Sequence Reference

Last Revised: 30th October 2013

Please cite this document if you use it in your homework.

The type `Seq.seq` represents sequences. Sequences are *parallel collections*: ordered collections of things, 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 type with only the operations we're about to describe. The differences between sequences and lists or trees is the cost of the operations, which we specify below.

1 The Signature

- `Seq.length : 'a Seq.seq -> int`
  `Seq.length s` evaluates to the number of items in `s`.
- `Seq.empty : unit -> 'a Seq.seq`
  `Seq.empty ()` evaluates to the sequence of length zero.
- `Seq.singleton : 'a -> 'a Seq.seq`
  `Seq.singleton x` evaluates to a sequence of length 1 where the only item is `x`.
- `Seq.append : 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`
  If `s1` has length l_1 and `s2` has length l_2, `Seq.append` evaluates to a sequence with length $l_1 + l_2$ whose first l_1 items are the sequence `s1` and whose last l_2 items are the sequence `s2`.
- `Seq.tabulate : (int -> 'a) -> int -> 'a Seq.seq`
  `Seq.tabulate f n` evaluates to a sequence `s` with length `n` where the i^{th} item of `s` is the result of evaluating `(f i)`. This is zero-indexed. `Seq.tabulate f i` raises `Range` if `n` is less than zero.
- `Seq.nth : int -> 'a Seq.seq -> 'a`
  `nth i s` evaluates to the i^{th} item in `s`. This is zero-indexed. `Seq.nth i s` will raise `Range` if `i` is negative or greater than `(Seq.length s)-1`.
- `Seq.filter : ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter p s` returns the longest subsequence `ss` of `s` such that `p` evaluates to `true` for every item in `ss`.[1]
- `Seq.map : ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`
  `Seq.map f s` maps `f` over the sequence `s`. That is to say, it evaluates to a sequence `s'` such that `s` and `s'` have the same length and the i^{th} item in `s'` is the result of applying `f` to the i^{th} item of `s`.
- `Seq.reduce : (('a * 'a) -> 'a) -> 'a -> 'a Seq.seq -> 'a`
  `Seq.reduce c b s` combines all of the items in `s` pairwise with `c` using `b` as the base case. `c` must be associative, with `b` as its identity.

[1] Here we use the term "subsequence" to mean any subsequence of a sequence, not necessecarily one whose elements are consecutive in the original sequence. For example, $\langle\rangle$, $\langle 3\rangle$, and $\langle 2, 4\rangle$ are subsequences of $\langle 1, 2, 3, 4\rangle$.

- `Seq.mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`
  `Seq.mapreduce l e n s` is equivalent to `Seq.reduce n e (Seq.map l s)`.

- `Seq.toString : ('a -> string) -> 'a Seq.seq -> string`
  `Seq.toString ts s` evaluates to a string representation of `s` by using `ts` to convert each item in `s` to a `string`.

- `Seq.repeat : int -> a -> 'a Seq.seq`
  `Seq.repeat n x` evaluates to a sequence consisting of exactly `n`-many copies of `x`.

- `Seq.flatten : 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten ss` is equivalent to `reduce append (empty ()) ss`

- `Seq.zip : ('a Seq.seq * 'b Seq.seq) -> ('a * 'b) Seq.seq`
  `Seq.zip (s1,s2)` evaluates to a sequence whose n^{th} item is the pair of the n^{th} item of `s1` and the n^{th} item of `s2`.

- `Seq.split : int -> 'a Seq.seq -> 'a Seq.seq * 'a Seq.seq`
  If `s` has at least `i` elements, `Seq.split i s` evaluates to a pair of sequences `(s1,s2)` where `s1` has length `i` and `append s1 s2` is the same as `s`. Otherwise it raises `Range`.

- `Seq.take : int -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.take i s` evaluates to the sequence containing exactly the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

- `Seq.drop : int -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.drop i s` evaluates to the sequence containing all but the first `i` elements of `s` if $0 \leq i \leq$ `length s`, and raises `Range` otherwise.

- `Seq.cons : 'a -> 'a Seq.seq -> 'a Seq.seq`
  If the length of `xs` is `l`, `Seq.cons x xs` evaluates to a sequence of length `l+1` whose first item is `x` and whose remaining `l` items are exactly the sequence `xs`.
  (**NOTE:** Please do not use `Seq.cons` unless it is specifically suggested. If it is overused, it can lead to a sequential style that is somewhat against the spirit of sequences.)

