Lecture 5 — More on Sequences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 13, 2011

Today, we'll continue our discussion of sequence operations. In particular, we'll take a closer look at `reduce`, how many divide-and-conquer algorithms can be naturally implemented using `reduce`, and how to implement `Seq.scan` and `Seq.fields`.

1 Divide and Conquer with Reduce

Let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
fun myDandC($S$) =
  case showt($S$) of
     EMPTY ⇒ $\boxed{\texttt{emptyVal}}$
   | ELT($v$) ⇒ $\boxed{\texttt{base}}$($v$)
   | NODE(L, R) ⇒ let
        val $L'$ = myDandC($L$)
        val $R'$ = myDandC($R$)
     in
        $\boxed{\texttt{someMessyCombine}}$($L'$, $R'$)
     end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. You have seen this in Homework 1 in which we asked for a reduce-based solution for the stock market problem. Turning such a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

$$\textit{reduce}\ \boxed{\texttt{someMessyCombine}}\ \boxed{\texttt{emptyVal}}\ (\textit{map}\ \boxed{\texttt{base}}\ S)$$

Let's take a look at two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm.

Stock Market Using Reduce. The first example is taken from your Homework 1. Given a sequence S of stock values the stock market problem returns the largest increase in price from a low point to a future high point—more formally:

$$\texttt{stock}(S) = \max_{i=1}^{|S|}\left(\max_{j=i}^{|S|}(S_j - S_i)\right)$$

Recall that the divide-and-conquer solution involved returning three values from each recursive call on a sequence S: the minimum value of S, the maximum value of S, and the desired result $\texttt{stock}(S)$. We will denote these as $\bot$, $\top$, and δ, respectively. To solve the stock markert problem we can then use the following implementations for *combine*, *base*, and *emptyVal*:

```
fun  combine((⊥_L, ⊤_L, δ_L), (⊥_R, ⊤_R, δ_R)) =
   (min(⊥_L, ⊥_R),   max(⊤_L, ⊤_R),   max(max(δ_L, δ_R), ⊤_R − ⊥_L))
fun  base(v) = (v, v, 0)
val  emptyVal = (0, 0, 0)
```

and then solve the problem with:

```
reduce combine emptyVal (map base S)
```

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 from the input. It turns out to be possible to merge two sequences S_1 and S_2 in $O(|S_1| + |S_2|)$ work and $O(\log(|S_1| + |S_2|)$ span. We can use our reduction technique to implement merge sort with a `reduce`. On particular, we use

```
val  combine = merge
val  base = singleton
val  emptyVal = empty
```

Stylistic Notes. We have just seen that we could spell out the divide and conquer steps in details or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide and conquer code or using the reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, but the divide-and-conquer code exposes more clearly the inductive structure of the code.

You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partition their input in two parts in the middle. For example, in the Euclidean Traveling Salesperson algorithm we briefly discussed, each split step splits the space along the larger dimension. This requires checking which is the larger dimension, finding the median line along that dimension, and then filtering the points based on that line. A similar construct can be used for the closest-pair problem from Homework 2. Neither of these algorithms fits the pattern.

2 Reduce: Cost Specifications

Thus far, we made the assumption that the combine function has constant cost (i.e., both its work and span are constant), allowing us to state the cost specifications of `reduce` on a sequence of length n simply as $O(n)$ work and $O(\log n)$. While this bound applies readily to problems such as the one for stock market, we cannot apply it to the merge sort algorithm because the combine step, which we run merge, does more than constant work.

As a running example, we'll consider the following algorithm:

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

where we use $\texttt{merge}_<$ to denote a merge function that uses an (abstract) comparsion operator $<$ and we further assume that if s_1 and s_2 are sequences of lengths n_1 and n_2, then

$$\begin{aligned} W(\texttt{merge}_<(s_1, s_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(s_1, s_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

What do you think the cost of `reduce_sort` is? We want to analyze the cost of `reduce_sort`. But as we begin to analyze the cost, we quickly realize that the `reduce` function is underspecified: we don't know the reduction order and for that reason, we don't know what `merge` is called with. This begs the question: does the reduction order matter? Or is it in fact doesn't matter because any order will be equally good?

