CA341 Comparative Programming Languages

Assignment 2: Comparing Functional Programming and Logic Programming

CASE3

Student Name: David Weir

Student Number/ID: 19433086

18/11/2021

Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): David Weir

Programme: Comparative Programming Languages

Module Code: CA341

Assignment Title: Comparing Functional Programming and Logic Programming

Submission Date: 18/11/2021

Module Coordinator: Dr Brian Davis

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml , https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

Name(s): David Weir Date: 18/11/2021

Comparative

Introduction

For this assignment I was instructed to implement a binary search tree (BST) in a functional and logic programming paradigm. Each implementation supports the following operations:

- Insert
- Search
- Traversals: in-order, preorder and post-order

To complete this assignment, I decided to implement the logic BST in Prolog. I chose Prolog because of my previous experience using that language in my CA208 Logic from 2nd year. During this module I briefly had the opportunity to work with BSTs in Prolog meaning I was not completely inexperienced.

I used Haskell to implement the BST using a functional programming paradigm. Haskell was the obvious choice for me because I am currently learning how to program in Haskell in CA320 Computability and Complexity. This means my memory of Haskell is current. It also provides me with an opportunity to practice my skills in Haskell in a larger project.

Functional Paradigm

Functional programming is a programming paradigm which has a number of key concepts including: expressions, functions, parametric polymorphism, data abstractions and lazy evaluation. It is a declarative programming paradigm using expressions that are executed when the program is run and produce a value.

In functional programming data is immutable, some functions are pure functions meaning they are deterministic and do not have any side-effects. Functional languages are heavily dependent on recursion as there are no for or while loops. [3]

Logic Paradigm

Logic programming is a programming paradigm based on formal logic. Logic programs are a set of sentences/statements that express facts and rules. [2] These rules are written as logical horn clauses with a head and a body and follow a declarative approach. Facts are written similarly but are expressed without a body. [2]

Functional VS Logic

Seen below are some tests of my BST in both implementations using the same data and operations. As we can see both implementations compute the same values but we will have a deeper look into what the behind the scenes differences are in how both languages manage to do this.

Prolog:

```
?- insert(5,bt(10, bt(3,bt(1,nil,nil),nil), bt(20,nil,bt(30,bt(22,nil,nil),bt(43,nil,nil)))), X).
X = bt(10, bt(3, bt(1, nil, nil), bt(5, nil, nil)), bt(20, nil, bt(30, bt(22, nil, nil), bt(43, nil, nil)))) .

?- search(30,bt(10,bt(3,bt(1,nil,nil),nil),bt(20,nil,bt(30,bt(22,nil,nil),bt(43,nil,nil))))).
true .

?- inorder(bt(10, bt(3,bt(1,nil,nil),nil), bt(20,nil,bt(30,bt(22,nil,nil),bt(43,nil,nil)))), X).
X = [1, 3, 10, 20, 22, 30, 43].

?- preorder(bt(10, bt(3,bt(1,nil,nil),nil), bt(20,nil,bt(30,bt(22,nil,nil),bt(43,nil,nil)))), X).
X = [10, 3, 1, 20, 30, 22, 43].

?- postorder(bt(10, bt(3,bt(1,nil,nil),nil), bt(20,nil,bt(30,bt(22,nil,nil),bt(43,nil,nil)))), X).
X = [1, 3, 22, 43, 30, 20, 10].

?-
```

Haskell:

```
[1 of 1] Compiling Main             ( functional.hs, interpreted )
Ok, one module loaded.
ghci> insert 5 (Root 10 (Root 3 (Root 1 Empty Empty) Empty) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty))))
Root 10 (Root 3 (Root 1 Empty Empty) (Root 5 Empty Empty)) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty)))
ghci> search 30 (Root 10 (Root 3 (Root 1 Empty Empty) Empty) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty))))
True
ghci> inorder (Root 10 (Root 3 (Root 1 Empty Empty) Empty) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty))))
[1,3,10,20,22,30,43]
ghci> preorder (Root 10 (Root 3 (Root 1 Empty Empty) Empty) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty))))
[10,3,1,20,30,22,43]
ghci> postorder (Root 10 (Root 3 (Root 1 Empty Empty) Empty) (Root 20 Empty (Root 30 (Root 22 Empty Empty) (Root 43 Empty Empty))))
[1,3,22,43,30,20,10]
ghci>
```

To begin implementing the BST in both, you define what a BST is. In Haskell this is done using a user-defined data type as we can see below. This displays a BST in the form we see above. This data type is a polymorphic tree where t could be any value and derives 3 typeclasses.

