Assignment 4

Yonatan Bitton 203244694
Meir Shomron 311499628

Question 2 - Generators / Lazy Lists

a.

1. Define an equivalence criterion for two given generators / lazy lists.

Answer:

Definition:

Let L1, L2 lazy-lists / lazy lists.

L1 and L2 will be considered equivalent, if for each natural number n,
(take n L1) is equals to (take n L2)).

* take – definition is :

```
function* take(n, generator) {
   for (let x of generator) {
      if (n <= 0) return;
      n--;
      yield x;
   }
}
```

2. Show that the following evenSquares1 and evenSquares2 generators equivalent according to your definition.

Answer:

We will proof by induction on n, that take(n evenSquares1) equals take(n evenSquares2).

Base case: n=1. By definition of take, n=1 is decreased to n=0, and yield x returns the first element of the naturalNumbers() ,0 which is returned to:

a. evenSquares1: 0 is mapped to 0^2 = 0, and then filtered by (x%2==0) and the result is 0.

b. evenSquares2: 0 is filtered by (x%2==0), and then mapped to 0^2 = 0, and the result is 0.

(equal [0] [0]) is true, therefore the base case is true.

Assume n<k holds: take(n evenSquares1) equals take(n evenSquares2).

Show n=k holds: take(n evenSquares1) equals take(n evenSquares2).

Distinction: For lazy-list L, take(n L) equals take(n-1 L) · (n'th member of L)

Therefore:

take(n evenSquares1) = take(n-1 evenSquares1) · (n'th member of evenSquares1)

take(n evenSquares2) = take(n-1 evenSquares2) · (n'th member of evenSquares2)

By the induction hypothesis for n-1, take(n-1 evenSquares1) = take(n-1 evenSquares2)
Lets observe the n'th member of evenSquares1 and evenSquares2.
By definitions of those lists, the n'th item is generated from the n'th member of naturalNumbers() for both. Lets mark n'th member of naturalNumbers() as x.
evenSquares1: takes x, maps it to x^2, and then filters by (x^2 %2 == 0).
evenSquares2: takes x, filters by (x%2 == 0), and then maps it to x^2.
If x is odd, x^2 is also odd. Then the filter won't pass, and then and the lists from the hypothesis will stay the same and equal.
If x is even:
evenSquares1: x^2 is also even. (x^2 % 2 == 0) will be passed, and x^2 will concatenated to (take n-1 evenSquares1).
evenSquares2: x is even, (x%2==0) is passed, and then mapped to x^2, which will be concatenated to (take n-1 evenSquares2).

So (n'th member of evenSquares1) is equal to (n'th member of evenSquares2).
By the distinction take(n evenSquares1) equals to take(n evenSquares2),
and by the definition of lazy-lists equivalence, evenSquares1 is equal to evenSquares2.

3. We will proof by induction on n, that (take n fibs1) equals to (take n fibs2).
1st Base case: n=1. By definition of take, fibs1 and fibs2 is not empty, so from this statement: (cons (head lzl) (take (tail lzl) (- n 1))) → if n=1, only (head lzl) is returned. (if n==0 '() is returned and being cons'ed to '(0)). (head fibs1) = (head fibs2) = 0, Therefore the 1st case holds.
2nd Base case: n=2. By definition of take, fibs1 and fibs2 is not empty, so from this statement: (cons (head lzl) (take (tail lzl) (- n 1))) → if n=2, the first element equals according to 1st base case, and the second element at the 2 cases is 1, because (take (tail lzl) 1) gives us the head of the tail, which is 1. Therefore the 2nd base case holds.

Assume n<k holds: (take n fibs1) equals (take n fibs2).
Show n=k holds: (take n fibs1) equals (take n fibs2).

Distinction: For lazy-list L, (take n L) equals (take n-1 L) · (n'th member of L)
Therefore:

(take n fibs1) equals take(n-1 fibs1) · (n'th member of fibs1)

(take n fibs2) equals (take n-1 fibs2) · (n'th member of fibs2)

By the induction hypothesis for n-1,(take n-1 fibs1) equals (take n-1 fibs2).
Lets observe the n'th member of fibs1 and fibs2.
By definitions of those lists, the n'th item is generated from the n'th-2 and n'th-1 members from both. (Which are equal by the induction hypothesis).

