PPL Assignment 4

By: cijhho123 and T.K.

Question 1b

We'll prove that **pipe$** is CPS-equivalent to **pipe**.

procedure **pipe$** is CPS-equivalent to procedure **pipe** if for every possible input **x1…xn** and for every continuation function **cont** the following equality holds:

(pipe$ \`(x1…xn) cont) = (cont (pipe \`(x1…xn)))

We'll prove it with induction.

Base case: (n=1)
(pipe$ \`(x1) cont) = (cont (pipe \`(x1))) by continuation's definition and from our code in the **pipe$** procedure.

Induction Step:
Let's assume that the statement is correct for some n >= 1, i.e. for every pipe with **n** elements:

(pipe$ \`(x1…xn) cont) = (cont (pipe \`(x1…xn)))

We'll show that the statement holds for n+1.

From out **pipe$** implementation, the following equality will hold:
(pipe$ \`(x1…xn+1) cont) = (pipe$ \`(x2…xn+1) cont2)

For **cont2** = (res$) => (cont (composite$ x1 $res id)) = cont(composite (x1 res)).

From our induction assumption, we can use the following equality:

(pipe$ \`(x2…xn+1) cont2) = (cont2 (pipe \`(x2…xn+1)))

And get:
(pipe$ \`(x1…xn+1) cont) => (pipe$ implementation)
(pipe$ \`(x2…xn+1) cont2) => (induction assumption)
(cont2 (pipe \`(x2…xn+1))) => (cont2 definition)
(composite (x1, (pipe \`(x2…xn+1))) => (pipe definition)
(composite (x1 , (xn+1 (xn (...(x3(x2))...))) => (composion's attribute)

(cont (xn+1 (xn (... (x3 (x2 (x1)))...))) = cont (pipe(x1,...,xn+1))
Proving that **pipe$** is indeed CPS-equivalent to **pipe**.

Question 2d

reduce1 will be used when we want to reduce elements from a lazy list, without knowing how many elements there are in said list.

reduce2 we will use when we know in advance how many elements of a lazy list we would like to reduce.

reduce3 we will use when we want to reduce an infinite lazy list or when we want to access each reduced member of the list.

Question 2g

The function taught in class to approximate pi is recursive without lazy lists, which makes computing the next items in the sequence faster, but more memory intensive.

While our implementation utilizes lazy lists, which hold only the current item in memory making computations a little slower, but a lot more scalable.

Question 3.1

1) unify[x(y(y), T, y, z, k(K), y), x(y(T), T, y, z, k(K), L)]

Eq1: x(y(y), T, y, z, k(K), y) = x(y(T), T, y, z, k(K), L)
Eq1 = Eq1 o {} = Eq1
add y(y) = y(T), … y=L to equations

Eq2: y(y) = y(T)
…
add T = y to equations

Eq3: T=y
add T = y to sub.

Eq4 : T = T
Eq4 o sub -> y = y
no change.

Eq5: y = y
no change

Eq6: z = z
no change

Eq7: k(K) = k(K)
add K = K to equations

Eq8: y = L
add L = y to substitution.

Eq9: K= K
no change

Final result: x(y(y), y, y, z, k(K), y)

MGU: {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))]

we remove truisms:

a = a => TRUE
M = x(Z)
f = f => TRUE
F = x(M)
Z = x(F)
f = f => TRUE
x(M) = x(M) => TRUE

acc = {}
M = x(Z),
F = x(M),
Z = x(F)
—
acc o M = x(Z) => acc = {M = x(Z)}
F = x(M),
Z = x(F)

—
{F = x(M)} o acc => {F = x(x(Z))}
acc o {F = x(x(Z))} => acc = {M = x(Z), F = x(x(Z))}
Z = x(F)

—
{Z = x(F)} o acc => {Z = x(x(x(Z)))}
acc o {Z = x(x(x(Z)))} => result in an ERROR

MGU: None.

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

acc = {}
t(A, B, C, n(A, B, C),x, y) = t(a, b, c, m(A, B, C), X, Y)
—
acc = {}
A = a
B = b
C = c
n(A, B, C) = m(A, B, C)
x = X
y = Y
—
acc = {
A = a,
B = b,
C = c,
x = X,
y = Y
}
n(A, B, C) = m(A, B, C)

—-
{n(A, B, C) = m(A, B, C)} o acc => { n(a, b, c) = m(a, b, c) }
acc o { n(a, b, c) = m(a, b, c) } => ERROR (predicate type mismatch)

MGU: None.

