Recitation 4 — Scan, Reduction, MapCollectReduce

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

September 18, 2013

1 Announcements

- How did HW 2 go?
- HW 3 is out—get an early start!
- Questions about homework or lecture?

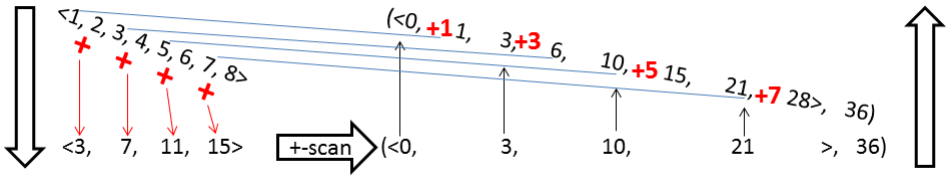
2 Scan Implementation Recap

Here is a neat diagram which summarizes how `+-scan` works:

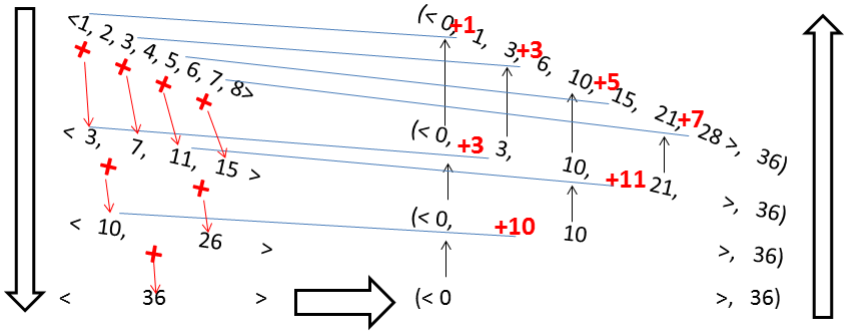
If $\texttt{s} = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$, then

`scan op+ 0 s` yields the following.

Here's one level of the recursive evaluation:



And here's the whole thing:



3 Parenthesis Distance

Remember the `parenDist` problem (I know, we are obsessed with those curvy things)? To solve this problem using `scan`, we need to find a way to associatively combine the solution we obtained from the smaller sub-

problems. We'll work with the sequence (which we obtained using `scan` last week) of the values the counter takes on as it increments for open parens and decrements for close parens. Here are some hints:

1. Candidates for the pair of matching parens with the largest distance are those delimited by zeroes (why is this the case?).
2. The number of characters between the zeroes is one more than the distance between the matching parens, because we included the open parenthesis.

We want to start from each zero and count the number of characters until the next zero, resetting the count once we reach the next zero. This sounds almost like `copy-scan`, except rather than simply copy over values until we reach the next zero, we want to maintain a counter that we increment each time (note that we're counting over values of a counter – wow, meta!)

Here's a first (wrong!) attempt at a solution:
Ler's use a datatype with two constructors: `MATCH` and `INCR`. Initially, all zeroes in the sequence are assigned a value of `MATCH 0` and other elements are assigned a value of `INCR`. This is done by mapping over the sequence with a function `preDist`. Intuitively, `MATCH x` indicates that we're `x` characters into a set of matched parens, and `INCR` indicates that the counter should be incremented. We define another operator `accDist` which acts somewhat like `copy`. It always returns `MATCH x`, where `x` is reset to 0 when we reach a `MATCH 0`, but is `MATCH (y + 1)` when we see an `INCR` and the previous value of the counter was `y`. We can do the counting and get the lengths of all the matches by scanning over the sequence using `accDist`.

```
datatype parenCount = MATCH of int | INCR

fun preDist 0 = MATCH 0
  | preDist _ = INCR

fun accDist (_, MATCH x) = MATCH x
  | accDist (MATCH x, INCR) = MATCH (x + 1)
```

This sounds like it should work, and it would if we were using `iter` instead of `scan`. What's wrong with it here?

`accDist` isn't associative! Consider the sequence (abbreviating `MATCH` with M and `INCR` with I) $\langle M(0), I, I\rangle$. `accDist(accDist(M(0), I), I) = accDist(M(1), I) = accDist(M(2))`, as we would expect. However, `accDist(M(0), accDist(I, I))` will not just return a different answer, it will raise a match exception, because we haven't specified the behavior of `accDist` when the first argument is `INCR`!