Lets mark n'th-2 member and n'th-1 member as x_2 x_1, and construct the n'th element.
fibs1: (fibgen x_2 x_1) returns (x_2 . (lambda () (fibgen (x_1 (+ x_1 x_2)))))
This application: (fibgen (x_1 (+ x_1 x_2))) returns (x_1 . (lambda () (fibgen (+ x_1 x_2) y)))
This application: (fibgen (+ x_1 x_2) y) returns ((+ x_1 x_2) . (lambda () …)))
The n'th element is the head of ((+ x_1 x_2) . (lambda () …))), which is (+ x_1 x_2).

fibs2: x_2 is (head fibs2), and x_1 is (head (tail fibs2)).
Lets observe these application: (lz-lst-add (tail fibs2) fibs2)
lz1: (x_1 . (lambda () (…))
lz2: (x_2 . (lambda () (…))
lzl1 and lzl2 are not empty, so from this line: (cons-lzl (+ (head lz1) (head lz2)) (lambda () ..))
The cons-lzl returns lazy list. The head of the returned lazy-list is (+ x_1 x_2) which is the n'th element.

So (n'th member of fibs1) is equal to (n'th member of fibs2).
By the distinction (take n fibs1) equals to (take n fibs2),
and by the definition of lazy-lists equivalence, fibs1 = fibs2.

Question 3 - CPS programming

1. Proove by induction that append is equivalent to append$.
Answer: Proof by induction on the 1st list's length - n.
(append$ x y cont)=(cont (append x y))
Base case: n=0
(cont (append x y))
= (cont (append '() y))
= (cont y)

(append$ x y cont) =
(cont y)
Therefore base case holds.

Assume n<k holds: (cont (append x' y')) = (append$ x' y' cont)
Show n=k holds: (cont (append x y)) = (append$ x y cont)

(append$ x y cont)=
(append$ (cdr x) y (lambda (appended-cdr) (cont (cons (car x) appended-cdr))))
By the induction hypothesis for n-1 [length of (cdr x) = n-1],
(append$ (cdr x) y (lambda (appended-cdr) (cont (cons (car x) appended-cdr))))
=
((lambda (appended-cdr) (cont (cons (car x) appended-cdr))) (append (cdr x) y))
a-e[((lambda (appended-cdr) (cont (cons (car x) appended-cdr))) (append (cdr x) y))] (app-exp)
replace appended-cdr with (append (cdr x) y)
a-e[(cont (cons (car x) (append (cdr x) y)))] =
a-e[(cont (append x y))].
Therefore append$ is CPS-equivalent to append.

Question 4 -

Logic programming: Lists and functors, Unification, Answer-query algorithm

b. What is the result of these operations? Provide all the algorithm steps. Answer in 'ex4.pdf' file. Explain in case of failure.

a. unify[p(v(v(d(1), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), KtM))]

b. unify[p(v(v(d(1), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), ntuf3))]

c. unify[p(v(v(d(M), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), KtM))]

d. unify[p(v(v(d(1), M, p), X)), p(v(d(B), v(B, ntuf3), KtM))]

Answer:

At all cases, the result is error.

The unify procedure calls unify-formulas, with the args of each functors recursively.

Therefore, there will be checking with the second v of the first formula, and the first v of the second formula. For example in question a:

First formula: v(v(d(1), M, ntuf3), X)

Second one: v(d(B), v(B, ntuf3), KtM)

We can see that the ARITY of the first formula is different from the second, because at the first formula v receives 2 arguments, and at the second formula v receives 3 arguments. At unify-formulas, a checking for the args length will be called, which aren't equal, therefore an error will thrown. Sadly, it's the case for all of the cases in the question.

c.