To answer this question, let's consider the following reduction order: the function `reduce` 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($\langle x_1 \rangle$, merge($\langle x_2 \rangle$, merge($\langle x_3 \rangle$, ...)))
```

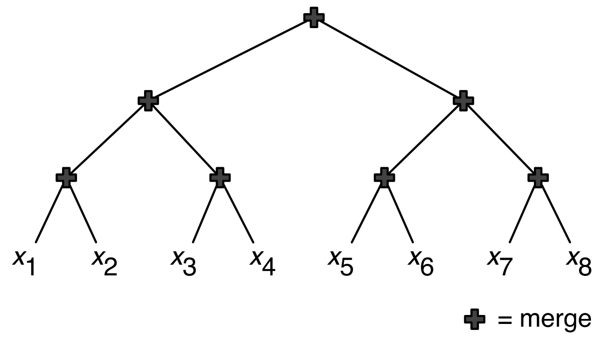
A closer look at this sequence shows that `merge` is always called with a singleton sequence as its left argument but the right argument is of varying sizes between 1 and $n - 1$. The final merge combines 1-element with $(n - 1)$-element sequences, the second to last merge combines 1-element with $(n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

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

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 essentially the order that the `iter` function uses. Furthermore, using this reduction order, the algorithm is effectively working backwards from the rear, "inserting" each element into a sorted suffix where it is kept at the right location to maintain the sorted order. This corresponds roughly to the well-known insertion sort.

Notice that in the reduction order above, the reduction tree was extremely unbalanced. Would the cost 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.



How much does this 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

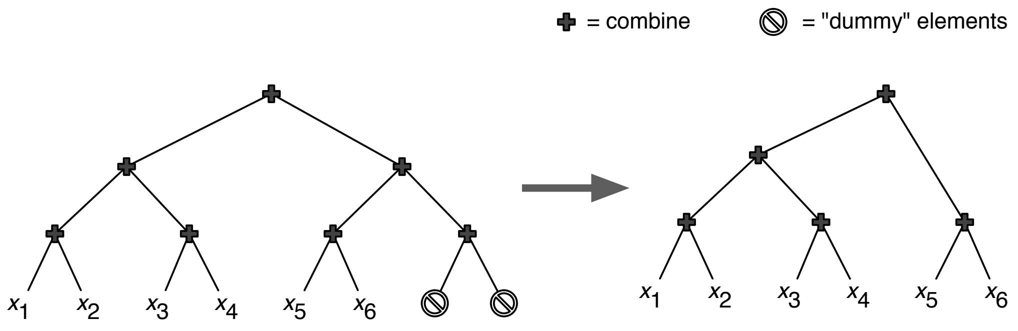
$$\begin{aligned} W(\texttt{reduction\_sort}\ x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

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

From these two examples, it is clearly important to specify the order of a reduction. These two examples illustrate how the reduction order can lead to drastically different cost. Moreover, there is a second reason to specify the reduction order: to properly deal with combining functions that are non-associative. In this case, the order we perform the reduction determines what result we get; because the function is non-associative, different orderings will lead to different answers. While we might try to apply reduce to only associative operations, unfortunately even some functions that seem to be associative are actually not. For instance, floating point addition and multiplication are not associative. In SML/NJ, integer addition is not associative either because of the overflow exception.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. But basically, this tree is the same as if we rounded up the length of the input sequence to the next power of two, and then put a perfectly balanced binary tree over the sequence. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.

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



How would we go about defining the cost of `reduce`? Given a reduction tree, we'll define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ or simply $\mathcal{R}(f, S)$ for brevity,

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

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

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) \;=\; O\left(n \;+ \sum_{f(a,b)\in\mathcal{R}(f,S)} W(f(a,b))\right)$$

$$S(\texttt{reduce}\ f\ \mathbb{I}\ S) \;=\; O\left(\log n \max_{f(a,b)\in\mathcal{R}(f,S)} S(f(a,b))\right)$$

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 $\max_{f(a,b)} S(f(a,b)$ thus, 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 $f\colon \alpha\times\alpha\to\alpha$ and a monotone size measure $s\colon \alpha\to\mathbb{R}_+$, if for any x, y,*

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

then

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

Applying this lemma to the merge sort example, we have

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

3 Contraction and Implementing Scan

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. In this lecture, we'll see a technique that allow us to implement `scan` in parallel.

Let's talk about another algorithmic technique: contraction. This is another common inductive technique in algorithms design. 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. In particular, the contraction technique involves the following steps:

1. Reduce 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 algorithms design, whether it applies to cars or not. For various reasons, it is more common in parallel algorithm than in sequential algorithms, usually because the contraction and expansion 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 by applying it to the scan problem. 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?* Let's look at an example for motivation.

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

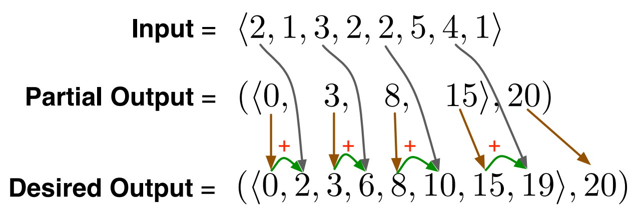
$$(\langle 0, 2, 3, 6, 8, 10, 15, 19\rangle, 20)$$

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.

Thought Experiment II: The crux of this problem is the realization that we can easily generate a sequence consisting of every other element of the final output, together with the final sum—and this is enough information to produce the desired final output with ease. Let's say we are able to somehow generate the sequence

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

Then, the diagram below shows how to produce the final output sequence:


Input = $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$
Partial Output = $(\langle 0, 3, 8, 15 \rangle, 20)$
+
+
+
+
Desired Output = $(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20)$

But how do we generate the "partial" output—the sequence with every other element of the desired output? The idea is simple: we pairwise add adjacent elements of the input sequence and recursively run `scan` on it. That is, on input sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$, we would be running `scan` on $\langle 3, 5, 7, 5 \rangle$, which will generate the desired partial output.

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

