Recitation 1 — Parenthesis Matching and SML Review

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

August 31, 2011

This recitation is aimed at helping you shake off some dust from the summer and getting you started on Homework 1. As with last semester in 15-150, we are using SML/NJ as our default programming language. We also expect you to write readable code and readable mathematical proofs.

1 Announcements

Where is my assignment? We are experimenting with a new way of handing out assignments. Unlike last semester, we won't be putting tar balls on the web for you to download. Instead, we'll distribute the assignments through a read-only `git`[1] repository. To start you off, we've put together a handout on `git` commands, with pointers to more advanced features:

```
http://www.cs.cmu.edu/~15210/resources/git.pdf
```

which will also be linked from the Resources page.

What is my grade? If you want to know your grades, as well as other class stats, visit the Gradebook page on the course web site and follow the instructions there. For security reasons, you can only view your own grades, and you will be authenticated via WebISO if you aren't currently logged on.

2 The Fun Starts Now

As a running example, we will look at the parenthesis matching problem, which is defined as follows:

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

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

  or a non-matching one

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

- **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 Sequence.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

[1] `git` is a fully distributed version control system, initially developed for Linux kernel development.

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

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

How would we go about solving this problem?

3 Sequence Fold (Ahh.. It's called "iterate" in 210)

We'll begin with the simplest sequential solution and work our way to a work-optimal parallel solution. The 15-210 package has various sequence implementations, all ascribing to the `SEQUENCE` signature. This is pretty similar to what you had in 15-150, so you should feel right at home.

For the current problem, we'll recall the function `iter` (for iterate) from the sequence library. It has the following type:

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

Now if `f` is a function, `b` is a value, and `s` is a sequence value, then `iter f b s` iterates f with left association on s using b as the base case.

How can we use this to solve the parenthesis matching problem? A simple way of thinking about the `iter` function is to think of it as a state transition. The function `f` that we pass to `iter` is responsible for transforming the state upon seeing an input element. For this problem, we want the state to keep track of the number of unmatched open parentheses so far. Therefore, when we see an open paren, the number goes up by 1, and when we see a close paren, the number goes down by 1. Using this rule, the number could go below zero if we see more close parens than open parens. This is when we know we can't possibly have a well-formed parenthesis expression—we'll designate a special state to represent this outcome.

More specifically, our state is an `int option`, where we use `SOME opens` to mean "*we have `opens` unmatched open parens*" and `NONE` to mean "*we have seen too many close parens and the expression is not well-formed.*" We start with `SOME 0` as our initial state because there is no unmatched open parens at the beginning—and it is not difficult to see that an expression is well-formed if and only if we leave *no* unmatched parens at the end (i.e., the state is `SOME 0`).

This leads to the following code:

```
fun match s =
    let
      fun check (NONE, _) = NONE
        | check (SOME c, OPAREN) = SOME(c+1)
        | check (SOME c, CPAREN) =
          case (Int.compare (c,0))
           of GREATER => SOME (c-1)
            | _ => NONE
```

```
in
  case (iter check (SOME 0) (preproc s))
   of SOME(0) => true
    | _ => false
end
```

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

4 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 the merge their results somehow. This leads to the next question: what would the recursive calls return?

The first thing that comes to mind might be that the function returns 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. The problem is that 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 crucially rely on the following observations (which can be formally shown by induction):

Observation 4.1. *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.*

Applying this reduction repeatedly, we can show that a parenthesis sequence is well-formed if and only if it eventually reduces to an empty string.

Observation 4.2. *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 $i, j, k, \ell \in \mathbb{Z}_+ \cup \{0\}$. 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 directly leads to a divide and conquer algorithm.

4.1 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. The library provides `showt` and `hidet` for cutting up and assembling sequences; these functions have similar functionality as `showt` and `hidet` from 15-150 last semester. 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 tree view. It has the following type:

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

The specification of `showt` can be summarized as follows: Let `s` be a sequence value.

- If $|s| = 0$, `showt s` evaluates to `EMPTY`.
- If $|s| = 1$, `showt s` evaluates to `ELT(`s_0`)`.
- If $|s| > 1$, and `NODE(take (s,` $(|s|/2)$`), drop (s,` $(|s|/2)$`))` evaluates to some value `v`, `showt s` evaluates to `v`.

Essentially, `showt s` splits the sequence in approximately half and returns both halves as sequences.

4.2 Implementing the algorithm in `treeview`

```
fun match s =
    let
      fun match' s =
          case (showt s)
           of EMPTY => (0,0)
            | ELT OPAREN => (0,1)
            | ELT CPAREN => (1,0)
            | NODE (L,R) =>
              let
                val (i,j) = match' L
                val (k,l) = match' R
              in
                case Int.compare(j,k)
                 of GREATER => (i, l + j - k)
                  | _ => (i + k - j, l)
              end
    in
      case (match' (preproc s))
```

```
    of (0,0) => true
     | _ => false
end
```

Running Time Analysis: Let's assume that `showt s NONE` takes $O(\log n)$ work and depth on any sequence of length n. We can formulate the work and depth recurrences as follows:

$$\begin{aligned} W(n) &= 2 \cdot W(n/2) + W_{showt}(n) = 2 \cdot W(n/2) + O(\log n) \\ D(n) &= D(n/2) + D_{showt}(n) = D(n/2) + O(\log n). \end{aligned}$$

Clearly, we have $D(n) = O(\log^2 n)$. We will see in class how to solve this type of recurrences in details. For now, if you recall from 15-251 or from your formula sheet last semester, $W(n)$ solves to $O(n)$.