Assignment 4

Question 1- CPS [20 points]

b. Prove that pipe$ is CPS-equivalent to pipe (8 points)

Given a function and its CPS version we will say that the functions are CPS-equivalent if for each series of operands and a continuation function cont:

Following this definition, we will show that compose$ is equivalent to compose:

(compose f g) is a function h such that h(x) = g(f(x))

So : g(f(x))

(compose$ f$ g$ cont) is a function h$ such that (h$ x cont2) = f$ x (lambda (f-res) (g$ f-res cont2)). Which is g(f(x)).

So compose and compose$ are indeed CPS-equivalent.

Now, we will show that pipe$ is equivalent to pipe, by induction on n = the size of the list (fs).

Assumption: fs$ is a list of equivalent CPS functions to fs.

To prove that pipe$ is CPS-equivalent to pipe. Specifically, for any list of unary functions fs and a final continuation cont, the result of (pipe$ fs cont) should be equivalent to applying cont to the result of (pipe fs).

**Base case:**

When fs contains only one function, fs = (list f):

Let be the function that pipe receives as an argument.

pipe: (pipe (list f)) ==> f

pipe$: (pipe$ (list f) cont) ==> (cont f)

In the base case, both pipe and pipe$ produce the same result: f. For pipe$, the result is passed to the continuation cont.

**Induction step:**

Assume that pipe$ works correctly for a list of n functions, i.e., for fs = (f1 f2 ... fn):

(pipe$ (list f1 f2 ... fn) cont) == cont((pipe (list f1 f2 ... fn)))

We need to show that pipe$ works for n+1 functions, i.e., for fs = (f1 f2 ... fn fn+1):

(pipe$ (list f1 f2 ... fn fn+1) cont)

Using the definition of pipe:

(pipe (list f1 f2 ... fn fn+1)) == (compose fn+1 (pipe (list f1 f2 ... fn)))

Using the definition of pipe$:

(pipe$ (list f1 f2 ... fn fn+1) cont)

==> (pipe$ (cdr (list f1 f2 ... fn fn+1)) (lambda (pipe-res) (compose$ (car (list f1 f2 ... fn fn+1)) pipe-res cont)))

==> (pipe$ (list f2 ... fn fn+1) (lambda (pipe-res) (compose$ f1 pipe-res cont)))

By the inductive hypothesis, pipe$ works for n functions:

(pipe$ (list f2 ... fn fn+1) (lambda (pipe-res) (compose$ f1 pipe-res cont)))

==> ((lambda (pipe-res) (compose$ f1 pipe-res cont)) (pipe (list f2 ... fn fn+1)))

==> (compose$ f1 (pipe (list f2 ... fn fn+1)) cont)

Since compose$ f1 is the CPS equivalent of compose f1, we have:

(compose$ f1 (pipe (list f2 ... fn fn+1)) cont)

==> cont((compose f1 (pipe (list f2 ... fn fn+1))))

This matches the definition of pipe for n+1 functions:

(compose fn+1 (pipe (list f1 f2 ... fn))) == (pipe (list f1 f2 ... fn fn+1))

Thus, we have shown that for any list of functions fs and a final continuation cont, the result of (pipe$ fs cont) is equivalent to applying cont to the result of (pipe fs).

Question 2-

**d. For which cases will you use each of the above reduce1-lzl, reduce2-lzl and reduce3-lzl procedures?**

* **reduce1-lzl** We will use this to perform a specific operation on all the elements in the lazy list, for example summing all elements.
* **reduce2-lzl**: We will use this to perform a specific operation on the first n elements of the lazy list, for example summing the first 10 elements.
* **reduce3-lzl**: We will use this to generate a lazy list of cumulative results, such as creating a lazy list of cumulative sums.

**g. What is the advantage and the disadvantage of generate-pi-approximations implementation, w.r.t. the pi-sum implementation taught in class.**

The generate-pi-approximations function has the advantage of lazy evaluation, which saves memory and gives continuous updates for π.

However, it can be slower and more complex because of extra function calls. The pi-sum function is simpler and faster for fixed ranges but does not save memory or provide continuous updates.

Question 3- Logic programing

3.1 Unification

What is the result of these operations? Provide algorithm steps and explain in case of failure.

