HW5 – PPL

Submitting:

Raz Monsonego: 313292120

Almog Zemach: 205789001

Question 1 – CPS

1.1.

** In the following proof, ‘a-e’ stands for ‘applicative-eval’ (as in the practical session).

b.

Claim: append$ is CPS – equivalent to append

i.e. :

$$(append\$\ lst1\ lst2\ cont) = (cont\ (append\ lst1\ lst2))$$

Proof by induction: (on length of the first list)

<u>**Base:**</u> $lst1 = '()$

$a - e[(append\$\ '()\ lst2\ cont)] \Rightarrow\ a - e[(cont\ lst2)] =$
$a - e[(cont\ (append\ '()\ lst2))]$

<u>**Assumption:**</u> $\forall i \in N\ \ s.t\ \ i \le n, |lst1| = i$:

$$a - e[(append\$\ lst1\ lst2\ cont)] = a - e[(cont\ (append\ lst1\ lst2))]$$

<u>**Step:**</u> $let\ n + 1 \in N\ \ s.t\ \ |lst1| = n + 1,\quad then$:

$a - e[(append\$\ lst1\ lst2\ cont)] \Rightarrow$

$a - e\left[\left(append\$\ (cdr\ lst1)\ lst2\ \left(lambda\ (res)\left(cont\ (cons\ (car\ lst1)\ res)\right)\right)\right)\right]$

Denote: $cont' = \left(lambda\ (res)\left(cont\ (cons\ (car\ lst1)\ res)\right)\right)$

Notice that $|(cdr\ lst1)| = n$ and therefore by our assumption,

$$a - e[(append\$\ (cdr\ lst1)\ lst2\ cont')] = a - e[\left(cont'\ (append\ (cdr\ lst1)\ lst2)\right)]$$

$\Rightarrow a - e\left[\left(cont\left(cons\ (car\ lst1)\ (append\ (cdr\ lst1)\ lst2)\right)\right)\right] = (*)$

$a - e\left[\left(cont\left(append\left(cons\ (car\ lst1)\ (cdr\ lst1)\right)\ lst2\right)\right)\right] =$

$a - e[\left(cont\ (append\ lst1\ lst2)\right)]$

Proof of $(*)$:

Claim: $\left(cons\ x\ (append\ lst1\ lst2)\right) = (append\ (cons\ x\ lst1)\ \ lst2)$

Proof by induction: (on length of first list)

<u>Base:</u> $lst1 = '()$

1. $a - e\left[\left(cons\ x\ (append\ '()\ lst2)\right)\right] \Rightarrow a - e[(cons\ x\ lst2)]$
2. $a - e[(append\ (cons\ x\ '())\ \ lst2)] = \ a - e[(append\ '(x)\ \ lst2)]$

$= a - e[(cons\ x\ lst2)]$

As we can see $1\ =\ 2$

<u>Assumption:</u> $\forall i \in N\ \ s.t\ \ i \le n, |lst1| = i$:

$a - e\left[\left(cons\ x\ (append\ lst1\ lst2)\right)\right] = a - e[(append\ (cons\ x\ lst1)\ \ lst2)]$

<u>Step:</u> $let\ n + 1 \in N\ \ s.t\ \ |lst1| = n + 1, \quad then$:

$a - e\left[\left(cons\ x\ (append\ lst1\ lst2)\right)\right] \Rightarrow$

$a - e\left[\left(cons\ x\ (append\ (cons\ (car\ lst1)\ (cdr\ lst1))\ lst2)\right)\right]$

Notice that $|(cdr\ lst1)| = n$ and therefore by our assumption,

$\Rightarrow a - e[(cons\ x\ (cons\ (car\ lst1)\ (append\ \ (cdr\ lst1)\ lst2)))]$

$\Rightarrow a - e[(cons\ (cons\ x\ '((car\ lst1))\ (append\ \ (cdr\ lst1)\ lst2)))]$

$\Rightarrow a - e[(append\ (cons\ \ \ (cons\ (cons\ x\ '(car\ lst1))\ (cdr\ lst1))\ lst2))]$

$\Rightarrow a - e[(append\ (cons\ x\ lst1)\ \ lst2)]$

Question 2 – Lazy lists

d. ***reduce1-lzl:***

We would like to use *reduce1-lzl* when we have a constructed lazy list with **finite size**, without a recursive construction that could cause an infinite loop.

