Lecture 5 — Data Abstraction and Sequences I

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

Lectured by Danny Sleator — 10 September 2013

Material in this lecture:

- Relationship between ADTs, cost specifications and data structures.
- The sequence ADT
- The scan operation: examples and implementation
- Using contraction : an algorithmic technique

1 Abstract Data Types, Cost Specifications, and Data Structures

So far in class we have defined several "problems" and discussed algorithms for solving them. The idea is that the problem is an abstract definition of what we want in terms of a function specification, and the algorithms are particular ways to solve/implement the problem. In addition to abstract functions we also often need to define abstractions over data. In such an abstraction we define a set of functions (abstractly) over a common data type. As mentioned in the first lecture, we will refer to the abstractions as abstract data types and their implementations as data structures.

An example of an abstract data type, or ADT, you should have seen before is a priority queue. Let's consider a slight extension where in addition to insert, and deleteMin, we will include a function that joins two priority queues into a single one. For historical reasons, we will call such a join a *meld*, and the ADT a "meldable priority queue". As with a problem, we like to have a formal definition of the abstract data type.

Definition 1.1. Given a totally ordered set $\mathbb{S}$, a *Meldable Priority Queue* (MPQ) is a type $\mathbb{T}$ representing subsets of $\mathbb{S}$, along with the following values and functions (partial list):

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \{\} \\
\texttt{insert}(Q,e) & : & \mathbb{T}\times\mathbb{S}\to\mathbb{T} & = Q\cup\{e\} \\
\texttt{deleteMin}(Q) & : & \mathbb{T}\to\mathbb{T}\times(\mathbb{S}\cup\{\bot\}) & = \begin{cases} (Q,\bot) & Q=\{\} \\ (Q\setminus\{\min Q\},\min Q) & otherwise \end{cases} \\
\texttt{meld}(Q_1,Q_2) & : & \mathbb{T}\times\mathbb{T}\to\mathbb{T} & = Q_1\cup Q_2
\end{array}
$$

Here we are using standard set notation: empty braces {} denotes an empty set, $A\cup B$ denotes taking the union of the sets A and B, and $A\setminus B$ denotes taking the set difference of A and B (i.e., all the elements in A except those that appear in B). Note that `deleteMin` returns the special element $\bot$ when the queue is empty; it indicates that the function is undefined when the priority queue is empty.

When translated to SML this definition corresponds to a signature of the form:

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

```
signature MPQ =
 sig
   structure S : ORD
   type t
   val empty : t
   val insert : t * S.t -> t
   val deleteMin : t -> t * (S.t option)
   val meld : t * t -> t
 end
```

Note that the type `t` is abstracted (i.e. it is not specified to be sets of elements of type `S.t`) since we don't want to access the queue as a set but only through its interface. Note also that the signature by itself does not specify the semantics but only the types (e.g., it could be `insert` does nothing with its second argument). To be an ADT we have to add the semantics as written on the righthand side of the equations in Definition 1.1.

In general SML signatures for ADTs will look like:

```
signature ADT =
 sig
   structure S1 : OtherADT
   ...
   type t
   helper types
   val v1 : ... t ...
   val v2 : ... t ...
   ...
 end
```

1.1 Cost Specifications

Now the operations on a meldable priority queue might have different costs depending on the particular data structures used to implement them. If we are a "client" using a priority queue as part of some algorithm or application we surely care about the costs, but probably don't care about the specific implementation. We therefore would like to have abstract cost associated with the interface. For example we might have for work:

	I1	**I2**	**I3**
`insert`(Q, e)	$O(\|Q\|)$	$O(\log \|Q\|)$	$O(\log \|Q\|)$
`deleteMin`(Q)	$O(1)$	$O(\log \|Q\|)$	$O(\log \|Q\|)$
`meld`(Q_1, Q_2)	$O(\|Q_1\| + \|Q_2\|)$	$O(\|Q_1\| + \|Q_2\|)$	$O(\log(\|Q_1\| + \|Q_2\|))$

You have already seen data structures that match the first two bounds. For the first one maintaining a sorted array will do the job. You have seen a couple that match the second bounds. What are they? We will be covering the third bound, which has a faster `meld` operation than the others, later in the course.

