PPL Assignment 3

Q-1, c:

Invariant-set-extend(n, Empty-set) = make-set(list(n))

Prove: applicative-eval[(set-extend n empty-set)] ➔*

applicative-eval [set-extend]➔

<Cl (e s) (if (part-of-set? s e) s (make-set (cons e (set->list s))))>

Applicative-eval[n]➔ n

Applicative-eval[empty-set]➔ empty-set

➔

Applicative-eval[(if (part-of-set? Empty-set n) empty-set (make-set (cons n (set->list empty-set))))]

Aplicative-eval[if]➔special operator if

Aplicative-eval[(part-of-set? Empty-set n)➔* false

Aplicative-eval[(make-set (cons n (set->list empty-set)))]➔

Aplicative-eval[make-set]➔ <Cl (elements) (foldr + 0 (map (lambda(x)(expt 2 x)) (remove-duplicates elements)))>

Aplicative-eval➔[(cons n (set->list empty-set))]➔* '(n)

Aplicative-eval[(foldr + 0 (map (lambda(x)(expt 2 x)) (remove-duplicates '(n))))]

⇨ * 2^n

Aplicative-eval [make-set (list n)]➔

Aplicative-eval[make-set]➔ <Cl (elements) (foldr + 0 (map (lambda(x)(expt 2 x)) (remove-duplicates elements)))>

Aplicative-eval[list n] ➔* '(n)

Aplicative-eval[(foldr + 0 (map (lambda(x)(expt 2 x)) (remove-duplicates '(n))))]

➔* 2^n

*invariant number 1 does not reflects the correctness of the ADT.

Say we have as set that contains only the number 1,and we are trying to add the same number

To our set. The set-extend procedure will leave the set unchanged.

So when we will use set-remove-el procedure on that result, we will get an empty set

And not the set that contains only the number 1.

Q-1, d:

In our implementation to the set(Number) ADT ,we used binary encoding, and

Every number instance is in fact a valid encoding to a set.

So we can implement a set of sets by using a list of numbers in which each number

represents a valid set.

Implementation using the set(Number) ADT and its make-set constructor:

; Signature: make-set-of-set(list)

; Type: [List (List(Number)) -> Set(Set(Number))]

; Purpose: Construct a set of sets

; Pre-conditions: list is a set of sets

```
 (define make-set-of-sets
(lambda (list)
(map (lambda (sublist) (make-set sublist)) list)
))
```

Q-1, f:

Re-implemented procedures – make-set, set-select, set->list, set?

The same- set-equal? ,set-empty?, set-remove, set-remove-el, set-extend

Q-3, b:

Point out an advantage of using a lazy-list based implementation over the implementation provided above

An advantage of implementing Newton's method as a lazy list is if the iterations do not converge to the root. If this is the case, the regular implementation will get stuck in an infinite loop. On the other hand, with the lazy list implementation, the process will stop after a fixed number of iterations every time.

a. Q-4, a.2: Prove that filter1$ is CPS-equivalent to filter

Procedure filter1$ is CPS-equivalent to filter if for every x1,x2,...,xn and for every continution cont, the following is true:

(filter1$ x1 … xn cont) = (cont (filter x1 … xn))

Claim:

Procedure filter1$ is CPS-equivalent to filter, meaning that for every n and for every cont

(filter1$ pred seq c) = (c (filter pred seq))

Proof:

Since filter1$ is recursive the proof is done by induction on the length of seq n

Base: n = 0

a-e[(filter1$ pred seq c)] ==>* a-e[(c (empty))] = a-e[(c (empty))]

Assumption:

For each n N the claim is true for k n

Step:

Let n = k + 1

Then:

```
a-e[ (filter1$ pred seq c) ] ==>*
a-e[ (pred$ (car seq)
          (lambda (pred-res)
            (if pred-res (filter1$ pred$ (cdr seq)
                           (lambda (filtered_cdr)
                              (cont (cons (car seq) filtered_cdr))))
              (filter1$ pred$ (cdr seq)
                           cont))) ]
```

Here we will split in two, if pred-res is true

From the induction assumption we get that since (cdr) shortens the length of seq, we can substitute and we get

a-e[(filter1$ pred$ (cdr seq) c)] = a-e[(c (filter pred (cdr seq)))]

==>*

a-e[c ((cons (car seq) (filter pred (cdr seq))))]==>*a-e[(c (filter pred seq))]

And if pred-res is false

a-e[(filter1$ pred$ (cdr seq) c)] = a-e[(c (filter pred (cdr seq)))]

Q-5, a (**Theory**):

The notion of substitution and its operations is used both for the construction of operational semantics for Scheme, and for the construction of type inference procedures for Scheme expressions. Compare the two notions: Substitution (as in Chapter 2) and type-substitution (as in Chapter 5). Note 2 similarities and 2 differences.

