of sequences as being implemented by lists or trees (though you could implement them as such); think of them as a new built-in abstract type with only the operations we're about to describe (the operations mentioned in the signature `SEQUENCE`). The differences between sequences and lists or trees show up in the cost of these operations, which we specify below.

We write `Seq.seq`, `Seq.map`, etc. to refer to the `seq` type and the `map` function defined in our given structure named `Seq`. This use of qualified names will help us to avoid confusion with the built-in SML `map` function on lists (which we could also refer to using the qualified name `List.map`).

Intuitively, the sequence operations have names that suggest that they do the same things as the corresponding operations on lists with which you are familiar. However, they have different work and span than the corresponding list functions. Firstly, sequences admit *constant-time* access to items — `nth s i` takes constant time, for instance. Secondly, sequences have better parallel complexity—many operations, such as `map`, act on each element of a sequence in parallel.

For each sequence operation described on the next few pages, we will (a) describe its behavior abstractly and (b) give a cost graph, that specifies the operation's work and span.

Before we start, recall that we use the math notation

```
<v1, . . . , vn>
```

for a sequence of length `n` whose entries are `v1` through `vn`. Actually, since the `nth` function uses zero-based indexing, it is sometimes more convenient to mimic this and write

```
<v0, . . ., v_(n-1)>
```

tabulate
The operation for building a sequence is *tabulate*, which constructs a sequence from a function that gives you the items at each position, the positions being indexed from 0 up to a specified bound. So `tabulate f n` builds a sequence of length `n` with 0'th item `f 0`, 1st item `f 1`, and so on. We can describe this operation via the following equivalence:

```
tabulate f n  ≅  <f 0, ..., f (n-1)>
```

and its evaluation properties are summarized by:

```
tabulate f n  ⟹  <v0 , ... , v_(n-1)>
   if f 0  ⟹  v0,  f 1  ⟹  v1,  ...,  f (n-1)  ⟹  v_(n-1).
```

(Again, this characterization does *not* imply any specific order of evaluation.)
The cost graph for `tabulate f n` is

```
       .
    /  |   \
  G0  G1 ... G(n-1)
    \  |   /
       .
```

where each `Gi` is the cost graph for `f i`.
So when `f i` is constant time, the work for `tabulate f n` is $O(n)$ and the span is $O(1)$.

nth
We define the behavior of `nth` as

```
nth <v0 , ... , v_(n-1)> i   ⟹   vi,                                  if 0 <= i < n
                                raise Range "nth index out of bounds", otherwise
```

For a sequence value `s` and integer value `i`, the cost graph for `nth s i` is simply

```
.
|
.
```

Thus, as promised, sequences provide constant-time access to elements;
`nth s i` has $O(1)$ work and $O(1)$ span.

length
The behavior of `length` is given by:

```
length < >  ⟹  0
length <v1 , ... , vn>  ⟹  n
```

The cost graph for `length s` when `s` is a sequence value is also simply

```
.
|
.
```

Thus, `length s` also has $O(1)$ work and span.

map
The behavior of `map f <v1,...vn>` is given by:

$$\texttt{map f <v1 , ... , vn>} \quad \cong \quad \texttt{<f v1,...,f vn>}$$

Each function application may be evaluated in parallel. This is shown by the cost graph

```
         .
  /      |      \
G1     G2 ... Gn
  \      |      /
         .
```

where we let `Gi` be the cost graph for `(f vi)`, `i = 1, ..., n`. (This cost graph resembles the cost graph for tabulate.)

As a consequence, if `f x` takes constant time for all values `x`, and `s` is a sequence value with length n, then `map f s` has $O(n)$ work and $O(1)$ span.[2]

reduce
`reduce` is intended to be used with an *associative* binary function `g` of type `t * t -> t`, for some type `t`, a value `z:t`, and a sequence of items of type `t`.

Recall, a function `g` is associative iff for all values `a,b,c` of type `t`,

$$\texttt{g(a, g(b, c))} \cong \texttt{g(g(a, b), c)}.$$

Associativity implies that if we pairwise combine a sequence of values using `g`, in any order that respects the original enumeration order, we will get equivalent results. For example,

$$\texttt{g(g(x1, x2), g(x3, x4))} \quad \cong \quad \texttt{g(x1, g(x2, g(x3, x4)))}.$$

Let's write `g` in infix notation using the operator $\odot$. So we will abbreviate `g(x1, x2)` by x1$\odot$x2.
When `g` is associative, we can abbreviate `g(x1, g(x2, g( ..., xn)))` by

$$\texttt{x1} \odot \texttt{x2} \odot \texttt{. . .} \odot \texttt{xn}.$$

Assuming that `g` is associative, `reduce g z <x1, . . ., xn>` has the following equivalence characterization:

$$\texttt{reduce g z}\,\langle x_1, \ldots x_n\rangle \ \cong \ x_1 \odot x_2 \odot \ldots \odot x_n \odot \texttt{z}$$

We assume that the implementation of `reduce` uses a balanced parenthesization format to form pairwise combinations. So, for example, the cost graph of `reduce g z <x1, x2, x3>` is the same as the cost graph for $(x1 \odot x2) \odot (x3 \odot z)$. (See the next page for the general picture.)