In any case, these cost definitions sit between the ADT and the specific data structures used to implement them. We will refer to them as *cost specifications*. We therefore have three levels:

1. **The abstract data type**: The definition of the interface for the purpose of describing functionality and correctness criteria.
2. **The cost specification**: The definition of the costs for each of the functions in the interface. There can be multiple different cost specifications for an ADT depending on the type of implementation. Although not required to show correctness this specification is required to analyze performance.
3. **The implementation as a data structure**: This is the particular data structure used to implement the ADT. There might be multiple data structures that match the same cost specification. If you are a user of the ADT you don't need to know what this is, although it is good to be curious.

2 The Sequence ADT

The first ADT we will go through in some detail is the sequence ADT. You have used sequences in 15-150. But we will add some new functionality and will go through the cost specifications in more detail. There are two cost specifications for sequences we will consider, one based on an array implementation, and the other based on a tree implementation.

We first do a quick review of set theory so we can more formally define a sequence. A *relation* is a set of ordered pairs. In particular for two sets α and β, ρ is a relation from α to β if $\rho \subseteq \alpha \times \beta$. Here $\alpha \times \beta$ indicates the set of all ordered pairs made from taking the first element from α and the second from β. A *function* is a relation ρ such that for every a in the domain of ρ there is only one b such that $(a, b) \in \rho$. A *sequence* is a function whose domain is $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$ (we use $\mathbb{N}$ to indicate the natural numbers, including zero). We define the sequence ADT based on the mathematical definition along with specific operations it supports. We only list a subset. A more complete list can be found in the library documentation.

Definition 2.1. A *Sequence* is a type $\mathbb{S}_\alpha$ representing functions from $\mathbb{N}$ to α with domain $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$, and supporting the following values and functions:

`empty`	:	$\mathbb{S}_\alpha$	=	$\{\}$
`length`(A)	:	$\mathbb{S}_\alpha \to \mathbb{N}$	=	$\lvert A \rvert$
`singleton`(v)	:	$\alpha \to \mathbb{S}_\alpha$	=	$\{(0, v)\}$
`nth`(A, i)	:	$\mathbb{S}_\alpha \to \alpha$	=	$A(i)$
`map`(f, A)	:	$(\alpha \to \beta) \times \mathbb{S}_\alpha \to \mathbb{S}_\beta$	=	$\{(i, f(v)) : (i, v) \in A\}$
`tabulate`(f, n)	:	$(\mathbb{N} \to \alpha) \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i, f(i)) : i \in \{0, \ldots, n-1\}\}$
`take`(A, n)	:	$\mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i, v) \in A \mid i < n\}$
`drop`(A, n)	:	$\mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i - n, v) : (i, v) \in A \mid i \geq n\}$
`append`(A, B)	:	$\mathbb{S}_\alpha \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha$	=	$A \cup \{(i + \lvert A \rvert, v) : (i, v) \in B\}$
...				

	ArraySequence		**TreeSequence**	
	Work	*Span*	*Work*	*Span*
`length`(T)	$O(1)$	$O(1)$	$---$	$---$
`nth`(T)	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
`tabulate` f n	$O\left(\sum_{i=0}^{n} W(f(i))\right)$	$O\left(\max_{i=0}^{n} S(f(i))\right)$	$---$	$O\left(\log n + \max_{i=0}^{n} S(f(i))\right)$
`map` f S	$O\left(\sum_{s\in S} W(f(s))\right)$	$O\left(\max_{s\in S} S(f(s))\right)$	$---$	$O\left(\log \lvert S\rvert + \max_{s\in S} S(f(s))\right)$
`append`(S_1, S_2)	$O(\lvert S_1\rvert + \lvert S_2\rvert)$	$O(1)$	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$

Figure 1: Sample *cost specifications* for the Array and Tree based implementations of Sequences. The $---$ means it is the same as in ArraySequence. A full list of costs can be found in the Cost Specifications part of `www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/docs/`.

