# Lecture 5 — More on Sequences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

*Lectured by Margaret Reid-Miller — 31 January 2012*

Today, we'll continue our discussion of sequence operations. In particular, we'll take a closer look at `reduce` and its costs with higher order functions, another example using `scan`, and finally how to implement `reduce` and `scan`.

## 1 Analyzing the Costs of Higher Order Functions

Last lecture we looked at using `reduce` to solve divide-and-conquer problems. In the example we gave, the maximum increasing subsequence sum, the combining function $f$ had $O(1)$ cost (i.e., both its work and span are constant). In that case the cost specifications of `reduce` on a sequence of length $n$ is simply $O(n)$ work and $O(\log n)$ span. Does that hold true when the combine function does not have constant cost?

For `map` it is easy to find its costs base on the cost of the function applied:

$$\begin{aligned} W(\texttt{map}\ f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\ S(\texttt{map}\ f\ S) &= 1 + \max_{s \in S} S(f(s)) \end{aligned}$$

Tabulate is similar. But can we do the same for `reduce`?

**Merge Sort.** As an example, let's consider merge sort. As you have seen from previous classes, merge sort is a popular divide-and-conquer sorting algorithm with optimal work. It is based on a function `merge` that takes two already sorted sequences and returns a sorted sequence that combines all inputs. We can use our reduction technique from last time to implement merge sort with a `reduce`. In particular, we can write merge sort as follows:

```
val combine = merge
val base = singleton
val emptyVal = empty()
reduce combine emptyVal (map base A)
```

More simply, we can write

```
fun reduce_sort s = reduce merge< (Seq.empty) (map singleton s)
```

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

In this case, we are using $\texttt{merge}_<$ to denote a merge function that uses an (abstract) comparsion operator $<$. It turns out to be possible to merge two sequences $S_1$ and $S_2$ with lengths $n_1$ and $n_2$ with the following costs:

$$\begin{aligned} W(\texttt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

*What do you think the cost of* `reduce_sort` *is?*

## 2 Reduce: Cost Specifications

We want to analyze the cost of `reduce_sort`. Does the reduction order matter? Or does it not matter because any order will be equally good?

To answer this question, let's consider the following reduction order based on `iter`: the function merges the divides the input into 1 and $n-1$ elements, recurse into the two parts, and run a merge. Thus, on input $x = \langle x_1, x_2, \ldots, x_n \rangle$, the sequence of `merge` calls looks like the following:

```
merge(... merge( merge( merge(I,⟨x1⟩), ⟨x2⟩), ⟨x3⟩), ...)
```

A closer look at this sequence shows that `merge` is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with 1-element sequences, so on so forth. Therefore, the total work using merge is

$$W(\texttt{reduce\_sort}\ x) \ \leq\ \sum_{i=1}^{n-1} c \cdot (1+i) \ \in\ O(n^2)$$

since merge on sequences of lengths $n_1$ and $n_2$ has $O(n_1 + n_2)$ work.

As an aside, this reduction order is the order that the `iter` function uses. Furthermore, using this reduction order, the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

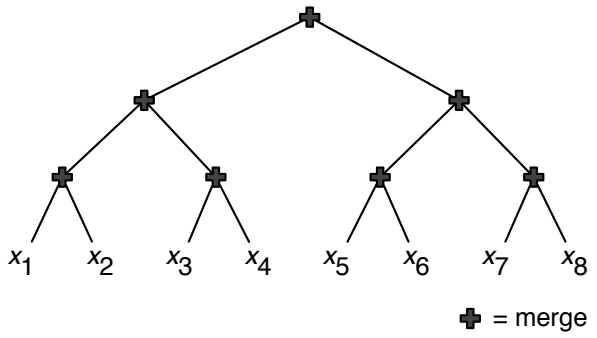
Clearly, this reduction order has no parallelism except within each merge, and therefore the span is

$$S(\texttt{reduce\_sort}\ x) \ \leq\ \sum_{i=1}^{n-1} c \cdot \log(1+i) \ \in\ O(n \log n)$$

since merge on sequences of lengths $n_1$ and $n_2$ has $O(\log(n_1 + n_2))$ span.

Notice that in the reduction order above, the reduction tree was extremely unbalanced. Would the costs change if the merges are balanced? For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a "full" binary tree[1] with $2^k$ leaves and merge according to the tree structure. As an example, the merge sequence for $|x| = 2^3$ is shown below.

[1] This is simply a binary tree in which every node other than the leaves have exactly 2 children.




Clearly using this balanced combining tree gives a smaller span than the imbalanced tree. But does it also reduce the work cost? At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each (these were generated using a `map`). Stepping up one level, there are $n/2$ nodes, each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level $i$ (with $i = 0$ at the root), we have $2^i$ nodes where each node is a merge with input two sequences of length $n/2^{i+1}$. Therefore, the work of `reduce_sort` using this reduction order is the familar sum

$$\begin{aligned} W(\texttt{reduction\_sort}\ x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$. In fact, this algorithm is essentially the merge sort algorithm.