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

In Prolog this is done using a fact.

```
bt(X, Left, Right).
```

Although both serve the same purpose, defining a tree with a root node and a left and right subtree, Prolog makes use of facts, predicate expressions that act as declarative statements. Whereas in Haskell I used the “data” keyword to define my own data type, BinTree.

After defining the BST I moved onto implementing an **insert** operation. In Haskell, I create an insert function that use the Ord typeclass and accepts a value to insert and a BST as parameters. This function will always evaluate to a new BST with the node inserted in its position. In Haskell, each parameter that is accepted has a type and the function will return a type (the last item in the declaration).

```
insert :: Ord t => t -> BinTree t -> BinTree t

insert a Empty = Root a Empty Empty
insert x (Root a left right)
    | x < a = Root a (insert x left) right
    | otherwise = Root a left (insert x right)
```

As we can see insert takes in a value t and a BST and evaluates to a BST. It first checks the base case, if the tree is empty, it creates a new tree with the value inserted at its root and 2 empty subtrees. If there is a left and/or right subtree then we use guards. If the inserted node is less than the root, we use recursion to insert the value in the left subtree. Otherwise (i.e. it is greater than the root) we recursively add the node to the right subtree.

In Prolog, we define an insert rule. We see an example of the insert operation in use above and the operation is defined below

```
insert(X, nil, bt(X, nil, nil)).
insert(X, bt(Y, Left, Right), bt(Y, Left1, Right)) :- X =< Y, insert(X, Left,
Left1).
insert(X, bt(Y, Left, Right), bt(Y, Left, Right1)) :- insert(X, Right, Right1).
```

Just like Haskell, the first rule is the base case. This base case is called when we are adding a node to an empty tree and will return the list with an inserted root and 2 empty subtrees. The final 2 rules are called if insert is passed on a non-empty tree. In this case, the first rule is called if the value to be inserted X is less than the root if so, we insert the node in the left subtree using recursion. Otherwise, recursive insert is called on the right subtree until the node is inserted.

As we can see the main difference is the function declaration in Haskell where you must declare the functions parameters and the parameters' types. This contrasts with Prolog's rules however, we can also see Haskell's function definition is nearly identical to Prolog and both languages depend heavily on recursion for iteration.

We then move on to the **search** operation that returns True if the value is in the tree and False if not. In Haskell, I create a function declaration with the Ord typeclass (for comparisons), a value t to check, a tree to search and returns the Boolean value. An example of the search operation in use in both languages can be seen above.

```
search :: Ord t => t -> BinTree t -> Bool

search x Empty = False
search x (Root a left right)
    | x == a = True
    | x < a = search x left
    | x > a = search x right
```

In Haskell, again we have a base case where we search an empty tree and this defaults to a False value. The second line in the definition is called if the tree is non-empty. We use Ord to check if the given value is equal to greater/less than the current root node. If they are equal we return True but otherwise we search the left subtree if it less than or right subtree if it is greater than, using recursion.

In Prolog, I defined 3 rules for search. The base case is used if the given BST in empty and returns False. The remaining 2 rules are called if there is a left and/or right subtree and use recursion to search the right or left subtree depending on if the value is less than/greater than

the root node. We use "_" to denote an anonymous variable that can be anything as we do not care what value it is.

Once again, we see the 2 languages similarities in Haskell's function definition and Prolog's rules as they both operate very similar to each other. The main differences being: Haskell's use of function declaration, typeclass and guards and Prolog's anonymous variable. However, we see more similarities in how the operation is programmed especially their dependence on recursion.

The final steps in both trees was to implement the following 1 tree traversals:

- In-order (left, root, right)
- Pre-order (root, left, right)
- Post-order (left, right, root)

In both implementations all traversals are very similar to each other except for the final order the nodes are returned in because of this we will focus mainly on in-order traversal for in-depth comparison.