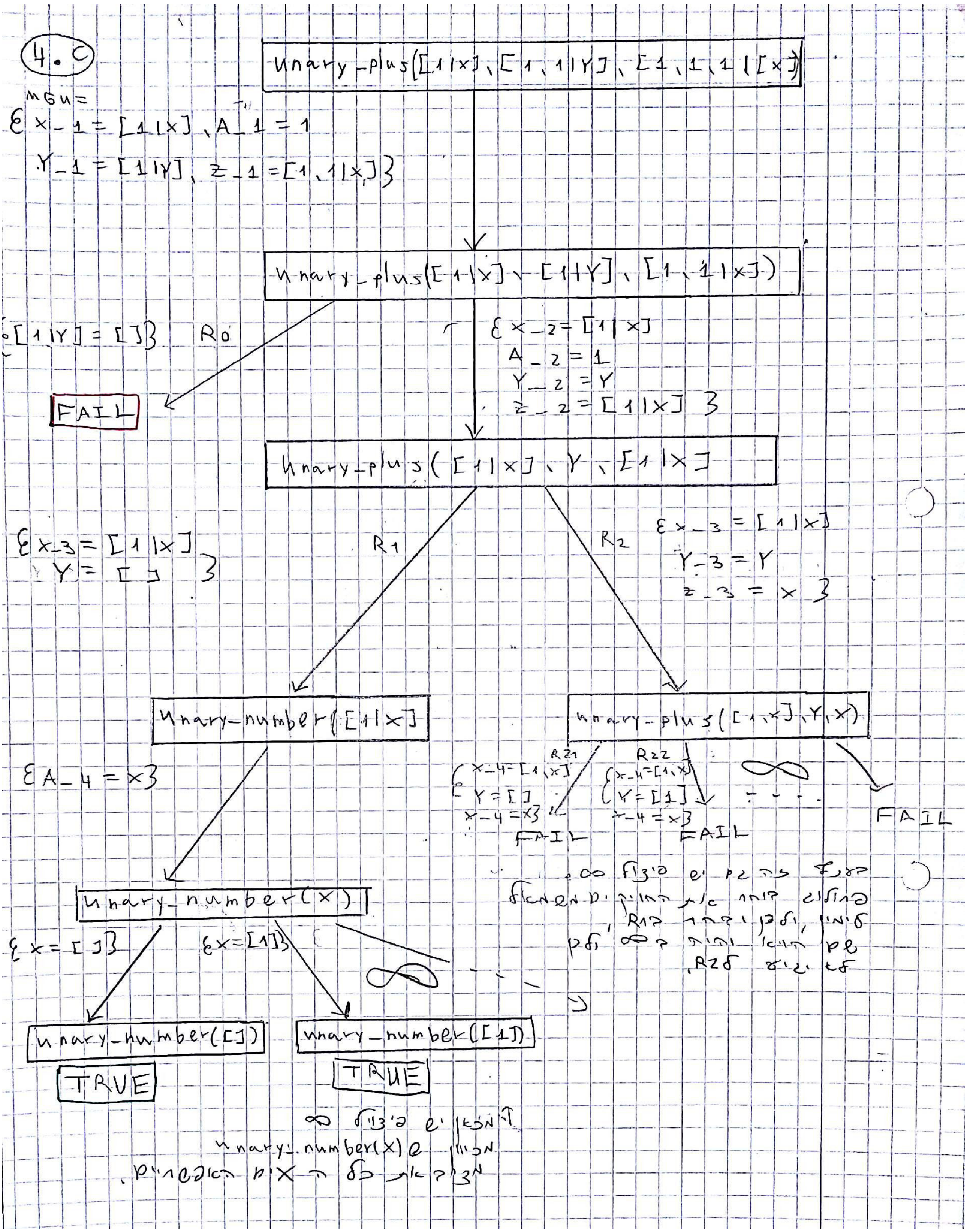
[1] Draw the proof tree for the query below and the given program. For success leaves, calculate the substitution composition and report the answer at each success leaf.

?- unary_plus ([1|X], [1,1|Y], [1,1,1|X]).

[2] Is this a success or a failure proof tree?

[3] Is this tree finite or infinite?

Answer:



[2].this is a success proof tree because it has at least 1 success path (infinite) in it as drawn above.

[3].this tree is infinite as explained in the graph.