As we discussed last time, to be deterministic, the reduction order for `reduce` must be defined precisely in the ADT. These two examples, however, illustrate how the particular reduction order we choose can lead to drastically different costs. When combining function has $O(1)$ work, using a balanced reduction tree improves the span from $O(n)$ to $O(\log n)$ but does not change the work. But with `merge` as the combining function, we see that the balanced reduction tree also improves the work from $O(n^2)$ to $O(n \log n)$.

In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree, we'll first define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ as

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) = \Big\{\text{all function applications } f(a,b) \text{ in the reduction tree}\Big\}.$$

Following this definition, we can state the cost of `reduce` as follows:

$$\begin{aligned} W(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(f(a,b))\right) \\ S(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f(a,b))\right) \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine operations. The span bound is more interesting. The $\log n$ term expresses the fact that the tree is at

most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,b))$, any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b)))$ span.

This can be used to prove the following lemma:

**Lemma 2.1.** *For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathbb{R}_+$, if for any $x, y$,*

1. $s(f(x,y)) \leq s(x) + s(y)$ *and*
2. $W(f(x,y)) \leq c_f \left(s(x) + s(y)\right)$ *for some universal constant $c_f$ depending on the function $f$,*

*then*

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log |S| \sum_{x \in S} s(x)\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce mergeI}\ \langle\rangle\ \langle\langle a\rangle : a \in A\rangle) = O(|A| \log |A|)$$

## Copy Scan

Previously, we used `scan` to compute partial sums to solve the maximum contiguous subsequence sum problem and to match parentheses. Scan is also useful when you want pass information along the sequence. For example, suppose you have some "marked" elements that you would like to copy across to their right until they reach another marked element. One way to mark the elements is to use options.

That is, suppose you are given a sequence of type $\alpha$ `option seq`. For example

$$\langle \texttt{NONE}, \texttt{SOME}(7), \texttt{NONE}, \texttt{NONE}, \texttt{SOME}(3), \texttt{NONE} \rangle$$

and your goal is to return a sequence of the same length where each element receives the previous SOME value. For the example:

$$\langle \texttt{NONE}, \texttt{NONE}, \texttt{SOME}(7), \texttt{SOME}(7), \texttt{SOME}(7), \texttt{SOME}(3) \rangle$$

Using a sequential loop or `iter` would be easy. How would you do this with `scan`?

If we are going to use a `scan` directly, the combining function $f$ must have type

$$\alpha\ option \times \alpha\ option \to \alpha\ option$$

How about

```
1  fun copy(a, b) =
2    case b of
3      SOME(_) ⇒ b
4    | NONE ⇒ a
```

What this function does is basically pass on its right argument if it is `SOME` and otherwise it passes on the left argument.

There are many other applications of scan in which more involved functions are used. One important case is to simulate a finite state automata.

## 3 Contraction and Implementing Scan

Now let's consider how to implement `scan`. For associated $f$, we can define `scan` as follows:

```
1  fun scan f I S =
2    (⟨ reduce f I (take(S, i) : i ∈ ⟨0, ... n − 1⟩ ⟩,
3     reduce f I S)
```

In this code the notation $\langle$ `reduce` $f\ I$ (`take`$(S,i)) : i \in \langle 0,\ldots,n-1\rangle\rangle$ indicates that for each $i$ in the range from 0 to $n-1$ apply reduce to the first $i$ elements of $S$. For example,

$$\begin{aligned}\texttt{scan + 0 }\langle 2,1,3\rangle &= (\langle \texttt{reduce + 0 }\langle\,\rangle, \texttt{reduce + 0 }\langle 2\rangle, \texttt{reduce + 0 }\langle 2,1\rangle\ \rangle \\ &\qquad \texttt{reduce + 0 }\langle 2,1,3\rangle) \\ &= (\langle 0,2,3\rangle, 6)\end{aligned}$$

Using a bunch of reduces, however, is not an efficient way to calculate the partial sums.

**Exercise 1.** *What is the work and span for the* `scan` *code shown above, assuming $f$ takes constant work.*

Beyond the wonders of what it can do, a surprising fact about `scan` is that it can be accomplished in parallel, although on the surface, the computation it carries out appears to be sequential in nature. How can an operation that computes all prefix sums possibly be parallel? At first glance, we might be inclined to believe that any such algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. It is this apparent dependency that makes `scan` so powerful. We often use `scan` when it seems we need a function that depends on the results of other elements in the sequence, for example, the copy scan above.

Suppose we're to run `plus_scan` (i.e. `scan (op +)`) on the sequence $\langle 2,1,3,2,2,5,4,1\rangle$. What we should get back is

$$(\langle 0,2,3,6,8,10,15,19\rangle, 20)$$

**Thought Experiment I:** At some level, this problem seems like it can be solved using the divide-and-conquer approach. Let's try a simple pattern: divide up the input sequence in half, recursively solve each half, and "piece together" the solutions. A moment's thought shows that the two recursive calls are not independent—indeed, the right half depends on the outcome of the left one because it has to know the cumulative sum. So, although the work is $O(n)$, we effectively haven't broken the chain of sequential dependencies. In fact, we can see that any scheme that splits the sequence into left and right parts like this will essentially run into the same problem.