2 Cost Bounds

Unfortunately, there is no known way of stating time complexity of a higher order function, such as map, itself abstractly in the function—there is no theory of asymptotic analysis for higher-order functions. Therefore, this chart **assumes that all functions that are given as arguments take constant time.** To consider the cost if these functions do not take constant time, we will need to go back to the cost graphs and expand them to include the additional work and span.

Function	Work	Span
`Seq.length S`	$O(1)$	$O(1)$
`Seq.empty ()`	$O(1)$	$O(1)$
`Seq.singleton x`	$O(1)$	$O(1)$
`Seq.append S1 S2`	$O(\|S1\| + \|S2\|)$	$O(1)$
`Seq.tabulate f n`	$O(n)$	$O(1)$
`Seq.nth i S`	$O(1)$	$O(1)$
`Seq.filter p S`	$O(\|S\|)$	$O(log\|S\|)$
`Seq.map f S`	$O(\|S\|)$	$O(1)$
`Seq.reduce c b S`	$O(\|S\|)$	$O(log\|S\|))$
`Seq.mapreduce l e n S`	$O(\|S\|)$	$O(log\|S\|))$
`Seq.toString ts S`	$O(\|S\|)$	$O(log\|S\|)$
`Seq.repeat n x`	$O(n)$	$O(1)$
`Seq.flatten S`	$O(\sum_{s \in S} \|s\|)$	$O(1)$
`Seq.zip (S1,S2)`	$O(min(\|S1\|, \|S2\|)$	$O(1)$
`Seq.split i S`	$O(\|S\|)$	$O(1)$
`Seq.take i S`	$O(i)$	$O(1)$
`Seq.drop i S`	$O(i)$	$O(1)$
`Seq.cons x S`	$O(\|S\|)$	$O(1)$

3 Detailed Cost Bounds

Intuitively, these sequence operations do the same thing as the operations on lists that you are familiar with. However, they have different time complexity than the list functions: First, sequences admit constant-time access to elements—`nth` takes constant time. Second, sequences have better parallel complexity—many operations, `map` act on each element of the sequence in parallel.

For each function, we (1) describe its behavior abstractly and (2) give a cost graph.

Length

The behavior of `length` is

```
length <x1 , ... , xn> == n
```

The cost graph for `length s` is

As a consequence, `length` has $O(1)$ work/span.

Nth

Sequences provide constant-time access to elements. Abstractly, we define the behavior of `nth` as

```
nth <x0 , ... , xn-1> i == xi if 0 <= i < n
                           or raises Range if i>=n
```

Here `<x1 , ... , xn>` is *not* SML syntax, but mathematical syntax for a sequence value.

The cost graph for `nth i s` is

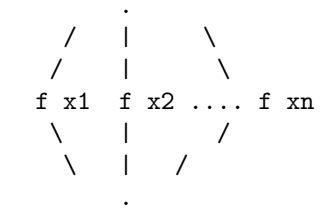
As a consequence, `nth` has $O(1)$ work/span.

Map

The the behavior of `map f <x1,...xn>` is

```
map f <x1 , ... , xn> ==> <f x1,...,f xn>
```

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



where we write `f x1`, etc. for the cost graphs associated with these expressions.

As a consequence, if `f` takes constant time, then `map f` has $O(n)$ work and $O(1)$ span.

Reduce

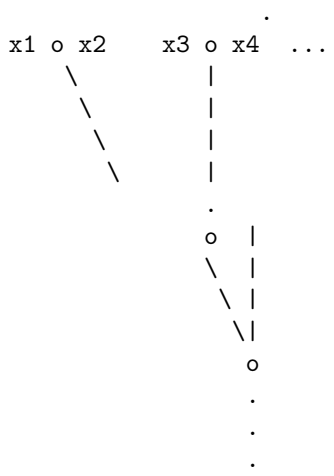
The behavior of `reduce` is

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

That is, `reduce` applied its argument function (which we write here as infix $\odot$) between every pair of elements in the sequence.

However, the right-hand side is ambiguous, because we have not parenthesized it. There are two options: First, we could assume the function $\odot$ is associative, with unit b, in which case these all mean the same thing. However, there are some useful non-associative operations (e.g. floating point). So the second option is to specify a particular parenthesization $(x_1 \odot (x_2 \odot \ldots \odot (x_n \odot b)) \ldots)$ or $(\ldots (x_1 \odot x_2) \odot \ldots \odot x_n)$ or the balanced tree $((x_1 \odot x_2) \odot (x_3 \odot x_4)) \odot \ldots$ (with b filled in at the end if the sequence has odd length). Unless we say otherwise, you can assume the balanced tree.