Two similarities are that we are trying to simplify the expressions, so we take a complicated or abstract expression, and substitute it, with something else that is simpler to deal with. Another similarity is that if they fill we will not be able to compute anything with the given program.
A couple of differences are that substitution is needed for the actual computation, as opposed to type-substitution that is needed in order to make sure the computation will succeed. Another difference is that type-substitution is non deterministic, as opposed to substitution that is.

i. {f:[T1 -> [T2->T1]], x:T1} |- (f x):[T2->T1]

True

Instance: T1: number T2: number x: number

ii. {f:[T1 -> [T2->T1]], x:Number} |- (f x):[T2->T1]

False

T1: number

iii. {f:[T1 -> [Boolean->T1]], x:T1} |- (f x):[T2->T1]

False

T2: boolean

iv. {f:[T1 -> [T2->T1]]} |- (f x):[T2->T1]

False

x: T1

v. {f:[T1 -> T3], x:T1} |- (f x):[T2->T1]

False

T3: [T2->T1]

1.

i. {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):T1
{f:T3, x:T4} |- (f x):[T1 -> T2]

Renaming:
{f1:[T1 -> [T2 -> T1]], x1:T1} |- (f1 x1):T1
{f2:T3, x2:T4} |- (f2 x2):[T1 -> T2]

Unifiable:
If we try to unify we get that:
T3 = [T1 -> T2]

T4 = T1
f1 =f2
X1 = x2

Note: This is still an ilegal ts but we were told to try to unify anyways.

ii. {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):T1
{f:T3, x:T1} |- (f x):[T1 -> T2]

Renaming:
{f1:[T1 -> [T2 -> T1]], x1:T1} |- (f1 x1):T1
{f2:T3, x2:T1} |- (f2 x2):[T1 -> T2]

Unifiable:
T3 = [T1 -> T2] = T1
f1=f2
x1=x2

iii. {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):Number
{f:T3, x:Number} |- (f x):[T1 -> T2]

Not unifiable:
Upon reviewing the first statement:
(f x):Number
{f:[T1 -> [T2 -> T1]]}
Since Number is atomic, {[T2 -> T1] = Number}
Is ilegal

Q-5, b (**Type inference**):

i.

Derive a type for the following defined variable (the naïve "partial evaluation" of the map procedure):

```
(define c-map-proc (lambda (proc)
        (lambda (lst)
                (if (empty? lst) empty
                        (cons (proc (car lst))
                                ((c-map-proc proc) (cdr lst)))) )))
```

Renaming: Not needed

Initializing derived-ts-pool: derived-ts-pool = { }
Typing leaf expressions:

1. {lst:T1}|- lst:T1

2. {}|- cdr:[List-> List]
3. {proc:T2}|- proc:T2
4. {c-map-proc:T3}|- c-map-proc:T3
5. {c-map-proc:T3}|- c-map-proc:T3
6. {}|- car:[List -> S]
7. {}|- cons:[S*List-> List]
8. {}|- empty?:[List(S) -> Boolean]
9. Typing (cdr lst) – Applying the Application typing rule to statements 1,2, with type substitution {_S1=T1=List, _S=List}:
   {lst:List} |- (cdr lst):List

10. Typing (c-map-proc proc) – Applying the Application typing rule to statements 3,4, with type substitution {_S1=T3:[T4->T5], S=T2 = T5}:

    {proc:T2} |- (c-map-proc proc):T5

11. Typing ((c-map-proc proc) (cdr lst)) - Applying the Application typing rule to statements 9, 10 with type substitution {_S1=List, _S = T3:[List->T6]}:

    {c-map-proc:[T2->T5], (cdr lst):List} |-
    ((c-map-proc proc) (cdr lst)):T6

12. Typing (car lst) - Applying the Application typing rule to statements 6 with type substitution {_S1=List, _S = T7}

    {lst:List} |- (car lst):S7

13. Typing (proc (car lst))- Applying the Application typing rule to statements 3, 12 with type substitution {_S1=T7, _S =T2 }

