Recitation 2 — Scan

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

7th September 2011

1 Announcements

- HW1 was due last night. If you're going to use a late day, please email us! Otherwise, we will grade whatever was in your hand in directory at midnight last night. This will not be the usual policy when homeworks are due on Mondays, but we want to grade today.
- HW2 is out. It's due at 23:59 EST on Monday 12th September, with late days as stated in the policy.
- Questions from lecture, homework, or life?

2 Scan

We mentioned the existence of a function `scan` quickly in lecture yesterday. We'll spend most of today describing exactly what `scan` is and showing off some unexpected applications of it.

`scan` takes a function as one of its arguments. The all of the text below makes the assumption that this function is *associative*. Recall the mathematical definition that a function f is said to be associative if and only if

$$\forall a \forall b \forall c. f(f(a,b),c) = f(a,f(b,c))$$

We also make the assumption that the initial value is a *right-identity* of the functional argument. Recall the mathematical definition that I is a right-identity of f if and only if

$$\forall a. f(a,I) = a$$

. We don't need these assumptions in general, and we'll come back to a version of `scan` later that doesn't have them, but it's a cleaner way to start thinking about `scan` with these properties.

2.1 Definition

`scan` has type

$$\texttt{scan} : (\alpha * \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} * \alpha)$$

Informally, scan computes both the reduction of the sequence using the supplied operator and the sequence of all the partial results that were computed along the way.

More formally, let f be an associative function, I a right-identity of f, and `s` a sequence value. If we use $\oplus$ as infix notation for f, `(scan f I s)` evaluates to

$$\left(\left\langle I, s_0, s_0 \oplus s_1, \ldots, s_0 \oplus \ldots \oplus s_{|s|-2} \right\rangle, s_0 \oplus \ldots \oplus s_{|s|-1}\right)$$

or

$$\left(\left\langle I, \bigoplus_{i=0}^{0} s_i, \bigoplus_{i=0}^{1} s_i, \bigoplus_{i=0}^{2} s_i, \quad \ldots \quad , \bigoplus_{i=0}^{|s|-2} s_i \right\rangle, \bigoplus_{i=0}^{|s|-1} s_i\right)$$

Note that these statements are not ambiguous only because f is associative. Since it is, all of the possible parenthesizations of the terms above compute the same thing, so the parentheses are not needed.

The terms in the sequence in the left component and the entire right component should look like applications of `reduce` with associative operation and an identity. Specifically, if f is associative and I is a right-identity of f, `(scan f b)` is logically equivalent to

```
fn s =>
  (tabulate (fn i => reduce f I (take(s,i))) (length s),
   reduce f I s)
```

It's worth noting that each of the calls to `reduce` applies f with I as the right argument exactly once, so this actually computes the pair

$$\left(\left\langle I, s_0 \oplus I, (s_0 \oplus s_1) \oplus I, \ldots, (s_0 \oplus \ldots \oplus s_{|s|-2}) \oplus I \right\rangle, (s_0 \oplus \ldots \oplus s_{|s|-1}) \oplus I\right)$$

If I wasn't the right-identity of f, the equivalence wouldn't hold.

With the assumption that `f` is associative, it's also the case that `(scan f b)` is logically equivalent to `(iterh f b)` in the same way that `(reduce f b)` is logically equivalent to `(iter f b)`. `iter` and `iterh` happen choose a very sequential association of the expressions in question, but again this doesn't matter. The associations only differ in that they have different implications for the cost of computing them, not what's computed.

Specifically, if `f` is a function that takes no more than a constant number of steps on all input, `(iterh f)` and `(iter f)` have both work and span in $O(n)$, where as `reduce` and `scan` both have work in $O(n)$ and span in $O(\lg n)$.

It's worth noting that while `reduce` and `scan` are highly parallel, unlike `iter` and `iterh`, they pay the price by having slightly less general types.