How can we make an operator that does what we want but is associative? To solve this problem, we should think about `accDist` as we would think of the argument to `reduce`. It will be applied to arbitrary subtrees, not to each element in order! Let's go through each case:

- `(INCR, INCR)`. When we combine the result of these two increments with a match on the left, we want it to increment twice. So, let's have the `INCR` constructor also carry an integer indicating the number of times to increment, and we'll add them when we combine two `INCR`s.
- `(MATCH x, INCR y)`. Here, we have `x` characters of a matching set on the left, and an instruction to increment `y` times. So, we now have a matching set of `x + y` characters.
- `(_, MATCH x)`. No matter what we've seen on the left, there's a new matching set starting on the right, and so we want to pass this on.

With these cases in mind, we can write out the code.

```
datatype paren = OPAREN | CPAREN

fun paren2int OPAREN = 1
  | paren2int CPAREN = ~1

val SOME maxInt = Int.maxInt

fun isMatch (prefixSum, total) =
    length prefixSum > 0 andalso
    reduce Int.min maxInt prefixSum >= 0 andalso total = 0

datatype parenCount = MATCH of int | INCR of int

fun preDist 0 = MATCH 0
  | preDist _ = INCR 1

fun accDist (_, MATCH x) = MATCH x
  | accDist (INCR x, INCR y) = INCR (x + y)
  | accDist (MATCH x, INCR y) = MATCH (x + y)

fun parenDist parens =
    let
      val iParens = map paren2int parens
      val (prefixSum, total) = scan op+ 0 iParens
    in
      if not (isMatch (prefixSum, total)) then NONE
      else let
          val preScan = map preDist (append (prefixSum, singleton 0))
          val (allVals, MATCH lastDist) = scan accDist (MATCH 0) preScan
          val distVals = map (fn MATCH x => x) allVals
        in
          SOME (Int.max (lastDist, reduce Int.max 0 distVals) - 1)
        end
    end
```

`parenDist` first checks if the input is balanced, and if so it computes the prefix sums on the sequence of 1's and -1's. Next, we perform another scan with `accDist` (you should convince yourself this new version is associative!) Finally we take the longest match over all prefixes.

For example:

$$\texttt{parens} = \langle (, (,), (,),), (,) \rangle$$

$$\texttt{iParens} = \langle 1, 1, -1, 1, -1, -1, 1, -1 \rangle$$

$$(\texttt{prefixSum}, \texttt{total}) = (\langle 0, 1, 2, 1, 2, 1, 0, 1 \rangle, 0)$$

and then

$$\texttt{preScan} = \langle M(0), I(1), I(1), I(1), I(1), I(1), M(0), I(1), M(0) \rangle$$

$$(\texttt{allVals, M lastDist}) = (\langle M(0), M(1), M(2), M(3), M(4), M(5), M(0), M(1), M(0) \rangle, M(0))$$

We take the maximum and subtract 1, getting `SOME 4` as the answer.

3.1 A Reduction

With some trickery (and some uses of `scan`, `map`, `reduce`), we can solve the `parenDist` problem in a different way. Recall the MCSS (maximum contiguous subsequence sum) problem which is to find $\max_{0 \le i \le j \le n} \left(\sum_{k=i}^{j} S_k \right)$[1]. The key observation is that by making the zeroes an extremely large negative number and counting the non-zero values once, we are effectively doing an MCSS problem!

```
fun parenDist s =
  let
    len = length s
    fun paren2int OPAREN = 1
      | paren2int CPAREN = ~1

    fun reduceToMCSS 0 = ~len
      | reduceToMCSS _ = 1

    val C = map paren2int s
    val (S,total) = scan (op+) 0 C
    val SOME(maxint) = Int.maxInt
    val C' = map reduceToMCSS S
  in
    if len = 0 orelse (reduce Int.min maxInt S) < 0 orelse total <> 0 then NONE
    else SOME((MCSS C') - 1)
  end
```

Laziness, cool!

4 `fields` and `tokens`

Two useful string parsing routines are `fields` and `tokens`, which break a string into a sequence of words. The only difference between the two is that fields will return empty words (which is preferable for parsing data written for or by computers), whereas tokens will not (which is more useful for human-centric data).