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

A = x(y(y), T, y, z, k(K), y)

B = x(y(T), T, y, z, k(K), L)

S = {T=y}

AoS = x(y(y), y, y, z, k(K), y)

BoS = x(y(y), y, y, z, k(K), L)

S = {T=y, L=y}

AoS = x(y(y), y, y, z, k(K), y)

BoS = x(y(y), y, y, z, k(K), y)

**Answer: {T=y, L=y}**

1. **unify[f(a, M, f, F, Z, f, x(M)), f(a, x(Z), f, x(M), x(F), f, x(M))]**

A = f(a, M, f, F, Z, f, x(M))

B = f(a, x(Z), f, x(M), x(F), f, x(M))

S = {M=x(Z)}

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)))

S = {M=x(Z), F= x(x(Z))}

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)))

S = {M=x(Z), F= x(x(Z)), Z = x(x(x(Z)))} - **substitution to the equation failed:**

**Z= x(x(x(Z)))- this creates circularity and therefore unification is not possible**.

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

G = t(A, B, C, n(A, B, C),x, y)

E = t(a, b, c, m(A, B, C), X, Y)

S = {a=A}

GoS = t(a, B, C, n(a, B, C),x, y)

EoS = t(a, b, c, m(a, B, C), X, Y)

S = {A=a, B=b}

GoS = t(a, b, C, n(a, b, C),x, y)

EoS = t(a, b, c, m(a, b, C), X, Y)

S = {A=a, B=b, C=c}

GoS = t(a, b, c, n(a, b, c),x, y)

EoS = t(a, b, c, m(a, b, c), X, Y)

S = {A=a, B=b, C=c, m=n}

**Answer: substitution to the equation failed: n(a,b,c) = m(a,b,c). Case 3- both complicated phrases but with different structure (m != n). This creates an error and therefore unification is not possible**

1. **unify[z(a(A, x, Y), D, g), z(a(d, x, g), g, Y)]**

B = z(a(A, x, Y), D, g)

C = z(a(d, x, g), g, Y)

S = {A=d}

BoS = z(a(d, x, Y), D, g)

CoS = z(a(d, x, g), g, Y)

S = {A=d, Y=g}

BoS = z(a(d, x, g), D, g)

CoS = z(a(d, x, g), g, g)

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

BoS = z(a(d, x, g), g, g)

CoS = z(a(d, x, g), g, g)

**Answer: {A=d, Y=g, D=g}**

**3.3 Proof tree**

Draw the proof tree for the query:

edge(a,b). %e1

edge(a,c). %e2

edge(c,b). %e3

edge(c,a). %e4

list([]). %l1

list([X|Xs]) :- list(Xs). %l2

path(Node1, Node2, [Node1 , Node2]) :- edge(Node1, Node2). %p1

path(Node1, Node2, [Node1 | Path]) :- %p2

edge(Node1, NextNode),

path(NextNode, Node2, Path).

* **Is it a finite or an infinite tree?**

The tree is infinite- There are infinite paths from a to b because you can go from a to c and back to a any number of times before eventually reaching b. Some of the paths include: (a,b), (a,c,b), (a,c,a,b), (a,c,a,c,b)…

* **Is it a success or a failure tree?**

The tree is a success tree because it has at least one path from the root to a node that returns true: the path that reaches [a,b].

?- path(a,b, P)

**Path(a,b,P)**

%p2 {N1=a, N2=b, P=P1}

%p1 {N1=a, N2=b, P=P1}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

edge(a, NextNode),

path(NextNode, b, P1)

edge(a,b)

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

%e1

%e1 {NextNode=b}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

%e2{NextNode=c}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

True **Path=[a,b]**

**[**

Path(c,b,P1)

Path(b,b,P1)

%p2 {N3=c,N4=b, P2=P1}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

%p1 and %p2

Don't apply

%p1 {N3=c,N4=b, P2=P1}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

edge(c, NextNode),

path(NextNode, b, P2)

edge(c,b)

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

false

**[**

%e3

True **Path=[a,c,b]**

**[**

%e4 {NextNode=a}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

%e3 {NextNode=b}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

Path(b,b,P2)

Path(a,b,P2)

%p1 {N5=a, N6=b, P3=P2}

%p2 {N5=c,N6=a, P3=P2}

path(Node1, Node2, [Node1], Path).

path(Node1, Node2, [Node1], Path).

)

**CONTINUES RECURSIVELY**

**.**

**.**

**.**

edge(a,b,P3)

**CONTINUES RECURSIVELY**

**.**

**.**

**.**

%e1

True **Path=[a,c,a,b]**