15-150 Fall 2019
Thursday, 24 October

Stephen Brookes *

1 Topics

• Functional programming and parallelism
• Cost semantics, work and span
• An abstract type of sequences with parallel operations

*Building on notes by Dan Licata.
2 Introduction

The big picture

Parallelism is relevant to computational 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, to evaluate the expression \((1 + 2) + (3 + 4)\) takes three units of work, one for each addition, but you cannot do the outer addition until you have done the inner two. With two processors you could do the work in two time-steps, but even with three or more processors, you cannot perform the calculation in fewer than two time-steps. This information is expressed in another way by the facts that the expression has work 3 and span 2.

One way to implement 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’s tedious and boring to think about assigning work to processors, when what you really want think about is the problem you’re trying to solve. After all, we might reasonably expect the scheduling details to be something a smart compiler can figure out better than the programmer, but it’s clearly the programmer’s job to design an algorithm that best solves the problem.

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 (that’s what the span is concerned with); the compiler should take care of scheduling the work onto processors.
Three things are necessary to make this separation of concerns work:

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 independently of the schedule.

3. You must be able to reason about the time complexity of your code independently of the schedule.

Our central tool for avoiding schedule-baking is functional programming. Let’s explain why functional programming is suitable.

First, taking advantage of higher-order functions, we can focus on bulk operations on (potentially large) collections of data. In many applications we want to combine the data in a collection (such as a list, or a tree) using a binary operation (such as +) which is associative, so it doesn’t matter in what order we pairwise combine the data. If we can represent the collection as an abstract type with efficient bulk operations for this kind of combination, it’s easy to 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 we 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.

**Sequences**

Today and in the next lectures we will talk about sequences. We will use the (math-like) notation \( \langle x_1, \ldots, x_n \rangle \) for a sequence of length \( n \) containing the items denoted by \( x_1 \) through \( x_n \). This syntax is not legal in ML programs, so don’t try to mix it into your programming notation. However, it is vital to allow ourselves a convenient notation for specifying and talking about sequences, and we can indeed write ML code that builds and manipulates them. We extend the notion of extensional equality to sequences in the obvious way: two sequences are equal if they have the same length, and contain equal values at all positions.

Our implementation of sequences includes a bunch of operations to be introduced shortly, including a map operation, a higher-order function similar
in spirit to \texttt{List.map}, with the following specification. When \( f \) is a function of type \( t_1 \rightarrow t_2 \) and \( x_1, \ldots, x_n \) are values of type \( t_1 \), we have:

\[
\text{map } f \langle x_1, \ldots, x_n \rangle = \langle f \, x_1, \ldots, f \, x_n \rangle.
\]

In terms of \textit{evaluation}, if each \( e_i \) is an expression of type \( t_1 \) and \( f \, e_i \) evaluates to \( v_i \), for \( i = 1, \ldots, n \), then

\[
\text{map } f \langle e_1, \ldots, e_n \rangle = \star \langle v_1, \ldots, v_n \rangle.
\]

This description also implicitly specifies the data dependencies (none!): to calculate \( \text{map } f \langle x_1, \ldots, x_n \rangle \) you need to calculate \( f \, x_1 \) through \( f \, x_n \), and there are \textit{no data dependencies} between these calculations. And we didn’t specify any particular schedule or evaluation strategy! Our description makes it clear that there is plenty of room for parallel evaluation, but also that other evaluation strategies may be available. You \textit{could} implement the evaluation of \( \text{map } f \langle x_1, \ldots, x_n \rangle \) sequentially, by evaluating \( f \, x_1 \) first, then \( f \, x_2 \), and so on. And if you had enough parallel processing power you \textit{could} implement the evaluation by using a separate processor for each \( f \, x_1 \), and you would end up with the same final result. Our “equational” specification for \texttt{map} is agnostic as to scheduling, as the schedule makes no difference to the value obtained.

If we had written the code for evaluating this kind of expression as an iterative (sequential) loop, on an inherently sequential data structure such as a list, we would have been gratuitously throwing away opportunities for exploiting parallelism. By using \texttt{map} on sequences like this – a bulk operation on a collection data structure – we avoided this problem.

Second, as noted in the above example, functional programming focuses on pure, mathematical functions, and evaluation has no side effects. This limits the dependence of one chunk of work on another to what it is obvious from the data-flow in the program; all that matters from a prior evaluation is the value it returned, and if there’s no data dependency we can do work in parallel and be sure to get the same results. For example, when you \texttt{map} a function \( f \) across a sequence, evaluating \( f \) on the first element has no influence on the value of \( f \) on the second element, etc. This is not the case for imperative programming, where one call to \( f \) might influence another via memory updates (or side effects).

