**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. :

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

**Base:**

**Assumption:**

**Step:**

Denote:

Notice that and therefore by our assumption,

Proof of

Claim:

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

**Base:**



As we can see

**Assumption:**

**Step:**

Notice that and therefore by our assumption,

**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:

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

On the other hand,

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.:

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.

**Advantages**: 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).

**Dis-advantages**: 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](https://bguppl.github.io/interpreters/class_material/5.1RelationalLogicProgramming.html).

We seek to find a unifier between:

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

**Step 1:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |

**Step 2:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Step 3:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Step 4:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |
|  |  |
|  |  |

**Step 5:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |
|  |  |

Here we get two atomic symbols (p, s) that are not the same, therefore we get that there is no unifier (substitution) for the given predicats.Now we seek to find a unifier between:

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

**Step 1:**

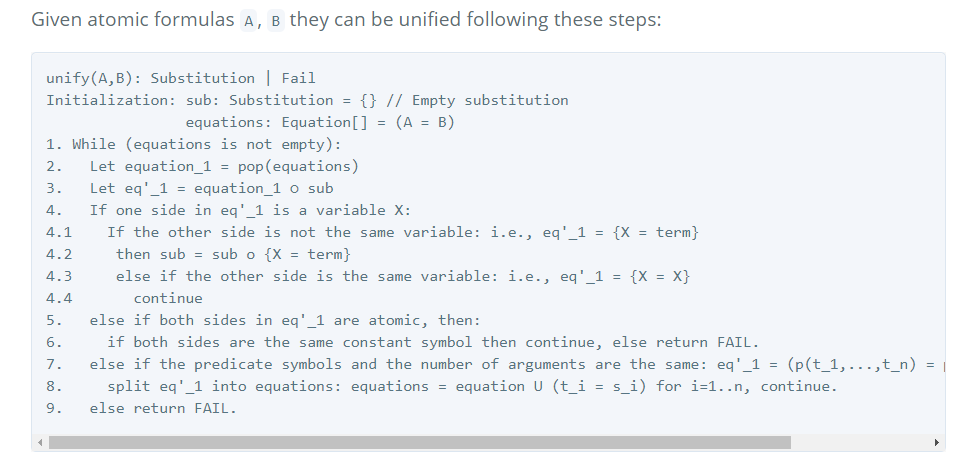
|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |

**Step 2:**

|  |  |
| --- | --- |
| Equation | Substitution |
|  |  |
|  |  |

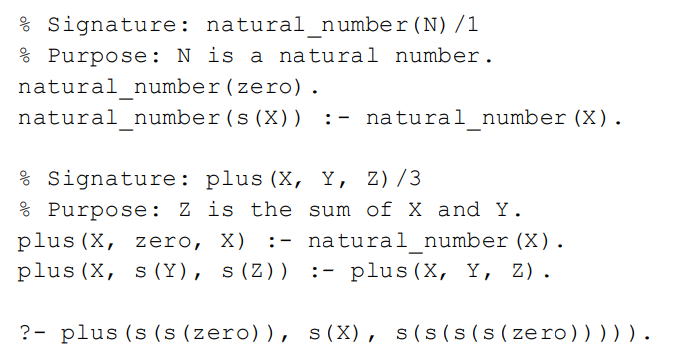
The algorithm returns ‘FAIL’ here, since we get that , which is cannot be true. A symbol cannot be equal to a list (this list also encapsulates that symbol).

**This is the algorithm:**



(3.3) Proof tree.

We draw the proof-tree for:



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.

P2

P2

P1

N1

N2

N2

P2

P2

We collect the substitution from the branch that returned ‘true’, and we get that   
. **TODO**