Remark: One sometimes asks as well that `z` be an *identity* for `g`, meaning
`g(x, z)` $\cong$ `x` $\cong$ `g(z, x)` for all values `x : t`. We do not require that in this document, but it can be helpful behind the scenes in implementing a nicely parallel version of `reduce`. You will likely make this assumption in 15-210.

[2]Unfortunately, there is no known way to express the time complexity of the `map` function itself, abstractly for all `f` — we lack a theory of asymptotic analysis for higher-order functions.

If `g(a,b)` is constant time, for all values `a` and `b`, it follows that the cost graph for `reduce g z` $\langle x_1, \ldots x_n\rangle$ looks something like

```
                    .
   /         /   /    \     \        \
  /         /   /      \     \        \
x1       x2   x3      x4     xn        z        (width n+1 here)
  \       /     \      / ...   \      /
   \     /       \    /         \    /
  x1 ⊙ x2       x3 ⊙ x4  ...  xn ⊙ z           (width (n+1)/2 here)
       \           /
        \         /                 .
         \       /   ...          .
          \     /               .
           \   /
             ⊙
              \            /
               \   ...    /
                \        /
                    .
                    .
                    .
                    ⊙
                    .                   (roughly log n levels deep)
```

Consequently, the graph has size $O(n)$ and critical path length $O(\log n)$, giving us the work and span, respectively. This computation does not have *constant* span, because later uses of `g` need the results of earlier combinations: there are data dependencies, because we can only combine two items at a time.

Contrast this with lists and `foldl` or `foldr`. When `L` is a list of length n, `foldr g z L` has work $O(n)$ and also span $O(n)$. Similarly for `foldl`.

mapreduce

`mapreduce` is intended to be used with a function `f` of type `t1 -> t2` for some types `t1` and `t2`, an *associative* binary function `g` of type `t2 * t2 -> t2`, a value `z:t2`, and a sequence of items of type `t1`.

The behavior of `mapreduce f z g <x1, . . ., xn>` (assuming that `g` is associative and that we again represent it by an infix operator $\odot$) is given by:

$$\texttt{mapreduce f z g}\,\langle x_1, \ldots x_n\rangle \;\cong\; (\texttt{f}\ x_1)\ \odot\ (\texttt{f}\ x_2)\ \odot\ \ldots\ \odot\ (\texttt{f}\ x_n)\ \odot\ \texttt{z}$$

The implementation of `mapreduce` uses a balanced parenthesization format for pairwise combinations, as with `reduce`. Its work and span are as for `reduce`, when `f` and `g` are constant time.

filter

`filter` is a higher order function for retaining only those elements in a sequence that satisfy a given predicate: Assuming `p` is total, `filter p s` $\cong$ `s'`, where `s'` consists of all elements `e` of `s` such that `p e` $\cong$ `true`. The order of retained elements in `s'` is the same as in `s`.

Assuming further that `p` is constant time, `filter p s` has work $O(n)$ and span $O(\log n)$.

5 Examples

Let's look at some examples. As before, let's assume we are working with a structure `Seq` that ascribes to signature `SEQUENCE`.

- Building an empty sequence. We might implement

  ```
   empty : unit -> 'a seq
  ```

  within `Seq` by

  ```
  fun empty () = tabulate (fn _ => raise Range "empty") 0
  ```

  `empty()` returns a sequence of length zero.

- We could also implement a `cons` operation for sequences within `Seq`, using `tabulate`. Let's specify that the function has type `'a -> 'a seq -> 'a seq` and that `cons x xs` returns a sequence with length one more than the length of `xs`, whose first (0'th) element is `x`, and whose remaining items are the items of `xs` shifted by one:

  ```
  fun cons (x : 'a) (xs : 'a seq) : 'a seq =
    tabulate (fn 0 => x | i => nth xs (i-1)) (length xs + 1)
  ```

- Outside `Seq`, let's define a function `first : int -> int Seq.seq` that constructs the sequence consisting of the first `n` natural numbers (with 0 being a natural number):

  ```
  fun first n = Seq.tabulate (fn x:int => x) n
  ```

  For `n` > 0, `first n` returns the sequence value

  ```
  <0, 1, . . ., n-1>
  ```

- Let's define a function `squares : int -> int Seq.seq` that constructs the sequence consisting of the first `n` natural numbers squared:

  ```
  fun squares n = Seq.map (fn x => x*x) (first n)
  ```

- We could also have defined `squares` with a single tabulate, rather than a tabulate followed by a map, as in:

  ```
  fun squares n = Seq.tabulate (fn i => i*i) n
  ```

  There is a general "fusion" law that expresses this kind of result. When `f : int -> t1` and `g : t1 -> t2` are total function values, it follows that `g o f : int -> t2` is also total, and for all values `n:int`,

  `Seq.map g (Seq.tabulate f n)` $\cong$ `Seq.tabulate (g o f) n`

  The `squares` example is a special case,
  with `f` being `(fn x:int => x)` and `g` being `(fn x:int => x*x)`.

