Lecture 4 — ADTs, Reduce, and Scan

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

Lectured by Margaret Reid-Miller — 26 January 2012

1 Abstract Data Types 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 you should have seen before (in 15-122) is a priority queue. Let's consider a slight extension where in addition to insert, and deleteMin, we will add a function that joins two heaps into a single heap. For historical reasons, we will call such a join a meld, and the ADT a "meldable priority queue".

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:

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \{\} \\
\texttt{insert}(S,e) & : & \mathbb{T}\times\mathbb{S}\to\mathbb{T} & = S\cup\{e\} \\
\texttt{deleteMin}(S) & : & \mathbb{T}\to\mathbb{T}\times(\mathbb{S}\cup\{\bot\}) & = \begin{cases} (S,\bot) & S=\{\} \\ (S\setminus\{\min S\},\min S) & otherwise \end{cases} \\
\texttt{meld}(S_1,S_2) & : & \mathbb{T}\times\mathbb{T}\to\mathbb{T} & = S_1\cup S_2
\end{array}
$$

We use the empty braces {} to denote an empty queue, set union $\cup$ to denote adding an element to a queue or joining two queues, and set difference $\setminus$ to remove an element from the queue. Note that `deleteMin` returns the special element $\bot$ when the queue is empty; it represents undefined.

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

```
signature MPQ
  sig
   struct 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
```

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

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:

```
sig
 struct S1 : ADT1
 ...
 type t
 helper types
 val v1 : ... t ...
 val v2 : ... t ...
 ...
end
```

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`(S,e)	$O(\|S\|)$	$O(\log \|S\|)$	$O(\log \|S\|)$
`deleteMin`(S)	$O(1)$	$O(\log \|S\|)$	$O(\log \|S\|)$
`meld`$(S1,S2)$	$O(\|S_1\| + \|S_2\|)$	$O(\|S_1\| + \|S_2\|)$	$O(\log(\|S_1\| + \|S_2\|))$

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: the abstract data type (specifying the interface), the cost specification (specifying costs), and the data structure (specifying the implementation).

2 Sequences

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. However in this course we will mostly be using the array implementation. In the lecture we will not go through the full interface, it is available in the documentation, but here are some of functions you should be familiar with:

```
nth, tabulate, map, reduce, filter, take, drop, showt,
```

and here are some we will discuss in the next couple lectures.

```
scan, inject, collect, tokens, fields
```

Two important sequence operations are `reduce` and `scan`. You have seen reduce before, but you may not be aware of some of its useful applications. Scan is a related operation that is surprisingly powerful. We will consider some examples of their use, clarify their semantics, and then how they might be implemented.

2.1 Reduce Operation

Recall that `reduce` function has the interface

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

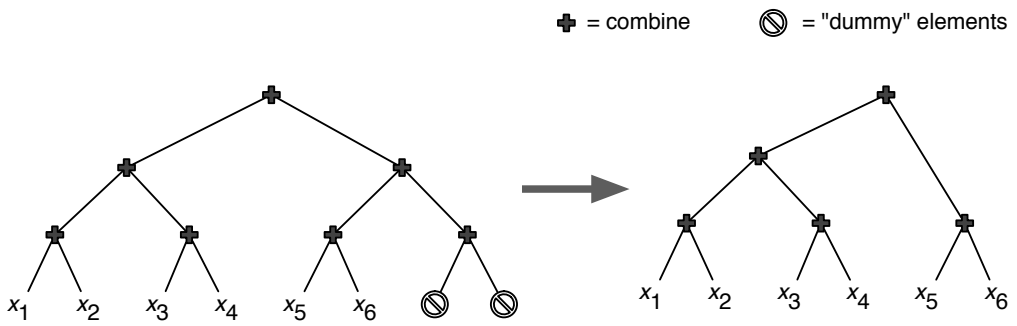
When the combining function f is associative, i.e., $f(f(x,y),z) = f(x,f(y,z))$, `reduce` returns the sum with respect to f of of the input sequence S. It is the same result returned by `iter`. The reason we include `reduce` is that it is parallel, whereas `iter` is strictly sequential. Note, though, `iter` can use a more general combining function.

The results of `reduce` and `iter`, however, may differ if the combining function is 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.

To properly deal with combining functions that are non-associative, it is clearly important to specify the order that the combining function is applied to the sequence. This order is part of the specification of the ADT `Sequence`. In this way, every (correct) implementation returns the same result when applying `reduce`; the results are deterministic regardless of which data structure you use.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. This tree is the same as if we rounded up the length of the input sequence to the next power of 2, i.e., $|x| = 2^k$, and then put a perfectly balanced binary tree[1] over the sequence with 2^k leaves. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.

