PPL222

Assignment 5

Responsible Lecturer: Meni Adler
Responsible TA: Reem Al-aasam

Submission Date: 30/6

General Instructions

Submit your answers to the theoretical questions in a pdf file called ex5.pdf and your code for programming questions inside the provided ex5.rkt, ex5.pl files. ZIP those 3 files together into a file called id1_id2.zip.
Do not send assignment related questions by e-mail, use the forum instead. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.
Use the forum in a responsible manner so that the forum remains a useful resource for all students: do not ask questions that were already asked, limit your questions to clarification questions about the assignment, do not ask questions "is this correct". We will not answer questions in the forum on the last day of the submission.

Question 1 - CPS [30 points]

The function `pipe` is implemented as follows:

```
; Signature: compose(f g)
; Type: [T1 -> T2] * [T2 -> T3]  -> [T1->T3]
; Purpose: given two unary functions return their composition, in the
same order left to right
; test: ((compose sqrt -) 16) ==> -4
;       ((compose not not) true)==> true
(define compose
  (lambda (f g)
    (lambda (x)
       (g (f x)))))


; Signature: pipe(lst-fun)
; Type: [[T1 -> T2],[T2 -> T3]...[Tn-1 -> Tn]]  -> [T1->Tn]
; Purpose: Returns the composition of a given list of unary
functions. For (pipe (list f1 f2 ... fn)), returns the composition
fn(....(f1(x)))
; test: ((pipe (list sqrt - - number?)) 16)) ==> true
;       ((pipe (list sqrt - - number? not)) 16) ==> false
;       ((pipe (list sqrt add1 - )) 100) ==> -11
(define pipe
  (lambda (fs)
    (if (empty? (cdr fs))
        (car fs)
        (compose (car fs) (pipe (cdr fs))))))
```

a. Write a CPS procedure `pipe$`, which gets a list of unary CPS functions and a continuation function and returns their compositions.
(10 points) [ex5.rkt]

```
; Signature: pipe$(lst-fun,cont)
; Type: [
          [T1 * [T2->T3]] -> T3,
          [T3 * [T4 -> T5]] -> T5,
          ...,
          [T2n-1 * [T2n * T2n+1]]-> T2n+1
         ]
         *
         [[T1 * [T2n -> T2n+1]] -> T2n+1] ->
                [[T1 * [T2n+1 -> T2n+2]] -> T2n+2]
         -> [T1 * [T2n+3 -> T2n+4]] -> T2n+4
; Purpose: Returns the composition of a given list of unry CPS
functions.


(
 (pipe$ (list add1$ square$ div2$)
        id)
 3 id
)
⇒ 8

(
 (pipe$ (list square$ add1$ div2$)
        id)
 3 id
)
⇒ 5
```

```
(
 (pipe$ (list add1$ square$ div2$)
        (lambda (f$) (compose$ f$ add1$ id)))
 3 id
)
⇒ 9

(
  (pipe$ (list square$ add1$ div2$)
         id)
  3 (lambda (x) (* x 10))
)
⇒ 50

(
 (pipe$ (list square$ add1$ div2$)
        (lambda (f$) (compose$ f$ add1$ id)))
 3 (lambda (x) (* x 10))
)
⇒ 60

(
 (pipe$ (list g-0$ bool-num$)
        id)
 3 id
)
⇒ 1
```

b. Prove that `pipe$` is CPS-equivalent to `pipe` (10 points) [ex5.pdf]

c. Define the type and implement the CPS version of *`reduce`* for two cases:

   `reduce-prim$` gets a **primitive reducer**, an init value, a list, and a cont function and returns the reduced value accordingly (the items are reduced by their list order).

   `reduce-user$` gets a **cps user reducer**, an init value, a list, and a cont function and returns the reduced value accordingly (the items are reduced by their list order).

