15-150 Fall 2013
Lab 11

6 November 2013

1 Introduction

In this lab, we will talk about what sequences are and you'll get some practice using them. Please take advantage of this opportunity to practice writing functions with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You should practice writing requires and ensures specifications, and tests on the functions you write on this assignment. In particular, every function you write should have both specs and tests.

1.3 Compiling This Lab

As is common with modular code, this lab is distributed across many files and relies on the SML/NJ compilation manager to introduce structures into the environment at the right time. The files that contain relevant code are listed in the file `sources.cm`, and the compilation manager takes it from there. When you want to run your code for this lab, at the REPL, you will enter

```
CM.make "sources.cm";
```

As you progress through the lab, you'll have to edit the `sources.cm` file to uncomment the files you've filled in. Make sure you're comfortable with this process! The current homework is organized in the same way, so ask your TA or a neighbour if you can't get this to work.

2 Sequences Cheat-Sheet

For your convenience a brief description of some of the functions on sequences is given here.

- `Seq.map : ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`, which takes a function and a sequence and returns a sequence whose elements are the result of applying the given function to the corresponding element in the given sequence.
- `Seq.reduce : ('a * 'a -> 'a) -> 'a -> 'a Seq.seq -> 'a`, which combines all the elements of a sequence using a particular function and base case.
- `Seq.mapreduce : ('a->'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`, which combines the operations of `Seq.map` and `Seq.reduce` by applying the given function of type `'a -> 'b` to each element of the sequence before combining them as in `Seq.reduce`.
- `Seq.length : 'a Seq.seq -> int`, which returns the number of elements in the sequence.
- `Seq.nth : int -> 'a Seq.seq -> 'a`, which returns the element of the given sequence at the indicated index, assuming it is in bounds.
- `Seq.tabulate : (int -> 'a) -> int -> 'a Seq.seq`, which computes a sequence of the given length such that the value of each element of the sequence is the result of applying the function to its index.
- `Seq.singleton : 'a -> 'a Seq.seq`, which takes a value and produces a sequence containing only that value.
- `Seq.append : 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`, which takes two sequences and appends the second to the end of the first

3 Mapreduce All Day

`mapreduce` is intended to be used with a function `f` of type `t1 -> t2` for some types `t1` and `t2`, an *associative* binary function `g` of type `t2 * t2 -> t2`, a value `z:t2`, and a sequence of items of type `t1`.

The behavior of `mapreduce f z g <x1, . . ., xn>` (assuming that `g` is associative) is given by:

$$\texttt{mapreduce f z g}\,\langle x_1, \ldots x_n\rangle \;=\; (\texttt{f}\ x_1)\ \texttt{g}\ (\texttt{f}\ x_2)\ \texttt{g}\ \ldots\ \texttt{g}\ (\texttt{f}\ x_n)\ \texttt{g z}$$

The implementation of `mapreduce` uses a balanced parenthesization format for pairwise combinations, as with `reduce`. Its work and span are as for `reduce`, when `f` and `g` are constant time.

Although asymptotically mapping and then reducing has the same complexity as `mapreduce`, it, `mapreduce`, is actually the more efficient of the two because it doesn't create an intermediate sequence.

Let's learn how to use `mapreduce`. First things first: it is so easy to construct lists.

Task 3.1 Write the function

```
seqFromList : 'a list -> 'a Seq.seq
```

and

```
seqToList : 'a Seq.seq -> 'a list
```

Hint: You should use `mapreduce` for `seqToList`.

Recall the function `List.exists : ('a -> bool) -> 'a list -> bool`, which determines whether an element of the list satisfies the given predicate. You will write an analogous function for sequences:

Task 3.2 Write the function

```
seqExists : ('a -> bool) -> 'a Seq.seq -> bool
```

to determine if the sequence has an element that satisfies the given predicate. Hint: You should use `mapreduce`.

Task 3.3 Write the function

```
acronym : (char Seq.seq Seq.seq) -> string
```

that, given a sequence of nonempty character sequences, finds the string whose characters are the first character of each sequence in their order of appearance. Example:

```
acronym <<#"S", #"O", #"P", #"R", #"A", #"N", #"O">,
         <#"A", #"L", #"T", #"O">,
         <#"T", #"E", #"N", #"O", #"R">,
         <#"B", #"A", #"S", #"S">>
        = "SATB"
```

Hint: you should use `mapreduce` for `acronym`. `str : char -> string` and `^: string * string -> string` (careful, it's infix) will also be useful.

4 Sequence Puzzles

The following functions ask you to become familiar with `Seq.tabulate`, `Seq.length`, and `Seq.nth`. Add your functions to `lab11.sml`.