The cost graph for $\texttt{reduce} \odot\ b\ \langle x_1, \ldots x_n \rangle$ is

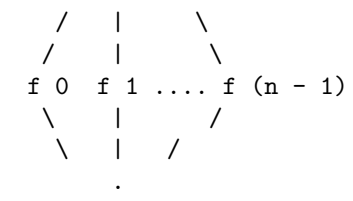


(with the last pair being either `xn-1 o xn` or `xn o b` depending on the parity of the length). That is, it is the graph of the balanced parenthesization described above. Consequently, if $\odot$ takes constant time (e.g. `op+`) then the graph has size (work) $O(n)$ and critical path length (span) $O(\log n)$. Reduce does not have constant span, because later applications of $\odot$ depend on the values of earlier ones.

Tabulate The way of introducing a sequence is *tabulate*, which constructs a sequence from a function that gives you the element at each position, from 0 up to a specified bound:

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

The cost graph (and therefore time complexities) for tabulate is analogous to the graph for `map`:



Note that with `nth` and `tabulate` you can write very index-y array-like code. Use this sparingly: it's hard to read! E.g. never write

```
Seq.tabulate (fn i => ... nth i s ...)
             (Seq.length s)
```

if the function doesn't otherwise mention `i`: you're reimplementing `map` in a hard-to-read way!

```
Seq.map (fn x => ... x ...) s
```

4 Thinking About Cost

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 such a way that it can make use of this ability to do work on multiple processors simultaneously. At the lowest level, this means deciding, at each moment in time, what to do on each processor. This is limited 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 addition, but you cannot do the outer addition until you have done the inner two. So even with three processors, you cannot perform the calculation in fewer than two timesteps. That is, the expression has work 3 but span 2.

The approach to parallelism that we're advocating in this class is based on raising the level of abstraction at which you can think, by *separating the specification of what work there is to be done from the schedule that maps it onto processors*. As much as possible, you, the programmer, worry about specifying what work there is to do, and the compiler takes care of scheduling it onto processors. Three things are necessary to make this separation of concerns work:

1. The code itself must not bake in a schedule.
2. You must be able to reason about the *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.

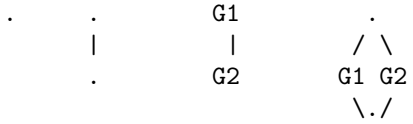
Our central tool for avoiding baking in a schedule is *functional programming*. First, we focus on bulk operations on big collections which do not specify a particular order in which the operations on each element are performed. For example, today, we will talk about *sequences*, which come with an operation `map f <x1,x2,...,xn>` that is specified by saying that its value is the sequence `<f x1, f x2, ... f xn>`. This specifies the data dependencies (to calculate `map`, you need to calculate `f x1` ...) without specifying a particular schedule. You can implement the same computation with a loop, saying "do `f x1`, then do `f x2`,...", but this is inlining a particular schedule into the code—which is bad, because it gratuitously throws away opportunities for parallelism. Second, functional programming focuses on pure, mathematical functions, which are evaluated by calculation. This limits the dependence of one chunk of work on another to what it is obvious from the data-flow in the program. For example, when you `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. It is in general undecidable to take an imperative program and notice notice, after the fact, that what you really meant by that loop was a bulk operation on a collection, or that this particular piece of code really defines a mathematical function.

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 some 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 dependencies between tasks? what can possibly be done in parallel?—and then figure out the details. You can use parallel functional programming to design algorithms, and then translate them down to whatever interface you need. Second, it's our thesis that eventually this kind of

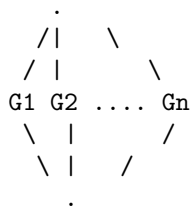
parallel programming will be practical and common: as language implementations improve, and computers get more and more cores, this kind of programming will become possible and even necessary. You're going to be writing programs for a long time, and we're trying to teach you tools that will be useful years down the road.

4.1 Cost Semantics Reminder

A cost graph is a form of *series-parallel graphs*. A series-parallel graph is a directed graph (we always draw a cost graph so that the edges point down the page) with a designated source node (no edges in) and 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:



The first is a graph with one node; the second is a graph with two nodes with one edge between them. The third, *sequential combination*, is the graph formed by putting an edge from the sink of `G1` to the source of `G2`. The fourth, *parallel combination*, is the graph formed by adding a new source and sink, and adding edges from the source to the source of each of `G1` and `G2`, and from the sinks of each of them to the new sink. We also need an n-ary parallel combination of graphs `G1 ... Gn`



The *work* of a cost graph is the number of nodes. The *span* is the length of the longest path, which we may refer to as the *critical path*, or the *diameter*. 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 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.