<u>Assignment 5</u>

<u>Q1.1b:</u>

We claim that the procedure append$ is CPS-equivalent to the procedure append. That is, for every lists ls1 ls2 and continuation 'c', (append$ lst1 lst2 c) = (c (append lst1 lst2)).

Proof. We will proof by induction on the length of lst1.

<u>Base case:</u> for lst1 = '() we get:
a-e[(append$ '() lst2 c)] ➔* a-e[(c lst2)] = a-e[(c (append '() lst2))]

<u>Inductive assumption:</u> Now we assume that for $k\epsilon\mathbb{N}$ the claim holds for every $i \leq k$.

<u>Inductive step:</u> let lst1 be a list with length k+1, so we get:
a-e[(append$ lst1 lst2 c)] ➔*
a-e[(append$ (cdr lst1) lst2 (lambda (res) (c (cons (car lst1) res))))] ➔*

Since (cdr lst1) is a list of length k, from the inductive assumption we get:

a-e[((lambda (res) (c (cons (car lst1) res))) (append (cdr lst1) lst2))] ➔*
a-e[(c (cons (car lst1) (append (cdr lst1) lst2))) =
a-e[(c (append lst1 lst2))]

Which is exactly what we wanted to proof.

<u>Q2.d:</u>

In order to use the reduce1-lzl procedure we must be sure first that lzl is finite, otherwise the calculation will not stop. If the lzl is finite, we will use the procedure only when we want the final result when reducing the whole list. That's because we can't control when to stop the reduce, only when we finish calculating the whole list, and we get only the final result.

When using reduce2-lzl we no longer need to check it is finite since we choose how many elements to reduce. So in contrast to the resule1-lzl case, we don't need to know its length. We still use this procedure only when want to know the final result of the calculation.

Similar to reduce2-lzl, when using the reduce3-lzl don't need to know the lzl length in advance. Since the procedure returns a lzl with each step of the reduce, we can use it to delay our computation and to take the results step by step. That is, we can take for instance the reduced first n elements in the lzl, and then later take the next m elements without starting the calculation all over.

<u>Q2.g:</u>

The main advantage in the procedure is since the calculation of the approximation is delayed, we can get a better approximation in each step without starting the calculation all over again and also without using extra memory for each step. That is, if after calculating some approximation we see it is not close enough, we can easily take more elements from our list without using more memory and there is no need to start the approximation from the beginning again. In these sense, the lzl approximation is better then the recursive version we saw at class.

On the other hand, in the recursive implementation we saw at class we can choose in advance what precession we want and get the final result in just one call of the procedure. This is in contrast to the lzl version where we can get a better approximation in each step but can't know how many steps are needed to get the wanted approximation. It means that we need to keep calling each time the next delayed computation.

Q3.1:

a) unify[t(s(s), G, s, p, t(k), s),
t(s(G), G, s, p, t(k), U)]

- Initialization:
  Sub = {}
  Eq = [t(s(s), G, s, p, t(k), s) = t(s(G), G, s, p, t(k), U)]
- Eq1 = t(s(s), G, s, p, t(k), s) = t(s(G), G, s, p, t(k), U)
- Eq1' = Eq1 o Sub
- Case where both sides are number of args: split into equations.
- Eq = [s(s)=s(G), G=G, s=s, p=p, t(k) = t(k), s=U]
- Eq is not empty, continue to another iteration:
- Eq1 = [s=U]
- Eq1' = Eq1 o Sub
- Case where one side is variable: update Sub.
- Sub = Sub o {U=s}
- Eq is not empty, continue to another iteration:
- Eq1 = [t(k) = t(k)]
- Eq1' = Eq1' o Sub = [t(k) = t(k)]
- Case where both sides are number of args: split into equations.
- Eq = [s(s)=s(G), G=G, s=s, p=p, k=k]
- Eq is not empty, continue to another iteration:
- Eq1 = [k = k]
- Eq1' = Eq1 o Sub
- Case where both sides are atomic with same constant symbol: continue.
- Eq is not empty, continue to another iteration:
- Eq1 = [p = p]
- Eq1' = Eq1 o Sub
- Case where both sides are atomic with same constant symbol: continue.
- Eq is not empty, continue to another iteration:
- Eq1 = [s = s]
- Eq1' = Eq1 o Sub
- Case where both sides are atomic with same constant symbol: continue.
- Eq is not empty, continue to another iteration:
- Eq1 = [G = G]

