Lecture 4 — Data Abstraction and Sequences

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

Lectured by Guy Blelloch — September 8, 2011

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. Lets 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:

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

Note that `deleteMin` returns the special element $\bot$ when empty. When translated to SML this 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
```

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	or I2	or 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 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
```

2.1 Scan Operation

We mentioned the scan function during the last lecture and you covered it in recitation. It has the interface:

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

When the function f is associative, i.e., $f(f(x,y),z) = f(x,f(y,z))$, 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,

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

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. You can assume* 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 a set of examples. You should have already seen how to use it for parenthesis matching and the stock market problem in recitation.

The MCSS problem : Algorithm 4 (using scan)

Lets consider how we might use scan operations to solve the MCSS problem. Any ideas? What if we do a scan on our input S using addition starting with 0? Lets say it returns X. Now for a position j lets 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 represents the total sum between the two. So how do we calculate the maximum sum R_j ending at j (exclusive).

Well this is

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

What is the last 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. This 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
```

the work of each of the steps (two scans, map and reduce) is $O(n)$ and the span is $O(logn)$. We therefore have a routine that is even better than any of our divide and conquer routines.

Copy Scan

We will go through one more example that will be helpful in your homework. Lets say you are given a sequence of type α `option seq`. For example

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

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

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

Anyone see how they could do this with a 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.

2.2 Analyzing the Costs of Higher Order Functions

We already covered map where we have:

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

Tabulate is similar. But what about functions such as `reduce` and `scan` when the combining function does not take constant time. For example for an integer sequence consider the following reduction:

$$\texttt{reduce}\ \texttt{merge}_{int}\ \langle\,\rangle\ \langle \langle a \rangle : a \in A \rangle$$

What does this return?