```
% implements: the Scan problem on sequences that have a power of 2 lenngth
fun scanPow2 $f$ $i$ $s$ =
  case $|s|$ of
      $0 \Rightarrow (\langle\rangle, i)$
    | $1 \Rightarrow (\langle i \rangle, s[0])$
    | $n \Rightarrow$
        let
          val $s' = \langle f(s[2i], s[2i+1]) : 0 \le i < n/2 \rangle$
          val $(r, t) = \mathtt{scanPow2}\ f\ i\ s'$
        in
          $(\langle p_i : 0 \le i < n \rangle, t)$,  where  $p_i = \begin{cases} r[i/2] & \textbf{if } even(i) \\ f(r[i/2], s[i-1]) & \textbf{otherwise}. \end{cases}$
        end
```

4 How to Parse Text in Parallel 101

One of the most common tasks in your career as a computer scientist is that of parsing text files. For example, you might have some large text file and you want to break it into words or sentences, or you might need to parse a program file, or perhaps you have a web log that lists all accesses to your web pages, one per line, and you want to process it in some way.

The basic idea of parsing is to take a string and break it up into parts; exactly how the string is split up depends on the "grammar" of what're parsing. For this reason, there are many variants of parsing. You probably have come across regular expressions, a class which is reasonably general and powerful enough for most simple text-processing needs. You may have wondered how Standard ML or your favorite compilers are capable of recognizing thousands of lines of code so quickly.

In this lecture, we are only going to talk about two particularly simple, but quite useful, forms of parsing. The first which we call `tokens` is used to break a string into a sequence of tokens separated by one or more whitespaces (or more generally a set of delimiters). A *token* is a maximal nonempty substring of the input string consisting of no white space (delimiter). By maximal, we mean that it cannot be extended on either side without including a white space. There are multiple characters we might regard as a white space, including the space character, a tab, or a carriage return. In general, the user should be able to specify a function that can be applied to each character to determine whether or not it is a space. Specifially, the `tokens` function, as provided by the sequence library, has type