More specifically, we pick some characters to be delimiters (the argument f takes a character and returns whether it is a delimiter) and define a word to be a maximal string without delimiters. Note the input string might consist of multiple consecutive delimiters (in which case `fields` which return an empty string for that word and `tokens` will simply omit it). For example, we'll parse a line from a comma-separated values (CSV) file.

```
val fields :  (char -> bool) -> string -> string seq
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
```

which would return

⟨"a", "", "", "line", "of", "a", "csv", "", "file"⟩.

Using `tokens`, instead of `fields`, the result would have been

[1] This is an alternate formulation to the one that sums up till `j-1`

⟨"a", "line", "of", "a", "csv", "file"⟩.

Traditionally `fields` and `tokens` have been implemented in a sequential fashion, starting at the front end and processing it one character at a time until reaching the end. During the processing, whenever a whitespace or delimiter is found, the algorithm declares the current token or field finished and starts a new one. However, `fields` and `tokens` can be implemented in parallel.

How do we go about implementing `fields` *in parallel?* We can figure out where each field starts by finding locations of the delimiters. Furthermore, this also tells us where each field ends—right before each delimiter (which starts the next field), and at the end of the string. Therefore, if there is a delimiter at location i and the next delimiter is at $j \geq i$, we have that the field starting after i contains the substring extracted from locations $(i+1)..(j-1)$, which may be empty. This leads to the following code, in which `delims` contains location of each delimiter. We use the notation $\oplus$ to denote sequence concatenation.

```
fun fields f s = let
  val delims = ⟨−1⟩ ⊕ ⟨ i :  0 ≤ i < |s| ∧ f(s[i]) ⟩ ⊕ ⟨|s|⟩
in
  ⟨ s[delims[i]+1, delims[i+1]) :  0 ≤ i < |delims|−1 ⟩
end
```

(If you're curious, we can implement the pseudocode $\langle\, i : 0 \leq i < |s| \wedge f(s[i]) \,\rangle$ using the sequence library functions `mapIdx` and `filter`). To illustrate the algorithm, let's run it on our familiar example.

```
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
delims = ⟨−1,1,2,3,8,11,13,17,18,23⟩
result = ⟨ s[0,1), s[2,2), s[3,3), s[4,8), s[9,11), s[12,13), ... ⟩
result = ⟨"a", "", "", "line", "of", "a", "csv" , "", "file" ⟩
```

4.1 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you might have seen in 15-150 which just involved a map, and then a reduce. The map-reduce paradigm actually involves a map, followed by a collect, followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.

The map-reduce paradigm processes a collection of documents based on a map function f_m and a reduce function f_r supplied by the user. The f_m function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the f_r function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map function f_m and reduce function f_r are the following:

$$\begin{aligned} f_m &: (\mathit{document} \to (\mathit{key} \times \alpha)\ \mathit{seq}) \\ f_r &: (\mathit{key} \times (\alpha\ \mathit{seq}) \to \beta) \end{aligned}$$

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the f_m

and f_r functions are sequential functions. Parallelism comes about since the f_m function is mapped over the documents in parallel, and the f_r function is mapped over the keys with associated values in parallel.

In ML, map-reduce can be implemented as follows:

```
1  function mapCollectReduce f_m f_r docs =
2    let
3        pairs = flatten ⟨ f_m(s) : s ∈ docs ⟩
4        groups = collect String.compare pairs
5    in
6      ⟨ f_r(g) : g ∈ groups ⟩
7    end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

$$\begin{array}{ll} & \texttt{flatten}\,\langle\langle a,b,c\rangle,\langle d,e\rangle\rangle \\ \Rightarrow & \langle a,b,c,d,e\rangle \end{array}$$

As an example application of the paradigm, suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following f_m and f_r functions.

function $f_m(\texttt{doc}) = \langle (w,1) : w \in \texttt{tokens doc}\rangle$

function $f_r(w,s) = (w, \texttt{reduce + 0 } s)$

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

```
val countWords = mapCollectReduce f_m f_r

countWords ⟨"this is a document",
            "this is is another document",
            "a last document"⟩
```

$\Rightarrow \langle (\text{“a”},\ 2), (\text{“another”},\ 1), (\text{“document”},\ 3), (\text{“is”},\ 3), (\text{“last”},\ 1), (\text{“this”},\ 2)\rangle$