For example:

$$(cons - lzl\ 3\ (\text{lambda}\ ()\ (cons - lzl\ 8\ (\text{lambda}\ ()\ `())))))$$

Is a relatively small (and finite) lazy list without recursion and therefore will be the optimal choice for *reduce1-lzl*.

On the other hand,

$$(integers - from\ 1)$$

will not be a good choice, it will cause an infinite loop.

***reduce2-lzl*:**

reduce2-lzl is a good choice when we want to compute a predefined number of components of a lazy-list, i.e.:

$$(reduce2 - lzl\ +\ 0\ (\text{integers} - \text{from}\ 1)\ 5)$$

This is especially useful when we want to calculate the reduce of some number of components of an infinite lazy-list (also good for a finite one, just will make the computation stop for an infinite one, which won't happen if we use reduce1-lzl for instance).

***reduce3-lzl*:**

We will use *reduce3-lzl* for computing a reduce of a lazy-list one component at a time, meaning that we use it as an Iterator of the reduce of the lazy-list, i.e., to make a delayed computation of reduce of a lazy-list (*reduce1-lzl* and *reduce2-lzl* basically acts like a regular reduce on a regular list and disable the application of delayed computation). Overall, this gives us the general advantage of a lazy-list, just here we get the reduce of the lazy-list relative to some reducer (function that operates on the components of the lazy-list).

g.

<u>Advantages</u>: First of all, in generate-pi-sum, we can decide 'live' whether we want a more accurate approximation or not, and apply the lazy-list again to get it, where in pi-sum we must decide up-front what is the accuracy we want, and just get the first approximation that satisfies this accuracy (difference w.r.t b).

<u>Dis-advantages</u>: In order to get a very good approximation, we will have to apply the lazy-list function allot of times, where in pi-sum we just give a different b that result in better resolution of the approximation. Even if we use take on the generate-pi-approximations, we will get a huge array in return (size as the amount of the times we applied the lazy-list).

(3.1) We perform the unification algorithm as written in link.

We seek to find a unifier between:

$$A = t(s(s), G, s, p, t(K), s)$$
$$B = t(s(G), G, s, p, t(K), U)$$

We do so by performing the unifying algorithm (link above).

Step 1:

Equation	Substitution
$t(s(s), G, s, p, t(K), s) = t(s(G), G, s, p, t(K), U)$	$\{\}$

Step 2:

Equation	Substitution
$s(s) = s(G)$	$\{\}$
$G = G$	
$s = s$	
$p = p$	
$t(K) = t(K)$	
$s = U$	

Step 3:

Equation	Substitution
$G = G$	$\{\}$
$s = s$	
$p = p$	
$t(K) = t(K)$	
$s = U$	
$s = G$	

Step 4:

Equation	Substitution
$s = s$	$\{\}$
$p = p$	
$t(K) = t(K)$	
$s = U$	
$s = G$	

Step 5:

Equation	Substitution
$p = p$	$\{\}$
$t(K) = t(K)$	
$s = U$	
$s = G$	

Step 6:

Equation	Substitution
$t(K) = t(K)$	$\{\}$
$s = U$	
$s = G$	

Step 7:

Equation	Substitution
$s = U$	$\{\}$
$s = G$	
$K = K$	

Step 7:

Equation	Substitution
$s = G$	$\{U = s\}$
$K = K$	

Step 7:

Equation	Substitution
$K = K$	$\{U = s, G = s\}$

Step 7:

Equation	Substitution
	$\{U = s, G = s\}$

We get that the unifier of the expressions is $\{U = s, G = s\}$.

Now we seek to find a unifier between:

$$A = p(\,[\,v \mid [\,V \ \mid W\,]\,]\,)$$
$$B = p(\,[\,[\,v \mid V] \mid W\,]\,)$$

We do so by performing the unifying algorithm (link above).

Step 1:

Equation	Substitution
$p(\,[\,v \mid [\,V \ \mid W\,]\,]\,) \ = p(\,[\,[\,v \mid V] \mid W\,]\,)$	$\{\,\}$

Step 2:

Equation	Substitution
$v = [\,v \mid V\,]$	$\{\,\}$
$[\,V \mid W\,] = W$	

The algorithm returns 'FAIL' here, since we get that $v = [v \mid V]$, which is cannot be true. A symbol cannot be equal to a list (this list also encapsulates that symbol).

