CA341 - Assignment 2

Comparing Functional Programming and Logic Programming

Adrian Irwin 20415624
Afolabi Fatogun 20409054

Introduction

In this assignment, we were tasked to implement a program that performs the following operations on a given ordered binary tree in both a Functional and Logic programming language.

- insert: adds a given integer into the given binary tree
- search: returns true if a given integer is in the given binary tree
- preorder: lists the elements of the given binary tree in preorder traversal
- inorder: lists the elements of the given binary tree in inorder traversal
- postorder: lists the elements of the given binary tree in postorder traversal

For the functional and logical aspects of this comparative project, we chose to use Haskell and Prolog respectively.

Haskell is an easy to understand language which supports polymorphism, through operator overloading, and solves redundancy of code, because of this we found that the program was easy to understand after we had written it.

Prolog is a language primarily used for AI development. Logic is expressed as facts and rules in this language, and because querying in Prolog is easy to use in a terminal, it made Prolog a good choice for our comparison.

Functional (Haskell)

The functional programming implementation starts with a type definition of a binary tree as seen below.

```
data BinTree t = Empty | Root t (BinTree t) (BinTree t)
               deriving(Ord, Eq, Show)
```

This new data structure defines the basic structure of a Binary tree which is used throughout the program.

The first function, insert, takes an element and a Binary tree and returns a new Binary tree. It consists of a base case for adding to an empty tree and adding to a tree with a root where it would check if the element is greater or less than the value of the root and assign a position on the left or right branch using recursion.

```haskell
insert :: (Ord a) => a -> BinTree a -> BinTree a
insert x Empty = Root x Empty Empty
insert x (Root y left right)
    | x < y = Root y (insert x left) right
    | otherwise = Root y left (insert x right)
```

The next function search takes the same data types as insert but this time returns a boolean value depending on whether the element is found in the Binary tree. The first instance is a base case for if the Binary tree is empty in which case the Boolean expression is False. The next instance finds the value in a Binary tree with a root and unknown branches. The program checks if the value is equal to the root in which case the Boolean expression is True then depending on if the value is less or greater than the root calls the search function recursively on either branch.

```haskell
search :: (Eq a, Ord a) => a -> BinTree a -> Bool
search x Empty = False
search x (Root y left right)
    | x == y = True
    | x < y = search x left
    | x >= y = search x right
    | otherwise = False
```

The last 3 functions, preorder, postorder, and inorder, these functions are given a binary tree and are to return a list with the respective orders.
Each function has a base case for an empty Binary tree, where an empty list would be returned and using recursion and list concatenation put each one into its right order.

```haskell
preorder :: BinTree t -> [t]
preorder Empty = []
preorder (Root x left right) = [x] ++ preorder left ++ preorder right

postorder :: BinTree t -> [t]
postorder Empty = []
postorder (Root x left right) = postorder left ++ postorder right ++ [x]

inorder :: BinTree t -> [t]
inorder Empty = []
inorder (Root x left right) = inorder left ++ [x] ++ inorder right
```

Logic (Prolog)

The first clause, insert, has 3 rules. Each rule takes an integer and a binary tree, with the third argument being the resulting binary tree. The first rule is the base case for when we are given an integer and an empty tree. The second rule is for when the integer we are adding is smaller than the current root which recursively calls the insert function on the left subtree. The third rule acts the same as the second rule but compares and inserts on the right subtree instead.

```
insert(X, nil, t(X, nil, nil)).
insert(X, t(Y, L, R), t(Y, L1, R)) :- X < Y, insert(X, L, L1).
insert(X, t(Y, L, R), t(Y, L, R1)) :- X >= Y, insert(X, R, R1).
```

The next clause, search, has 3 rules. Each rule takes an integer and a binary tree. Instead of having the resulting binary tree being returned to us, we are instead returning a boolean depending on if the value is in the given binary tree. The first rule is our base case for when the integer we are given is the root of the given tree and will return True. The second and third rule is for when the integer we are given is not the root of the given tree but is either less or greater than the current root, in this case, search will be recursively called on the respective subtree. If the integer is not found in the tree False will be returned.

```
search(X, t(X, _, _)) :- !.
search(X, t(Y, L, _)) :- X < Y, search(X, L), !.
search(X, t(Y, _, R)) :- X >= Y, search(X, R), !.
```

The last three clauses, preorder, postorder, and inorder, are all clauses that list the given binary tree depending on their specific orders. Each clause has both a fact for when only nil is given to the clause stating that it is true and a base case rule for when there is a root but no subtrees.

