PPL - Assignment 5

Part 1: Theoretical Questions

1.1.a.

Two lazy lists (lzl1, lzl2) are similar if they fulfill one of the following conditions:

1. Both lists are empty '()

2. In case the two lazy lists are endless, for every n, applying n times a tail function on lzl1 and lzl2 separately, and afterwards applying a head function on each result, will yield the same value.

3. In case the two lazy lists are finite, there exists n, such that applying n times a tail function on lzl1 and lzl2 separately, will both yield an empty list, also, for every n, applying n times a tail function on lzl1 and lzl2 separately, and afterwards applying a head function on each result, will yield the same value.

1.1.b.

We'll show that **even-squares-1 and even-squares-2 lists are equivalent using induction**:

Base n=0: Both lists produce their integers in the same way (using integers-from) starting from 0. Therefore, applying a tail function zero times, along with applying the head function once afterwards will return the following:

For the even-squares-1 procedure we receive (head even-squares-1) which returns zero, since zero squared return zero and zero is even.

For the even-squares-2 procedure we receive (head even-squares-2) which also return 0, since zero is even and zero squared return zero

Therefore, the base case exists.

Induction assumption: For each list, applying the tail function k times (k<n), along with applying the head function once afterwards, will yield the same values for both procedures.

Inductive step: In the nth step, applying the tail function n times, along with applying the head function once afterwards will return the same squared even lazy lists of integers.

We know that a squared even integer returns an even integer and a squared odd integer returns an odd integer, therefore the even predicate will filter the same elements, no matter if we apply it before the squaring or afterwards. In our case, mapping first each integer to a squared integer, and then filtering with the even predicate, will return the same result as first filtering with the even predicate and then mapping each even integer to a squared integer. Also, since both even-squares-1 procedure and even-squares-2 procedure use the same integer-form definition, they will have the same initial lazy lists, and eventually return the same squared even lazy lists of integers.

Part 2: Theoretical Questions

2.a.

Let’s define procedure f type as: [x1 * x2 * … * xn -> T1 U T2]

and procedure f$ type as: [x1 * x2 * … * xn * [T1->T1] * [Empty -> T2] -> T1 U T2]

We’ll say f$ is an equivalent success-fail continuations version to f, if for the same (x1 x2 … xn) parameters they fulfill the following conditions:

1. If Applying (f x1 x2 … xn) will return a T1 value, then applying (f$ x1 x2 … xn succ fail) will return the same T1 value

2. If Applying (f x1 x2 … xn) will return a T2 value, then applying (f$ x1 x2 … xn succ fail) will return the same T2 value

3. If Applying (f x1 x2 … xn) will not terminate, then applying (f$ x1 x2 … xn succ fail) will not terminate

2.d.

We’ll show that **get-value and get-value$ are equivalent using induction**:

Base n=0: Assoc-list is empty thus we’ll receive:

a-e [(get-value assoc-list key)] -> a-e [(get-value ‘() key)] = 'fail

a-e [(get-value$ assoc-list key success fail)] -> a-e [(get-value$ ‘() key success fail)] = a-e [(fail)]

Therefore, get-value and get-value$ are equivalent

Induction assumption: For each assoc-list which contains k pairs (k<n), the procedures get-value and get-value$ are equivalent

Inductive step: In the nth step, for each n pairs assoc-list which contains one pair ‘((k0 . v0)) we’ll consider two case:

Case key= k0:

a-e [(get-value assoc-list key)] -> a-e [(if (eq? (car (car assoc-list)) key) (…) (…)] ->*

a-e [(cdr (car assoc-list))] = v0

a-e [(get-value$ assoc-list key success fail)] -> a-e [(get-value$ ‘((key . v0)) key success fail)] ->*

a-e [(success (cdr (car ‘((key . v0)))))] = a-e [(success v0)]

Case key!= k0:

a-e [(get-value assoc-list key)] ->a-e [(if (eq? (car (car assoc-list)) key) (…) (…)] ->*

a-e [(get-value (cdr assoc-list) key)] = ‘fail

a-e [(get-value$ assoc-list key success fail)] -> a-e [(if (eq? (car (car assoc-list)) key) (…) (…)] ->*

a-e [(get-value$ (cdr assoc-list) key success fail)] = a-e [(fail)]

Therefore, get-value and get-value$ are equivalent

Part 3: Theoretical Questions

3.1.a. The result is: Failure.

We’ll start from an empty substitution { }.

We’ll build the following equations: s(s) = s(H), G = G, H = p, p = p, t(E) = t(E), s = K

(For the following equations: G=G, p=p, t(E)=t(E), there is no change)

For the equation s(s) = s(H), we’ll add {H = s} to our substitution

For the equation H = p, will use the substitution and get s=p.

