CA341 Assignment 2

Comparing Functional and Logic Programming

By Stephen McAleese, CASE3, 18921756

# Overview

The purpose of this assignment is to implement a binary search tree using a functional and logic programming language. I chose Haskell to implement the functional implementation and Prolog as the logic language.

Each implementation has the following features: a tree data structure to store the nodes of the binary search tree, an insert function, a search function and three types of traversal functions: preorder, inorder and postorder.

# Functional Implementation

In the Haskell implementation, the binary search tree is stored in a custom data structure called *Tree*.

data *Tree* a = Node a (*Tree* a) (*Tree* a)

| Null deriving(*Read*, *Show*)

New nodes can only be inserted into the tree recursively as functional programming languages support recursion but not iteration.

The function *insert* starts at the root node and recurses down the tree to where the new node should be inserted. If the element to be inserted is less than the root, it recurses left. If it is greater, it recurses right. The base case function is called when the function encounters an empty node and it returns a new node. Since the base case replaces an empty node with a new node, it inserts a new node into the tree.

insert :: *Ord* a => *Tree* a -> a -> *Tree* a

insert (Node v l r) x

| x < v = Node v (insert l x) r

| x > v = Node v l (insert r x)

| otherwise = Node v l r

insert \_ x = Node x Null Null

If we wanted to create a tree with three nodes, we could define it like this:

main = do

let tree = Node 8 Null Null

print $ insert (insert tree 4) 12

The problem here is that we have to manually insert each node which will become tedious if we want to insert a large number of nodes. The solution is to have a function that can insert a list of nodes into a tree recursively. In my implementation, the function *makeTree* does this:

makeTree :: *Ord* a => [a] -> *Tree* a

makeTree [] = Null

makeTree list =

let treeRoot = Node (head list) Null Null

in foldl insert treeRoot (tail list)

The function first creates a root node before recursively inserting the rest of the nodes with the built-in function *foldl*.

Searching the binary search tree is also done recursively. The function starts at the root node and searches left or right depending on whether the value to be searched is less than or greater than the value of the current node.

search :: *Ord* a => Tree a -> a -> Bool

search (Node b l r) a

| a < b = search l a

| a > b = search r a

| otherwise = True

search Null a = False

If the node being searched for exists in the tree, the search function will find it via recursion. When this happens, the function returns True because it has found a match. The function returns false when it reaches a Null node because this means the search value does not exist in the tree.

The preorder, inorder and postorder traversals traverse through the binary search tree recursively to build up a list of nodes:

preorder :: *Tree* a -> [a]

preorder Null = []

preorder (Node v l r) = [v] ++ preorder l ++ preorder r

inorder :: *Tree* a -> [a]

inorder Null = []

inorder (Node v l r) = inorder l ++ [v] ++ inorder r

postorder :: *Tree* a -> [a]

postorder Null = []

postorder (Node v l r) = postorder l ++ postorder r ++ [v]

The preorder traversal function adds nodes to the list from the top of the binary tree down. The inorder traversal adds the lowest node first and the highest last resulting in an ordered list. The postorder traversal is a bottom-up traversal of the binary tree as it always recurses as far as possible, both left and right, before adding a node to the list.

## Implementing Deletion

Here is how I implemented deletion in Haskell:

deleteFromList :: *Eq* a => a -> [a] -> [a]

deleteFromList a [] = []

deleteFromList a (x:xs)

| a == x = deleteFromList a xs

| otherwise = x : deleteFromList a xs

deleteFromTree :: *Ord* a => *Tree* a -> a -> *Tree* a

deleteFromTree (Node v l r) a =

makeTree nodes

where nodes = deleteFromList a (inorder (Node v l r))

The function *deleteFromTree* collects the nodes from the tree by doing an inorder traversal of it. It then uses the *deleteFromList* function to delete the node *a* from the list of nodes. Finally, it creates a brand new tree using this new list.

Since variables in Haskell are immutable, it is necessary to create a new and different tree.

# Logic Implementation

The Prolog implementation has the same functions as the Haskell implementation: makeTree, insert, search, preorder, inorder and postorder. Similarly, these functions are implemented recursively as Prolog does not support iterations (for or while loops).

As Prolog is a logic programming language, there are no functions, only predicates. A predicate is a statement which evaluates to true or false based on the values of the variables within it. As Prolog predicates cannot return any values, the only way to get values from a predicate is to ask the predicate to derive the value of an unknown parameter.

For example, below is the implementation of the *search* predicate in Prolog, which will return true if the value being searched for exists in the binary search tree supplied as a second parameter and false otherwise.

search(X, t(X,\_,\_), Y) :- Y = true.

search(\_, nil, Y) :- Y = false.

search(X, t(V,L,\_), Y) :- X < V, search(X, L, Y).

search(X, t(V,\_,R), Y) :- X > V, search(X, R, Y).

Because predicates cannot return a value, we need to derive a third parameter to store our result. This is done via the pattern matching. For example, if we call:

search(8, t(4, t(8, nil, nil), t(12, nil, nil)), Y)

We’re asking the predicate to derive the value Y which will either be true or false. The predicate will recurse until it becomes the line below:

search(8, t(8, nil, nil), Y)

At this point, the search predicate will recognize this pattern (the first of the four lines above) and derive the value of Y which will be true in this case. If a leaf node is encountered, this means that the value couldn’t be found causing the predicate to return false.