```
val tokens :  (char -> bool) -> string -> string seq
```

where the first argument to `tokens` is a function that checks whether a given character is a white space (delimiter). As an example, we'll parse the string `"this␣is␣␣␣a␣␣short␣string"` (the symbol ␣ denotes a white space),

```
tokens (fn x => (x = #"␣")) "this␣is␣␣␣a␣␣short␣string"
```

which would return the sequence

```
⟨"this", "is", "a", "short", "string"⟩.
```

More formally, we can define the string tokenizer problem as follows:

Definition 4.1 (The String to Token Problem)**.** Given a string (sequence) of characters $S = \Sigma^*$ from some alphabet σ and a function $f : \Sigma \to \{\mathsf{True}, \mathsf{False}\}$, the *string to token* problem is to return a sequence of tokens from S in the order as they appear in S, where a token is a nonempty maximal substring of S such that f evaluates to False for all its characters.

This function appears in the sequence library and is often used to break up text into words. The second useful parsing routine is `fields` and is used to break up text into a sequence of fields based on a delimiter. A field is a maximal possibly empty string consisting of no delimiters. Like in `tokens`, the input string might consist of multiple consecutive delimiters. The key difference here is that a field can be empty so if there are multiple delimiters in a row, each creates a field. For example, we'll parse a line from a comma-separated values (CSV) file.

```
val fields :  (char -> bool) -> string -> string seq
tokens (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
```

which would return

```
⟨"a", "", "", "line", "of", "a", "csv", "", "file"⟩.
```

The `fields` function is useful for separating a file that has fields delimited by certain characters. As mentioned before, a common example is the so-called CSV files that are often used to transfer data between spreadsheet programs.

Traditionally, `tokens` and `fields` have been implemented in a sequential fashion, starting at the front end and processing it one character at a time until reaching the end. During the processing, whenever a whitespace or delimiter is found, the algorithm declares the current token or field finished and starts a new one. Part of the reason for this sequential nature probably has to do with how strings are loaded from external storage (e.g. tape/disk) many years ago. Here, we'll assume that we have random access to the input string's characters, a common assumption now that a program tends to load a big chuck of data into main memory at a time. This makes it possible to implement `fields` and `tokens` in parallel.

In the rest of this lecture, we will discuss a parallel implementation of `fields`. You will think about implementing `tokens` in parallel in your homework.

How do we go about implementing `fields` *in parallel?* Notice that we can figure out where each field starts by looking at the locations of the delimiters. Further, we know where each field ends—this is necessarily right before the delimiter that starts the next field. Therefore, if there is a delimiter at location i and the next delimiter is at $j \geq i$, we have that the field starting after i contains the substring extracted from locations $(i+1)..(j-1)$, which may be empty. This leads to the following code, in which `delims` contains the starting location of each field. We use the notation $\oplus$ to denote sequence concatenation.

```
fun fields f s = let
  val delims = ⟨0⟩ ⊕ ⟨i+1 : i ∈ [0,|s|) ∧ f(s[i])⟩ ⊕ ⟨|s|+1⟩
in
  ⟨s[delims[i], delims[i+1]-1) :  i ∈ [0, |delims|]⟩
end
```

To illustrate the algorithm, let's run it on our familiar example.

```
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"

delims = ⟨0,2,3,4,9,12,14,18,19,24⟩

result = ⟨ s[0,1), s[2,2), s[3,3), s[4,8), s[9,11), s[12,13), ... ⟩

result = ⟨"a", "", "", "line", "of","a", ... ⟩.
```

5 SML Code

5.1 Scan

```
functor Scan(Seq : SEQUENCE) =
struct
  open Seq

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