- Similarly, when `f : t1 -> t2` and `g : t2 -> t3` are total function values, so is `g o f : t1 -> t3`, and

  ```
      Seq.map (g o f)  ≅  (Seq.map g) o (Seq.map f)
  ```

  As an example, consider

  ```
    fun square x = x*x
    fun quads s = Seq.map square (Seq.map square s)
  ```

  Here `quads` is extensionally equivalent to `(Seq.map square) o (Seq.map square)`. And `quads` is extensionally equivalent to `Seq.map (square o square)`. Since `square o square` is extensionally equivalent to `fn x:int => x*x*x*x`, we have

  ```
      quads  ≅  Seq.map (fn x:int => x*x*x*x)
  ```

- Evens.
  You may recall writing a function in lab once to filter a list of integers, retaining only the even integers. That took $O(n)$ work and span. Here is an implementation using sequences:

  ```
      fun evens (s : int Seq.seq) : int Seq.seq =
              Seq.filter (fn n => n mod 2 = 0) s
  ```

  Alternatively, we could simply have written:

  ```
      val evens = Seq.filter (fn n => n mod 2 = 0)
  ```

  The predicate `(fn n => n mod 2 = 0)` takes constant time, so the work and span of the filter operation are $O(n)$ and $O(\log n)$, respectively, where n is the length of `s`. This is the only operation performed by `evens`, so this gives the work and span of `evens` as well.

- Counting, revisited.
  In the first lecture, we considered at a high-level some SML functions that operate on sequences of integers, combining the integers together by adding them. Later we saw how to use `foldl` and `foldr` to accomplish these tasks using lists. Now we can revisit these ideas using actual sequences.

  Here is a function `sum : int Seq.seq -> int` for adding the integers in a sequence:

  ```
     fun sum (s : int Seq.seq) : int = Seq.reduce (op +) 0 s
  ```

  (Note that `(op +)` is associative.)

  And a function `count : int Seq.seq Seq.seq -> int` for combining the integers in a sequence of integer sequences:

  ```
     fun count (s : int Seq.seq Seq.seq) : int = sum (Seq.map sum s)
  ```

  `sum` takes an integer sequence and adds up all the numbers in it using `reduce`, just like we did with folds for lists and trees. `count` sums up all the numbers in a sequence of sequences, by (1) summing each individual sequence and then (2) summing the sequence that results.

(Exercise: rewrite `count` with `mapreduce`, so it takes only one pass).

We can now see that `count` on a sequence of m sequences, each of length n, requires $O(mn)$ work, as follows. First, `sum s` is implemented using `reduce` with a constant-time argument function, and thus has $O(n)$ work, where n is the length of s. Each call to `sum` inside the `map` is on a sequence of length n, and thus takes $O(n)$ work. This function is mapped across m sequences, yielding $O(mn)$ work for the `map`. The sequence sums now form a sequence of length m, so the final `sum` contributes $O(m)$ more work, which is subsumed by the $O(mn)$. So the total work here is $O(mn)$, as promised.

Similarly, we can see that the span is $O(\log m + \log n)$. Again, `sum s` is implemented using `reduce` with a constant-time argument function, and thus has $O(\log n)$ span, where n is the length of s. Each call to `sum` inside the `map` is on a sequence of length n, and thus has $O(\log n)$ span. These inner calls all occur in parallel, and so together have $O(\log n)$ span. The outer `sum` is on a sequences of length m, and therefore has $O(\log m)$ span. The total span is the sum of the inner span and the outer span, because of data dependency: the outer summation must wait to happen until after all the inner sums have been computed. Therefore the total span is $O(\log m + \log n)$.

- Reversing a sequence, by re-indexing:

```
fun reverse (s : 'a Seq.seq) : 'a Seq.seq =
    Seq.tabulate (fn i => Seq.nth s (Seq.length s - i - 1)) (Seq.length s)
```

This implementation of `reverse` has linear work and constant span.

Caution: index calculations tend to be hard get right and just as hard to read. Moreover, you run the risk of writing very index-y array-like code rather than stylish higher-order functional code. So avoid doing that where possible.