    {lst:List} |- (proc (car lst)):T2

14. Typing (cons (proc (car lst))
    ((c-map-proc proc) (cdr lst))))- Applying the Application typing rule to statements 7, 11, 13 with type substitution
    {_S1=T2, _S2 =T6=List, _S = List}
    (cons (proc (car lst))
        ((c-map-proc proc) (cdr lst)))):List

15. Typing (empty? lst)- Applying the Application typing rule to statements 8 with type substitution {_S1=List, _S =boolean }
    {lst:List} |-(empty? lst):boolean

16. Typing **if** (if (empty? lst) empty
    (cons (proc (car lst))
    ((c-map-proc proc) (cdr lst))))))- Applying the Application typing rule to statements 13, 14, 15 with type substitution {_S1 = boolean, _S2 = List}
    {} |- (if (empty? lst) empty
    (cons (proc (car lst))
    ((c-map-proc proc) (cdr lst)))))):List

17. Typing **procedure** (lambda (lst)
    (if (empty? lst) empty
        (cons (proc (car lst))
            ((c-map-proc proc) (cdr lst)))))
    Applying the Procedure rule to statement 16, with type substitution {_S1=List, _U1=List}:
    {} |- (lambda (lst)
    (if (empty? lst) empty
        (cons (proc (car lst))
            ((c-map-proc proc) (cdr lst))))): List

18. Typing **procedure** (lambda (proc)

```
(lambda (lst)
  (if (empty? lst) empty
      (cons (proc (car lst))
            ((c-map-proc proc) (cdr lst)))))
```

Applying the Procedure rule to statement 16, with type substitution {_S1=[T7 -> T2], _U1=List}:

```
{} |-  (lambda (proc)
(lambda (lst)
  (if (empty? lst) empty
      (cons (proc (car lst))
            ((c-map-proc proc) (cdr lst))))): List
```

19. Typing **define** (define c-map-proc (lambda (proc)

```
(lambda (lst)
(if (empty? lst) empty
(cons (proc (car lst))
((c-map-proc proc) (cdr lst)))) )))
```

Applying the Define typing rule to statements 18 with type substitution {_S= List}

```
{} |- (define c-map-proc (lambda (proc)
(lambda (lst)
(if (empty? lst) empty
(cons (proc (car lst))
((c-map-proc proc) (cdr lst)))) ))): List
```

ii. Assume that the naively Curried previous version is incorrectly written as:

```
(define c-map-proc
    (lambda (proc)
        (lambda (lst)
            (if (empty? lst) empty
                (cons (proc (car lst))
                    (c-map-proc proc (cdr lst)))) )))
```

Show at which step your proof from previous entry fails. There is no need to repeat identical steps. Just write the step where the proof diversify until it ends.

The proof fails when typing (c-map-proc proc (cdr lst)).
C-map-proc is of type [T1->T2] but here we get [T1*T2 - > T3], and since it receives too many variables it is not well typed

iii. Assume that the naively Curried map from entry (b.1) is modified as:

```
(define c-map-proc
    (lambda (proc)
        (lambda (lst)
        (if (empty? lst) lst
            (cons (proc (car lst))
                (c-map-proc proc (cdr lst)))) )))
```

Explain and show how this change affects the results.
When the list is empty, instead of returning empty it returns the lst. This changes the type inference we use in that case. So instead of inferring that a list is returned, we only assume that a type T is returned. The inference that lst is a list will only happen later.

b.

Add a typing rule for let* expressions:

The difference her is that every new expression recognizes the previous previous expressions.

For every: type environment _Tenv,
variables _v1, _v2, ..., _vn, n>=0,
expressions _b1, _b2, ..., _bm, m>=1, and
type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv |- _e1:_S1,
...,
_Tenv |- _en:_Sn,
_Tenv1 ° {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b1:_U1,
_Tenv1_Tenv2 ° {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b2:_U2,
...,
_Tenv1_Tenv2..._Tenvn ° {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _bm:_Um
Then _Tenv |- (let ((_v1 _e1) (_v2 _e2) ... (_vn _en)) _b1 _b2 ... _bm) : _Um

ii. Derive the type of:

```
(let* ((x 1)
      (y (+ x 1)))
  (+ x y))
```

Applying typing rule let*:
{} |- (x 1) :N
{} |- (y(+ x 1)):N
{} |- (+ x y):N

iii. Show how the type derivation in entry (2) differ from the type derivation of:

```
(let ((x 1)
     (y (+ x 1)))
  (+ x y))
```

We try to assign the value (+ x 1) to y, but x isnt bound and is a free variable, therefore we have to assume that x is a number, and our Tenv is bigger.