The insert predicate works in a similar way:

insert(X, nil, t(X, nil, nil)). % y1

insert(X, t(V, L, R), t(V2, L2, R2)) :-

X < V,

insert(X, L, N), % x1 (becomes y1)

(V2, L2, R2) = (V, N, R) ;

X > V,

insert(X, R, N),

(V2, L2, R2) = (V, L, N) ;

(V2, L2, R2) = (V, L, R).

The first argument *X,*  is the value to be inserted. The second argument is the input tree and the third argument is the output tree, the tree after having the new node inserted.

## Example

Here is the recursive call stack (first box below) for the insert predicate. The statement above labeled ‘x1’ is called repeatedly causing the predicate to recursively move down the left branch of the tree. It starts at 8, then goes to 4 and stops at the base case which is the left branch of 4 (which is nil). When it hits this base case, the statement labeled ‘y1’ above becomes true which causes a new node to be inserted left of node 4 in the tree as the second box below shows.

%A

insert(2, t(8, t(4, nil, nil), t(12, nil, nil)))

insert(2, t(4, nil, nil)) % go into left subtree

insert(2, nil, t(2, nil, nil)) % base case, insert

t(2, nil, nil) at %A

number to insert: 2

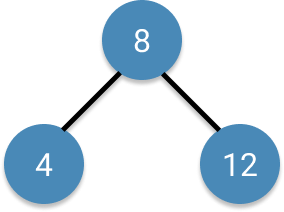
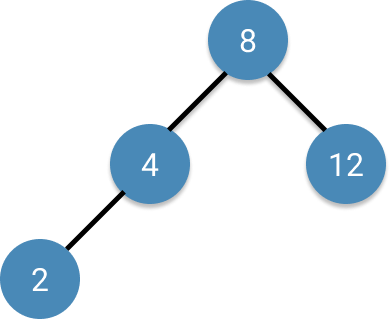
% V L R

t(8, t(4, nil, nil), t(12, nil, nil) % input tree

% V N R

t(8, t(4, t(2, nil, nil), nil), t(12, nil, nil) % output tree

**Input tree: Output tree:**



## Implementing Deletion

I implemented the deletion predicate in Prolog in a similar way to how I implemented the deletion function in Haskell:

deleteFromTree(X, T, Z) :-

inorder(T, NL), % create list of nodes

delete(NL, X, Y),

makeTree(Y, nil, Z). % create new tree

An inorder traversal collects nodes, deletes one of the nodes and creates a brand new tree from this new list.

# Comparison of the Two Implementations

Haskell is a purely functional programming language, meaning that all operations are carried out by applying functions to inputs to produce outputs. The Haskell implementation creates the root of the binary tree by creating an instance of the *Tree* data type and then recursively adding to it.

For example, given the node defined on line 1, a new node can be added to the left of that node by calling the insert function in place of its left node (line 2).

Node v l r -- line 1

Node v (insert l x) r -- line 2

What’s happening here is that the expression *(insert l x)* returns a value. In other words, the expression is evaluated. Haskell executes operations via nested expressions without any external variables which makes it stateless and without side effects (it doesn’t modify variables outside the function scope). Each expression evaluates and the value that results is used by the next function and so on until a final value is outputted by the function.

To modify the binary search tree, Haskell recursively finds the right place in the tree to put the expression  *(insert l x)* or *(insert r x).* This expression is evaluated to a value when the base case is reached and this value then becomes part of the tree.

Like Haskell, Prolog is also a declarative programming language where the programmer specifies what is needed rather than a list of exact instructions to get the desired output. However, whereas Haskell is a functional programming language, Prolog is a logic programming language.

Prolog does not have functions which return values and does not have expressions which are then evaluated. Instead, it only has facts and predicates which can either be true or false. Answers can be obtained from predicates by asking them to find the value of an unknown parameter.

For example, the insert predicate is defined as follows:

insert(X, nil, t(X, nil, nil)). % x4

insert(X, t(V, L, R), t(V2, L2, R2)) :-

X < V,

insert(X, L, N), % x2

(V2, L2, R2) = (V, N, R) ; % x3

X > V,

insert(X, R, N),

(V2, L2, R2) = (V, L, N) ;

(V2, L2, R2) = (V, L, R).

Like Haskell, the predicate is recursive and uses a base case to find the answer. But whereas Haskell uses a function call to obtain a value, Prolog instead uses a predicate to find the value of a parameter.

In the *insert* predicate above, the first two arguments of the predicate are known inputs and the final argument is the output. The first two arguments are supplied as defined inputs but the last argument is left undefined. It is left as a variable because we want the predicate to find its value.

The predicate above finds the value of the last argument using the base case on line ‘x4’. When *X* and *nil* are inputted to the base case, the third argument is inferred from the first two arguments. This third argument, in the form *N* on line ‘x2’ modifies part of the binary search tree on line ‘x3’. Once the right part of the binary search tree has been modified (after several recursive steps), Prolog then backtracks to the first function call so that the entire modified binary search tree is outputted in the form *t(V2, L2, R2)* which can then be outputted or used elsewhere.

the last parameter X is found from the first two inputs:

insert(4, t(8, nil, nil), X).

% X = t(8, t(4, nil, nil), nil)

Overall, Haskell and Prolog achieve their goals in similar ways. They are both declarative and recursive. Haskell accomplishes goals using expressions consisting of functions which are then evaluated to values. In contrast, Prolog is based on first-order logic. Instead of functions, it has recursive predicates which find the answer via reverse induction from one or more base cases. Known parameters are supplied to the predicate which can then find the value of the unknown parameter by induction. Whereas Haskell replaces the entire function call with the value that results from it, Prolog stores the answer found by a predicate as another parameter in that predicate.