CS 558, Homework 2

Soumya Banerjee

October 12, 2010

1 Binary Trees

```
main :: IO()
main =
        do
                print (allpaths tree1)
                print (allpaths tree2)



data T = Leaf | Node T T
        deriving Show
data P = GoLeft P | GoRight P | This
        deriving Show


tree1 :: T
tree1 = Node (Leaf) (Node Leaf Leaf)
tree2 :: T
tree2 = (Node (Node Leaf Leaf) (Node Leaf Leaf))

allpaths :: T -> [P]
allpaths Leaf = [This]
allpaths (Node ltree rtree) =  [This] ++ (map GoLeft (allpaths ltree)) ++ (map GoRight (allpat
```

2 General Trees

```
main :: IO()
main =
        do
                print (foldTree (*) (id) 1 tree1)
                print (foldTree (+) (id) 0 tree1)

data Tree a = Node a [Tree a]
                deriving Show

tree1 :: Tree Int
tree1 = Node 5 [ Node 1 [], Node 2 [ Node 3 [], Node 4 [Node 6 []] ] ]


foldTree1 fNode fLeaf value ([Node x []]) = fLeaf x
foldTree1 fNode fLeaf value ([]) = value
foldTree1 fNode fLeaf value ((Node x (y:ys)):ls) = fNode (fNode x (fNode (foldTree1 fNode fLea
foldTree fNode fLeaf value (Node x ys)  = foldTree1 fNode fLeaf value [Node x ys]
```

```
mapTree f = foldTree ( \[(Node x (y:ys))] ls -> ((Node (f x) (y:ys) ):ls) ) ( \([Node x []]) -
```

3 Graphs

```
main :: IO()
main =
        do
                print (isTree graph1)
                print (isTree graph2)
                print (isDAG graph1)
                print (isDAG graph2)
                print (isDAG graph3)
                print (isRootedDAG graph1)
                print (isRootedDAG graph2)
                print (isRootedDAG graph3)
                print (calcDepth graph1)

type Graph = [(Int,Int)]


graph1 :: Graph
graph1 = [(1,2),(2,3),(1,4)]

graph2 :: Graph
graph2 = [(1,2),(2,3),(1,4),(4,2)]

graph3 :: Graph
graph3 = [(1,2),(2,3),(1,4),(4,2),(2,1)]

findVertices v zs = [y | (x,y) <- zs, x == v]

dfs :: [Int] -> Graph -> [Int]
dfs [] l = []
dfs (y:ys) l = [y] ++ (dfs (findVertices y l) l) ++ (dfs ys l)


removeDuplicates [] = []
removeDuplicates (x:xs) = x : (removeDuplicates (filter (\y -> not(x == y)) xs))


makeVertexList l = removeDuplicates ((fst (unzip l)) ++ (snd(unzip l)))
```

```
{- takes the vertex list and the original graph and returns the list of list of vertices reach
iterateDfs xs l = [[x] ++ (removeDuplicates(dfs (findVertices x l) l)) | x <- xs]


mysort [] = []
mysort (x:xs) = mysort y1 ++ [x] ++ mysort y2
                where
                        y1 = [p | p <- xs, p <= x]
                        y2 = [q | q <- xs, q > x]

{- will take the list of list of vertices reached by dfs from each vertex and the list of vert
connectedTest ls l2 = or[True  | l1 <- ls, (mysort l1) == (mysort l2)]

{- the graph is a tree if the graph is connected and the number of vertices is one less than t
isTree :: Graph -> Bool
isTree g = (length (vl)) == (1 + length (g)) &&  (connectedTest (iterateDfs vl g) vl)
                where
                        vl = makeVertexList g




isDAG :: Graph -> Bool
isDAG g = and[ not(v `elem` (dfs (findVertices v g)  g)) | v <- ( removeDuplicates ((fst (unzi

{- find a list of vertices in the graph with indegree 0 -}
findIndegree0 zs = removeDuplicates ([ x | (x,y) <- zs, not (x `elem` (map snd zs)) ])

{- if there is at least one vertex with outdegree 0 and there is one vertex from which all oth
isRootedDAG' g vs l = ((length vs) > 0) && (or [ (mysort ( [v] ++ (removeDuplicates(dfs (findV


isRootedDAG :: Graph -> Bool
isRootedDAG [] = True
isRootedDAG g = isRootedDAG' g (findIndegree0 g) (makeVertexList g)


dfsDepth [] v g dist = []
dfsDepth (u:us) v g dist
                | not (u == v) =  (dfsDepth (findVertices u g) v g (dist + 1)) ++ (dfsDepth us
                | otherwise = [dist]

formTuple r g = [(v,head(mysort (dfsDepth r v g 0))) | v <- (makeVertexList g)]

findRoot' g vs l = [v | v <- vs, (mysort ( [v] ++ (removeDuplicates(dfs (findVertices v g) g))
```

```
findRoot g = findRoot’ g (findIndegree0 g) (makeVertexList g)

calcDepth g = formTuple (findRoot g) g
```