Q-6, a:

Expression	Equation
(let*((V1 E1)….(Vn En)) b1….bm)	Tlet*=Tbm

Q-6, b:

Expression	Variable
(let* ((x 1) (y (+ x 1))) (+ x y))	T0
(+ x 1)	T1
+	T+
X	Tx
1	Tn1
+(x y)	T2
Y	Ty
3	Tn3

Construct type equations:

Expression	Equations:
(let* ((x 1) (y (+ x 1))) (+ x y))	T0=Tn3
(+ x 1)	T+ = [Tx *Tn1 ➔ T1]

+	T+ = [N *N → N]
X	Tx=Tn1
1	T1=N
(+ x y)	T+ = [Tx * Ty] →T2
Y	Ty
3	Tn3

Solve the equations:

Expression	Substitution
T0=T2	{ }
T+ = [N *N → N]	
T+ = [Tx * Ty -> T2]	
T+ = [Tx * Tn1 -> T1]	
Tx = Tn1	
Tn1=N	
T2=Tn3	
Ty=T1	
Tn3=N	

Expression	Substitution
T+ = [N *N → N]	{T0=T2 }

T+ = [Tx * Ty -> T2]	
T+ = [Tx * Tn1 ->T1]	
Tx = Tn1	
Ty=T1	
T2=Tn3	
Tn1=N	
Tn3=N	

Expression	Substitution
T+ = [N *N → N]	{T0=T2 , T+ = [Tx * Ty -> T2]}
T+ = [Tx * Tn1 ->T1]	
Tx = Tn1	
Ty=T1	
T2=Tn3	
Tn1=N	
Tn3=N	
Tn2=N	

Expression	Substitution
T+ = [N *N → N]	{T0=T2 , T+ = [Tx * Ty -> T2] , T+= [Tx * Tn1 ->T1]}
Tx = Tn1	

Ty=T1	
T2=Tn3	
Tn1=N	
Tn3=N	
Tn2=N	

Expression	Substitution
T+ = [N *N ➔ N]	{T0=T2 , T+ = [Tx * Ty -> T2] , T+= [Tx * Tn1 ->T1], Tx=Tn1}
Ty=T1	
T2=Tn3	
Tn1=N	
Tn3=N	
Tn2=N	

Expression	Substitution
T+ = [N *N ➔ N]	{T0=T2 , T+ = [Tx * Ty -> T2] , T+= [Tx * Tn1 ->T1], Tx=Tn1, Ty=T1}
T2=Tn3	
Tn1=N	

Tn3=N	

Expression	Substitution
T+ = [N *N ➔ N]	{T0=Tn3 , T+ = [Tn1 * Tn1 -> T1] , T+= [Tn1 * T1 ->Tn3], Tx=Tn1, Ty=T1,T2=Tn3}
Tn1=N	
Tn3=N	

Expression	Substitution
T+ = [N *N ➔ N]	{T0=Tn3 , T+ = [N * N -> T1] , T+= [Tx * T1 ->Tn3], Tx=N, Ty=T1, T2=Tn3, Tn1=N}
Tn3=N	

Expression	Substitution
T+ = [N *N ➔ N]	{T0=N , T+ = [N * N -> T1] , T+= [N * T1 ->N], Tx=N, Ty=T1, T2=N, Tn1=N, Tn3=N }

Expression	Substitution
	{T0=N , Tx=N ,Ty=N ,T2=N ,Tn1=N, Tn3=N, T1=N, T+ =[N * N➔N] }

Q-6, c:

Axiomatic type-inference

+

1) Renaming is optional
2) General Algorithm

-

1) Non-deterministic decisions in application of typing axioms ands ruld

2) Difficult implementation-involves management of multiple data types

Type-constraint type-inference

+

1) Deterministic Algorithm
2) The Algorithm makes sure the internal expressions are also well-typed

-

1) Has to use renaming

2) Derives from the first type-inference method