CSE 130: Programming Languages – Principles and Paradigms Fall 2000

Problem Set 2

Instructor: Daniele Micciancio Oct. 11, 2000

Due: October 23, beginning of class. No late submissions accepted.
Guidelines for the programming assignments: This time, some of the problems (2.a, 2.b and 3.c) have a programming component. When solving these problems you should use only the subset of scheme that has been described in class. In particular, you should not use the "imperative" features of scheme. For all programs you are required to type them and run them using the scheme interpreter. Together with your solutions you should submit a print out of the programs, and the result of executing them in scheme. You should also include a brief english description of how your programs work. Clarity of your programs (as well as the english description) is as important as correctness. In particular, in order to get full credit your programs should be properly formatted so to make them easy to read. No electronic copy of your programs is required.
Collaboration policy: As usual, you should do your homeworks individually: this is not a team project. See collaboration policy on the web page for details.

Problem 1 (λ-calculus, 10 points)

(a) In class we studied how the boolean values can be represented as λ-terms as follows:

$$\mathbf{true} = (\lambda xy.x) \quad \text{and} \quad \mathbf{false} = (\lambda xy.y)$$

and showed how to define the boolean functions:

$$\mathbf{not} = (\lambda z.(z\ \mathbf{false}\ \mathbf{true})), \quad \mathbf{and} = (\lambda z_1 z_2.(z_1\ z_2\ \mathbf{false})), \quad \mathbf{or} = (\lambda z_1 z_2.(z_1\ \mathbf{true}\ z_2)).$$

Give a λ-term (in the style of the "not", "and" and "or" functions above) for the exclusive-or operation defined by

$$\begin{aligned}\mathbf{xor}(\mathbf{true},\mathbf{true}) &= \mathbf{false}\\ \mathbf{xor}(\mathbf{false},\mathbf{false}) &= \mathbf{false}\\ \mathbf{xor}(\mathbf{true},\mathbf{false}) &= \mathbf{true}\\ \mathbf{xor}(\mathbf{false},\mathbf{true}) &= \mathbf{true}\end{aligned}$$

(b) In class we showed also how to represent pairs $[x, y] \equiv (\lambda z.(z\ x\ y))$ as functions that return the first element of the pair on input **true**, and the second element on input **false**. We also defined constructor and selectors:

$$\mathbf{pair} = (\lambda xy.(\lambda z.(z\ x\ y))) \quad \mathbf{first} = (\lambda x.(x\ \mathbf{true})) \quad \mathbf{second} = (\lambda x.(x\ \mathbf{false})).$$

with the property that $(\mathbf{first}\ (\mathbf{pair}\ x\ y)) = x$ and $(\mathbf{second}\ (\mathbf{pair}\ x\ y)) = y$

Pairs can be used to build more complicate data structures, e.g. tuples can be represented applying the pair constructor repeatedly

$$[x_1, x_2, \ldots, x_n] \equiv (\mathbf{pair}\ x_1\ (\mathbf{pair}\ x_2\ (\mathbf{pair}\ x_3\ \ldots(\mathbf{pair}\ x_{n-1}\ x_n)))).$$

In this problem we study how to represent numerals using λ-terms. A non-negative integer $n \geq 0$ can be represented by the $(n+2)$-tuple

$$\begin{aligned} n &\equiv [\underbrace{\mathbf{false}, \mathbf{false}, \ldots, \mathbf{false}}_{n}, \mathbf{true}, \mathbf{true}] \\ &\equiv (\underbrace{\mathbf{pair\ false}\ (\mathbf{pair\ false}\ \ldots (\mathbf{pair\ false}}_{n}\ (\mathbf{pair\ true\ true})))) \end{aligned}$$

Using this representation of the numbers give λ-temrs for the following constants and operations:

- Zero constant "**zero**": the λ-term representing the numeral 0.
- Zero predicate "**iszero**": on input the representation of a numeral n, should return **true** if $n = 0$ and **false** is $n > 0$.
- Increment function "**inc**": on input a representation of an integer n, should return the representation of integer $n + 1$.
- Decrement function "**dec**": on input a representation of an integer $n > 0$, should return the representation of integer $n - 1$. If the input is $n = 0$ the result should also be 0.

(c) Using the representation of the numerals and the functions given in part b., define an addition function "**add**" that on input the representations of two numerals n, m, returns the representation of $n + m$. [Notice: you will need the fix-point operator we will describe next week to do part (c). In the meantime you can work on problem 2.]

Problem 2 (Functional programming in SCHEME, 10 points)

(a) In this part you are asked to define two higher order functions in scheme to perform numerical derivation and integration of univariate functions.