So functional programming is well suited for algorithm design without commitment to a schedule.
The key to reasoning about program behavior, independently of schedule, is determinism: the (extensional) behavior of your program is the same for any schedule! This is true because functional programs have a well-defined mathematical meaning independent of implementation: the values produced by expression evaluation don’t depend on what scheduling decisions occurred during evaluation. For example, assuming that $f$ is a total function and the $x_i$ are suitably typed, we say that

$$\text{map } f \langle x_1, \ldots, x_n \rangle = \langle f x_1, \ldots, f x_n \rangle$$

because the left-hand and right-hand expressions evaluate to equal sequence values. We don’t need to say here exactly how the left-hand expression or the right-hand expression take steps, and indeed there are many possible stepping strategies that are consistent with this equation. In all cases the left- and right- expressions evaluate to equal final values, and that’s what the equation says.

So functional programming nicely supports reasoning about behavioral correctness, independently of scheduling. And equational reasoning can be carried out without ever speaking of schedules.

Our central tool for reasoning about time complexity, independently of the schedule, is a cost semantics. This involves the asymptotic work/span analyses that we have been doing all semester. The cost semantics lets you reason abstractly, yet actually implies some concrete constraints on how well any potential scheduler might be able to divide work among processors. For example, we may have an implementation of sequences with a map function whose cost semantics tells us that when $s$ is an integer sequence of length $n$, $\text{map } (\text{fn } x \Rightarrow x + 1) \ s$ takes $O(n)$ work but $O(1)$ span—each function application can be done in parallel. As we will see, work and span analysis can be phrased in terms of cost graphs, which can be used to reason abstractly about the running time of your program for any schedule, and are also a useful data structure for explaining the ideas behind scheduling.

We should note that sequences can be implemented in several different ways: as lists, trees, arrays, vectors, with or without built-in guarantees about balance. With each choice of a sequence implementation we will get operations like map, and the work/span characteristics for these functions may vary across implementations. For example, with a list-based implementation of sequences map is sequential (like List.map), and with a balanced tree implementation map is logarithmic in the size of the tree (linear in the depth
of the tree). No matter what implementation we start from, the same kind of work/span analysis can be done.

Caveat programmer

There is an important caveat (warning) that we should mention. Even with today’s technology, this methodology based on separation of concerns — you design the algorithm, the compiler schedules it — may not deliver good practical performance. It’s hard 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). So don’t get the impression that writing programs our way will magically guarantee good performance. Nevertheless, our methodology does impose good discipline and can help you to design clear, correct, and potentially efficient solutions to parallelizable problems. And we want you to get used to the idea of figuring out how to design for parallelizability, and how to strive for correct code that meets desirable work/span requirements.

There are some implementations of functional languages that address these issues, to varying degrees of success. Manticore, MultiMLton, and PolyML are implementations of Standard ML that have a multi-threaded run-time, so allow you to run code truly in parallel, but it’s tricky to get actual speedups. NESL is a research language by Guy Blelloch (who designed 15-210), and that’s where a lot of the ideas that we will discuss today 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, and has been used to implement sequences in a way that produces demonstrable performance gains. A parallel implementation of ML is being developed here at CMU by a team including Umut Acar, with the aim of using this in 15-210 (and eventually 15-150).
Rationale

So why are we teaching you this style of parallel programming? There are two reasons: First, even if you have to get into more of the gritty details of scheduling to get your code to run fast today, it’s good to be able to think about problems at a high level first, and then figure out the details. If you’re writing code for an internship this summer using a low-level parallelism interface, it can be useful to first think about the abstract algorithm—what are the data dependencies, and what can be done in parallel?—and then figure out the details. You can use parallel functional programming to design algorithms without worrying about scheduling, and then translate down to whatever interface you need. Second, it’s our thesis that eventually this kind of parallel programming will be practical and common: as language implementations improve, and computers get more and more cores, this kind of programming will become feasible, and even necessary in order to fully exploit the potential speed-up of using parallelism. You’re going to be writing programs for a long time, and we’re trying to teach you tools that will be useful years down the road.