The *arraySequence* and *treeSequence* cost specificatios for the Sequence ADT are given figure 1 for a few of the functions.

In the pseudocode used in the course notes we will use a variant on set notation to indicate various operations over sequences. In particular we use triangular brackets $\langle\ \rangle$ instead of curly brackets $\{\}$. Here are some examples of the notation we will use:

S_i	The i^{th} element of sequence S
$\lvert S\rvert$	The length of sequence S
$\langle\ \rangle$	The empty sequence
$\langle v \rangle$	A sequence with a single element v
$\langle i, \ldots, j \rangle$	A sequence of integers starting at i and ending at $j \geq i$.
$\langle e : p \in S \rangle$	**Map** the expression e to each element p of sequence S. The same as "`map` (**fn** $p \Rightarrow e$) S" in ML.
$\langle p \in S \mid e \rangle$	**Filter** out the elements p in S that satisfy the predicate e. The same as "`filter` (**fn** $p \Rightarrow e$) S" in ML.

More examples are given in the "Syntax and Costs" document.

We will now cover some of the sequence functions in more detail, including

- `reduce`, `iter`, `scan` and `iterh`
- `tokens`, `fields`
- `collect`

You have most likely seen `reduce` before, but we will cover more applications of it, and how to analyze its cost when the combining function requires more than constant work and span.

3 The Scan Operation

A function closely related to `reduce` is `scan`. We mentioned it during the last lecture and you covered it in recitation. It has the interface:

$$\texttt{scan}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} \times \alpha)$$

As with `reduce`, when the function f is associative, the scan function returns the sum with respect to f of each prefix of the input sequence S, as well as the total sum of S. Hence the operation is often called the *prefix sums* operation. For a function f which is associative it can be defined as follows:

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

This uses our pseudocode notation and the $\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S,i)) : i \in \langle 0,\ldots,n-1\rangle \,\rangle$ 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.*

We will soon see how to implement a scan with the following bounds:

$$\begin{aligned}W(\texttt{scan}\ f\ I\ S) &= O(|S|) \\ S(\texttt{scan}\ f\ I\ S) &= O(\log|S|)\end{aligned}$$

assuming that the function f takes constant work. For now we will consider some useful applications of scans.

Note that the scan operation takes the "sum" of the elements before the position i. Sometimes it is useful to include the value at position i. We therefore also will use a version of such an *inclusive scan*.

$$\texttt{scanI}\ +\ 0\ \langle 2,1,3\rangle = \langle 2,3,6\rangle$$

This version does not return a second result since the total sum is already included in the last position.

3.1 The MCSS Problem Algorithm 5: Using Scan

Let's consider how we might use scan operations to solve the Maximum contiguous subsequence (MCSS) problem. Recall, this problem is given a sequence S to find:

$$\max_{0 \le i \le j \le n} (\sum_{k=i}^{j-1} S_k) .$$

As a running example for this section consider the sequence

$$S = \langle 1, -2, 3, -1, 2, -3 \rangle .$$

What if we do an inclusive scan on our input S using addition? i.e.:

$$X = \texttt{scanI + 0}\ S = \langle 1, -1, 2, 1, 3, 0 \rangle$$

Now for any j^{th} position consider all positions $i < j$. To calculate the sum from immediately after i to j all we have to do is return $X_j - X_i$. This difference represents the total sum of the subsequence from $i+1$ to j since we are taking the sum up to j and then subtracting off the sum up to i. For example to calculate the sum between the -2 (location $i+1=1$) and the 2 (location $i=4$) we take $X_4 - X_0 = 3 - 1 = 2$, which is indeed the sum of the subsequence $\langle -2, 3, -1, 2 \rangle$.