4) unify[z(a(A, x, Y), D, g), z(a(d, x, g), g, Y)]

acc = {}
z(a(A, x, Y), D, g) = z(a(d, x, g), g, Y)]
—
acc = {}
a(A, x, Y) = a(d, x, g)
D = g
g = Y
—
acc = {}
D = g
g = Y
A = d
x = x
Y = g

—--
{D = g} o acc => {D = g}
acc o {D = g} => acc = {D = g}

g = Y
A = d
x = x
Y = g
—---
{g = Y} o acc => {g = Y}
acc o {g = Y} => acc = {D = Y, g = Y}
A = d
x = x
Y = g

—
{A = d} o acc => {A = d}
acc o {A = d} => acc = {D = Y, g = Y, A = d}
x = x
Y = g
—
acc = {D = Y, g = Y, A = d}
Y = g

—-
acc = {D = Y, g = Y, A = d}
Y = g
{Y = g} o acc => {Y = Y}
acc o {Y = Y} => acc = {D = Y, g = Y, A = d, Y = Y}

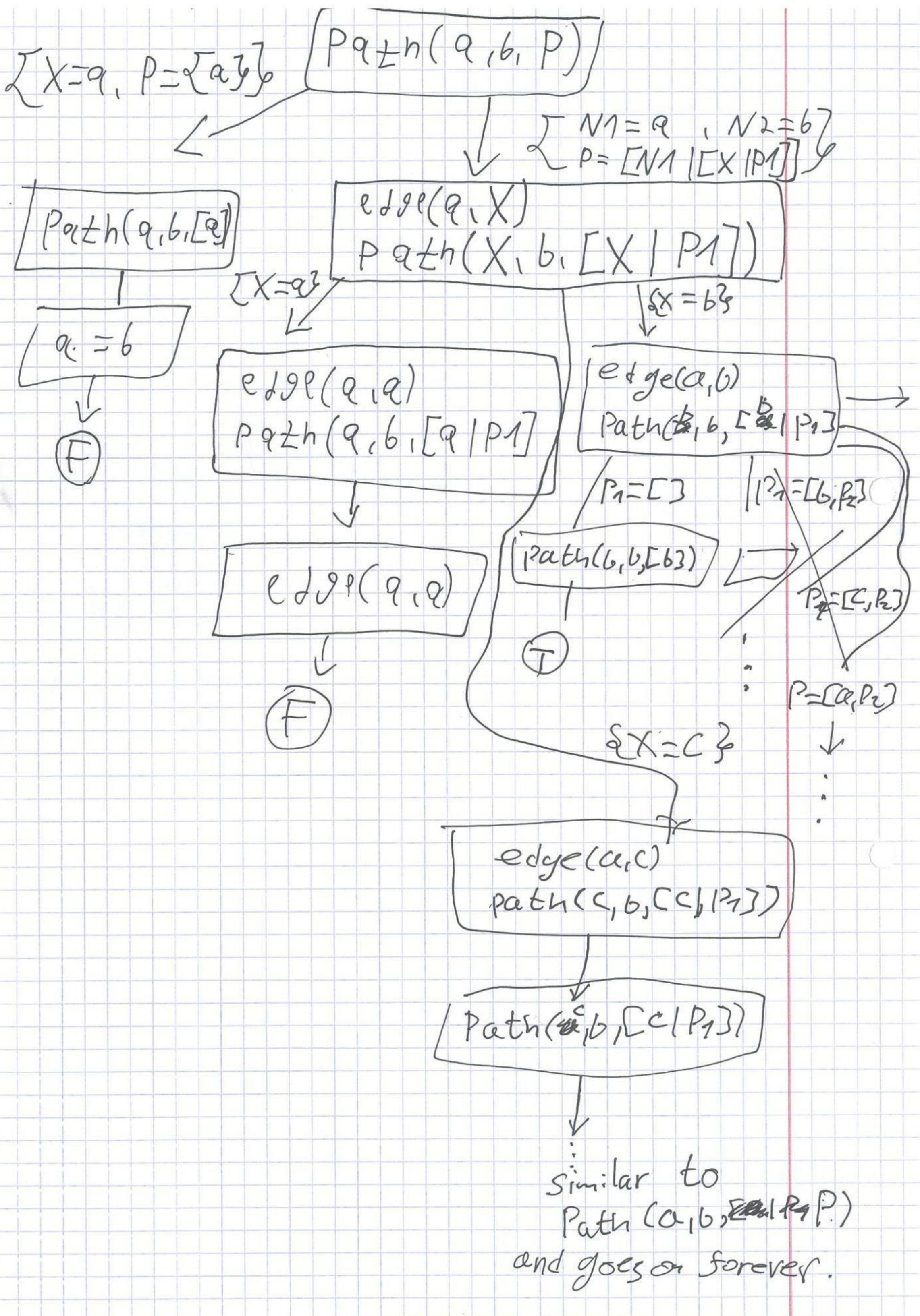
MGU: {D = Y, g = Y, A = d, Y = Y}

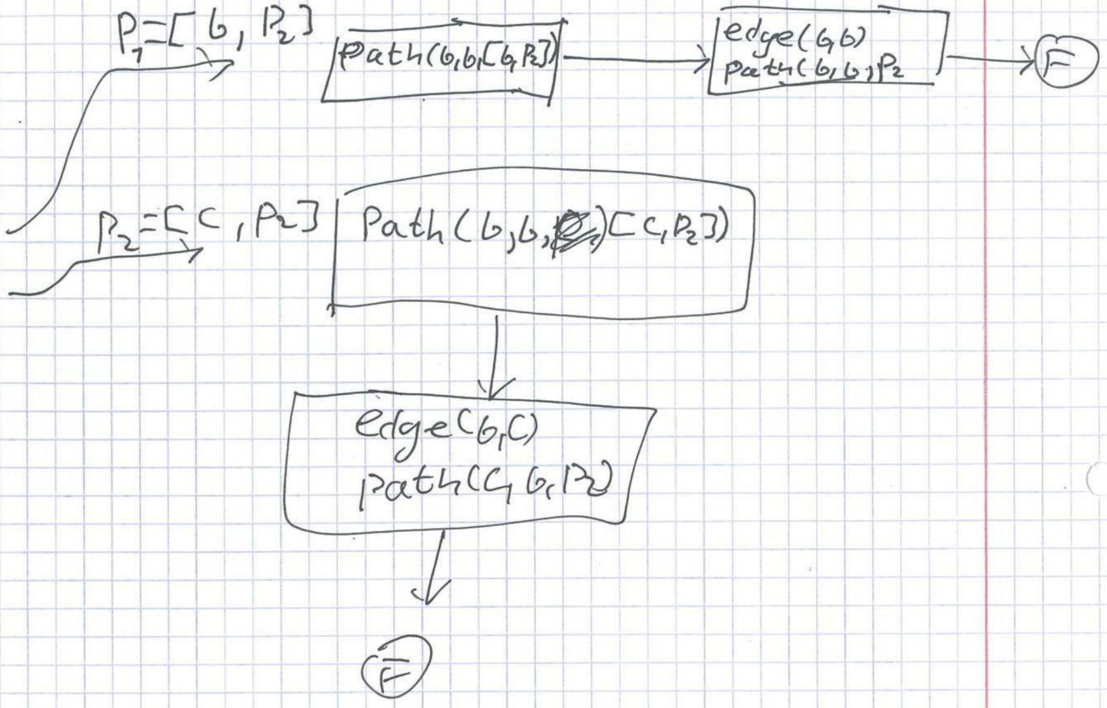
task 3C

The tree is an infinite tree, because it contains a loop (a,c,a) which can be extended forever.

The tree is a success tree because it has at least one (and infinitely many in general) paths in the tree to a TRUE.

Here’s a tree diagram: (next page)


Path(a, b, P)
{X=a, P=[a]}
{N1=a, N2=b
P=[N1|[X|P1]]}
Path(a,b,[a])
a = b
F
edge(a, X)
path(X, b, [X | P1])
{X=a}
edge(a, a)
path(a, b, [a | P1]
edge(a, a)
F
{X=b}
edge(a, b)
Path(b, b, [b | P1]
P1=[]
path(b, b, [b])
T
P1=[b, P2]
P2=[c, P2]
P=[a, P2]
{X=c}
edge(a, c)
path(c, b, [c | P1])
Path(c, b, [c | P1])
similar to
Path(a, b, P)
and goes on forever.


P1=[b, P2]
Path(b,b,[b,P2])
edge(b,b)
path(b,b,P2
F
P2=[c, P2]
Path(b,b,[c,P2])
edge(b,c)
path(c,b,P2)
F