Recitation 3 — Scan

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

January 30, 2013

1 Announcements

- HW 2 is due next Monday. Hopefully you have all started by now; if not, now would be a good time.
- Questions from lecture or homework?

2 Scan

Yesterday, we covered the function `scan`. We'll recap the definition of `scan` briefly today, and show you how to solve interesting problems with it.

`scan` takes a function as one of its arguments. 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 *left-identity* of the functional argument. Recall the mathematical definition that I is a left-identity of f if and only if

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

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.

With the assumption that `f` is associative, `(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)`.

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 $O(n)$, whereas `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.

2.1 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
```

2.2 Recap

If $s = \langle 1, 6, 3, -2, 9, 0, -4 \rangle$, then

`(scan op+ 0 s)` yields the following:

$(\langle 0, 1, 7, 10, 8, 17, 17 \rangle, 13)$

Remember that in the result, location i stores the "sum" of the values at locations **before** i in the original sequence. There is a variant of `scan` called `scanI` which sums the values at locations before and including i.

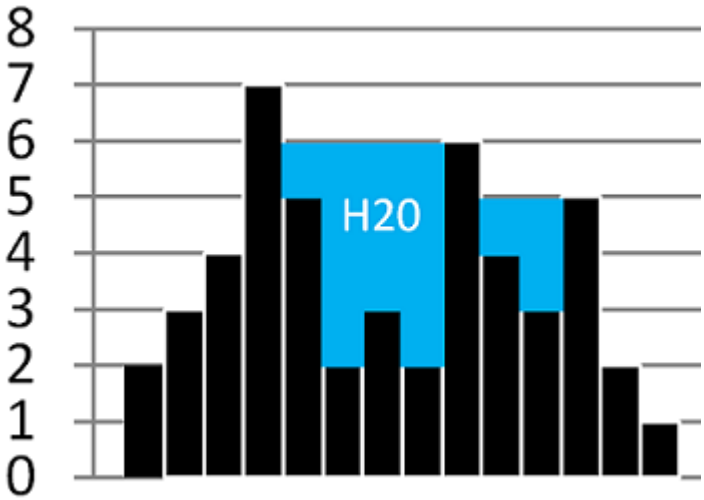
2.3 Example Uses of Scan

At first glance, `scan` seems to offer not 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 Histogram

Consider the following problem:

Given a histogram, if we were to pour water over it, how much water (in terms of area) would it hold? Fof simplicity we will represent a histogram as a sequence of non-negative integers. For example the histogram shown below is represenented by the sequence $s = \langle 2, 3, 4, 7, 5, 2, 3, 2, 6, 4, 3, 5, 2, 1 \rangle$, and holds 15 units of water.



Any ideas on how we might solve this problem?

The idea is to single out one bar b_i. If we know the maximum of the bar heights to the left of b_i ($maxl$) and the maximum of the bar heights to the right of b_i ($maxr$), given that $maxl > height(b_i)$ and $maxr > height(b_i)$ then the water b_i will hold above it is $MIN(maxl, maxl) - height(b_i)$.

Do we know of any functions that could be useful for generating these sequences of max-bar-heights? How about `scan`. Using a few `scan`'s, a `map` and a `reduce`, this problem becomes very simple.

```
fun rev s =
    let val n = length s
    in tabulate (fn i => nth s (n - i - 1)) n
    end

fun histogramFill (hist : int seq) =
    let
      val (lHeights, _) = scan Int.max 0 hist
      val (rHeightsRev, _) = scan Int.max 0 (rev hist)
      val heights = map2 Int.min lHeights (rev rHeightsRev)

      fun collect (maxHeight, thisHeight) =
          Int.max (maxHeight - thisHeight, 0)
    in
      reduce op+ 0 (map2 collect heights hist)
    end
```

To get the maximum height to the left of each position, we use a `scan` with the `Int.max` operator. How about getting the maximum height to the right of each position? Use a `scan` on the reverse of the list, and then reverse it again! We can then compute the total fill with a `map` and a `reduce`.

2.3.2 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]

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

[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.

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: $n = 0$

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,

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

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:

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

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 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 $O(\lg n)$.

2.3.3 Matching Parentheses

We can use `scan` to solve the parenthesis matching problem that we went over two weeks ago. 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 parentheses 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.

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

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