Assignment 5 – PPL

Question 1:

1.1.a) The equivalence criterion for two lazy lists(lz1,lz2) is: one of the three

1. Both of lists are empty

2. If both lists are infinity: for each n, to make n times tail on lz1 and then make head on the result will be the same value as make n times tail on lz2 and then make head on the result

3. If both lists aren't infinity: there is an n that make n times tail on lz1 will get the result of empty list, and also make n times tail on lz2 will get the result of empty list.

In addition, for each k<n the condition is also happening.

1.1.b) We will show that the even-squares-1 and even-squares-2 lazy lists are equivalent according to our definition.

We will define what each list is doing.

Even-squares-1: make natural number, put him on square and check if he is even.
If yes return him, else make the next natural number.

Even-squares-2: make natural number, check if he is even.
If yes put him on square and return him, else make the next natural number.

Diagnosis: natural number is even if and just if his square is also even, Therefore each of the lists return the same values and they are infinity.
Now, we just have to show that they return the values on the same order.
We will do it with induction.

<u>Base: n=0</u>
Both lists make the natural numbers in the same way start from zero(integers-from).
So, make tail 0 times and then make head on the return value is actually:

1)(head even-squares-1)

2)(head even-squares-2)

We will look at both actions.

<u>Action 1: (head even-squares-1)</u>

Make (integers-from 0) and we get 0, then put him on square with x=0.

Then, make lzl-filter on the return value(0) and check if it is even(he is) and return him.

<u>Action 2: (head even-squares-2)</u>

Make (integers-from 0) and we get 0, then make lzl-filter on the return value(0) and check if it is even.

Hi is even, So forward him to square action with x=0, and the return value is 0.

Both lists return zero, base is good.

<u>The induction assuming:</u>

For each list, make tail n-1 times on the list and then make head on the return value will get the same value.

<u>Step:</u>

We will look at the n action of even-squares-1 making tail on the list and then head.
First of all, even-squares-1 make lzl-map on the return value(integers-from 0).
At the end of the last run of even-squares-1 there was a value of k^2, Therefore at the last run of even-squares-1 the last action of (integers-from 0) return the value k.
Now, the value that came back from (integers-from 0) was k+1.
On k+1 we make lzl-map that put him on square when x=k+1.
Since k was even, So k+1 isn't even and also $(k+1)^2$ isn't even.
In conclusion, k is even, k+1 isn't even and $(k+1)^2$ isn't even.
Therefore, the predict (lambda (x) (= (modulo x 2) 0) return false, and
lzl-filter make tail another time on even-squares-1.
Now, the return value from (integers-from 0) is k+2, and finally the return value is $(k+2)^2$.

We will look at the n action of even-squares-2 making tail on the list and then head.
From the induction assuming, at the end of the last run of even-squares-2 there was a value of k^2 like even-squares-1.
Now, the value that came back from (integers-from 0) was k+1.
Since k was even, So k+1 isn't even.
Therefore, the predict (lambda (x) (= (modulo x 2) 0) return false, and
lzl-filter make tail another time on even-squares-2.
Now, the return value from (integers-from 0) is k+2.
On k+2 we make lzl-map that put him on square when x=k+2.
And finally the return value is $(k+2)^2$.

Thus, Both lists return the same value after making same number of times tail and after that head.
According to the rules we defined on the previous section the even-squares-1 and even-squares-2 lazy lists are equivalent.

Question 2:

2.a) We will define procedure f from type [T1*…*Tn->(U1 union U2)].
And procedure f$ type [T1*…*Tn*[U1->U1]*[Empty->U2]->(U1 union U2)].

Say that f$ is the version of Success-Fail-Continuations that equivalence to f just if f and f$ have the same range and for all parameters x1,…,xn and for each function fail,succ one of the following is happening:

1. if make (f x1 … xn) return value from type U1 so make (succ(f x1 … xn)) and make (f$ x1 … xn succ fail) return the same value from type U1.
2. if make (f x1 … xn) get to flow of an error and return value from type U2 so make fail return the same value from type U2 and also make (f$ x1 … xn succ fail) return the same value from type U2.
3. if make (f x1 … xn) doesn't finish so also make (f$ x1 x2 … xn succ fail) doesn't finish.