The Plan

In the next lectures, we will discuss cost semantics, in more detail; and we will introduce sequences, an important data structure with good parallel complexity.
3 Cost Semantics

Cost semantics provides a way to assess the work and span of functional code. The key idea is to use cost graphs. Earlier we showed how to derive recurrence relations for work and span of recursive function applications. Cost graphs are more generally useful (and can be used for arbitrary expressions, not just for recursive function applications). Cost semantics can also be used for asymptotic analysis of work and span, and when used for recursive functions will yield the same results as the method we introduced earlier, based on recurrence relations.

3.1 Cost graphs

A cost graph is a form of series-parallel graph. A series-parallel graph is a directed graph with a designated source node (no edges in) and sink node (no edges out). Such graphs are constructible using two operations, called sequential and parallel composition. We can draw a cost graph as a picture, with nodes that represent units of work, with the source at the top and the sink at the bottom. Edges indicate data dependency, and are shown pointing downward by convention. (We don’t draw arrowheads, relying on the downward convention.)

The particular series-parallel graphs we need are of the following form:

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

The first is a trivial graph with one node (both the source and the sink). The second is the sequential combination of graphs \( G_1 \) and \( G_2 \), formed by by putting an edge from the sink of \( G_1 \) to the source of \( G_2 \); its source is the source of \( G_1 \) and its sink is the sink of \( G_2 \). The third is the parallel combination of graphs \( G_1 \) and \( G_2 \), formed by adding a new source and sink, and adding edges from the source to the sources of \( G_1 \) and \( G_2 \), and from the sinks of \( G_1 \) and the sink of \( G_2 \) to the new sink. We also use \( n \)-ary versions of sequential and parallel composition. We draw an \( n \)-ary parallel combination of graphs \( G_1 \) through \( G_n \) as
Again, a new source and a new sink, with edges from the new source to the sources of the $G_i$, and edges from the sinks of the $G_i$ to the new sink.

(Unfortunately, it's not easy to typeset these pictures in \LaTeX.)

We may sometimes put arrows on the edges to emphasize the flow of control. In the diagrams above we relied on verticality to do the same job.

The work of a cost graph $G$ is the number of nodes in $G$. The span of $G$ is the length of the longest path from the source of $G$ to the sink of $G$, which we may refer to as the critical path, or the diameter of the graph. 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, in which a computation forks into various subcomputations that are run in parallel, but these come back together at a well-defined joint 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

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

in which we labeled nodes to indicate what units of work occur. The structure of this graph says that the summands 1 and 2 are evaluated in parallel; because 1 and 2 are already values, these evaluations are trivial; after this, there is one step for doing the addition. The work of this graph is 5 and the span is 4. (Note: these numbers are an additive constant-factor higher than
we may have said earlier in the course for the work and span of the expression \((1+2)\), because in the graph we count the fork and the join as separate nodes, and we count a step for evaluating a value to itself; asymptotically, this minor detail doesn’t matter.)

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 \(G_1\) and \(G_2\) 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 max of the spans of the individual graphs.

### 3.2 Brent’s Principle

Cost graphs can be used to reason abstractly about the time complexity of your program, independently of the schedule. For example, below, we associate a cost graph with each operation on sequences. You 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.

But 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 (and it’s even unlikely that you’re always going to have “enough” processors). What can we say, based on work and span, about what happens when you evaluate ML code using the 2 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 asymptotically best running time that can be achieved:

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

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, and you 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, it will take at least 8 steps on 2 processors. But if you increase the number of processors to 3, you can achieve the span 5. If you have 5 units of work to do, the fact that you have 2 processors doesn’t help: you still need 5 steps.

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 you design; if not, you got the cost semantics 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.

### 3.3 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 pebble game with \( p \) pebbles. The rules of the game are: in each step, you can pick up a pebble from a node or from your hand and place it on a node all of whose predecessors have been visited, marking that node as visited. To generate a schedule that says what each pebble (processor) does in each timestep, we divide the pebbling up into steps; at each step you can play at most \( p \) pebbles. The nodes played on are the units of work completed in that timestep. The restriction to playing on nodes whose predecessors have been visited ensures that dependencies are respected. The nodes whose predecessors have been visited, but have not themselves been visited, form the frontier, the currently available work.

Consider the following cost graph:
A 2-pebbling with pebbles X and O might start out like this:

<table>
<thead>
<tr>
<th>Step</th>
<th>X</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

In the first step, we can only play 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 to a processor is idle because there is not enough available work that can be done. In the second step, we can play on both of the next two nodes. In the third step, we can play on any of the four nodes at the next level, but we can (and did) choose to play on only one of them. This is a step in which a processor was idle, even though there is work that could have been done. A greedy schedule assigns as many processors as possible work at each time step. (A non-greedy scheduler might be more efficient overall if not all units of work took the same time.) The schedule outlined above is not greedy!