   (10 points) [ex5.rkt]

```
; Signature: reduce-prim$(reducer, init, lst, cont)
; Type: @TODO
```

```
      ; Purpose: Returns the reduced value of the given list, from
left
      ;          to right, with cont post-processing
      ; Pre-Condition: reducer is primitive

      (reduce-prim$ + 0 '(3 8 2 2) (lambda (x) x)) ⇒ 15
      (reduce-prim$ * 1 '(1 2 3 4 5) (lambda (x) x)) ⇒ 120
      (reduce-prim$ - 1 '(1 2 3 4 5) (lambda (x) x)) ⇒ -14


      ; Signature: reduce-user$(reducer, init, lst, cont)
      ; Type: @TODO
      ; Purpose: Returns the reduced value of the given list, from
left
      ;          to right, with cont post-processing
      ; Pre-Condition: reducer is a CPS user procedure

      (define plus$
       (lambda (x y cont)
          (cont (+ x y))))

     (define div$
       (lambda (x y cont)
         (cont (/ x y))))

      > (reduce-user$ plus$ 0 '(3 8 2 2) (lambda (x) x))
      15
      > (reduce-user$ div$ 100 '(5 4 1) (lambda (x) (* x 2)))
      10
```

Question 2 - Lazy lists [25 points]

a. Define an equivalence criterion for two lazy lists (when do we say that two lazy lists are equivalent)
(3 points) [ex5.pdf]

b. Show that the following fibs1 and fibs2 are lazy-lists equivalent according to your definition.
(7 points) [ex5.pdf]

```
; Signature: fibs1
; Type: [Void -> LzL(Number)]
(define fibs1
   (letrec ((fibgen
                        (lambda (a b)
                              (cons-lzl a
                                (lambda () (fibgen b (+ a b))))) ))
         (fibgen 0 1)))

; Signature: fibs2
; Type: [Void -> LzL(Number)]
(define fibs2
   (cons 0
         (lambda ()
           (cons 1
           (lambda () (lz-lst-add (tail fibs2) fibs2))))))

; Signature: lz-lst-add(lz1,lz2)
; Type: [LzL(Number)*LzL(Number) -> LzL(number)]
(define lz-lst-add
   (lambda (lz1 lz2)
      (cond ((empty-lzl? lz1) lz2)
            ((empty-lzl? lz2) lz1)
            (else (cons-lzl (+ (head lz1) (head lz2))
              (lambda () (lz-lst-add (tail lz1) (tail
lz2))))))))
```

c. Using lazy-lists you cannot see the whole list because sometimes it is infinite. Thus you have learned the `take` function, which receives a lazy list and n, a number of elements, as parameters, and returns a regular list of the first n lazy list elements. Expand the interface of the lazy lists with a new function, `take-while`, which still accepts two parameters, and the first parameter is still the lazy list. But compared to the previous one, the second parameter is a function. This is a predicate, which examines the terms to decide if it should continue. Each element in the lazy list is examined by the predicate. As long as it succeeds, additional elements are taken. Once an element is found that does not pass the predicate, the action stops, and the identified element is not returned. That is, the longest sequence of elements is required to match the condition of the predicate. For example, if we create a lazy list that calculates powers, we can get:

```
> (take (integers-from 0) 10)
'(0 1 2 3 4 5 6 7 8 9 10)
> (take-while (integers-from 0) (lambda (x) (< x 9)))
'(0 1 2 3 4 5 6 7 8)
> (take-while(integers-from 0) (lambda (x)  (= x 128)))
'()
```

c1. Implement `take-while`.

```
; Signature: take-while(lz-lst,pred)
; Type: [LzL<T>*[T -> boolean] -> List<T>]
; Purpose: while the pred holds return the list elements
```
(5 points) **[ex5.rkt]**

c2. Implement `take-while-lzl` which has the same semantics of `take-while` but returns a lazy-list rather than a list.

```
(take (take-while-lzl (integers-from 0) (lambda (x) (< x 9))) 10)  ⇒
'(0 1 2 3 4 5 6 7 8)
(take-while-lzl(integers-from 0) (lambda (x)  (= x 128)))) ⇒ '()

; Signature: take-while-lzl(lz-lst,pred)
; Type: [LzL<T>*[T -> boolean] -> LzL<T>]
; Purpose: while the pred holds return the list elements as a lazy
list
```

