Lab 1 - *ParenLab*

Due: *Mon, Sept 8th, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Fall '14)

1 Introduction

This assignment is meant to give you some practice implementing a divide-and-conquer algorithm, end-to-end. You will implement two solutions to the *parenthesis distance* problem, and perform some analysis of your solutions. Note that this lab is conceptually a lot more difficult than the previous one, so get started early!

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf parenlab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `support/ArrayParenPackage.sml`
3. * `MkBruteForcePD.sml`
4. * `MkDivideAndConquerPD.sml`
5. * `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `parenlab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 The Parenthesis Distance Problem

We define a string s to be *closed* if it contains only '(' and ')' characters, and is one of the following:

empty: The empty string.

concatenated: The concatenation of two closed strings, $s_1 s_2$

matched: A single closed string s_0 surrounded by a pair of *matched* parentheses, i.e. (s_0)

Definition 4.1 (The Maximum Parenthesis Distance (MPD) Problem)**.** Given a **closed string** s of parentheses, return:

$$\max\left\{|x| : x \in \text{Substrings}(s) \mid x \text{ is matched}\right\}$$

Substrings(s) refers to all (contiguous) substrings of s, including s itself and the empty string. For example, the string "(()())(())", has a maximum parenthesis distance of 6. **Note** that the solution to the MPD problem may not be defined on some inputs (for example, the empty string has no matched contiguous substring).

4.1 Logistics

4.1.1 Representation

When solving this problem, instead of interacting with strings, you will work with sequences of `paren` values, where the type `paren` is defined in a structure that ascribes to `PAREN_PACKAGE` as:

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

with `OPAREN` corresponding to a left parenthesis and `CPAREN` corresponding to a right parenthesis.

4.1.2 Implementation

In this lab, you will implement two solutions to the parenthesis distance problem as the function

```
val parenDist : paren seq -> int option
```

such that `parenDist S` evaluates to `SOME m`, where m is the maximum parenthesis distance in S (if it is defined), and `NONE` otherwise (if the solution to the MPD is not defined for S).

In your solutions, you will also have access to the `Option210` structure, which you may find useful (you should avoid reimplementing functions available as part of this structure). This structure is located in `support/ArrayParenPackage.sml`.

4.1.3 Indicating Parallelism

As seen in recitation, you should use the 210 library function `par` (inside the structure `Primitives`) to express parallel evaluation. Parallel operations can also be expressed in terms of operations on sequences such as `map` or `reduce`. In this class, *you must be explicit about what calls are being made in parallel* to receive full credit.

4.2 The Brute-Force Algorithm

It is possible to give a brute-force algorithm by generating all possible solutions and picking the best. Note that this is different from the sequential solution which is provided for you in `MkSequentialPD.sml`.

Task 4.1 (15%). Complete the functor `MkBruteForcePD` in the file `MkBruteForcePD.sml` with a brute-force solution to the maximum parenthesis distance problem. You may use the solution to the parenthesis matching problem from recitation 2. You may also find `Seq.subseq` to be useful for your solution.

Remember that a brute-force solution is one which generates all possible solutions, filters out those that don't meet the requirements of the problem, and finally selects the best solution from those that remain. You will not receive full credit if you deviate from this strategy.

4.3 The Divide-and-Conquer Algorithm

You will now implement a solution to the maximum parenthesis distance problem using a divide-and-conquer algorithm. The work and span of your solution must satisfy the recurrences:

$$W(n) = 2 \cdot W(n/2) + W_{\texttt{showt}}(n) + O(1)$$

$$S(n) = S(n/2) + S_{\texttt{showt}}(n) + O(1)$$

where n is the length of the input `paren seq`, and $W_{\texttt{showt}}$ and $S_{\texttt{showt}}$ are the work and span of `showt` respectively. Assume that $W(1) = S(1) \in O(1)$. A solution with correct behavior but with work or span that is not described by the appropriate recurrence will not receive full credit.

Task 4.2 (40%). Complete the functor `MkDivideAndConquerPD` in `MkDivideAndConquerPD.sml` with a divide-and-conquer solution as described above. For this assignment, you are not required to submit a proof of correctness of your implementation. However, we advise that you work out a proof by mathematical induction for your solution as an exercise.

4.4 Style

Style grading for this assignment will be pass/fail. **You should review the full policy on the home page of the course website.**

4.5 Testing

Task 4.3 (5%). Add test cases to test your code in `Tests.sml`. For this assignment you should make sure you thoroughly and carefully test both of your implementations of the `PAREN_DIST` signature. Your tests should include edge cases and also more general test cases on specific sequences.

To aid with testing, we have provided a testing structure in `support/Tester.sml`, which should simplify the testing process. The structure `Tester` will test your implementations against test cases specified in `Tests.sml`. Test cases should be added as strings to the `tests` list. Each test case string should consist of the characters '(' and ')' only, which `Tester` will translate for you into a `paren seq`.

In order to test your code, run the following commands in the terminal. This lets you test your brute-force and divide-and-conquer implementations separately:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
...
- Tester.testBF ();
...
- Tester.testDC ();
...
```

Alternatively, you can simply play with your code at the REPL:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
...
- open Tester;
...
- BF.parenDist (iFromTC "()");
val it = SOME 2 :  int option
- DC.parenDist (iFromTC "()");
val it = SOME 2 :  int option
```

5 Written Questions

Write all answers to the following questions in `written.pdf`.

5.1 Analysis

Task 5.1 (5%). Give the work and span of your brute-force `parenDist` solution (from Task 4.1).

Task 5.2 (10%). Recall the work recurrence given for the divide-and-conquer solution to `parenDist`:

$$W(n) = 2 \cdot W(n/2) + W_{\texttt{showt}}(n) + O(1)$$

This recurrence is *parametric* in the cost of `showt`. Naturally, this depends on the implementation of `showt`. Complete the following tasks:

1. Solve the work recurrence with the assumption that $W_{\texttt{showt}}(n) \in \Theta(\log n)$.
2. Solve the work recurrence with the assumption that $W_{\texttt{showt}}(n) \in \Theta(n)$.

5.2 Recurrences

Find tight big-O bounds for the following recurrences. You can use any method taught in class (brick method, tree method, substituion method) to solve them. **Do not just give a final answer**. We expect to see some explanations to show how you solve the problem. You can assume for all of them that $T(1) = T(2) = 1$.

Task 5.3 (5%). $T(n) = 9T(n/3) + O(n^2)$

Task 5.4 (5%). $T(n) = T(\sqrt{n}) + O(\log n)$

Task 5.5 (5%). $T(n) = T(n/4) + O(\log^2 n)$

Task 5.6 (5%). $T(n) = 2T(\sqrt{n}) + O(1)$

Task 5.7 (5%). $T(n) = T(n/5) + T(7/10n) + O(\log n)$
Hint: This one is hard. Check the recitation for a very similar example.