Recitation 2 – *Parenthesis Matching*

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

January 21st, 2014

1 Announcements

- Lab 1 – *ParenLab*, has been released! It is due next Monday, January 27th.

2 Parenthesis Matching

We define the parenthesis matching problem as follows:

- **Input:** a character sequence `s : char seq`, where each s_i is either a "`(`" 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 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.

2.1 Exercise

We'll begin by looking for a sequential solution (no parallelism yet!). In the space below, write the function `parenMatch : paren seq -> bool`. You should strive for $O(n)$ work and span, where n is the length of the input sequence. You might find the `SEQUENCE` function `iter` useful in this example:

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

Solution 2.0

```
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
```

Explanation: As we iterate across the sequence, we can keep track of the number of open parentheses that we have seen so far. We subtract from this number when we find a closing parenthesis. For a well-formed parenthesis expression, we should end with a value of 0.
Note that if the number goes below 0 at any point, then we can't possibly have a well-formed expression - we propagate a `NONE` to handle this situation.

$O(n)$ work and span is pretty good, but we can do better!

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.*

3.1 Splitting a sequence in half

The sequence library provides a conceptual view of sequences called `treeview`, which lends itself quite nicely to divide-and-conquer algorithms. We have

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

as well as a means for examining a sequence in `treeview`:

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

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

3.2 Calling functions in parallel

We introduce a function

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

Specifically, `par (f, g)` is logically equivalent to `(f (), g ())` for functions `f` and `g`. You should use `par` to indicate which functions should be run in parallel.

3.3 Exercise

Using the above observations and code recommendations, implement the function `parenMatch : paren seq -> bool` using a divide-and-conquer strategy. You should strive for $O(n)$ work and $O(\log n)$ span. (Note that `showt` is $O(1)$ work and span).

Solution 3.0

```
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
```

Explanation: We keep simplifying s *conceptually* until it contains no substring "()" and return the pair (i, j) as our result. 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 "$)^i($".