C311/B521/A596 Programming Languages

Assignment 9: ParentheC Interpreter

"Code should run as fast as necessary, but no faster; something important is always traded away to increase speed."

—*Richard Pattis*

Introduction

Your task is to transform the interpreter below into a language that can be automatically converted to C. We will be using ParentheC for this task, so you'll be CPSing and registerizing the interpreter below before making final modifications to make it ready for ParentheC.

You cannot receive a grade for this course until you complete this assignment and demonstrate your understanding with one of the instructors. See the end of this page for more details.

Preliminaries

- If you haven't done so, you should probably read the ParentheC paper, *Using ParentheC to Transform Scheme Programs to C or How to Write Interesting Recursive Programs in a Spartan Host (Program Counter)*. It is slightly out of date viz. registerization, but can still prove a useful resource.
- Download parenthec.rkt and pc2c.rkt

Language

The language we will be using for the interpreter in the assignment is characterized by the grammar here:

```
          <expr>   ::=  (exp_const <integer>)
                    |   (exp_var <integer>)
                    |   (exp_if <expr> <expr> <expr>)
                    |   (exp_mult <expr> <expr>)
                    |   (exp_sub1 <expr>)
                    |   (exp_zero <expr>)
                    |   (exp_capture <expr>)
                    |   (exp_return <expr> <expr>)
                    |   (exp_let <expr> <expr>)
                    |   (exp_lambda <expr>)
                    |   (exp_app <expr> <expr>)
```

This is quite simple, and identical to the other languages that we have written interpreters for. However, we now explicitly tag all language constructs. When we refer to a variable (using `exp_var`) we refer to the variable's lexical address.

We provide a parse.rkt file, which contains a function `parse` that will convert normal Scheme-like syntax into the language we are using on this assignment. For example,

```
> (require "parse.rkt")
> (parse '((lambda (a b c) (* a c)) 5 6 7))

          (exp_app
            (exp_app
              (exp_app
                (exp_lambda
                  (exp_lambda
                    (exp_lambda (exp_mult (exp_var 2) (exp_var 0)))))
                (exp_const 5))
              (exp_const 6))
            (exp_const 7))
```

Use this to generate tests for your program. Notice that the output is curried for you. Note also that our parser produces quoted output, but your `value-of` will not take quoted expressions as input.

Here is a fully-functioning interpreter that implements this language for us. It uses the ParentheC *union* type to represent the language.

```
#lang racket
(require "parenthec.rkt")

(define-union exp
  (const n)
```

```
   (var v)
   (if test conseq alt)
   (mult rand1 rand2)
   (sub1 rand)
   (zero rand)
   (capture body)
   (return vexp kexp)
   (let vexp body)
   (lambda body)
   (app rator rand))

(define value-of
   (lambda (expr env)
     (union-case expr exp
       [(const n) n]
       [(var v) (apply-env env v)]
       [(if test conseq alt)
        (if (value-of test env)
     (value-of conseq env)
     (value-of alt env))]
       [(mult rand1 rand2) (* (value-of rand1 env) (value-of rand2 env))]
       [(sub1 rand) (- (value-of rand env) 1)]
       [(zero rand) (zero? (value-of rand env))]
       [(capture body)
        (call/cc
  (lambda (k)
    (value-of body (envr_extend k env))))]
       [(return vexp kexp)
        ((value-of kexp env) (value-of vexp env))]
       [(let vexp body)
        (let ((v (value-of vexp env)))
          (value-of body (envr_extend v env)))]
       [(lambda body) (clos_closure body env)]
       [(app rator rand)
        (apply-closure (value-of rator env) (value-of rand env))])))

(define-union envr
   (empty)
   (extend arg env))

(define apply-env
   (lambda (env num)
     (union-case env envr
       [(empty) (error 'env "unbound variable")]
       [(extend arg env)
        (if (zero? num)
     arg
     (apply-env env (sub1 num)))])))

(define-union clos
   (closure code env))

(define apply-closure
   (lambda (c a)
     (union-case c clos
       [(closure code env)
        (value-of code (envr_extend a env))])))

                                        ; Basic test...should be 5.
(pretty-print
  (value-of (exp_app
              (exp_app
               (exp_lambda (exp_lambda (exp_var 1)))
               (exp_const 5))
              (exp_const 6))
             (envr_empty)))

     ; Factorial of 5...should be 120.
(pretty-print
  (value-of (exp_app
       (exp_lambda
        (exp_app
         (exp_app (exp_var 0) (exp_var 0))
         (exp_const 5)))
       (exp_lambda
        (exp_lambda
         (exp_if (exp_zero (exp_var 0))
          (exp_const 1)
          (exp_mult (exp_var 0)
     (exp_app
      (exp_app (exp_var 1) (exp_var 1))
      (exp_sub1 (exp_var 0))))))))
     (envr_empty)))

      ; Test of capture and return...should evaluate to 24.
 (pretty-print
  (value-of
   (exp_mult (exp_const 2)
      (exp_capture
       (exp_mult (exp_const 5)
          (exp_return (exp_mult (exp_const 2) (exp_const 6))
                                     (exp_var 0)))))
   (envr_empty)))

;; (let ([fact (lambda (f)
;;               (lambda (n)
;;                 (if (zero? n)
;;                     1
;;                     (* n ((f f) (sub1 n))))))])
;;   ((fact fact) 5))

(pretty-print
 (value-of (exp_let
      (exp_lambda
       (exp_lambda
        (exp_if
         (exp_zero (exp_var 0))
         (exp_const 1)
         (exp_mult
   (exp_var 0)
   (exp_app
    (exp_app (exp_var 1) (exp_var 1))
    (exp_sub1 (exp_var 0)))))))
      (exp_app (exp_app (exp_var 0) (exp_var 0)) (exp_const 5)))
     (envr_empty)))
```