**Contraction:** To compute `scan` in parallel, we'll introduce a new inductive technique common in algorithms design: contraction. It is inductive in that such an algorithm involves solving a smaller instance of the same problem, much in the same spirit as a divide-and-conquer algorithm. But with contraction, there is only one recursive call. In particular, the contraction technique involves the following steps:

1. Contract the instance of the problem to a (much) smaller instance (of the same sort)
2. Solve the smaller instance recursively
3. Use the solution to help solve the original instance

For intuition, we'll look at the following analogy, which might be a stretch but should still get the point across. Imagine designing a new car by building a small and greatly simplified mock up of the car in mind. Then, the mock-up can be used to help build the actual final car, which probably involve a number of refining iterations that add in more and more details. One could even imagine building multiple mock-ups each smaller and simpler than the previous to help build the final car.

The contraction approach is a useful technique in algorithm design, whether it applies to cars or not. For various reasons, it is more common in parallel algorithms than in sequential algorithms, usually because both the contraction and expansion steps can be done in parallel and the recursion only goes logarithmically deep because the problem size is shrunk by a constant fraction each time.

We'll demonstrate this technique first by applying it to a slightly simpler problem, `reduce`. To begin, we have to answer the following question: *How do we make the input instance smaller in a way that the solution on this smaller instance will benefit us in constructing the final solution?*

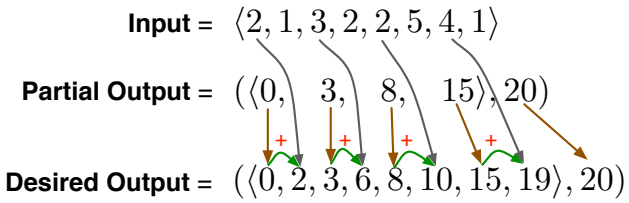
The idea is simple: We apply the combining function pairwise to adjacent elements of the input sequence and recursively run `reduce` on it. In this case, the third step is a "no-op"; it does nothing. For example on input sequence $\langle 2,1,3,2,2,5,4,1\rangle$ with addition, we would contract the sequence to $\langle 3,5,7,5\rangle$. Then we would continue to contract recursively to get the final result. There is no expansion step.

**Thought Experiment II:** How can we use the same idea to evaluate span? What would be the result after the recursive call? In the example above it would be

$$(\langle 0,3,8,15\rangle, 20).$$

But notice, this sequence is every other element of the final scan sequence, together with the final sum—and this is enough information to produce the desired final output. This time, the third expansion step is needed to fill in the missing elements in the final scan sequence: Apply the combining function element-wise to the even elements of the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recursive call:




This leads to the following code. We'll first present `scan` in pseudocode for when $n$ is a power of two, and then an actual implementation of `reduce` and `scan` in Standard ML.

```
1  % implements: the Scan problem on sequences that have a power of 2 length
2  fun scanPow2 f i s =
3    case |s| of
4      0 ⇒ (⟨⟩, i)
5    | 1 ⇒ (⟨i⟩, s[0])
6    | n ⇒
7        let
8          val s' = ⟨f(s[2i], s[2i+1]) : 0 ≤ i < n/2⟩
9          val (r, t) = scanPow2 f i s'
10       in
11         (⟨p_i : 0 ≤ i < n⟩, t), where p_i = { r[i/2]          if even(i)
                                               { f(r[i/2], s[i-1]) otherwise.
12       end
```

Notice that the reduction tree for `reduce` that we showed in the previous lecture is the same tree that the contract step uses in `scan`. In this way, the final sum in the `scan` output (the second of the pair) is the same as for `reduce`, even when the combining function is non-associative. Unfortunately, the same is not true for the scan sequence (the first of the pair); when the combining function is non-associative, the resulting scan sequence is not necessarily the same as applying `reduce` to prefixes of increasing length.

## 4 SML Code

```
fun reduce f i s =
   case length s
    of 0 =>  i
     | 1 =>  f(i, nth s 0)
     | n =>
       let
         val s' = tabulate
                     (fn i => case (2*i = n - 1) of
                                 true =>  (nth s (2*i))
                               | _    => f(nth s (2*i), nth s (2*i + 1)))
                     ((n+1) div 2)
       in
         reduce f i s'
       end

fun scan f i s =
   case length s
    of 0 => (empty(), i)
     | 1 => (singleton i, f(i, nth s 0))
     | n =>
       let
         val s' = tabulate
```

```
              (fn i => case (2*i = n - 1) of
                           true =>  (nth s (2*i))
                         | _    => f(nth s (2*i), nth s (2*i + 1)))
              ((n+1) div 2)
  val (r, t) = scan f i s'
  fun interleave i = case (i mod 2) of
                       0 =>  (nth r (i div 2))
                     | _ => f(nth r (i div 2), nth s (i-1))
in
  (tabulate interleave n, t)
end
```