For a fixed number of processors $p$ there are 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, you play as many pebbles (a maximum
of $p$) as you 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):

<table>
<thead>
<tr>
<th>Step</th>
<th>X</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>h</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Step</th>
<th>X</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>j</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

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 2-BFS 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.
Note
The only reason we introduced pebbling and schedules is to illustrate the
ideas behind Brent’s Theorem. You should not think in terms of scheduling
when designing functional code, and you should not worry about scheduling
strategy to figure out how functional programs behave. Instead we advocate
a high-level approach that abstracts away from scheduling issues!

4 Sequences

To introduce sequences, here is a signature containing some of the main
operations that we’ll begin with.

signature SEQ =
  sig
    type 'a seq
    exception Range
    val tabulate : (int -> 'a) -> int -> 'a seq
    val length : 'a seq -> int
    val nth : int -> 'a seq -> '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
  end

(We may add additional functions later.)

Let’s assume we have an implementation of this signature, a structure
Seq:SEQ. 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 SEQ). 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 ML 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 that you are familiar with. However, they have different work and span than the corresponding list functions. Firstly, sequences admit constant-time access to items — nth i s takes constant time, when i is an integer value and s is a sequence value. Secondly, sequences have better parallel complexity—many operations, such as map, act on each element of a sequence in parallel.

For each sequence operation, we will:

• describe its type and (evaluational) behavior abstractly
• give a cost graph, which specifies the work and span.

Before we start, recall that we use the math notation

\[ \langle v_1, \ldots, v_n \rangle \]

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

\[ \langle u_0, \ldots, u_{n-1} \rangle, \]

which denotes the same sequence value if we let \( u_0 = v_1, \ldots, u_{n-1} = v_n \).

tabulate

\[ \text{tabulate} : (\text{int} \to 'a) \to \text{int} \to 'a \text{ seq} \]

The main operation for building sequences is tabulate, which uses a function to generate sequence items, at positions indexed from 0 up to a specified bound, which must be non-negative. So when \( n \geq 0 \) and f is a function value of type \( \text{int} \to t \), \text{tabulate} f n builds a sequence of type t seq, of length n, consisting of the values of f 0 through f(n − 1). We can characterize this operation equationally:

\[ \text{tabulate} f n = \langle f 0, \ldots, f (n - 1) \rangle \]

and its evaluation properties are summarized by:

\[ \text{tabulate} f n \Rightarrow^* \langle v_0, \ldots, v_{n-1} \rangle \]
if \( f \circ \ldots \circ f^{(n-1)} \) \( \Rightarrow^* \) \( v_{n-1} \). Again, this characterization does not imply any specific order of evaluation, and the function calls to \( f \) are independent, so may be evaluated in parallel. The cost graph for \texttt{tabulate f n} looks like

\[
\begin{array}{c}
\text{\( G_0 \)} \quad \text{\( G_1 \)} \ldots \quad \text{\( G(n-1) \)} \\
\end{array}
\]

where each \( G_i \) is the cost graph for \( f^i \). In general,

\[
W(\text{tabulate } f \ n) = \sum \{W(f^0), \ldots, W(f^{(n-1)})\} + c \\
S(\text{tabulate } f \ n) = \max \{S(f^0), \ldots, S(f^{(n-1)})\} + c
\]

So when \( f \) is a constant time function value, the work for \texttt{tabulate f n} is \( O(n) \) and the span is \( O(1) \).

\texttt{nth}

\[\texttt{nth : int -> 'a seq -> 'a}\]

\[\text{exception Range}\]

We define the behavior of \texttt{nth} as

\[
\texttt{nth } i \langle x_0, \ldots, x_{n-1} \rangle = x_i \quad \text{if } 0 \leq i < n \\
= \text{raise Range} \quad \text{otherwise}
\]

For a sequence value \( s \) and integer value \( i \), the cost graph for \texttt{nth i s} is trivial, and \texttt{nth i s} has \( O(1) \) work and \( O(1) \) span. As promised, (using this implementation) sequences provide constant-time access to elements. Notice that indexing starts at 0, not 1. The first item in sequence \( s \) is at index 0, and the final item in \( s \) is at index \( \texttt{length s} - 1 \). Attempts to extract an item at an invalid index will raise the \texttt{Range} exception.

