Assignment 4

Question 1 – CPS, Section b:

Proof: `pipe$` is CPS-equivalent to `pipe`.

We will proof it by induction on the length of fs list (pipe$ parameter).
To Prove that pipe$ is CPS-equivalent to pipe we need to show that for any list of unary functions $fs = (list\ f_1\$ \dots f_n\$)$, their CPS equivalents $f_1 \dots f_n$, and a continuation function $cont$, the following equivalence holds: $(cont\ (pipe\ \ fs))\ =\ (pipe\$\ fs\ cont)$.

Proof by induction on the length of fs list (pipe$ parameter).

Base case: if fs of length *1* , the value of $(cont\ (pipe\ \ fs)) =\ (cont\ (pipe\ \ (f1\$\ .'(\ )\ )))$ is $\big(cont\ (\text{car fs})\big) =\ (cont\ f1\$)$ since $(cdr\ fs)$ is empty.
The value of $(pipe\$\ (fs'\ cont))$ is $(cont\ (lambda\ (x\ next - cont)\ ((car\ fs')\ x\ next - cont)))$ = $(cont\ (lambda\ (x\ next - cont)\ (f_1\$\ x\ next - cont)))$ = $(cont\ f1\$)$, and therefore $((cont\ (pipe\ \ fs))\ =\ (pipe\$\ fs\ cont)$.

Induction assumption: We assume the equivalence holds for input list fs of length n, and show that the equivalence $(cont\ (pipe\ \ fs'))\ =\ (pipe\$\ fs'\ cont)$ holds for $fs' = (list\ f_1\$ \dots f_n\$\ f_{n+1}\$)$.

Induction step: : if fs' of length *n+1* , $(cdr\ fs)$ is not empty and the value of $(pipe\ \ fs'cont\ )$ is
$(pipe\$\ (cdr\ fs')\ (lambda\ (res)\ (compose\$\ (car\ fs')\ res\ cont)))$
=$(pipe\$\ (list\ f_2\$ \dots f_n\$\ f_{n+1}\$)\ (lambda\ (res)\ (compose\$\ f_1\$\ \ res\ cont))))))$
=$((cont\ (pipe\ \ (list\ f_2\$ \dots f_n\$\ f_{n+1}\$)))\ (lambda\ (res)\ (compose\$\ f_1\$\ \ res\ cont))))))$

By the definition of `compose$`, this is equivalent to:
=$((cont\ (pipe\ \ (list\ f_2\$ \dots f_n\$\ f_{n+1}\$)))\ (lambda\ (res)$

$(lambda\ (f_1\$\ res\ cont)$

$(cont\ (lambda\ (x\ next\_cont)\ (f_1\$\ \ x\ (lambda\ (res\_compose)\ (res\ res\_compose\ next\_cont))))))$

By the induction assumption:

$((lambda\ (res)\ (cont\ (lambda\ (x\ next\_cont)\ (f_1\$\ \ x\ (lambda\ (res\_compose\ )$

$(res\ res\_compose\ next\_cont))))))\ (pipe\ (list\ f_2\$ \dots f_n\$\ f_{n+1}\$)))$

Which simplifies to
$(cont\ (lambda\ (x\ next\_cont)\ ((pipe\ (list\ f_2\$\ \dots\ f_n\$\ f_{n+1}\$))\ x\ (lambda\ (res\_compose)$

$(res\ res\_compose\ next\_cont))))\ =\ (cont\ (pipe\ \ (list\ f_1\$ \dots f_n\$\ f_{n+1}\$))$

Thus, $(pipe\ \ fs'cont\ )\ =\ (cont\ (pipe\ \ fs')$.

Therefore, by induction, we provrd that: $(cont\ (pipe\ \ fs))\ =\ (pipe\$\ fs\ cont)$ for any list of unary functions $fs = (list\ f_1\$ \dots f_n\$)$.