Note that the programs we pass to `value-of` do not have a quote in front of them. Why is this? Because the programs are no longer lists. We are actually passing a series of nested unions created by calling these tagged constructors.

Note: ParentheC does not support the `pretty-print` command, so you will need to `printf` akin to `(printf "~s\n" *v)` in your main function to display the results of the execution. Notice we use `\n` is used as the newline character.

Assignment

Your assignment is to turn this interpreter into C programs using ParentheC, and run the `factorial` and `capture` test programs. Here are the steps you will need to accomplish. Save a new copy of your interpreter after you finish every step. **We will expect you to have all of these intermediate files available during your demonstration**. Also, you will often need to go back to an older version of your interpreter and having copies of all of them will save a lot of time.

1. CPS this interpreter. It will be easier if you treat `apply-env` and `apply-closure` as serious, `closure` and `extend` as simple.
2. Make the interpreter representation-independent with respect to continuations (using functional reps).
3. Make your continuations be data structures (tagged lists).
4. Make your continuations be ParentheC unions (remove the old ds constructors), and change uses of pmatch to uses of union-case.
5. Write your interpreter using our A-normal form style, by adding `let*` above your serious calls.
6. Registerize the interpreter.
7. Add a function `main` (of no arguments, whose body is a single `(begin …)`) block (no nested begins) that does whatever computations your program needs to accomplish.
8. Change all of your `(define name (lambda () …))` statements to use `define-label`.
9. Define your registers and program counter at the top of the program.
10. Convert non-simple tail calls to assignments to the program counter, and then add calls to `mount-trampoline` and `dismount-trampoline`.
11. Verify that everything works in Scheme.
12. In a **fresh** Scheme process with **no** other files loaded, load `pc2c.rkt`, generate C code for your interpreter, compile the C program, and run the resulting executable, verifying that you see the correct output. (If you don't know how to do this step, check with the AIs for help.)

Save a new copy of your interpreter after you finish every step. **We will expect you to have all of these intermediate files available during your demonstration**. Also, you will often need to go back to an older version of your interpreter and having copies of all of them will save a lot of time.

- You should turn in a file named `interp.pc` that contains the exact Scheme code you used to generate your C programs.
- This assignment is due on **Wednesday, November 5th at 11:59 p.m.** Once you've done the assignment, you must meet with one of the AIs to demonstrate your knowledge of your code. This meeting is a required part of the assignment and must take place on or before **Friday, November 14th**.

- You must book an appt in advance (at least 1 day in advance). Make sure you get an email confirming your appointment.

Book Now!
Powered By: schedulething

- Remember: successful completion of this assignment and code review is required in order to receive a grade for this course. For assignments handed in after the due date, credit is reduced proportionally to lateness.

Just Dessert

Add a `callcc` form to your interpreter that behaves like Scheme's `call/cc`. Include at least one test program, as well as any necessary changes to the parser or other files in an email to your instructor.

assignment-9.txt · Last modified: 2014/10/24 12:38 by ihemann