\texttt{length}

\[\texttt{length : 'a seq -> int}\]

The behavior of \texttt{length} is given by:

\[\texttt{length } \langle x_0, \ldots, x_{n-1} \rangle = n\]

When \( s \) is a sequence value the cost graph for \texttt{length s} is trivial, and \texttt{length s} has \( O(1) \) work and span.
map

\[ \text{map} \quad : \quad (\text{`a} \to \text{`b}) \to \text{`a} \quad \text{seq} \to \text{`b} \quad \text{seq} \]

The behavior of map is given by

\[ \text{map} \quad f \quad \langle x_1, \ldots, x_n \rangle = \langle f \quad x_1, \ldots, f \quad x_n \rangle. \]

Each of the function applications may be evaluated in parallel. This is shown by the cost graph of map \( f \langle x_1, \ldots, x_n \rangle \), which looks like

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

where each \( G_i \) is the cost graph for \( f \quad x_i \). (The shape of this graph obviously resembles the cost graph for tabulate.)

If \( f \) is a constant time function value and \( s \) is a sequence value with length \( n \), then map \( f \quad s \) has \( O(n) \) work and \( O(1) \) span.

reduce

\[ \text{reduce} \quad : \quad (\text{`a} \times \text{`a} \to \text{`a}) \to \text{`a} \to \text{`a} \quad \text{seq} \to \text{`a} \]

reduce is intended to be applied to an associative binary function \( g \) of type \( t \times t \to t \) for some \( t \), a value \( z \) that is an identity element for \( g \), and a sequence of items of type \( t \), to produce a value of type \( t \). Since the type is polymorphic, we can choose any type for \( t \) here. A function \( g \) of type \( t \times t \to t \) is associative iff for all values \( a, b, c \) of type \( t \),

\[ g(a, g(b, c)) = g(g(a, b), c). \]

Associativity implies that if we pairwise combine a sequence of values using \( g \), in any order that respects the original enumeration order, we'll get the same result. For example,

\[ ^{1} \text{It's also possible to apply reduce in other situations, such as a non-associative function and/or a value z that is not an identity element. In such cases our specification does not say what happens. If you ever need to use reduce in more relaxed situations like this you would need to adopt a more detailed specification!} \]
\[ g(g(x_1, x_2), g(x_3, x_4)) = g(x_1, g(x_2, g(x_3, x_4))). \]

When \( g \) is associative, we may write

\[ x_1 \ g \ x_2 \ g \ . \ . \ g \ x_n \]

to stand for the result of (any such) combination. This infix form of notation abstracts away from parenthesization, and resembles the ML notation when \( g \) is actually an *infix* function. The value \( z \) is an *identity* for \( g \) if, for all values \( x \) of type \( t \), \( g(x, z) = g(z, x) = x \).

Assuming that \( g \) is associative and \( z \) is an identity for \( g \), the applicative behavior of \texttt{reduce} \( g \ z \) has the following equational characterization: for all \( n \geq 0 \) and all values \( x_1, \ldots, x_n \) of type \( t \),

\[ \texttt{reduce} \ g \ z \langle x_1, \ldots, x_n \rangle = x_1 \ g \ x_2 \ g \ . \ . \ g \ x_n \ g \ z \]

We interpret the empty case as being equal to \( z \), so

\[ \texttt{reduce} \ g \ z \langle \rangle = z. \]

And since we assume that \( z \) is an identity for \( g \), when \( n > 0 \) the first equation is equivalent to

\[ \texttt{reduce} \ g \ z \langle x_1, \ldots, x_n \rangle = x_1 \ g \ x_2 \ g \ . \ . \ g \ x_n. \]

In explaining cost analysis we \emph{do} need to refer to order of evaluation. The implementation of \texttt{reduce} is assumed to use a balanced parenthesization format for the pairwise combinations, so for example the cost graph of

\[ \texttt{reduce} \ g \ z \langle x_1, x_2, x_3 \rangle \]

is the same as the cost graph for

\[ g(g(x_1, x_2), g(x_3, z)) \]

If \( g(a, b) \) is constant time, for all values \( a \) and \( b \), it follows that the cost graph for \texttt{reduce} \( g \ z \langle x_1, \ldots, x_n \rangle \) looks something like the following picture \footnote{In class we give a recurrence relation that explains why!}, with a binary branching structure and two phases (first just branching the data, then combining the data pairwise in a cascade):
Consequently, the graph has size (work) $O(n)$ and critical path length (span) $O(\log n)$. 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**

$$\text{mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b}$$

`mapreduce` is intended to be used with a function $f$ of type $t1 \rightarrow t2$ for some types $t1$ and $t2$, an *associative* function $g$ of type $t2 \times t2 \rightarrow t2$, an identity value $z : t2$ for $g$, and a sequence of items of type $t1$.

