CSE 130: Programming Languages – Principles and Paradigms Fall 2000

Problem Set 2 - Solutions

Instructor: Daniele Micciancio Oct. 23, 2000

Problem 1 (10 points)

a. The following expression can be used as the **xor** operator given the codification of **true** and **false**:

xor= $\lambda xy.(x\ (y\ \textbf{false}\ \textbf{true})\ y)$

b.
- **zero** = (**pair true true**) = $\lambda z.(z\ \lambda xy.x\ \lambda xy.x)$
- **iszero** = $\lambda n.(\textbf{first}\ n)$
- **inc** = $\lambda n.(\textbf{pair false}\ n)$
- **dec** = $\lambda n.(\textbf{iszero}\ n)\ n\ (\textbf{second}\ n)$

c. Define

$\mathbf{T} = \lambda\mathbf{f}.\lambda\mathbf{m}.\lambda\mathbf{n}.(\textbf{iszero m})\ \textbf{n}\ (\textbf{f}\ (\textbf{dec m})\ (\textbf{inc n}))$

Essentially this is the following recursive definitions for addition:

$$m + n = \begin{cases} n & \texttt{if}\ \ m = 0 \\ dec(m) + inc(n) & \texttt{if}\ \ 0 < m \end{cases}$$

The function we are looking for is obtained by applying the fix point operator to the functional **T**. Recall that the fixpoint operator $\mathbf{\Theta}$ was defined as $\mathbf{\Theta} = (\mathbf{B}\ \mathbf{B})$ with $\mathbf{B} = \lambda xy.(y\ (x\ x\ y))$. The function we are looking for is then $(\mathbf{\Theta}\ \mathbf{T})$

Problem 2 (10 points)

a. Function computing the approximate derivative of function f:

```
(define (deriv f d)
       (lambda (x)
              (/ ( - (f (+ x (/ d 2)))
                 (f (- x (/ d 2)))) d)
       )
)
```

For computing the integral, observe that using the approximation formula and fixing k we have:

$$\int_a^b f(x)dx \approx f(a + \frac{1}{2}\frac{b-a}{k}) \cdot \frac{b-a}{k} + \int_{a+\frac{b-a}{k}}^b f(x)$$

We obtain the following program:

```
(define (int f a b k)
     ( if (= k 0 ) 0
          ( + ( * ( f ( + a ( / (- b a) (* 2 k)))) (/ (- b a) k))
              (int f (/ (+ b ( * (- k 1) a)) k) b (- k 1)))))
```

To test the programs one can use the following:

```
(define(f x) (* x x))

(define (g x) ((deriv f 0.01) x))

(define (h x) (int f 0 x 1000))

(define y (- (int g 0 100 1000) (- (f 100) (f 0))))

(define x (- ((deriv h 0.01) 3) (f 3)))
```

Running the program gives us the values for `x` and `y`:

```
1 ]=> (load "dif.scm")

;Loading "dif.scm" -- done
;Value: x

1 ]=> x

;Value: 6.083342348972565e-6

1 ]=> y

;Value: -3.954482963308692e-9
```

Altough not exactly zero because of the approximations done in numerical computation, the difference between the two values is very small in both cases.

b. Program for *sums*

```
(define (sums list)
  (if (null? list) '()
      (cons (car list)
            (map (lambda (x) (+ x (car list)))
                 (sums (cdr list))))))
```

Program for *smus*

```
(define (smus list)
  (cond ((null? list) '())
        ((null? (cdr list)) list)
        (else (let ((s (smus (cdr list))))
                (cons (+ (car list) (car s)) s)))))
```

If you run (sums (numbers 1000)) and (smus (numbers 1000)) you will notice that while smus terminates in a fraction of a second, running sums takes several seconds. A possible way to define a faster function to compute sums is the following:

```
(define (sums list) (reverse (smus (reverse list))))
```

Using this implementation, the running time of (sums (numbers 1000)) is much smaller, just a little bit more than (smus (numbers 1000)).

A fast version of sums can also be defined directly, e.g., by the program:

```
(define (sums list)
  (if (null? list) '()
      (cons (car list)
            (sums (cons (+ (car list) (cadr list))
                        (cddr list))))))
```

If you run this program you will notice that the running times of sums and smus are roughly the same (a fraction of a second in both cases).

Problem 3 (10 points)

a. One can think of the difference between the static and dynamic scoping as follows: in static scoping, the value of unbound variables in a function is given by their values in the environment where the function is defined, whereas under dynamic scoping, the value of the same variables is given by their value in the environment where the function is called.

   If executed under static scoping the program returns 3. The reason is that after `p2` is bound to `(lambda (y) x)`, the returned value of `p2`, will be 1, (which is the value of `x` in the environment where `p2` is defined.) This can be tested by simply running the program (remember that Scheme uses static scoping).

   Under dynamic scoping the value returned is 4, since now the value returned by `p2` is 2. (In the environment where `p2` is called, `x` is 2.)

b. Under static scoping the program returns 5. When printing the value of `x` when `sub1` is called, `x` is bound to the variable `x` of the program `main`, which is assigned value 5. If dynamic Under dynamic scoping rules, when `sub2` is called, the variable `x` will be bound to the `x` defined in the `sub2` procedure, therefore the value printed will be 10.

c. The idea to transfom a call-by-name program into an equivalent call-by-value program is to use function abstractions to delay the evaluation of the parameters. We first consider a simple example, and then describe the general transformation. Consider the non-terminating program

```
1 ]=> (define A (lambda (x) (x x)))
;Value: A

1 ]=> (A A)
....
```

   Even if we pass (A A) as an argument to a program that does not use it the program will loop:

```
1 ]=> (define (one x) 1)
;Value: one

1 ]=> (one 1)
;Value: 1

1 ]=> (one (A A))
....
```

However, if we pass to "one" a function that evaluates to (A A), the execution terminates:

```
1 ]=> (one (lambda () (A A)))
;Value: 1
```

[Notice: scheme allows the definition of functions with no arguments (lambda () (A A)).] The reason this time the program terminates is that for scheme (lambda () (A A)) is just a function abstraction, and the function is not executed until you apply it to a (possibly empty) list of parameters.

So, in order to get a call by name semantics we can substitute all actual parameters S in function calls by corresponding abstractions (lambda () S) and all references to formal parameters x in function definitions with corresponding functions invocations (x). Of course the translation function should be recursively applied to all subexpressions.

The complete program is the following:

```
1 ]=> (define (Trans S)
        (cond ((symbol? S) (list S))
              ((number? S) S)
              ((eq? (car S) 'lambda)
               (list 'lambda (cadr S) (Trans (caddr S))))
              (else
               (cons (Trans (car S))
                     (map (lambda (x) (list 'lambda '() (Trans x)))
                          (cdr S))))))
; Value: trans
```

Now consider the program S1 and its translation from call-by-name to call-by-value:

```
1 ]=> (define S1 '((lambda (x) 1)
                   ((lambda (x) (x x)) (lambda (x) (x x)))))
;Value: s1

1 ]=> (define S2 (Trans S1))
;Value: s2

1 ]=> S2
;Value: ((lambda (x) 1)
         (lambda () ((lambda (x) ((x) (lambda () (x))))
                     (lambda () (lambda (x)
                                  ((x) (lambda () (x)))))))))
```

The argument passed to (lambda (x) 1) in S1 defines an infinite computation. Therefore, S1 does not terminate if executed call by value, however it terminates in call by name because the formal parameter x is never used:

```
1 ]=> (eval S2 '())
;Value: 1

1 ]=> (eval S1 '())
...
```