2.d) We will show that get-value and get-value$ are equivalent according to our definition.
Get-value is recursive, So we will prove in induction on the length of the list association-list, define the length n.

<u>Base: n=0</u>

So, association-list = '().
"Key" will be such symbol and fail, success such procedures.
So, the value "key" isn't in association-list and the calculation ends with error.
a-e[(get-value association-list key)]->*a-e [(get-value ‘() key)] ->* 'fail'.

And what is really happening is:
a-e[(get-value$ association-list key success fail)]->*a-e[(get-value$ ‘() key success fail)]->*a-e[(fail)].

From our definition, get-value and get-value$ are equivalent for n=0.

<u>Base: n=1</u>

Association-list contains one pair, lets say (key0, value0).
"Key" will be such symbol and fail, success such procedures.
So there is 2 opportunities:

<u>First option- key=key0:</u>

Key is in association-list and the calculation ends with returning value0.

a-e[(get-value association-list key)]->*a-e [(get-value‘((key0.value0))key0)]->*value0.

And also happening:

a-e[(get-value$ association-list key success fail)]->* a-e[(get-value$ ‘((key0.value0)) key0 success fail)]->*a-e[(success(cdr (car ‘((key0.value0)))))]->*a-e[(successvalue0)]

So the functions are equivalent according to our definition.

<u>Second option- key != key0:</u>

Key isn't in association-list and the calculation ends with error.

a-e[(get-value association-list key)]->*a-e [(get-value‘((key0.value0))key0)]->*'fail.

And also happening:

a-e[(get-value$ association-list key success fail)]->* a-e[(get-value$ ‘((key0.value0)) key success fail)]->*a-e[(fail)]

So the functions are equivalent according to our definition.

<u>The induction assuming:</u>

for $n = k \in \mathbb{N}$ the assuming happens for each $k \geq i \in \mathbb{N}$.

We mean that for each association-list from length I happens that for each key,succ,fail the procedures get-value and get-value$ are equivalent.

<u>Step</u>

Will be $k \in \mathbb{N}, n = k + 1$, so:
1) The first pair in the list is (key.value0).
So, the condition (eq? (car (car association-list)) key) return #t and happens:
a-e [(get-value association-list key)]->*
a-e [(if (eq? (car (car association-list)) key) (…) (…)]->*
a-e [(cdr (car association-list))]->* value0

As well,
a-e [(get-value$ association-list key success fail)]->*
a-e [(get-value$ ‘((key . value0)) key success fail)]->*
a-e [(success (cdr (car ‘((key . value0)))))]->* a-e [(success value0)]

And the functions are equivalent for association-list from n=k+1 length according to our definition.

2) The first pair in the list is (key0.value0) so y != key0.
So, the condition (eq? (car (car association-list)) key) return #f and happens:
a-e [(get-value association-list key)]->*
a-e [(if (eq? (car (car association-list)) key) (…) (…)]->*
a-e [(get-value (cdr association-list) key)]

As well,
a-e [(get-value$ association-list key success fail)]->*
a-e [(if (eq? (car (car association-list)) key) (…) (…)] ->*
a-e [(get-value$ (cdr association-list) key success fail)]

The list (cdr association-list) length is n=k.
Therefore, from the induction assuming both of the functions are equivalent for this list, for the parameter key and for the functions fail and success.
In conclusion, both of the functions get-value and get-value$ are equivalent for the original list(association-list) from n=k+1 length.

Question 3:

3.1.a) Failure.

We will make the Unify algorithm on:

Unify [t (s (s) , G , H , p, t (E) , s) ,

t (s (H) , G , p , p, t (E) , K)].

First of all, we define empty substitution: {}.

The composite terms are atomic formula with the predict t, who contains different terms.

The amount of terms in each atomic formula is the same, Therefore we will compare among all the terms that in the predict t.