In Haskell, all 3 traversal types have the same function declaration that takes a BST and returns a list of nodes. The first 2 lines of the definition are also identical. The base case if the tree is empty will always return an empty list and if the tree only consists of one node the list will always consist of just one value. It is only if the tree has left and right subtrees that it differentiates. The only change however is the order in which we recurse through the subtrees and where we place the root node, as described above.

```
inorder :: BinTree t -> [t]

inorder Empty = []
inorder (Root x Empty Empty) = [x]
inorder (Root x left right) = (inorder left) ++ [x] ++ (inorder right)

preorder :: BinTree t -> [t]

preorder Empty = []
preorder (Root x Empty Empty) = [x]
preorder (Root x left right) = [x] ++ (preorder left) ++ (preorder right)

postorder :: BinTree t -> [t]

postorder Empty = []
postorder (Root x Empty Empty) = [x]
postorder (Root x left right) = (postorder left) ++ (postorder right) ++ [x]
```

My implementation of each traversal in Prolog follows the same idea, all three being very similar to each other. The only difference being the order we traverse the subtrees and root node. Each traversal has the same base case for when the tree is empty, we return an empty list. If the tree

has subtrees in order and preorder are nearly identical. It traverses the left tree and then the right subtree and appends the nodes from each and the original root node in the correct index in the list.

```
inorder(nil, []).
inorder(bt(X, Left, Right), R) :-
    inorder(Left, R1),
    inorder(Right, R2),
    append(R1, [X|R2], R).

preorder(nil, []).
preorder(bt(X, Left, Right), R) :-
    preorder(Left, R1),
    preorder(Right, R2),
    append([X|R1], R2, R).
```

Post-order differs slightly, first we traverse the 2 subtrees, however before we append to the final list we create a “NewList” that contains the nodes of the left subtree first and then the right subtree. The original root node is appended to the end of the NewList to create the final list with the correct order of node traversal.

```
postorder(nil, []).
postorder(bt(X, Left, Right), List) :-
    postorder(Left, LL),
    postorder(Right, RL),
    append(LL, RL, NewList),
    append(NewList, [X], List).
```

Comparing both we can see a major difference is the use of the append function in Prolog to create a list and concatenate lists, whereas in Haskell we can make use of the “++” operator.

Conclusion

After my analysis of both implementation the differences between the Logic and Functional paradigms becomes obvious but so too does some of their similarities.

The first big difference is Prolog’s use of rules and predicates versus Haskell’s function declarations and definitions. In Haskell you must declare the typeclass (optional) (e.g. Eq, Ord, Show), the number of parameters the function accepts, the parameters types and the type of the value that the function returns. Each function will return a value and the value will always be of the same type. [1]

In Prolog a predicate can only evaluate to a Boolean value. To get a return value in Prolog I used relations that will find the correct values of any undefined values to make the predicate true.

Haskell also makes extensive use of pattern matching in its functions compared to Prolog that uses backtracking until it reaches the correct destination in the tree. [5]

Both languages are declarative as mentioned above however, Haskell uses mathematical expressions mapping values to other values where Prolog uses logic sentences to express facts and rules. [4]

Despite their differences, both languages share a heavy dependence on recursion to iterate through the tree and both use base cases to terminate the recursion.

References

Please find my implementation of a Binary Search Tree in Prolog and Haskell in the same folder as this report. This code is commented and can be used as reference on top of the code snippets above.

[1] D, Sinclair. Introduction to Haskell. https://www.computing.dcu.ie/~davids/courses/CA320/CA320_Haskell_2p.pdf, online [Accessed 17/11/21

[2] D, Sinclair. Introduction to Prolog. https://www.computing.dcu.ie/~davids/courses/CA208/CA208_Prolog_2p.pdf Online, [Accessed 17/11/21]

[3] Taktak, A., Ganney, P., Long, D. and Axell, R., 2020. *Clinical engineering*. 2nd ed. pp.131-168. Accessed 18/11/2021

[4] P. C. Lawrence and A. W. Smith, *Logic programming, functional programming, and inductive definitions*. 2005.

[5] P, Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. Accessed 18/11/2021