Both preorder and postorder have only one extra rule that would recursively call the rule for both the left and right subtrees either before or after printing the current root depending on which order traversal is being used.

Inorder has 3 extra rules, the first and second rules are for when the given tree only has either a left or right subtree with the other being nil, the root for the subtree will be printed then inorder will be recursively called on the respective subtree. The final rule is used to recursively call inorder when both subtrees are not nil, inorder is recursively called on both the left and right subtree, while the

root of the given tree is printed.

```
preorder(nil).
preorder(t(X, nil, nil)) :- write(X), write(" "), !.
preorder(t(X, L, R)) :- write(X), write(" "), preorder(L), preorder(R), !.

postorder(nil).
postorder(t(X, nil, nil)) :- write(X), write(" "), !.
postorder(t(X, L, R)) :- postorder(L), postorder(R), write(X), write(" "),!.

inorder(nil).
inorder(t(X, nil, nil)) :- write(X), write(" "), !.
inorder(t(X, L, nil)) :- inorder(L), write(X), write(" "), !.
inorder(t(X, nil, R)) :- write(X), write(" "), inorder(R), !.
inorder(t(X, L, R)) :- inorder(L), write(X), write(" "), inorder(R), !.
```

Comparison

Haskell is a purely functional programming language which was designed for large-scale industrial applications.[1] It follows the functional programming paradigm which means that all computations are mathematical functions, this also means that it implements type classes, lazy evaluation and parametric polymorphism. Being a language that follows the functional programming paradigm, this also means that functions have no side effects.

Prolog is a logic programming language that has an important role in artificial intelligence [2] and is associated with computational linguistics. It follows the logic programming paradigm which means that all programs are written in terms of relations, under the names of facts and rules. [3]

Mathematical vs Logical

The biggest difference between both programming paradigms is that Haskell(Functional) uses mathematical functions while Prolog(Logic) uses logical expressions. A mathematical programming expression would return values while Logical expressions return Boolean values i.e True or False.

Similarities

- Both Programming paradigms fall under a declarative method which is a building structure that involves stating what the desired outcome of a task is before the program is developed. [4]
- In the two implementations, recursive methods were used extensively to build the programs.

Advantages/Disadvantages

Functional Programming

Advantages	**Disadvantages**
Shorter Code	Not suitable for all tasks
Code is easy to test	Recursion is not easy to understand and can lead to errors
Polymorphism is available to this programming paradigm	Variables cannot be changed after declaration

Logical Programming

Advantages	**Disadvantages**
It is easy to identify patterns due to the nature of logical programming	Logic programming does not support features to make large programming easy to manage
Built-in list handling in Prolog	Less efficiency
Shorter Code	

References

[1] “What is Haskell Programming Language?,” *GeeksforGeeks*, Feb. 27, 2022. https://www.geeksforgeeks.org/what-is-haskell-programming-language/ (accessed Nov. 25, 2022).

[2] “Prolog | An Introduction - GeeksforGeeks,” *GeeksforGeeks*, May 26, 2018. https://www.geeksforgeeks.org/prolog-an-introduction/ (accessed Nov. 25, 2022)

[3] “Dublin City University Loop: Log in to the site,” *loop.dcu.ie*. https://loop.dcu.ie/pluginfile.php/4621350/mod_resource/content/1/CA341_Logic_Programming_Paradigm.pdf (accessed Nov. 25, 2022)

[4] “What Is Declarative Programming? Definition from SearchITOperations,” *SearchITOperations*. https://www.techtarget.com/searchitoperations/definition/declarative-programming#:~:text=Declarative%20programming%20is%20a%20method (accessed Nov. 25, 2022)