Recitation 1 — Parenthesis Matching

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

August 28, 2013

Welcome to 210! This recitation is aimed at getting you started on Homework 1, which has been released. We will be using SML/NJ as our default programming language, which you should be familiar with if you have taken 15-150 previously. We will expect you to write clean and readable (self-documenting!) code as well as mathematical proofs.

1 Administrivia

Where is my assignment? We will be distributing the assignments for this course through Autolab (`https://autolab.cs.cmu.edu/`). You will also be submitting your assignments on Autolab.

When are Office Hours? Office Hours will be posted on the course webpage at

`http://www.cs.cmu.edu/~15210/staff.html`

These times are subject to change. If you have time conflicts and cannot attend any of the listed office hours, please contact one of the course staff.

2 Let's Begin!

We'll begin with a running example: the parenthesis matching problem. We define it as follows:

- **Input:** a `char` sequence `s : char seq`, where each s_i is either an "`(`" or "`)`". For instance, we could get a parenthesis-matched sequence

  $$s = \langle (, (,), (,),) \rangle$$

  or an unmatched one

  $$t = \langle), (,), (,),) \rangle$$

- **Output:** `true` if `s` represents a parenthesis-matched string and `false` otherwise. In the above examples, the algorithm should output `true` on input s and `false` on input t.

To simplify the presentation, we will be working with a `paren` data type instead of characters. Specifically, we will write a function `match` of type `paren seq -> bool` that determines whether the input is a *well-formed parenthesis expression* (i.e., it is a parenthesis-matched sequence). The type `paren` is given by

```
datatype paren = OPAREN | CPAREN
```

where `OPAREN` represents an open parenthesis and `CPAREN` represents a close parenthesis.

So, how would we go about solving this problem? Lets begin with a simple sequential solution and work our way to a work-optimal parallel solution.

2.1 Sequence `iter`

As in 15-150, you will be making extensive use of a `SEQUENCE` library throughout this course. For the current problem, we'll use the function `iter` (for iterate) from the sequence library. It has the following type:

```
val iter : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
```

If `f` is a function, `b` is a value, and `s` is a sequence, then `iter f b s` iterates f with left association on s using b as the base case. You may think of f as a state transition function and b as a base state.

2.2 Back to Parentheses

How can we use this to solve the parenthesis matching problem? As we iterate across the sequence, we can keep track of the number of open parentheses that we have seen so far and subtract from this number when we find a closing one. For a well-formed parenthesis expression, when we reach the end of the sequence we should have a value of 0.

Is this sufficient?

No, if the number goes below zero at any point, then when we know we can't possibly have a well-formed parenthesis expression—we'll designate a special state to represent this outcome.

What type should we use to represent the state? `int option`.

```
fun parenMatch p =
    let
      fun pm ((NONE, _) | (SOME 0, CPAREN)) = NONE
        | pm (SOME c, CPAREN) = SOME (c-1)
        | pm (SOME c, OPAREN) = SOME (c+1)
    in
      iter pm (SOME 0) p = (SOME 0)
    end
```

You can show that this solution has $O(n)$ work and span, where n is the length of the input sequence. *How can we make it more parallel?*

3 Divide and Conquer

As you have already seen in previous classes, divide and conquer is a powerful technique in algorithms design that often leads to efficient parallel algorithms. A typical divide and conquer algorithm consists

of 3 main steps (1) divide, (2) recurse, and (3) combine.

To follow this recipe, we first need to answer the question: how should we divide up the sequence? We'll first try the simplest choice, which is to split it in half—and attempt to merge the results together somehow. This leads to the next question: what would the recursive calls return?

Let's try returning whether the given sequence is well-formed. Clearly, if both s_1 and s_2 are well-formed expressions, s_1 concatenated with s_2 must be a well-formed expression. However, we could have s_1 and s_2 such that neither of which is well-formed but $s_1 s_2$ is well-formed (e.g., "(((" and ")))"). This is not enough information to conclude whether $s_1 s_2$ is well-formed.

We need more information from the recursive calls. You are probably already familiar with a similar situation from mathematical induction—you often need to strengthen the inductive hypothesis. We'll rely crucially on the following observations (which can be formally proven by induction):

Observation 3.1. *If s contains "()" as a substring, then s is a well-formed parenthesis expression* ***if and only if*** *s' derived by removing this pair of parenthesis "()" from s is a well-formed expression.*

Observation 3.2. *If s does* ***not*** *contain "()" as a substring, then s has the form "$)^i($". That is, it is a sequence of close parens followed by a sequence of open parens.*

That is to say, on a given sequence s, we'll keep simplifying s *conceptually* until it contains no substring "()" and return the pair (i, j) as our result. This is relatively easy to do recursively. Consider that if $s = s_1 s_2$, after repeatedly getting rid of "()" in s_1 and separately in s_2, we'll have that s_1 reduces to "$)^i($" and s_2 reduces to "$)^k($" for some i, j, k, ℓ. To completely simplify s, we merge the results. That is, we merge "$)^i($" with "$)^k($". The rules are simple:

- If $j \leq k$ (i.e., more close parens than open parens), we'll get "$)^{i+k-j}($".
- Otherwise $j > k$ (i.e., more open parens than close parens), we'll get "$)^i($".

This leads directly to a divide and conquer algorithm.

3.1 How to split a sequence in half?

The sequence library we give you provides a conceptual view of sequences called `treeview` that lends itself particularly well to divide-and-conquer algorithms. For those of you who have used `listview` in 15-150, this concept will be very familiar. To review, we have a data type `'a treeview` defined as follows:

```
datatype 'a treeview =
    EMPTY
  | ELT of 'a
  | NODE of ('a seq * 'a seq)
```

The function `showt` provides a means to examine the sequence in the `treeview`:

```
val showt : 'a seq -> 'a treeview
```

Essentially, `showt s` splits the sequence in approximately half and returns both halves as sequences, provided that the input sequence has length at least 2. The two base cases are for empty and singleton sequences.

3.2 Implementing the algorithm in `treeview`

To make it more obvious which calls are being made in parallel, we will also introduce a function

```
par : (unit -> 'a) * (unit -> 'b) -> 'a * 'b
```

If you run `par (f, g)`, this construct allows you to execute the two functions `f` and `g` in parallel.

```
fun parenMatch s =
    let
      fun pm s =
          case showt s
           of EMPTY => (0,0)
            | ELT OPAREN => (0,1)
            | ELT CPAREN => (1,0)
            | NODE (l, r) =>
              let
                val ((i,j),(k,l)) =
                  par (fn () => pm l, fn () => pm r)
              in
                if j > k then (i, l+j-k)
                else (i+k-j, l)
              end
    in
      pm s = (0, 0)
    end
```

4 To Be Continued...

We will discuss running time analysis in future lectures and recitations. Homework 1 has been released (on Autolab). There will be a SML coding portion related to parenthesis matching and also problems on asymptotic analysis. Please check the course website for office hours if you have trouble.