The behavior of `mapreduce` $f$ $z$ $g$, assuming that $g$ is associative and $z$ is an identity element for $g$, is given by:

$$\text{mapreduce } f \; z \; g \; \langle x_1, \ldots, x_n \rangle = (f \; x_1) \; g \; (f \; x_2) \; g \; \cdots \; g \; (f \; x_n) \; g \; z$$

The implementation of `mapreduce` uses a balanced parenthesization format for pairwise combinations, as with `reduce`. Write out formulas for the work
and span of mapreduce $f \ z \ g \ s$ and deduce that the asymptotic work and span are the same as for reduce $g \ z \ s$ when $f$ and $g$ are constant time.

5 Examples

In these examples, let’s assume we have open-ed the structure Seq:SEQ, so we don’t need to use qualified names to refer to the sequence functions defined in this structure.

- Building an empty sequence: the function

  \[ \text{empty} : \text{unit} \to \text{'a seq} \]

  is defined by

  \[
  \text{fun empty ( ) = tabulate (fn _ => raise Range) 0}
  \]

  It’s easy to see (using the specs for tabulate) that empty( ) defined like this does return a sequence of length zero. The work and span for empty( ) are both $O(1)$.

- You can implement a cons operation for sequences, with tabulate. This function has type

  \[ \text{cons} : \text{'a} \to \text{'a seq} \to \text{'a seq} \]

  We specify that cons $x \ s$ returns a sequence with length one more than the length of $s$, whose first (0'th) element is $x$, and whose remaining items are the items of $s$ with positions shifted by one:

  \[
  \text{fun cons (x : 'a) (s : 'a seq) : 'a seq =}
  \text{tabulate (fn 0 => x | i => nth (i-1) s) (1 + length s)}
  \]

  When $x$ is a value and $s$ is a sequence value of length $n$, the work for \text{cons} $x \ s$ is $O(n)$ and the span is $O(1)$, because the function

  \[
  \text{fn 0 => x | i => nth (i-1) s}
  \]

  is constant-time. (Remember that \text{nth} (i-1) $s$ takes constant time.)
• A function \texttt{first : int \rightarrow int seq} that constructs the sequence consisting of the first \( n \) integers:

\[
\text{fun first } n = \text{tabulate (fn x:int => x) } n
\]

For \( n \geq 0 \), \texttt{first } \( n \) returns the sequence value

\(<0, 1, \ldots, n-1>\)

of type \texttt{int seq}. The work for \texttt{first } \( n \) is \( O(n) \) and the span is \( O(1) \).

For \( n < 0 \) what happens?

• A function \texttt{squares : int \rightarrow int seq} that constructs the sequence consisting of the first \( n \) square numbers, starting with 0:

\[
\begin{array}{l}
(* \texttt{squares : int \rightarrow int seq *}) \\
(* \text{REQUIRES } n \geq 0 *) \\
(* \text{ENSURES squares } n = \text{the first } n \text{ square numbers *}) \\
\text{fun squares } n = \text{map (fn x => x*x) (first } n) \\
\end{array}
\]

Note that we could have defined \texttt{squares} instead by using a single \texttt{tabulate}, rather than as above, which is actually a \texttt{tabulate} followed by a \texttt{map}. The alternative way is:

\[
\text{fun squares } n = \text{tabulate (fn i => i*i) } n
\]

There is a general “fusion” law that expresses the validity of this kind of result. When \( f : \texttt{int \rightarrow t1} \) and \( g : \texttt{t1 \rightarrow t2} \) are total functions, it follows that \( g \circ f : \texttt{int \rightarrow t2} \) is also total, and for all \( x : \texttt{int}, \)

\[
\text{map } g \ (\text{tabulate } f \ x) = \text{tabulate } (g \circ f) \ x
\]

The fact that the two alternative ways to define \texttt{squares} are equivalent is just a special case of this fusion property, obtained by letting \( f \) be \( \text{fn x:int => x} \) and \( g \) be \( \text{fn x:int \rightarrow x*x} \).