(10 points) [ex5.rkt]

d. Implement the *reduce-lzl* procedure (in ex5.rkt), which gets a binary function, an init value, and a lazy list, and returns the reduced value of the given list (where the items are reduced according to their list order).
(5 points) [ex5.rkt]

```
; Signature: reduce-lzl(reducer, init, lzl)
; Type: [T2*T1 -> T2] * T2 * LzL<T1> -> T2
; Purpose: Returns the reduced value of the given lazy list


> (reduce-lzl + 0
    (cons-lzl 3 (lambda () (cons-lzl 8 (lambda () '()))))))
11

> (reduce-lzl / 6
    (cons-lzl 3 (lambda () (cons-lzl 2 (lambda () '()))))))
1   ;;; [6 / 3 / 2]
```

Question 3 - Logic programing [45 points]

3.1 Unification (10 points) [ex5.pdf]

What is the result of the operations? Provide all the algorithm steps. Explain in case of failure.

```
a. unify[ p(v(v(d(M),M,ntuf3),X)), p(v(d(B),v(B,ntuf3),KtM))]
b. unify[n(d(D),D,d,k,n(N),K),n(d(d),D,d,k,n(N),d)]
```

3.2 Logic programming (15 points) [ex5.pl]

Write a procedure `unique(List, UniqueList, Dups)/3` that succeeds if and only if `UniqueList` contains the same elements of `List` without duplicates (according to their order in `List`), and `Dups` contains the duplicates.
For example:

```
?- unique([1,2,3,4,5],X,Y).
X = [1,2,3,4,5], Y =[].
?- unique([1,2,3,4,5,3,4,5],X,Y).
X = [1,2,3,4,5], Y=[3,4,5].
?- unique([1,2,3,4,5,3,4,5],[1,2,3,4,5],[3,4,5]).
true.
?- unique([1,2,3,4,5,3,4,5],[1,2,4,5,3],[3,4,5]).
false.
?- unique([1,2,3,4,5,3,4,5],[1,2,3,4,5],[4,3,4]).
false
```

[15 points]

Your program should be written according to the logic language presented in class, i.e., **you cannot use any additional Prolog procedure** (number arithmetics, list procedures like *append member*, *not*, *!*, etc.). You can use `\=`, since it is part of our logic programming language. In case you need such procedures, implement them or use the implementation given in class.

3.3 Proof tree (20 points) [ex5.pdf]

Unary numbers provide symbolic representation for the natural numbers. They are defined inductively as follows: zero is the atom [], and for a Unary number c, [1|c] represents the number c+1. The numbers 0, 1, 2, 3 etc. are represented by the terms [], [1], [1, 1], [1, 1, 1], and so forth. The following program is given.

```
% Signature: unary_number(L)/1
% Purpose: L is a unary number
unary_number([]).        %1
unary_number ([1|A]) :-  %2
    unary_number (A).
```

```
% Signature:  unary_plus(X,Y,Z)/3
% Purpose: X append Y = Z
unary_plus(X, [], X) :-           %1
   unary_number(X).
unary_plus (X, [A|Y], [A|Z]):-     %2
   unary_plus(X, Y, Z).
```

a. Draw the proof tree for the query below and the given program. For success leaves, calculate the substitution composition and report the answer at each success leaf.
   `?- unary_plus ([1|X], [1,1|Y], [1,1,1|X]).`
   (10 points)

b. Is this a success or a failure proof tree?
   (2 points)

c. Is this tree finite or infinite?
   (2 points)

d. Is this query provable from the given program?
   (3 points)

e. Let us define L9 as the *rational logic programming* language with the addition of one functor only - *cons*. Is L9 decidable?
   (3 points)