4 Numbers

```
main :: IO()
main =
        do
                print (makeLongInt 123 10)
                print (evaluateLongInt (10,[1,2,3]))
                print (changeRadixLongInt (10,[1,2,3]) 8)
                print (changeRadixLongInt (10,[1,2,3]) 16)
                print (addLongInts (10,[1,2,3]) (3,[1]))
                print (multLongInts (10,[1,2,3]) (3,[2]))

type Numeral = (Int,[Int])

makeLongInt n r = (r, (makeLongInt’ n r))

makeLongInt’ 0 r = []
makeLongInt’ n r = (makeLongInt’ (n ‘div‘ r) r) ++ [n ‘mod‘ r]

evaluateLongInt (r,l) = foldr (\ (x,y) l -> x*y + l) 0 (zip (iterate (*r) 1) (reverse l))

myAdd a b = a + b

myMult a b = a * b

myIntegerMod n r
        | n < r = n
        | n == r = 0
        | otherwise = myIntegerMod (n - r) r

myIntegerDiv n r = myIntegerDiv’ n r 0

myIntegerDiv’ n r c
        | n < 0 = c - 1
        | otherwise = myIntegerDiv’ (n - r) r (c + 1)

withoutBuiltinMakeLongInt’ 0 r = []
withoutBuiltinMakeLongInt’ n r = (withoutBuiltinMakeLongInt’ (myIntegerDiv n r) r) ++ [myInteg

withoutBuiltinMakeLongInt n r = (r, (withoutBuiltinMakeLongInt’ n r))
```

```
withoutBuiltinEvaluateLongInt (r,l) = foldr (\ (x,y) l -> (myAdd (myMult x y) l) ) 0 (zip (ite

changeRadixLongInt (r1, l) r2 = makeLongInt (withoutBuiltinEvaluateLongInt (r1, l)) r2


myzip [] [] = []
myzip [] (x:xs) = (0,x):(myzip [] xs)
myzip (x:xs) [] = (x,0):(myzip xs [])
myzip (x:xs)(y:ys) = (x,y):(myzip xs ys)

addLongInts (r1,l1) (r2,l2) = if r1 >= r2 then (r1, (addLong (r1,l1) (changeRadixLongInt (r2,l

addLong (r,l1) (t,l2) = reverse (addLong' r  (reverse ( zip (reverse (myzip (reverse l1) (reve

addLong' r [] = []
addLong' r [((x,y),w)]
                | x + y >= r = [x + y - r, 1]
                | otherwise = [x + y]
addLong' r (((x1,y1),w1):(((x2,y2),w2):zs))
                | x1 + y1 >= r = ((x1 + y1 - r):(addLong' r  (((x2 + 1,y2),w2):zs)))
                | otherwise = ((x1 + y1):(addLong' r  (((x2,y2),w2):zs)))


mulDigit x [] r z = []
mulDigit x [(y1,c1)] r z
                | x * y1 + c1 >= r = [x * y1 + c1 - r,1]
                | otherwise = [x * y1 + c1]
mulDigit x ( (y1,c1):((y2,c2):ys) ) r z
                | x * y1 + c1 >= r  = ( (x * y1 + c1 -r):( mulDigit x ((y2,1):(ys)) r z ) )
                | otherwise  = ( (x * y1 + c1):( mulDigit x ((y2,0):(ys)) r z ) )

stripSecondNum [] l r z = []
stripSecondNum (x:xs) l r z = ((reverse (mulDigit x (reverse(zip l (iterate (id) 0))) r z)) ++

multLongInts' (r,l1) (t,l2) = foldr (addLongInts) (r,[0]) (zip (repeat r) (stripSecondNum (rev

multLongInts (r,l1) (t,l2)
                | r >= t = multLongInts' (r,l1) (changeRadixLongInt (t,l2) r)
                | otherwise = multLongInts' (changeRadixLongInt (r,l1) t) (t,l2)
```

5 HW 2.3 (Graphs)

5.1 1. Are all directed graphs representable in this fashion?

No, all directed graphs cannot be represented in this fashion. The edges are represented in the form [(Int,Int)] and one would run out of numbers to enumerate the vertices if the number of vertices exceeded

$2^{31} - 1$.

