

Recitation 1 — Parenthesis Matching and SML Review

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

January 18, 2012

This recitation is aimed at helping you shake off some snow from the winter and getting you started on Homework 1, which will be released later today. 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 also expect you to write clean and readable code as well as mathematical proofs.

1 Administrivia

Where is my assignment? We will be distributing the assignments for this course 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.

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.

What is my grade? If you want to know your grades, visit the Gradebook page on the course website and follow the instructions there. You will need to log in with your WebISO credentials.

2 The Fun Begins

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

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

or a non-matching one

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

¹`git` is a fully distributed version control system, initially developed for Linux kernel development. Since nobody reads footnotes, we won't go into any more detail here.

- **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

```
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 simplest sequential solution and work our way to a work-optimal parallel solution.

2.1 Sequence iter

We'll first introduce you to the `SEQUENCE` library which we will be using throughout the course. Recall the standard 15-150 `SEQUENCE` signature:

```
val length : 'a seq -> int
val nth : 'a seq -> int -> 'a
val tabulate : (int -> 'a) -> int -> 'a seq
val filter : ('a -> bool) -> 'a seq -> 'a seq
val map : ('a -> 'b) -> 'a seq -> 'b seq
val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
```

The 15-210 `SEQUENCE` library includes these as well as many other functions which will come in handy later on in the semester. 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
```

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.

2.2 Back to Parentheses

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 0, CPAREN) = NONE
      | check (SOME c, CPAREN) = SOME (c-1)
  in
    case (iter check (SOME 0) s)
    of SOME 0 => true
      | _ => false
  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 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_1s_2 is well-formed (e.g., “(((” and “)))”). This is not enough information to conclude whether s_1s_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 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 $)^i(j^j$. 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 $)^i(j^j$ and s_2 reduces to $)^k(\ell$ for some $i, j, k, \ell \in \mathbb{Z}_+ \cup \{0\}$. To completely simplify s , we merge the results. That is, we merge $)^i(j^j$ with $)^k(\ell$. The rules are simple:

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

This directly leads 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 `treeview` 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 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),(k,l)) = par (fn () => match' L, fn () => match' R)
         in
           case Int.compare(j,k)
           of GREATER => (i, l + j - k)
            | _ => (i + k - j, l)
         end
       end
  in
    case (match' s)
    of (0,0) => true
     | _ => false
  end
end

```

Running Time Analysis: Let's assume that `showt s` takes $O(\log n)$ work and span on any sequence of length n . We can formulate the work and span recurrences as follows:

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

It is not too hard to see that $S(n)$ is $O(\log^2 n)$

$$\begin{aligned}
 S(n) &= \log(n) + \log(n/2) + \dots + \log(1) \\
 &= \sum_{i=1}^{\log(n)} i \in O(\log^2 n)
 \end{aligned}$$

It is a little more work to see $W(n) = O(n)$ (see Lecture 3).

$$\begin{aligned}
 W(n) &= \log(n) + 2W(n/2) \\
 &= \log(n) + 2(\log(n) - 1) + 4(\log(n) - 2) + \dots + n(\log(n) - \log(n)) \\
 &= n \cdot 0 + n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 3 + n/16 \cdot 4 + n/n \cdot \log(n) \\
 &= n \sum_{i=1}^{\log(n)} i 2^{-i} \\
 &\leq n \sum_{i=1}^{\infty} i 2^{-i}
 \end{aligned}$$

Let $T = \sum_{i=1}^{\infty} i2^{-i}$. We can show $T = 2$:

$$\begin{aligned}
 T &= \sum_{i=1}^{\infty} i2^{-i} \\
 &= 1/2 \sum_{i=1}^{\infty} (i-1)2^{-(i-1)} + \sum_{i=1}^{\infty} 2^{-1} \\
 &= 1/2 \sum_{j=0}^{\infty} j2^{-j} + 1 \\
 &= T/2 + 1
 \end{aligned}$$

This implies that $T = 2$. So, $W(n) \leq 2n$ and $W(n) = O(n)$.

For the curious, here's a quick way to see it.

$$\begin{aligned}
 W(n) &= 2W(n/2) + c \log n \\
 &= c + 2(c \log \frac{n}{2} + 2W(n/4)) \\
 &= c + 2c \log \frac{n}{2} + 4c \log \frac{n}{4} + 8W(n/8) \\
 &\vdots \\
 &= c \log n + 2c \log \frac{n}{2} + 4c \log \frac{n}{4} + 8c \log \frac{n}{4} + \dots + n \cdot C \\
 &\leq 2cn = O(n)
 \end{aligned}$$

Of course, we could be more formal by proving it using induction.

4 To Be Continued...

Homework 1 will be released later today, and due next Wednesday. There will be a proof of correctness, but note that we will not be looking for a step-by-step code evaluation trace. You should be familiar with SML evaluation by now, so we'll be more interested in a higher-level discussion of the algorithm itself. Please check the course website for updates on office hours if you have trouble.