Project Part 3

Jason Hemann

January 31, 2018

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- Name your file `main.ss`, and use a command like the following: (`zip <username>-p3.zip main.ss parenthec.ss`)

2 To begin:

This assignment **relies** on your successful completion of part 2. If you haven't successfully completed part 2 (and maintained the versions of your code along the way!), please complete that before starting this part.

Save versions of your files. You will not want to have to go back and debug this. **Don't debug!** If it goes wrong, just throw version `n` away and start back at `n-1`.

You should be able to test your program between each step. Sometimes you can even test your program as you work through the pieces of a step.

If you haven't done so, you might consider reading the ParentheC paper, "Using ParentheC to Transform Scheme Programs to C or How to Write Interesting Recursive Programs in a Spartan Host". It is slightly out of date viz. registerization, but can still prove a useful resource.

Download from Piazza the three files: `parenthec.ss`, `pc2c.ss`, and `pc2j.ss`.

You will also need to use the following `define-union` for expressions and the `main` program:

```
(define-union exprn
  (const cexp)
  (var n)
  (if test conseq alt)
  (mult nexp1 nexp2)
  (sub1 nexp)
  (zero nexp)
  (letcc body)
  (throw kexp vexp)
  (let exp body)
  (lambda body)
  (app rator rand))
```

```
;; (let ((f (lambda (f)
;;            (lambda (n)
;;              (if (zero? n)
;;                  1
;;                  (* n ((f f) (sub1 n))))))))
;;   (* (letcc k ((f f) (throw k ((f f) 4)))) 5))

(define main
  (lambda ()
    (value-of-cps
     (exprn_let
      (exprn_lambda
       (exprn_lambda
        (exprn_if
         (exprn_zero (exprn_var 0))
         (exprn_const 1)
         (exprn_mult (exprn_var 0)
                     (exprn_app (exprn_app (exprn_var 1) (exprn_var 1))
                                (exprn_sub1 (exprn_var 0)))))))
      (exprn_mult
       (exprn_letcc
        (exprn_app
         (exprn_app (exprn_var 1) (exprn_var 1))
         (exprn_throw (exprn_var 0)
                      (exprn_app (exprn_app (exprn_var 1) (exprn_var 1))
                                 (exprn_const 4)))))
       (exprn_const 5)))
     (empty-env)
     (empty-k))))
```

Notice that this test program is **not** quoted data.

3 Part 3

Your assignment is to complete the transformation of your interpreter from part 2 to a version we can translate to **C**. When your interpreter is complete, you can turn it into a C program using **pc2c**, and it will run the test program provided. Here are the steps you will need to accomplish. Once again, **save a new copy of your interpreter after you finish every step**. You will often need to go back to an older version of your interpreter and restart; having copies of all of them will save a **lot** of time.

1. At the top line of your file, add:

   ```
   (load "parenthec.ss")
   ```

2. Next add the `exprn define-union` to your file, change the **pmatch**-expression in **value-of-cps** to instead be a **union-case**-expression. Consult the "ParentheC" paper or the example from class to

see how to do this. Make sure to remove the commas in the patterns of what was your **pmatch** expression. Add **main** to the bottom of your file, and make sure it returns **120** when you invoke it.

3. Transform your closure constructor to a `define-union` (I named mine `clos`), change the `pmatch` in `apply-closure` to instead use `union-case`, and ensure that your constructor invocations are preceded with `clos_`, or something other than `clos` if you use a different name for your union. Make sure to remove the and commas in the patterns in what was your `pmatch` expression.

4. Transform your environment constructors to a `define-union` (I named mine `envr`), change the `pmatch` in `apply-env-cps` to instead use `union-case`, and ensure all constructor invocations are preceded with `envr_`, or something other than `envr` if you use a different name for your union. Make sure to remove the and commas in the patterns in what was your `pmatch` expression.

5. Transform your continuation constructors to a `define-union` (I named mine `kt`), change the `pmatch` in `apply-k` to instead use `union-case`, and ensure all constructor invocations are preceded with `kt_`, or something other than `kt` if you use a different name for your union. Make sure to remove the commas in the patterns in what was your `pmatch` expression.

6. Transform all your serious function calls to our "A-normal form" style, by adding `let*` above your serious calls, and through sequencing the sub-expressions in your serious calls, ensuring that the names of the actual parameters **to** the serious calls are **exactly** the names of the formal parameters in their definitions.

7. Registerize the interpreter. Turn each `let*` expression to a `begin` block: the former `let*` bindings will become `set!` expressions, and the body of that `let*`, what was your serious call, now becomes the invocation of a function of no arguments. Change all serious functions to be functions of no arguments. Define your global registers using `define-registers` at the top of the program.

8. Change all of your `(define <name> (lambda () ...))` statements to instead use `define-label`. Define your program counter at the top of the program using `define-program-counter`.

9. Convert all label invocations (that is, the invocations at the end of your `begin` blocks) into assignments to the program counter, and then add calls to `mount-trampoline` and `dismount-trampoline`. Note this will require modifying `empty-k` in your `kt` union, and the `empty-k` clause in the union-case inside apply-k. On the last line of `main`, print the register containing the final value of the program, e.g. `(printf "Fact 5:  ~s\n" v)`. See the parentheC document for notes on these steps.

10. That's the end! You're done. (`zip <username>-p3.zip main.ss parenthec.ss`) and submit it to the PLC grading server.

4 But we aren't at C yet (or Java)!

I promised you C code. So here's how to get it from here.

1. Comment out the lines `(load "parenthec.ss")` and if you added it to your file, comment out your invocation of `main` (that is, comment out `(main)`). Save a copy of this file as `interp.pc`.

2. In a clean, fresh REPL, open and run `pc2c.ss`. This should load without errors. In the associated Scheme REPL with no other files loaded, type `(pc2c "interp.pc" "a9.c" "a9.h")`. This which will generate C code from your interpreter.

3. Compile the C program with a C compiler of your choice. You can find here `gcc` binaries for many different systems at `http://gcc.gnu.org/install/binaries.html`. Alternately, you could use an online C compiler such as the following: `http://tutorialspoint.com/compile_c_online.php` Run the resulting executable, verifying that you see the correct output.

4. If you are more of a Java person, you might want to use the `pc2j.ss` file to instead generate Java code. In a fresh REPL

   ```
   > (load "pc2j.scm")
   > (pc2j "interp.pc")
   ```

   You can then run these as usual, or if you have `javac` and `java` in your `PATH`:

   ```
   > (compile/run "interp.pc")
   ```