2.2 Note on Terminology

If f is a function and I is a relevant identity for f, we'll often say "f-scan" to mean

```
fn s => scan f I
```

For example, a "+-scan" is

```
fn s => scan (op+) 0
```

2.3 Example Uses of Scan

At first glance, `scan` seems not to offer much that isn't already available through `reduce`. With clever choices of associative functions, though, `scan` can be used to compute some surprising things efficiently in parallel.

2.3.1 `+-scan`

We saw a simple use of `scan` in lecture to compute all the prefix sums of a sequence:

```
fun prefixsum s = scan (op+) 0 s
```

For example, for the sequence $\langle 5, 2, 3, 2, -1\rangle$, `prefixsum` computes the pair

$$(\langle 0, 5, 7, 10, 12\rangle, 11)$$

2.3.2 Matching Parentheses

We can use `scan` to solve the parenthesis matching problem that we went over last week. The idea is that we first map each open parenthesis to 1 and each close parenthesis to -1. We then do a $+$-`scan` on this integer sequence. The elements in the sequence returned by `scan` exactly correspond how many unmatched parenthesis there are in that prefix of the string.

For example:

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

becomes

$$\langle 1, -1, 1, 1, -1, -1, -1\rangle$$

and then

$$\langle 0, 1, 0, 1, 2, 1, 0, -1\rangle$$

and then fails, because the counter went negative at some point indicating an imbalance.

```
functor ParensScan (S : SEQUENCE) : PARENS =
struct
  structure Seq = S
  structure SeqUtil = Util (Seq)

  open Seq
  open SeqUtil

  fun match s =
      let
        fun paren2int OPAREN = 1
          | paren2int CPAREN = ~1

        val C = map paren2int (preproc s)
        val (S,total) = scan (op+) 0 C
        val SOME(maxint) = Int.maxInt
      in
        (reduce Int.min maxint S) >= 0  andalso total = 0
      end
end
```

2.3.3 Stock Market Problem

We can also use `scan` to solve the Stock Market problem from this week's homework. This is somewhat more complicated than matching parentheses, so the code is slightly more involved. Here's the idea:

- We do a min-`scan`. The each element in the resultant sequence is the minimum of the corresponding prefix of prices.

- We use `map2` to map subtraction down the zip of this resultant sequence and the original sequence of prices. Each element of this new sequence is the jump in price that would be obtained by buying at the minimum and selling at that time.
- We maximize over this sequence of jumps to get the correct answer.

```
functor StocksScan (Seq : SEQUENCE) : STOCKS =
struct
  structure Seq = Seq
  open Seq

  fun stock s =
      let
        val SOME(maxint) = Int.maxInt
        val (p,_) = scan Int.min maxint s
      in
        case length s
         of 0 => NONE
          | _ => SOME (reduce Int.max 0e (map2 (op-) s p))
      end
end
structure ScanTest = StocksTest (StocksScan(ArraySequence))
```

2.3.4 Computing Fibbonacci Numbers

With a carefully chosen matrix, we can use `scan` to compute the Fibonnaci numbers. In the extremely unlikely event that you've forgotten, the Fibbonacci numbers are defined as follows:

Definition: The Fibbonacci numbers are an integer sequence given by the following recurrence[1]

- $F_{-1} = 1$
- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

We make the following claim about this definition, which we will prove by induction:

Claim:

For all natural numbers n,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Proof: We'll prove this by induction on n.
Base Case: $n = 0$

[1] It is slightly contrived, but harmless, to define the -1^{st} element of the Fibbonacci sequence. The other base cases are such that the recursive case will never use it, so this could be any constant and produce the same sequence of integers. This one happens to make the proof work, though.

Any $n \times n$ matrix to the zero power is the $n \times n$ identity matrix, so

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} F_1 & F_0 \\ F_0 & F_{-1} \end{pmatrix}$$

which is exactly as desired.

Inductive Case:

Assume that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

We want to show that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

It suffices to show that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

Recall matrix multiplication, specifically in the case of taking the product of two 2×2 matrices:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Therefore,

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} \\ &= \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \\ &= \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \end{aligned}$$

This is exactly as desired and concludes the proof.