<u>This is the algorithm:</u>

Given atomic formulas `A`, `B` they can be unified following these steps:

```
unify(A,B): Substitution | Fail
Initialization: sub: Substitution = {} // Empty substitution
                equations: Equation[] = (A = B)
1. While (equations is not empty):
2.   Let equation_1 = pop(equations)
3.   Let eq'_1 = equation_1 o sub
4.   If one side in eq'_1 is a variable X:
4.1    If the other side is not the same variable: i.e., eq'_1 = {X = term}
4.2     then sub = sub o {X = term}
4.3     else if the other side is the same variable: i.e., eq'_1 = {X = X}
4.4       continue
5.   else if both sides in eq'_1 are atomic, then:
6.     if both sides are the same constant symbol then continue, else return FAIL.
7.   else if the predicate symbols and the number of arguments are the same: eq'_1 = (p(t_1,...,t_n) =
8.     split eq'_1 into equations: equations = equation U (t_i = s_i) for i=1..n, continue.
9.   else return FAIL.
```

(3.3) Proof tree.

We draw the proof-tree for:

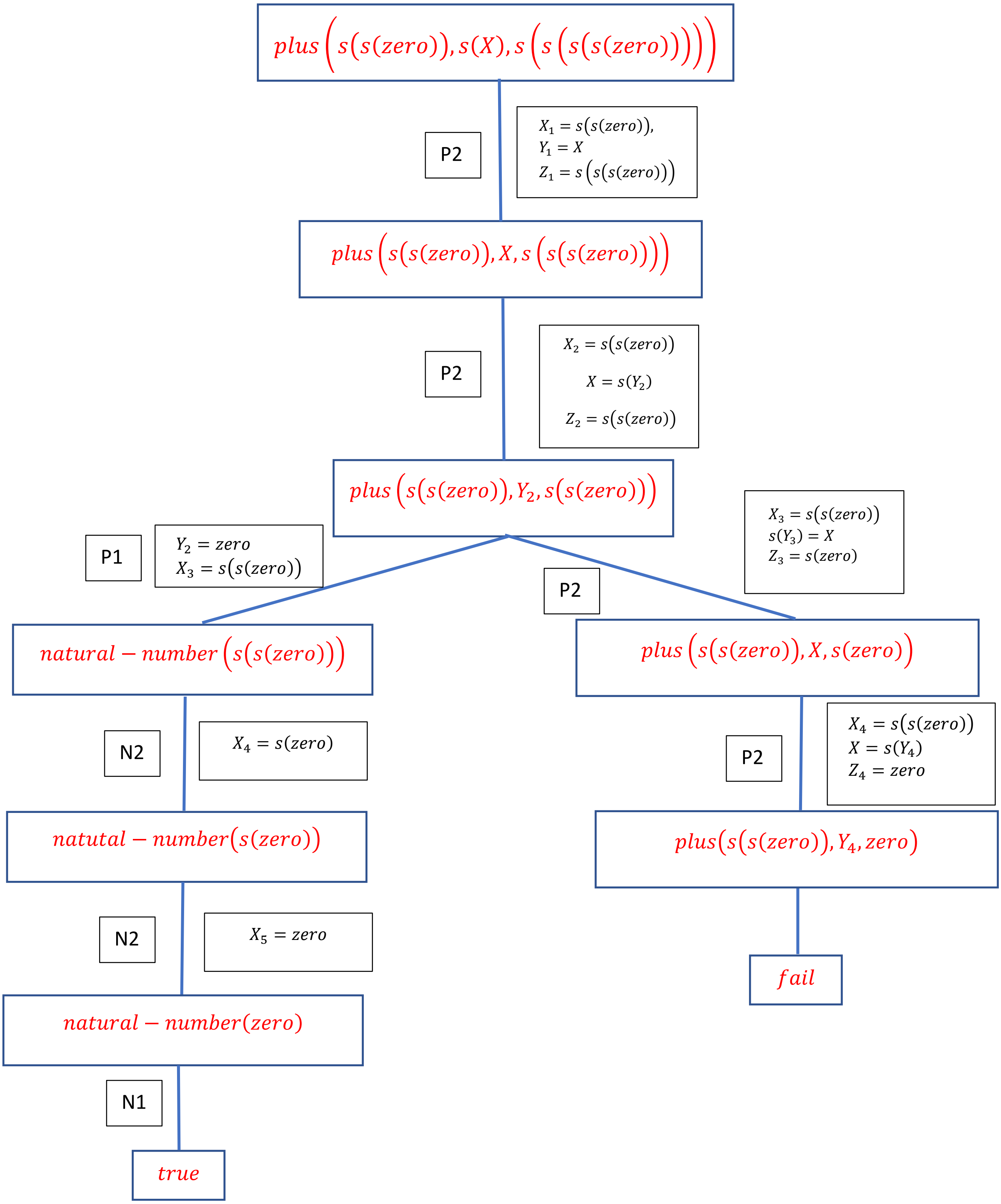
```
% Signature: natural_number(N)/1
% Purpose: N is a natural number.
natural_number(zero).
natural_number(s(X)) :- natural_number(X).

% Signature: plus(X, Y, Z)/3
% Purpose: Z is the sum of X and Y.
plus(X, zero, X) :- natural_number(X).
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).

?- plus(s(s(zero)), s(X), s(s(s(s(zero))))).
```