Question 2 - Lazy lists

d. Use cases of reduce1-lzl: when we want to combine all elements of a lazy list into a single result. This function processes the entire lazy list recursively and therefore won't stop for infinite lazy lists. Hence we can use it if you want to preform reduce on final lazy lists.

We will notice that in very long and finite lazy list we have to go through all the elements recursively, so it is possible that one of the other functions will be more efficient for certain purposes (For example, if you don't have to include all the elements and an approximation is enough).

Use cases of reduce2-lzl: when we want to combine only the first n elements of a lazy list into a single result. This function stops processing after n elements and can be used in cases where we have seen that reduce1-lzl is less useful.

Cases in which reduce2-lzl will be useful when we want to preform reduce on infinite or very long lazy lists but it's not necessary to preform reduce on all the elements, the first n are enough (for example- approximately the value of a long and even infinite polynomial).

Use cases of reduce3-lzl: when we want to generate a new lazy list where each element is the reduce result of all previous elements of the input lazy list. reduce3-lzl also can be used by us in cases where we have seen that reduce1-lzl is less useful because it doesn't calculate the input list elements entirety and can therefore handle infinite input lists and also be more efficiently for very long lists which not required to calculate entirety.

Cases in which reduce3-lzl will be useful are similar to reduce2-lzl but in cases where we do not know in advance how many elements we will have to preform reduce on, for example, if we look for an approximation up to a certain value, it is efficient and comfortable to use reduce3-lzl and not to perform a reduce2-lzl many times. But if we know for sure that we want to perform on reduce the first n elements, it is efficient and convenient to use reduce2-lzl and not call cbr n times on reduce3-lzl output and then prevent the creation of unnecessary closures.

g. The number pi is infinite and therefore it is required to make sure in the implementation that it is defined where to stop in its calculation. pi-sum implementation deal with it by receiving two parameters, a - the initial value and b - which defines the level of approximation. In our implementation we create a lazy list (which each time calculates the next element).

An advantage of the current implementation: flexibility in the level of approximation – in generate-pi-approximations the level of approximation can be improved if needed more effectively than in pi-sum. In cases where we don't know in advance which approximation is required or often needs to improve the approximation, pi-sum recalculation all the previous approximations each call (with different level of approximation b) and in generate-pi-approximations it is not required since there is no determination of level of approximation in advanced.

Disadvantage in the current implementation: Calculating unnecessary clousers - every approximation is required in the creation of clousers (while calculating lambda - the second element in the lazy list pair) which can be heavy, while in the implementation we saw in the lecture, the same clouser is activated with different parameters. Therefore, in cases where an update of the approximation level is not required and the level is known in advance, there is an advantage to using pi-sum.

Question 3 - Logic programing –

3.1 Unification:

We denote the Substitution by S.

1. unify[x(y(y), T, y, z, k(K), y), x(y(T), T, y, z, k(K), L)]
   Initial: Substitution - S = {} , A=x(y(y),T,y,z,k(K),y), B=x(y(T),T,y,z,k(K),L).
   Both have 'x' as the outermost term and the same number of parameters, so we proceed to compare their arguments.

   Compare arguments pairs:
   - Equation: y(y)= y(T) - both have 'y' as the outermost function and the same number of parameters, therefore T=y and hence:
     S = {T=y}
     Apply S to both A and B: AoS=x(y(y),y,y,z,k(K),y)
     BoS=x(y(y),y,y,z,k(K),L)
   - Equation: y= y. Arguments y and y are identical and the equation y=y is true.
   - Equation: y= y. Arguments y and y are identical and the equation y=y is true.
   - Equation: z = z. Arguments z and z are identical and the equation z=z is true.
   - Equation: k(K) = k(K). Both have 'y' as the outermost function, therefore K=K - identical.
   - Equation: L=y
     S = {T=y, L=y}
     Apply S to both A and B: AoS=x(y(y),y,y,z,k(K),y)
     BoS=x(y(y),y,y,z,k(K),y)

<u>**Result:**</u> S = {T=y, L=y}.