Remember that matrix multiplication is an associative operation on square matrices. We'll only need 2×2 `int` matrices, so for simplicity let's represent them as values of type `int * int * int * int`.

The above proof means that we can compute the Fibbonacci numbers by applying scan to a matrix multiplication function:

```
functor FibboScan (S : SEQUENCE) : FIBBO =
struct
  structure Seq = S
  open Seq

  (* very simple representation of 2x2 matrices *)
  fun mmult ((a,b,c,d),(e,f,g,h)) = (a*e + b*g, c*e + d*g,
                                     a*f + b*h, c*f + d*h)
```

```
  (* returns the first n fibbonacci numbers *)
  fun fib n =
      let
        val s = tabulate (fn _ => (1,1,1,0)) (n+1)
        val (ans,_) = scan mmult (1,0,0,1) s
      in
        map (fn (_,_,_,x) => x) (drop(ans,1))
      end
end
```

Note that we have to produce $n + 1$ terms of our version of the Fibbonacci sequence and discard the first. This exactly corresponds to the choice we made to make the base case is correct.

Since the matrices are of a constant 2×2 size, the matrix multiplication is actually a constant time operation—we're really just doing a handful of integer additions and multiplications. That means that we can compute n Fibbonacci numbers with total work in $O(n)$ and span $O(\lg n)$.

3 Homework 2

This week's homework asks you to produce algorithms solving two different problems. As a way to introduce the problems to you, we'll now go over the brute force solutions to both. Your solutions will need to be substantially more clever; these brute force solutions are basically just direct transcriptions of the formal statement of the problem into SML syntax.

3.1 Closest Pair

The closest pair problem is to find the closest pair of points when given an unordered list of points in some two dimensional Euclidian space. Specifically, if d is the distance function for the space and s is a sequence set of points, you want to compute

$$\min \{d(p_i, p_j) | 0 \leq i < |s|, 0 \leq j < |s|, i \neq j\}$$

```
functor NaiveClosestPair (P : CP_PACKAGE) : CLOSEST_PAIR  =
struct
  structure Seq = P.Seq
  structure Point = P.Point
  open Seq
  open Point

  fun closestDist (s : point seq) =
      let
        (* generate all not-equal index pairs *)
        val indices = tabulate (fn x => x) (length s)
        val allindpair = flatten (map
                                    (fn x =>
                                        map (fn y => (x,y)) indices)
                                    indices)
        val neqs = filter (op=) allindpair

        (* compute all the pair wise distances *)
```

```
        val dists = map (fn (x,y) => dist(nth s x, nth s y)) neqs
      in
        (* minimize over them *)
        reduce Real.min Real.maxFinite dists
      end
end
```

3.2 Pittsburgh Skyline Problem

Given a sequence of buildings $B = \langle b_1, b_2, \ldots, b_n \rangle$, where each $b_i = (l_i, h_i, r_i)$, the *Pittsburgh skyline problem* is to find a sequence of points $S = \langle p_1, p_2, ..., p_{2n} \rangle$, $p_i = (x_i, y_i)$ such that if

$$X = \cup_{(l,h,r) \in B} \{l, r\}$$

then

$$S = sort_x\{(x, \max\{h : (l, h, r) \in B, l \leq x < r\}) : x \in X\}$$

where $sort_x$ sorts the points by their x-coordinate.

```
functor NaiveSkyline (Seq : SEQUENCE) : SKYLINE =
struct
  structure Seq = Seq
  open Seq

  fun skyline b =
      let
        fun L (l,h,r) = l
        fun H (l,h,r) = h
        fun R (l,h,r) = r

        fun between x (l,_,r) = l <= x andalso x < r
        fun only x = filter (between x)

        (* all x coordinates of any building *)
        val X = merge Int.compare (map L b) (map R b)

        val col = map (fn x => (x, reduce Int.max 0
                                          (map H (only x b))))
                      X

        fun ord ((x,_),(y,_)) = Int.compare(x,y)
      in
        sort ord col
      end
end
```