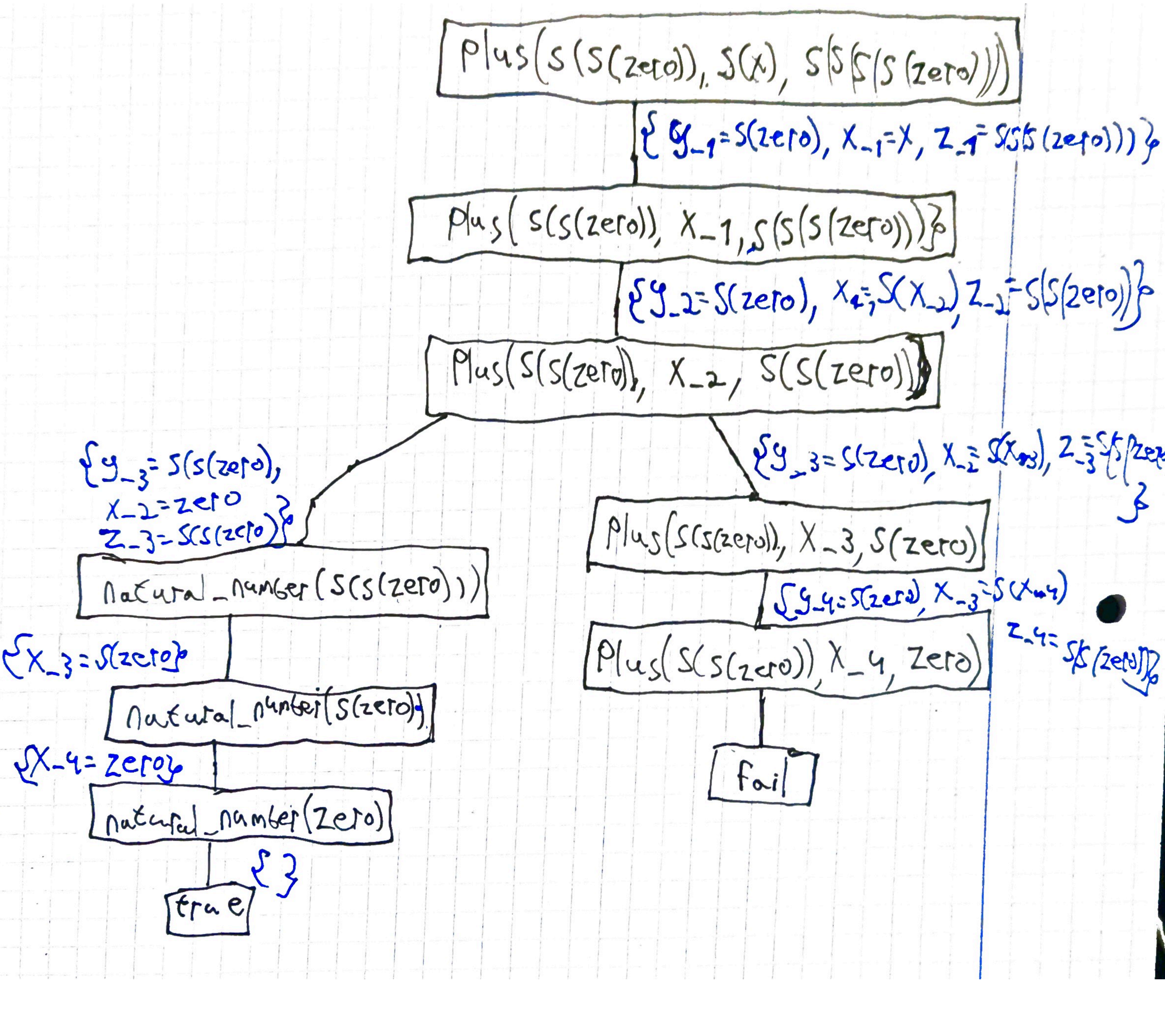
- Eq1' = Eq1 o Sub
- Case where both sides are atomic with same variable: continue.
- Eq is not empty, continue to another iteration:
- Eq1 = [s(s) = s(G)]
- Eq1' = Eq1 o Sub
- Case where both sides are number of args: split into equations.
- Eq = [s=G]
- Eq is not empty, continue to another iteration:
- Eq1 = [s = G]
- Eq1' = [s= G] o Sub
- Case where one side is variable: update Sub.
- Sub = Sub o {s=G}
- Eq is empty

By these steps we can see that the MGU is {G=s, U=s}

b) unify[p(v | [V | W]]),
p([[v | V] | W])]

- Initialization:
  Sub = {}
  Eq = [p([v | [V | W]])= p([[v | V] | W])]
- Eq1 = [p([v | [V | W]])= p([[v | V] | W])]
- Eq1' = Eq1 o Sub
- Case where both sides are number of args: split into equations.
- Eq = [v= [v|V], [V | W] = [W]]
- Eq1' = Eq1 o Sub

Fail because v can't be defined with a list [v | V].



plus(s(s(zero)), s(X), s(s(s(s(zero)))))
{Y_1=s(zero), X_1=X, Z_1=s(s(s(zero)))}
plus(s(s(zero)), X_1, s(s(s(zero))))
{Y_2=s(zero), X_1=s(X_2), Z_2=s(s(zero))}
plus(s(s(zero)), X_2, s(s(zero)))
{Y_3=s(s(zero)), X_2=zero, Z_3=s(s(zero))}
natural_number(s(s(zero)))
{X_3=s(zero)}
natural_number(s(zero))
{X_4=zero}
natural_number(zero)
{}
true
{Y_3=s(zero), X_2=s(X_3), Z_3=s(s(zero))}
plus(s(s(zero)), X_3, s(zero))
{Y_4=s(zero), X_3=s(X_4), Z_4=s(s(zero))}
plus(s(s(zero)), X_4, zero)
fail