• Similarly to what happened with lists, there is a map-fusion law for sequences. When \( f : \texttt{t1 \rightarrow t2} \) and \( g : \texttt{t2 \rightarrow t3} \) are total functions, so is \( (g \circ f) : \texttt{t1 \rightarrow t3} \), and we have the following equation:
\[ \text{map } (g \circ f) = (\text{map } g) \circ (\text{map } f) \]

As an example, consider

\[
\begin{align*}
\text{fun } & \text{square } x = x \times x \\
\text{fun } & \text{quads } s = \text{map } \text{square} \ (\text{map } \text{square} \ s)
\end{align*}
\]

Here \( \text{quads} \) is extensionally equal to \( (\text{map } \text{square}) \circ (\text{map } \text{square}) \). And \( \text{quads} \) is extensionally equal to \( \text{map } (\text{square } \circ \text{square}) \). Since \( \text{square } \circ \text{square} \) is extensionally equal to \( \text{fn } x: \text{int } => x \times x \times x \times x \), we have

\[
\text{quads} = \text{map } (\text{fn } x: \text{int } => x \times x \times x \times x)
\]

- Counting, revisited.

In the first lecture, we defined some (recursive) ML functions that operate on lists of integers, combining the integers together by adding them. Later we saw how to use \text{foldl} and \text{foldr} to do the same job. Now we can revisit these ideas using sequences.

Here is a function \( \text{sum} : \text{int seq } \rightarrow \text{int} \) for adding the integers in a sequence:

\[
\begin{align*}
\text{fun } & \text{sum } (s : \text{int seq}) : \text{int } = \text{reduce } (\text{op }+) \ 0 \ s
\end{align*}
\]

Note that \( (\text{op }+) \) is associative, and 0 is an identity element for addition, so this use of \text{reduce} meets the requirements of the specification given earlier.

And here is a function \( \text{count} : \text{int seq seq } \rightarrow \text{int} \) for combining the integers in a sequence of integer sequences:

\[
\begin{align*}
\text{fun } & \text{count } (s : \text{int seq seq}) : \text{int } = \text{sum } (\text{map } \text{sum} \ s)
\end{align*}
\]

\( \text{sum} \) takes an integer sequence and adds up all the numbers in it using \text{reduce}, just like we did with folds for lists and trees. \( \text{count} \) 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 \( \text{count} \) with \text{mapreduce}, so it takes only one pass).
We can now see that \texttt{count} on a sequence of \( n \) sequences, each of length \( n \), requires \( O(n^2) \) work: \texttt{sum} \( s \) is implemented using \texttt{reduce} with constant-time arguments, and thus has \( O(n) \) work and \( O(\log n) \) span, where \( n \) is the length of \( s \). Each call to \texttt{sum} inside the \texttt{map} is on a sequence of length \( n \), and thus takes \( O(n) \) work. This function is mapped across \( n \) rows, yielding \( O(n^2) \) work for the \texttt{map}. The row sums also form a sequence of length \( n \), so the final \texttt{sum} contributes \( O(n) \) more work, which is subsumed by the \( O(n^2) \). So the total work here is \( O(n^2) \), as promised. However, the span is \( O(\log n) \): the \texttt{map} doesn’t contribute anything, and both the inner and outer \texttt{sums} are on sequences of length \( n \), and therefore have \( O(\log n) \) span. The total span is the sum of the inner span and the outer span, because of the data dependency: the outer additions happen after the inner sums have been computed. The sum of \( \log n \) and \( \log n \) is still \( O(\log n) \), so the total span is \( O(\log n) \).

(In the lecture slides this example is explained using cost graphs, and you should compare the two work/span derivations to make sure you understand how cost graphs can be used.)

• Reversing a sequence, a nice use of re-indexing:

\begin{verbatim}
fun reverse (s : 'a seq) : 'a seq =
  tabulate (fn i => nth (length s - i - 1) s) (length s)
\end{verbatim}

This implementation of reverse has linear work and constant span! Reason: this follows from the work/span properties of \texttt{length} and \texttt{tabulate} stated earlier.