6 HW 2.6 (Programs)

foldr f e xs = foldl (flip f) e (reverse xs)

axioms:
foldr f v [] = v (Eq. 1)
foldr f v (x:xs) = f x (foldr f v xs) (Eq. 2)
foldl f v [] = v (Eq. 3)
foldl f v (x:xs) = foldl f (f v x) xs (Eq. 4)
flip f x y = f y x (Eq. 5)
reverse [] = [] (Eq. 6)
reverse (x:xs) = (reverse xs) ++ [x] (Eq. 7)
[] ++ ys = ys (Eq. 8)
(x:xs) ++ ys = x:(xs ++ ys) (Eq. 9)
Proof by calculation
Let xs = [x_0, x_1, x_2, ..., x_n] (Eq. 10)
LHS = foldr f e xs
= { by Eq. 10 }
foldr f e [x_0, x_1, x_2, ..., x_n]
= { by Eq. 2 definition of foldr }
f x_0 (foldr f v [x_1, x_2, ..., x_n])
= { by Eq. 2 definition of foldr }
f x_0 (f x_1 (foldr f v [x_2, ..., x_n]))
= { by Eq. 2 definition of foldr }
f x_0 (f x_1 (f x_2 (foldr f v [x_3, ..., x_n])))
= { after applying Eq. 2 $n - 2$ times}
f x_0 (f x_1 (f x_2 (.... (f x_n (foldr f v []))...)))
= { by Eq. 1 base case definition of foldr}
f x_0 (f x_1 (f x_2 (.... (f x_n e)...)))
RHS = foldl (flip f) e (reverse xs)
= { by Eq. 10 }
foldl (flip f) e (reverse [x_0, x_1, x_2, ..., x_n])
= { by Eq. 7 definition of reverse }
foldl (flip f) e ((reverse [x_1, x_2, ..., x_n]) ++ [x_0])
= { by Eq. 7 definition of reverse }
foldl (flip f) e (((reverse [x_2, ..., x_n]) ++ [x_1]) ++ [x_0])
= { by Eq. 7 definition of reverse }
foldl (flip f) e ((((reverse [x_3, ..., x_n]) ++ [x_2]) ++ [x_1]) ++ [x_0])
= { after applying Eq. 7 $n - 2$ times }
foldl (flip f) e ((((...... ((reverse [ ]) ++ [x_n]) ++ ....) ++ [x_2]) ++ [x_1]) ++ [x_0])
= { after applying Eq. 6 base case of reverse }
foldl (flip f) e ((((...... ([ ] ++ [x_n]) ++ ....) ++ [x_2]) ++ [x_1]) ++ [x_0])
= { after applying Eq. 8 base case of ++ }

foldl (flip f) e ((((...... ($[x_n]$ ++ $[x_{n-1}]$) ....) ++ $[x_2]$) ++ $[x_1]$) ++ $[x_0]$)
= { after applying Eq. 9 definition of ++ }
foldl (flip f) e ((((...... ($[x_n, x_{n-1}]$) ....) ++ $[x_2]$) ++ $[x_1]$) ++ $[x_0]$)
= { after applying Eq. 9 $n-1$ times }
foldl (flip f) e ($[x_n, x_{n-1}$, ..., $x_2, x_1, x_0]$)
= { after applying Eq. 4 definition of foldl }
foldl (flip f) (flip f e x_n) ($[x_{n-1}$, ..., $x_2, x_1, x_0]$)
= { after applying Eq. 5 definition of flip }
foldl (flip f) (f x_n e) ($[x_{n-1}$, ..., $x_2, x_1, x_0]$)
= { after applying Eq. 4 definition of foldl }
foldl (flip f) (flip f (f x_n e) x_{n-1}) ($[x_{n-2}$, ..., $x_2, x_1, x_0]$)
= { after applying Eq. 5 definition of flip }
foldl (flip f) (f x_{n-1} (f x_n e)) ($[x_{n-2}$, ..., $x_2, x_1, x_0]$)
= { after applying Eq. 5 definition of flip and Eq. 4 definition of foldl $n-4$ times }
foldl (flip f) (f x_3 (.... (f x_{n-1} (f x_n e)) ....)) ($[x_2, x_1, x_0]$)
= { after applying Eq. 4 definition of foldl }
foldl (flip f) (flip f (f x_3 (.... (f x_{n-1} (f x_n e)) ....)) x_2) ($[x_1, x_0]$)
= { after applying Eq. 5 definition of flip }
foldl (flip f) (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....))) ($[x_1, x_0]$)
= { after applying Eq. 4 definition of foldl }
foldl (flip f) (flip f (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....))) x_1) ($[x_1, x_0]$)
= { after applying Eq. 5 definition of flip }
foldl (flip f) (f x_1 (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....)))) ($[x_0]$)
= { after applying Eq. 4 definition of foldl }
foldl (flip f) (flip f (f x_1 (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....)))) x_0) ([])
= { after applying Eq. 5 definition of flip }
foldl (flip f) (f x_0 (f x_1 (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....))))) ([])
= { after applying Eq. 3 base case of foldl }
f x_0 (f x_1 (f x_2 (f x_3 (.... (f x_{n-1} (f x_n e)) ....))))
Hence LHS = RHS by the calculational proof method and foldr f e xs = foldl (flip f) e (reverse xs).