APL – Final Exam

Mayer Goldberg

June 29, 2013

Contents

1 Instructions

You are to work on this final exam alone, without any partners, without copying any material from the internet, and without doing anything else that is ordinarily understood to be *academic dishonesty*.

You should include a `readme.txt` file with all your other files. This file should include your name, ID, email, and any explanations that are likely to be needed in order to run your code. As always, your submission should include some demos that show that your code works properly.

I want to give you some motivation to work on the Coq section, so: **If you complete the problems in the Coq section, you don't need to do anything else – Just had in your files** `2013-coq-final.v` & `readme.txt` **and you're done.**

If you don't want to work on the Coq part of the final, then please hand in two problems from the remaining 3 sections: *The Lambda Calculus*, *Threaded Code* and *Parsing Combinators*.

Please submit the exam by July 15, give-or-take a day.

2 Assisted Theorem Proving in Coq

The website contains the files `2013-coq-final.v` and `2013-coq-final.pdf`. The `*.v` file defines a theory of *real numbers*. You are given axioms, and some theorems *T0-T18*. I included proofs for *T0* & *T1*, so that you can have concrete examples of how to work with this theory. Theorems *T2-T18* contain `admit.` in their proofs. Your job is to complete these proofs and remove the `admit.` tactic, thereby completing the proofs.

3 The Lambda Calculus

3.1 Decompiler in Scheme

Recall the equation:

$$(\langle M, a\rangle\ \langle M, b\rangle) \quad = \quad MMba$$

Implement a variadic version of this in Scheme, so that you could have such applications with *arbitrarily-many* arguments. Use them to write a decompiler for the set of proper combinators in Scheme.

Below is a sample interaction with Scheme to show how your program should behave:

```
> (decomp '(3 2 4)
   (lambda (a b c)
     (lambda (x y)
       (lambda (z r f v)
         (((a a) (b b) (a b c x))
          (z (r (f a) (v v v v) (v v)) (y x))
          (a a a))))))
(lambda (x0 x1 x2)
  (lambda (x3 x4)
    (lambda (x5 x6 x7 x8)
      (((x0 x0) (x1 x1) (x0 x1 x2 x3))
        (x5 (x6 (x7 x0) (x8 x8 x8 x8) (x8 x8)) (x4 x3))
        (x0 x0 x0)))))
```

4 Threaded Code

Please submit **two** problems from this section.

4.1 Address Interpretation

Implement your own threaded, interactive language (TIL) using address interpretation. The implementation language should be C (not C++). Your system should support basic arithmetic & Boolean operations, conditionals (an *if-then-else*-like mechanism), function calls.

4.2 Multi-threading

Change the underlying architecture of your TIL, to add multi-threading capabilities. This includes

- Changes to the underlying architecture, to support data and control stack switching.
- Primitives for managing threads: Creation, suspension, resuming, and killing.

4.3 Decompilation of threads

Support online decompilation of threaded code by adding a thread named DECOMPILE that takes a name of a thread and returns its source code.

4.4 Deployment

Support deployment of an "executable":

- Create an export routine for extracting a word and all of its dependencies, as a minimal dictionary, into a binary file.
- Create a load procedure that will load an executable from a file.
- Modify the address interpreter to take the name of a binary file on the command line, preceded by a command-line switch.

5 Parsing combinators

5.1 Parsing combinators for ambiguous grammars

Modify the parsing-combinators package so as to work with ambiguous grammar. Assume that each of your parsers can now return a list of ASTs. This list will contain more than one element, in case the grammar is ambiguous and there is more than one way to parse a single input. Note that the source of more than one AST is really the `disj` procedure, and you need to modify the other procedures just for the sake of compatibility with the new API (i.e., parsers the success-continuation of which return lists of ASTs).