4.1 Transpose

Recall the function `transpose` from Homework 5:

```
transpose [[1,2,3],
           [4,5,6]]
  ==>
[[1,4],
 [2,5],
 [3,6]]
```

Task 4.1 Write

```
fun transpose (s : 'a Seq.seq Seq.seq) : 'a Seq.seq Seq.seq
```

that transposes a sequence of sequences. You may assume that s is rectangular, with dimensions $m \times n$, where $m, n > 0$. Your solution should have $O(m \times n)$ work and $O(1)$ span.

4.2 Filter

We've used the function `filter`, which takes a predicate and a list, and evaluates to a list with only the items that satisfy the given predicate, before. One of your tasks for this lab is to write an analogous function for sequences.

Task 4.2 Write

```
fun filter' (p: 'a -> bool) (s: 'a Seq.seq) : 'a Seq.seq
```

such that `filter' p s` evaluates to a sequence that includes only the elements `x` of `s` for which `p x` = `true`. **Your implementation must not use `Seq.filter`.**

4.3 Reduce

Contraction is an algorithmic technique in which we take a problem, reduce it to a smaller problem and then recurse on the smaller problem. It's similar to Divide and Conquer (the technique behind merge sort) but differs in a key way. Divide and Conquer is based around the idea that we take our problem, divide it up, perform a recursive call on each part, then combine the results together. In contraction, we take the input, *make it smaller*, and then

perform a single recursive call on the smaller input. Note that this difference can result in contraction algorithms having much better runtime then divide and conquer algorithms. Contraction is a very powerful technique that you will explore more in 15-210 (if you go on to take it). For our purposes we will use use contraction to implement the sequence function `reduce` with O(n) work and O(log n) span. The idea is to do pairwise reduction, and is illustrated by the following trace:

```
    reduce op+ 0 <1,2,3,4,4,3,2,1>
==> reduce op+ 0 <3,  7,  7,  3 >
==> reduce op+ 0 <10,     10    >
==> reduce op+ 0 <20>
==> 20
```

Task 4.3 Write

```
fun reduce (f : 'a * 'a -> 'a)(b : 'a)(s : 'a Seq.seq) : 'a Seq.seq
```

such that `reduce f b s` functions the same as `Seq.reduce` and runs in $O(n)$ work and $O(\log n)$ span. You may assume that `Seq.length s` is a power of 2. **Your implementation must not use `Seq.reduce` or `Seq.mapreduce`.**

Have a TA check your code before proceeding!

5 Finitely Branching Parallel Trees

A *Finitely Branching Parallel Tree* is similar to the binary trees introduced earlier in the class, with the exception that each node can now have an *arbitrary* number of children (rather than exactly 2). To represent this, we define each node to be a *sequence* of `fbtrees` (this allows us to evaluate children of nodes in parallel). The datatype for these `fbtrees` is as such:

```
datatype 'a fbtree = Leaf of 'a | Node of 'a fbtree seq
```

All of the functions we've previously defined for binary trees have analogs for `fbtree`s as well. For example, here is the size function on `fbtree`s.

```
fun size (Leaf x) = 1
  | size (Node s) = Seq.reduce (op +) 0 (Seq.map size s);
```

Task 5.1 Write the function

```
depth : 'a fbtree -> int
```

Depth for fbtrees is defined as the longest path from the root to a leaf in the tree. A leaf should have depth 1.

Task 5.2 Write the function

```
trav : 'a fbtree -> 'a list
```

such that `trav T` will evaluate to the inorder traversal of the tree T. This function might be useful when testing your code. Hint: the function `@` may be useful.

Higher order functions, like `map` and `reduce`, can be implemented for `fbtree`s as well. Here is the implementation for map:

```
(* fbmap : ('a -> 'b) -> 'a fbtree -> 'b fbtree *)
(* REQUIRES: f is a total function *)
(* ENSURES: the output of map f T is equivalent to applying f
   to all the leaves of T *)
fun fbmap f (Leaf x) = Leaf (f x)
  | fbmap f (Node s) = Node (Seq.map (fbmap f) s)
```

Task 5.3 Write the function

```
fbreduce : ('a * 'a -> 'a) -> 'a -> 'a fbtree -> 'a
```

When g is associative and z is a zero for g, `fbreduce g z T` will evaluate to the result of pairwise combining the items in `trav T` using `g`. `g` is associative when, for all `a`,`b`,`c`, we have `g(a,g(b,c))` = `g(g(a,b),c))`. `z` is a zero for `g` when, for all `x`, we have `g(x,z)` = `x`.