2. unify[f(a, M, f, F, Z, f, x(M)), f(a, x(Z), f, x(M), x(F), f, x(M))]
   Initial: Substitution - S = {} , A= f(a,M,f,F,Z,f,x(M)), B= f(a,x(Z),f,x(M),x(F),f,x(M)).
   Both have 'f' as the outermost term and the same number of parameters, so we proceed to compare their arguments.

   Compare arguments pairs:
   - Equation: a = a. Arguments a and a are identical and the equation a=a is true.
   - Equation: M=x(Z)
     S = {M=x(Z)}
     Apply S to both A and B: AoS=f(a,x(Z),f,F,Z,f,x(x(Z)))
     BoS=f(a,x(Z),f,x(x(Z)),x(F),f,x(x(Z)))

   - Equation: f = f. Arguments f and f are identical and the equation f=f is true.
   - Equation: F = x(x(Z))
     S = {M= x(Z) F = x(x(Z))}
     Apply S to both A and B: AoS=f(a,x(Z),f,x(x(Z)),Z,f,x(x(Z)))
     BoS=f(a,x(Z),f,x(x(Z)),x(x(x(Z))),f,x(x(Z)))
   - Equation: Z = x(x(x(Z))) - **fail** (as Z on both sides of the equation).

<u>**Result:**</u> fails on occurs check

3. unify[t(A, B, C, n(A, B, C),x, y), t(a, b, c, m(A, B, C), X, Y)]

Initial: Substitution - S = {} , A= t(A,B,C,n(A,B,C),x,y), B= t(a,b,c,m(A,B,C),X,Y).
Both have 't' as the outermost term and the same number of parameters, so we proceed to compare their arguments.

Compare arguments pairs:

- Equation: A = a
  S = {A = a}
  Apply S to both A and B: AoS=t(a,B,C,n(a,B,C),x,y)
  BoS=t(a,b,c,m(a,B,C),X,Y)
- Equation: B = b
  S = {A=a, B=b}
  Apply S to both A and B: AoS=t(a, b,C,n(a, b,C),x,y)
  BoS=t(a,b,c,m(a, b,C),X,Y)
- Equation: C = c
  S = {A=a, B=b, C=c}
  Apply S to both A and B: AoS=t(a,b,c,n(a,b,c),x,y)
  BoS=t(a,b,c,m(a,b,c),X,Y)
- Equation: n(a,b,c)=m(a,b,c) – **fail** (different function symbols)

