Recitation 4 – *Scan Reloaded and Reductions*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2014)

*September 16th, 2014*

1 Announcements

- How did Skyline go?
- Bignum is out—get an early start!
- Questions about homework or lecture?

2 Scan Implementation

Scan is a complex operation, so we're going to work through one level of recursion (not a whole trace like you did for `iterh` on Minilab).

Let $S = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$. We'll look at `scan op+ 0 S`.

First, `scan` contracts the sequence to a new $S'$ by contracting every other pair of elements, giving us $S' = \langle \_, \_, \_, \_ \rangle$.

Then, a recursive call to `scan op+ 0 S'` will return:

$$(\langle \_, \_, \_, \_ \rangle, \_ )$$

We then interleave values from $S$ into $S'$, giving us the final scan of

$$(\langle \_, \_, \_, \_, \_, \_, \_, \_ \rangle, \_ )$$

Note that this particular sequence is much easier to scan than certain other sequences—why?
3 Scan That Combines In Groups of $k$

Real-world implementations of scan don’t actually combine in pairs of two. If a level in scan has $n$ elements, combining in pairs of two requires us to have $\frac{n}{2}$ threads working in parallel. The reality is that parallelizing execution has a significant overhead; thus, real-world implementations often separate the $n$ elements into groups of $k$ and each group combines its $k$ elements sequentially. This allows us to reduce the number of parallel threads to $\frac{n}{k}$. For simplicity, let’s assume our input sequences are powers of $k$.

Let’s see what this looks like for $k = 3$ for \( \text{scan} (\text{op} +) 0 \%[1, 2, 3, 4, 5, 6, 7, 8, 9] \).

Give the recurrences for work and span when we combine in groups of $k$ assuming we’re scanning with a constant time function.
4 Reduction

1. Write a function `rev` which reverses the input sequence. Here's the twist: you can only use the following functions: `map`, `reduce`, `empty`, `singleton`, `append`, `length`, `filter`.

   \[
   \text{fun rev (S : 'a seq) : 'a seq} =
   \]

2. Give a closed form for the work and span of `rev` under both the `ArraySequence` and `TreeSequence` implementations. Given two sequences `S` and `T` of size `n` and `m`, the cost bounds in `ArraySequence` for the above functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>[ \sum_{e \in S} W(f(e)) ]</td>
<td>[ \max_{e \in S} S(f(e)) ]</td>
</tr>
<tr>
<td>reduce</td>
<td>[ O(n) + \sum_{f(x,y) \in O_r(f,b,s)} W(f(x,y)) ]</td>
<td>[ O(\log n) \max_{f(x,y) \in O_r(f,b,s)} S(f(x,y)) ]</td>
</tr>
<tr>
<td>empty</td>
<td>[ O(1) ]</td>
<td>[ O(1) ]</td>
</tr>
<tr>
<td>singleton</td>
<td>[ O(1) ]</td>
<td>[ O(1) ]</td>
</tr>
<tr>
<td>append</td>
<td>[ O(n + m) ]</td>
<td>[ O(1) ]</td>
</tr>
<tr>
<td>length</td>
<td>[ O(1) ]</td>
<td>[ O(1) ]</td>
</tr>
<tr>
<td>filter</td>
<td>[ \sum_{e \in S} W(p(e)) ]</td>
<td>[ O(\log n) + \max_{e \in S} S(p(e)) ]</td>
</tr>
</tbody>
</table>

`TreeSequence` has identical bounds except `append` has \[ O(\log(n + m)) \] work and span, and the span of `map` is \[ \log n \] plus the max term.
5 BignumLab Starter!

Let’s help motivate some students to start this lab early! We have two input sequences $X$ and $Y$, both of which are bit sequences where \( \text{datatype bit} = \text{ZERO} \mid \text{ONE} \). We say that $X$ and $Y$ are non-negative, and they are represented from the least significant bit to the most, i.e. \( \text{nth } X \ 0 \) is the least significant bit of $X$. Let’s sequentially compute the sum of $X$ and $Y$.

\[
\text{fun sum (X, Y) =}
\]

6 Bonus: A Different Kind of Recurrence

Often in the real world, there’s no neat closed form for a lot of recurrences, but we still need a fast way to compute them! So here’s your chance to see how to do this for linear recurrences. You are given a sequence of tuples, \( S = \langle (A_i, B_i) \rangle_{0 \leq i < n} \), and told \( x_{i+1} = A_i x_i + B_i \), \( x_0 = k \).

Compute \( x_n \) in terms of \( k \) in \( O(n) \) work and \( O(\log(n)) \) span.

Hint: Be careful about associativity.

(Note: Our solution is 1 line)