We will define the next equations: 1)s(s)=s(H), 2)G=G, 3)H=p, 4)p=p, 5)t(E)=t(E), 6)s=k

We make unification between s(s) and s(H), it's compare of two atomic formulas, the predict is equal and the amount of terms is equal.

So, we will compare between the terms inside the s's.

We get H=s, we add him to the substitution and get : {H=s}.

We look at the equation G=G.

Same variable appears on both sides, therefore we continue to the next step without any influence on our substitution.

We will activate our substitution on H=p and we get s=p.

Both sides of the equation there are different atomic expressions.

So, the algorithm return failure.

3.1.b) Failure.

We will make the Unify algorithm on:

Unify [g (c , v (U) , g , G , U , E , v (M)) ,

g (c , M , g ,v (M) , v (G) , g , v (M)].

First of all, we define empty substitution: {}.

The composite terms are atomic formula with the predict g, who contains different terms.

The amount of terms in each atomic formula is the same, Therefore we will compare among all the terms that in the predict g.

We will define the next equations: 1)c=c, 2)v(U)=M, 3)g=g, 4)G=v(M), 5)U=v(G), 6)E=g, 7)v(M)=v(M)

We will look at the first equation, we activate the substitution on c=c.

Both sides are identical atomics, Therefore we continue to the next step without any influence on our substitution.

We look at the next equation, on one side we have a variable so we activate the substitution on v(U)=M and we add the result to the substitution and get: {M=v(U)}.

We will look at the next equation, we activate the substitution on g=g.

Both sides are identical atomics, Therefore we continue to the next step without any influence on our substitution.

We look at the next equation, on one side we have a variable so we activate the substitution on G=v(M), we get G=v(v(U)).

We add the result to the substitution and get: {M=v(U), G=v(v(U))}.

We look at the next equation, on one side we have a variable so we activate the substitution on U=v(G), we get U=v(v(v(U))).

We get failure since we get the same variable on both sides.

3.1.c) Failure.

We will make the Unify algorithm on:

Unify s ([v | [[v | V] | A]]) ,

s ([v | [v | A]])

First of all, we define empty substitution: {}.

The composite terms are atomic formula with the predict s, who contains different terms.

The amount of terms in each atomic formula is the same, Therefore we will compare among all the terms that in the predict s.

We define the equation: [v | [[v | V] | A]] = [v | [v | A]]

We activate our empty substitution on the equation, the terms will stay without change.

We are doing unification for two pairs, therefore now we make unification between each pairs from the same index.

We get the equations: v=v, [[v | V] | A] = [v | A]

Activate substitution on the first equation doesn't change the terms because both sides are identical atomics.

So, we continue to the next step without influence on our substitution.

Now, we look at the second equation, activate substitution doesn't change the terms.