We denote the first rule of ‘natural_numbers’ as N1, and the second N2.

In the same manner we denote the first rule of ‘plus’ as P1, and the second P2.



$plus(s(s(zero)), s(X), s(s(s(s(zero)))))$
P2
$X_1 = s(s(zero)),$
$Y_1 = X$
$Z_1 = s(s(s(zero)))$
$plus(s(s(zero)), X, s(s(s(zero))))$
P2
$X_2 = s(s(zero))$
$X = s(Y_2)$
$Z_2 = s(s(zero))$
$plus(s(s(zero)), Y_2, s(s(zero)))$
P1
$Y_2 = zero$
$X_3 = s(s(zero))$
$X_3 = s(s(zero))$
$s(Y_3) = X$
$Z_3 = s(zero)$
P2
$natural - number(s(s(zero)))$
$plus(s(s(zero)), X, s(zero))$
N2
$X_4 = s(zero)$
P2
$X_4 = s(s(zero))$
$X = s(Y_4)$
$Z_4 = zero$
$natutal - number(s(zero))$
$plus(s(s(zero)), Y_4, zero)$
N2
$X_5 = zero$
$fail$
$natural - number(zero)$
N1
$true$

$\{X_1 = s(s(zero)),\ Y_1 = X\ \ Z_1 = s\left(s(s(zero))\right)\} \circ$

$\{\ X_2 = s(s(zero)), X = s(Y_2), \qquad Z_2 = s(s(zero))\} \circ$

$\{Y_2 = zero, X_3 = s(s(zero))\} \circ$

$\{\ X_4 = s(zero)\ \} \circ$

$\{\ X_5 = zero\ \}$

$= \ \{X_1 = s(s(zero)),\ Y_1 = s(Y_2),\ Z_1 = s\left(s(s(zero))\right)\ , X_2 = s(s(zero)), X = s(Y_2), Z_2 = s(s(zero))\} \circ$

$\{Y_2 = zero, X_3 = s(s(zero))\} \circ$

$\{\ X_4 = s(zero)\ \} \circ$

$\{\ X_5 = zero\ \}$

$= \ \{X_1 = s(s(zero)),\ Y_1 = s(zero)\ \ Z_1 = s\left(s(s(zero))\right)\ , X_2 = s(s(zero)), X = s(zero),$
$Z_2 = s(s(zero)), Y_2 = zero, X_3 = s(s(zero))\} \circ$

$\{\ X_4 = s(zero)\ \} \circ \{\ X_5 = zero\ \}$

$= \{X_1 = s(s(zero)),\ Y_1 = s(zero)\ \ Z_1 = s\left(s(s(zero))\right)\ , X_2 = s(s(zero)), X = s(zero),$

$Z_2 = s(s(zero)), Y_2 = zero, X_3 = s(s(zero)), X_4 = s(zero)\} \circ \ \{\ X_5 = zero\ \}$

$= \{X_1 = s(s(zero)),\ Y_1 = s(zero)\ \ Z_1 = s\left(s(s(zero))\right)\ , X_2 = s(s(zero)), X = s(zero),$

$Z_2 = s(s(zero)), Y_2 = zero, X_3 = s(s(zero)), X_4 = s(zero), X_5 = zero\ \}$

We take only the variables in the query, which in this case is only X, and get:

$$\{\boldsymbol{X = s(zero)}\}$$