Define a *deriv* function in scheme that on input a univariate function real function $f : R \rightarrow R$ and a real parameter δ, returns an approximation of $\frac{df(x)}{dx}$ computed according to the rule

$$\frac{df(x)}{dx} \approx \frac{f(x + (\delta/2)) - f(x - (\delta/2))}{\delta}$$

where the smaller is δ the better the approximation.

Then, define another scheme function *int* that on input a function f, a lower bound a, an upper bound b and an integer parameter k, computes an approximate value for the integral of f in the interval $[a, b]$ using the approximation formula:

$$\int_a^b f(x)\mathrm{dx} \approx \sum_{i=1}^{k} f\left(a + \left(i - \frac{1}{2}\right)\left(\frac{b-a}{k}\right)\right) \cdot \left(\frac{b-a}{k}\right)$$

where the larger is k the better the approximation.

Finally, test your dunctions doing the following:

- Pick some function f. (e.g., low degree polynomial) and three integers a, b, c.
- Define functions $g(x) = df/dx$ and $h(x) = \int_0^x f(x)dx$ using your scheme programs on input f.
- Check that f, g, h satisfy the familiar properties

$$\int_a^b g(x)dx = f(b) - f(a) \quad \text{and} \quad (dh/dx)(c) = f(c)$$

Notice: most probably the equations will be satisfied only approximately because the numerical computations of integral and derivative only give approximate results.

b. Define a recursive function *sums* in scheme that on input a list of numbers $(x_1 \dots x_n)$ outputs the list $(y_1 \dots y_n)$ of partial sums $y_i \sum_{j=1}^{i} x_i$. [Notice: your solution should be *reasonably* efficient (i.e., polynomial time). In particular, the definition of "sums" should not make more than one recursive call to itself.]

Now define another function *smus* that on input a list of numbers $(x_1 \dots x_n)$ outputs the list $(z_1 \dots z_n)$ of trailing sums $y_i = \sum_{j=i}^{n} x_i$. [Notice: you should define the function "smus" from scratch, without using "sums" as a subroutine. As in "sums", only one recursive call is allowed.]

Compare the running time of the two functions executing them on the list (1000 999 998 ... 1) of the first 1000 integers in decreasing order. [Notice: in your solutions, include only the result of running the functions on shorter lists, say (10 ... 1).] You can build the input list using the function

(define (numbers n) (if (zero? n) '() (cons n (numbers (- n 1)))))

[Notice: this is different from the function numbers defined in class which outputs the numbers in increasing order.] What is the running time of the two functions? Which function runs faster? Explain. If one of the two functions (say "smus") runs faster then the other one (say "sums"), show how to give an alternative implementation of "sums" combining "smus" and the scheme library function "reverse". What is the running time of the new function on input (nubers 1000)?

[Note: if your computer is particularly slow, or particularly fast, you can use a shorter or longer list of numbers in your experiments.]

Problem 3 (Scoping and Parameter passing, 10 points)

(a) Consider the following "scheme" program:

```
(let ((x 1))
   ((lambda (p1 p2)
       (let ((x 2)) (p1  x (p2 x))))
    (lambda (x y) (+ x y))
    (lambda (y) x)))
```

What is the result of the program if static scoping is used? What if dynamic scoping is used? Give a brief explaination in both cases.

(b) This is the same as problem 9 in ch.4 from the textbook. Consider the following program:

```
program main;
  var x : integer;
  procedure sub1;
    begin writeln('x=',x) end;
  procedure sub2;
    var x: integer;
    begin x:=10; sub1 end;
  begin {main}
  x:=5;
  sub2
  end. {main}
```

What value of x is printed if the progran is executed under static scoping? And under dynamic scoping? Briefly explain your answers.

(c) In class we have studied how recursion can be implemented in the λ-calculus using the fix-point operator:

$$\Theta = (\lambda xy.y\ (x\ x\ y))(\lambda xy.y\ (x\ x\ y)).$$

Unfortunately, for this implementation of recursion to work, programs should be executed using a call-by-name parameter passing method, while scheme implements call-by-value.

In this problem you are asked to write a translation function *trans* (in scheme) that compiles call-by-name λ-calculus into call-by-value λ-calculus. Both the input and the output should use scheme syntax as generated by the grammar

S ::= (lambda (x) S) | (S S) | x | c

where x is any variable name, and c any numerical constant. In other words, for any expression S generated by the above grammar, (trans S) should return another expression S' such that the result of executing S' using call-by-value is the same as the result of executing S using call-by-name. [You can use scheme predicates "symbol?" and "number?" to check if an expression x represents a variable name or a numberical constant.]

Give a small expression S1 such that S1 terminates when executed call-by-name, but it does not terminate if executed call-by-value. Run (trans S1) to obtain another expression S2. Then run both S2 and S1 in scheme using the commands (eval S2 '()) and (eval S1 '()) to show that S2 terminates while S1 does not when executed in a call by name fashion.