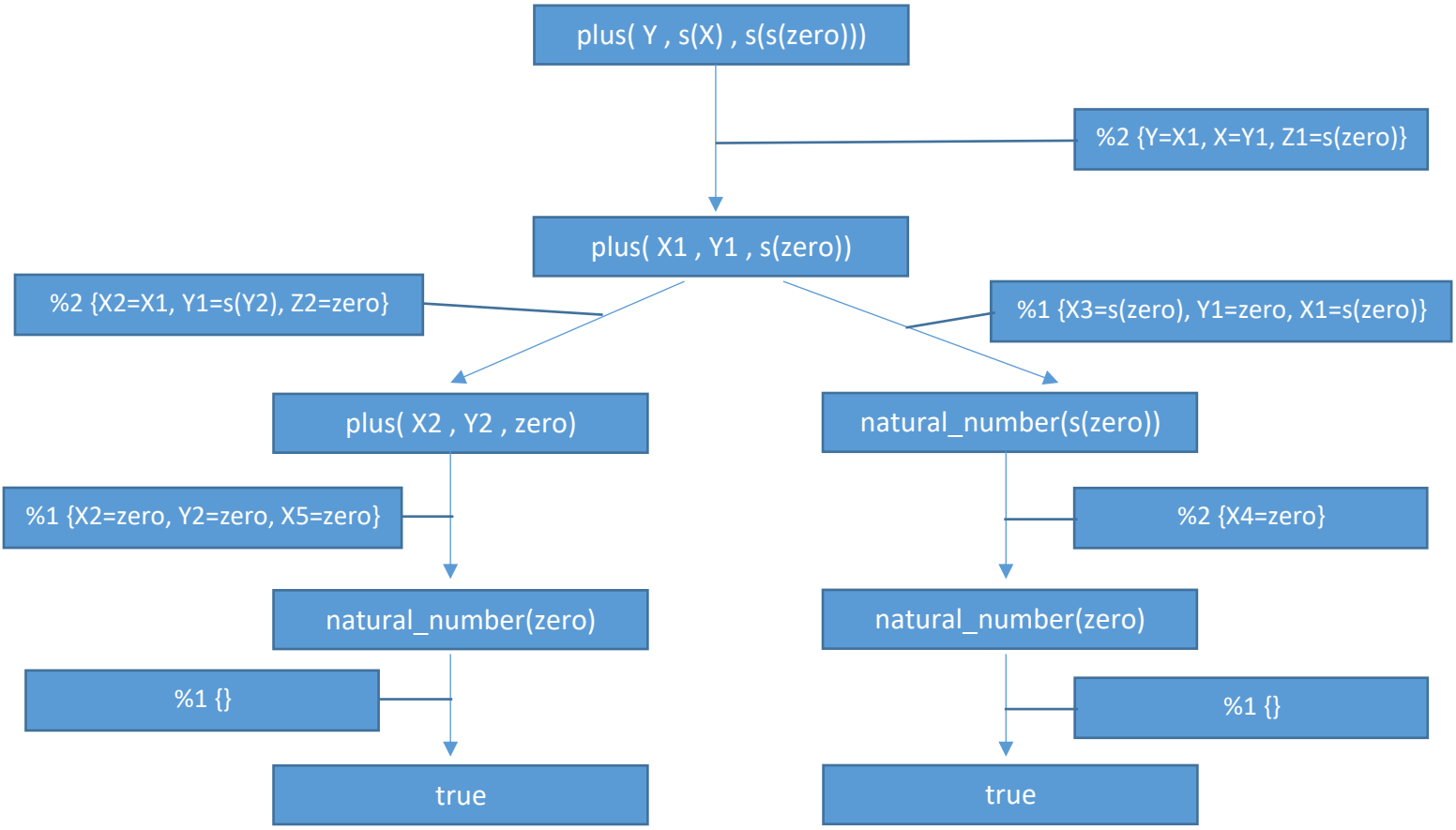
Same as before, we make unification for two pairs, Therefore we will do unification between each pair from the same index.

We get the following equations: v=[v | V], A=A

We look at the first one, activate substitution doesn't change the terms.

But now we get symbol (v) in one side and in the other side we get complicated expression, So the unification return failure.

3.3.a) The proof tree for the query - plus(Y , s(X) , s(s(zero))) is:



The composition substitution of the right path is:

{} -> {X4=zero} -> {X3=s(zero), Y1=zero, X1=s(zero)} -> {Y=X1, X=Y1, Z1=s(zero)} = {X4=zero,X3=s(zero), Y1=zero, X1=s(zero), Y=s(zero), X=zero, Z1=s(zero)}

So the restriction for the variables get the substitution: {Y=s(zero), X=zero}.

The composition substitution of the left path is:

{} -> {X2=zero, Y2=zero, X5=zero} -> {X2=X1, Y1=s(Y2), Z2=zero} -> {Y=X1, X=Y1, Z1=s(zero)} =
{X2=zero, Y2=zero, X5=zero, X1=zero, Y1=s(zero), Z2=zero,
Y=zero, X=s(zero), Z1=s(zero)}

So the restriction for the variables get the substitution: {Y=zero, X=s(zero)}.

3.3.b) The answers of the answer-query algorithm for this query are:

{{Y=s(zero), X=zero}, {Y=zero, X=s(zero)}}

3.3.c) This is a success proof tree because he has successful computation path, for instance: the left path that is finite route from the rout until the leaf.
And the leaf is successful.

3.3.d) The proof tree doesn't contain infinity routes, therefore he is finite.