Result: fails on occurs check

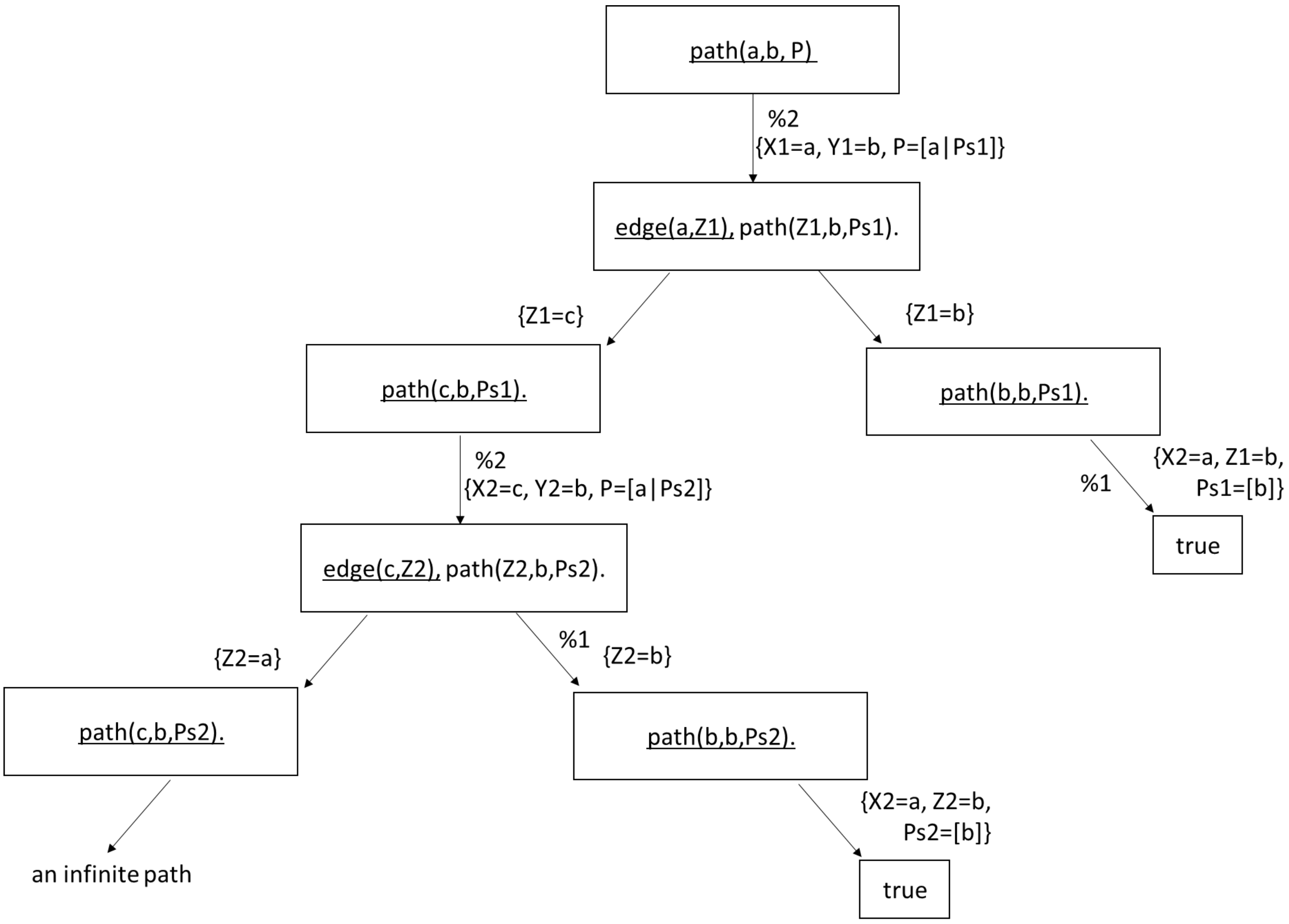
4. unify[z(a(A, x, Y), D, g), z(a(d, x, g), g, Y)]
   Initial: Substitution - S = {} , A=z(a(A, x, Y), D, g), B=z(a(d, x, g), g, Y).
   Both have 'z' as the outermost term and the same number of parameters, so we proceed to compare their arguments.

   Compare arguments pairs:

   - Equation: a(A, x, Y)= a(d, x, g) - both have 'a' as the outermost function and the same number of parameters, therefore we proceed to compare their arguments:
     - Equation: A=d
       S = {A=d}
       Apply S to both A and B: AoS=z(a(d, x, Y), D, g)
       BoS=z(a(d, x, g), g, Y)
     - Equation: x = x Arguments x and x are identical and x=x is true.
     - Equation: Y=g
       S = {A=d, Y=g}
       Apply S to both A and B: AoS=z(a(d, x, g), D, g)
       BoS=z(a(d, x, g), g, g)
   - Equation: D=g
     S = {A=d, Y=g, D=g}
     Apply S to both A and B: AoS=z(a(d, x, g), g, g)
     BoS= z(a(d, x, g), g, g)
   - Equation: g = g. Arguments g and g are identical and the equation g = g is true.

Result: S={A=d, Y=g, D=g}

3.3 Proof tree:



The tree includes at least one success path (and in practice more) and is therefore a success tree.

Also, the tree includes an infinite path and is therefore an infinite tree (all the paths that start at a, reach c and return to a in the tree, and there are infinite ones because they can go and return from a to c an infinite number of times).