So we have here slick way to do reversal of sequences that avoids the quadratic runtime of the naïve list reversal function. Yet another situation in which lists don’t facilitate parallelism but sequences do. But index calculations like those done inside the tabulation function tend to be hard to read and get right, so try to avoid them when there’s an obvious alternative (or prove that you got them right!). For example, the following function definition is almost identical to the one above but doesn’t work as intended because the indices are off by 1:

\begin{verbatim}
fun reverse2 (s : 'a seq) : 'a seq =
  tabulate (fn i => nth (length s - i) s) (length s)
\end{verbatim}

23
What does it do?

Note that with `nth` and `tabulate` you can (if you are not careful) write very index-y code in which you keep extracting items from sequences. Sometimes this is fine, but in general use this style sparingly: it can be hard to read, and you may be doing unnecessary computation! For example, never write something like

\[
\text{tabulate } (\text{fn } i \Rightarrow \ldots \text{nth } i \ s \ldots) \ (\text{length } s)
\]

if the function body doesn’t otherwise mention \(i\): you’d be reimplementing `map` in a hard-to-read way, and you would probably be able to express your algorithmic intentions more elegantly as something like

\[
\text{map } (\text{fn } x \Rightarrow \ldots \ x \ldots) \ s
\]

Stylistically this use of `map` is likely to be clearer and just as effective.

**Reflection**

Bear in mind that the above discussion assumed we had chosen a specific implementation `Seq` of the signature `SEQ`. There are many different implementations that could be built, including:

- a list-based implementation
- binary trees with sequence items at the nodes
- balanced binary trees
- balanced binary trees enhanced with extra information at the nodes, to help speed up some operations.

You might want to think about the consequences of choosing one implementation over another. For example, some operations may be faster in one sequence implementation than in others. However, there may not always be a clear advantage.
6 Self-test

Assume we have a structure that implements the signature SEQ from above, with sequence functions that have the work and span properties discussed above. Assume we have opened this structure, so these functions are available at the top level.

1. The count function of type int seq seq -> int was defined earlier as:

       fun sum L = reduce (op +) 0 L;
       fun count S = sum (map sum S)

Show that this function can also be defined, again using sum, as:

       fun count S = mapreduce sum (op +) 0 S

In other words, these two versions of the function are extensionally equivalent.

2. Let n ≥ 0 and let f and g be functions of type int -> t such that f i = g i for i = 0,...,i = n - 1. Recall what it means to say that two sequence values are “equal”: they have the same length and equal items, in the same order. Show that tabulate f n = tabulate g n.

3. Here is a function

       append : 'a seq * 'a seq -> 'a seq

that appends two sequences:

       fun append (s1 : 'a seq, s2 : 'a seq) : 'a seq =
           let
               val (n1, n2) = (length s1, length s2)
           in
               tabulate (fn i => if i < n1
                           then nth i s1
                           else nth (i - n1) s2) (n1 + n2)
           end
(i) What are the work and span for \texttt{append(s1, s2)} when \texttt{s1} and \texttt{s2} are sequence values of length \texttt{n1} and \texttt{n2}, respectively? Contrast this with the work and span for \texttt{L1@L2} when \texttt{L1} and \texttt{L2} are list values of these lengths.

(ii) Show that for all sequence values \texttt{s}, \texttt{append(s, empty()) = s} and \texttt{append(empty(), s) = s}.

(iii) Show that for all sequence values \texttt{s1, s2, s3} of the same type, \texttt{append(s1, append(s2, s3)) = append(append(s1, s2), s3)}.

(iv) Write a function \texttt{flatten : 'a seq seq -> 'a seq} that appends all the sequences in a sequence of sequences. What are the work and span for \texttt{flatten S} when \texttt{S} is a sequence of \texttt{n} sequences, each of which has length \texttt{n^2}?

4. Using \texttt{tabulate} and \texttt{nth}, write a function

\texttt{zip : 'a seq * 'b seq -> ('a * 'b) seq}

that satisfies the following specification:

\texttt{REQUIRES s1 and s2 are sequences with the same length}

\texttt{ENSURES zip (s1 s2) = a sequence of the same length as s1 and s2}

\texttt{and for all i:int,}

\texttt{nth i (zip (s1, s2)) = (nth i x1, nth i x2)}

What are the work and span for \texttt{zip (s1, s2)}, when \texttt{s1} and \texttt{s2} are sequence values of length \texttt{n}?

5. Show that for all suitably typed function values \texttt{f} and \texttt{g}, and all \texttt{n \geq 0},

\texttt{zip (tabulate f n, tabulate g n) = tabulate (fn i => (f i, g i)) n}

6. Now think about some different ways to implement the \texttt{SEQ} signature:

(a) as balanced binary trees, with the first sequence item at the root

(b) as balanced binary trees, with the first sequence item \texttt{paired with the tree’s size} at the root

For each one, figure out how you would write the various sequence functions and what the work and span would be. In (b) be careful about picking a sensible representation invariant. You should find that keeping size information at nodes allows \texttt{length} to be faster!