Now consider how for each j we might calculate the maximum sum that starts at any $i \le j$ and ends at j. Call it R_j. This can be calculated as follows:

$$\begin{aligned} R_j &= \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\ &= \max_{i=0}^{j} (X_j - X_{i-1}) \\ &= X_j + \max_{i=0}^{j} (-X_{i-1}) \\ &= X_j + \max_{i=0}^{j-1} (-X_i) \\ &= X_j - \min_{i=0}^{j-1} X_i \end{aligned}$$

The last equality is because the maximum of a negative is the minimum of the positive. This indicates that all we need to know is X_j and the minimum previous $X_i, i < j$. This can be calculated with a scan using the minimum operation. Furthermore the result of this scan is the same for everyone, so we need to calculate it just once. The result of the scan is:

$$(M, \_) = \texttt{scan min 0}\ X = (\langle 0, 0, -1, -1, -1, -1 \rangle, -1) ,$$

and now we can calculate R:

$$R = \left\langle X_j - M_j : 0 \le j < |S| \right\rangle = \langle 1, -1, 3, 2, 4, 1 \rangle .$$

You can verify that each of these represents the maximum contiguous subsequence sum ending at position j.

Finally, we want the maximum string ending at any position, which we can do with a reduce using max. This gives 4 in our example.

Putting this all together we get the following very simple algorithm:

```
1  function MCSS(S) =
2  let
3      X = scanI + 0 S
4      (M, _) = scan min 0 X
5  in
6     max ⟨ X_j − M_j : 0 ≤ j < |S| ⟩
7  end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $O(n)$ work and $O(\log n)$ span.

3.2 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 α `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  function 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 automaton.

3.3 Contraction and Implementing Scan

Now let's consider how to implement `scan` efficiently and at the same time apply one of the algorithmic techniques from our toolbox of techniques: *contraction*. Throughout the following discussion we assume the work of the binary operator is $O(1)$. As described earlier a brute force method for calculating scans is to apply a reduce to all prefixes. This requires $O(n^2)$ work and is therefore not efficient (we can do it in $O(n)$ work sequentially).

Beyond the wonders of what it can do, a surprising fact about `scan` is that it can be accomplished efficiently in parallel, although on the surface, the computation it carries out appears to be sequential in nature. At first glance, we might be inclined to believe that any efficient 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 are 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.

Exercise 2. *Can you see a way to implement a scan in $O(n \log n)$ work and $O(\log n)$ span using divide and conquer? Hint: what can you do with the total sum that is returned on the left?*

Contraction: To compute `scan` in $O(n)$ work in parallel, we 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, the subproblems do not have to be independent—they can be evaluated sequentially. In particular, the contraction technique involves the following steps:

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

The contraction approach is a useful technique in algorithm design in general but for various reasons it is more common in parallel algorithms than in sequential algorithms. This is 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?*

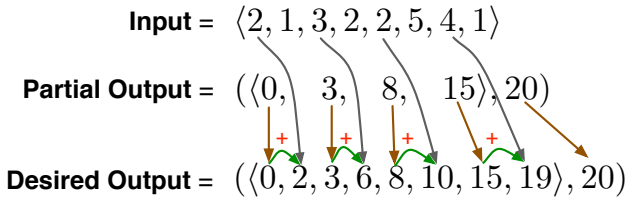
The idea is simple: We apply the combining function pairwise to adjacent elements of the input sequence and recursively run `reduce` on it. In this case, the third step is a "no-op"; it does nothing. For example on input sequence $\langle 2,1,3,2,2,5,4,1\rangle$ with addition, we would contract the sequence 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  function scanPow2 f i s =
3    case |s| of
4      0 ⇒ (⟨⟩, i)
5    | 1 ⇒ (⟨i⟩, s[0])
6    | n ⇒
7      let
8        s' = ⟨f(s[2i], s[2i+1]) : 0 ≤ i < n/2⟩
9        (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 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
```