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



In the next lecture we will offer an explanation why we chose this particular combining order.

2.2 Divide and Conquer with Reduce

Now, 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.

```
1  fun myDandC(S) =
2    case showt(S) of
3      EMPTY ⇒ [emptyVal]
4    | ELT(v) ⇒ [base](v)
5    | NODE(L, R) ⇒ let
6        val L' = myDandC(L)
7        val R' = myDandC(R)
8      in
9        [someMessyCombine](L',R')
10     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:

```
reduce [someMessyCombine] [emptyVal] (map [base] S)
```

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. The first example we will consider today; the second we will considered in the next lecture.

Algorithm 4: MCSS Using Reduce.

The first example is the Maximum Contiguous Subsequence Sum problem from last lecture. Given a sequence S of numbers, find the contiguous subsequence that has the largest sum—more formally:

$$\text{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \ : \ 1 \le i \le n, i \le j \le n\right\}.$$

Recall that the divide-and-conquer solution involved returning four values from each recursive call on a sequence S: the desired result $\texttt{mcss}(S)$, the maximum prefix sum of S, the maximum suffix sum of S, and the total sum of S. We will denote these as M, P, S, T, respectively. We use v_+ to denote $\max\{v, 0\}$. To solve the mcss problem we can then use the following implementations for `combine`, `base`, and `emptyVal`:

fun `combine`$((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_L)) =$
$(\max(S_L + P_R, M_L, M_R), \quad \max(P_L, T_L + P_R), \quad \max(S_R, S_L + T_R), \quad T_L + T_R)$

fun `base`$(v) = (v_+, v_+, v_+, v)$

val `emptyVal` $= (0, 0, 0, 0)$

and then solve the problem with:

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

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail 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 reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. 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 partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also cannot be used for the closest-pair problem from Homework 2. Neither of these algorithms fits the pattern.

2.3 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:

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

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 associated f, it can be defined 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 $\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S,i)) : i \in \langle 0,\ldots,n-1\rangle \,\rangle$ indicates that for each i in the range from 0 to $n-1$ apply reduce to the first i elements of S. For example,

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

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.*

In the next lecture we will discuss 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 why the operation is useful by giving an examples. You should have already seen how to use it for parenthesis matching and the stock market problem in recitation.

Algorithm 5: The MCSS problem using scan

Let's consider how we might use scan operations to solve the Maximum contiguous subsequence (MCSS) problem. That is, find $\max_{0\le i\le j\le n}(\sum_{k=i}^{j-1} S_k)$. Any ideas?

$$\begin{aligned}
\texttt{mcss}(S) &= \max_{0\le i\le j\le n}(\sum_{k=i}^{j-1} S_k) \\
&= \max_{0\le i\le j\le n}(\sum_{k=0}^{j-1} S_k - \sum_{l=0}^{i-1} S_l) \\
&= \max_{0\le i\le j\le n}(X_j - X_i) \\
&= \max_{0\le j\le n}(X_j - \min_{0\le i\le j} X_i)
\end{aligned}$$

What if we do a scan on our input S using addition starting with 0? Suppose it returns X. Now for a position j let's consider all positions $i < j$. To calculate the sum from i (inclusive) to j (exclusive) all we have to consider is $X_j - X_i$. This difference represents the total sum of the subsequence from i to j. For each j, how do we calculate the maximum sum that ends at j (exclusive), call it R_j.

Well this is

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

The last equality is because the maximum of a negative is the minimum of the negative. What is the term? It is just the minimum value of X up to j (exclusive). Now we want to calculate it for all j, so we can use a scan. Finally, we want the $\max_{j=0}^{|S|} R_j$.

Putting it altogether, we get

$$\begin{aligned}
\texttt{MCSS}(S) &= \max_{0 \le i \le j \le |S|} \left(\sum_{k=i}^{j-1} S_k\right) \\
&= \max_{0 \le i \le j \le |S|} \left(\sum_{k=0}^{j-1} S_k - \sum_{l=0}^{i-1} S_l\right) \\
&= \max_{0 \le i \le j \le |S|} (X_j - X_i) \\
&= \max_{0 \le j \le |S|} (X_j - \min_{0 \le i \le j} X_i)
\end{aligned}$$

This final formula gives the following algorithm:

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