But since we received two different atomic expressions in both sides the algorithm failed.

3.1.b. The result is: Failure.

We’ll start from an empty substitution { }.

We’ll build the following equations: c = c, v(U) = M, g = g, G = v(M), U = v(G), E = g, v(M) = v(M)

(For the following equations: c = c, g = g, v(M) = v(M), there is no change)

For the equation v(U) = M, we’ll add {M = v(U)} to our substitution.

For the equation G = v(M), we’ll use the substitution and update our substitution to {M = v(U), G = v(v(U))}.

For the equation U = v(G), we’ll use the substitution and get U = v(v(v(U))).

But since we received the same variable in both sides the algorithm failed.

3.1.c.

We’ll start from an empty substitution { }.

We’ll build the equation: [v | [[v | V] | A]] = [v | [v | A]]

From each index in that equation we'll receive the following equations: v=v, [[v|V] | A] = [v | A]

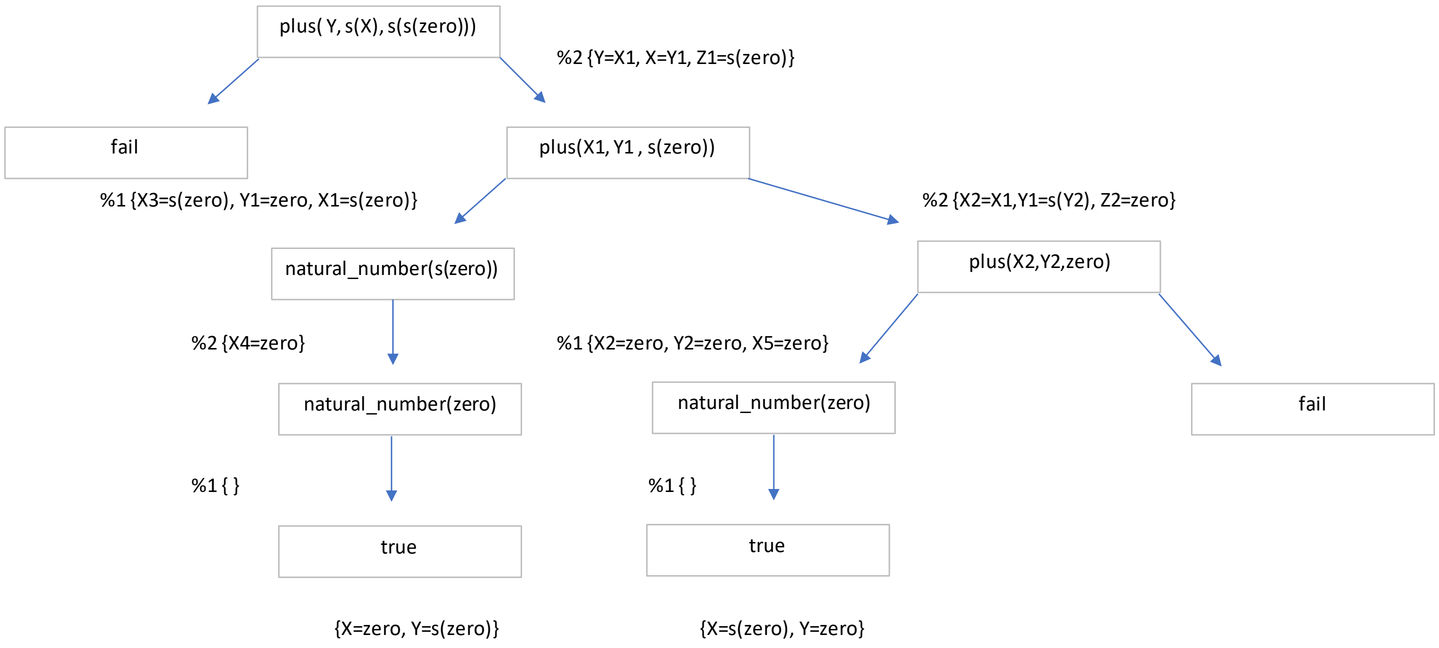
(For the following equations: v=v, there is no change)

From each index in the equation [[v|V] | A] = [v | A], we'll receive the following equations: v=[v|V], A=A

(For the following equations: A=A, there is no change)

But since we received a symbol in one side and a compound expression (list) in the other side, the algorithm failed.

3.3.a.



3.3.b. The answers are: { {Y=s(zero), X=zero}, {Y=zero, X=s(zero)} }

3.3.c. This is a success proof tree, because it has a successful computation paths (a complete finite path from the root to a true leaf)

3.3.d. The tree does not have any infinite paths, and since all the tree's paths are finite, the proof tree is finite