CA341 Assignment 2
Comparing Functional Programming and Logic Programming

Joao Pereira
19354106
joao.pereira2@mail.dcu.ie

Name(s): Joao Pereira

Programme: Computer Applications and Software Engineering

Module Code: ca341

Assignment Title: Comparing Functional Programming and Logic Programming

Submission Date: 23/11/2021

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):  Date: 23/11/2021

Table of Contents

Introduction - Choice of Languages

Functional Programming

For Functional Programming I chose Haskell. My decision to select Haskell is due to the fact we are currently learning Haskell for the module " CA320 Computability & Complexity" and therefore it is ideal to use Haskell to further develop my Haskell skills and experience. As a bonus, I enjoy programming in Haskell and the compiler in Haskell is a huge benefit when searching for errors.

Logic Programming

In terms of Logic Programming, Prolog was my choice of language. It seemed only right to appoint Prolog as it is a language for Logic. Although I am not a huge fan of Prolog and especially how it operates and how the code is structured and written, it was a different yet challenging experience with a huge learning curve. Like Haskell, the fact I got to further develop my Prolog skills and knowledge, meant it was a good pick for a language.

Code Analysis

Functional Programming Code Analysis

For my Haskell program, I initiated with a Binary Tree data type and followed onwards with the five required functions (insert, search, inorder, preorder and postorder). For my BST data type, I set the BinTree as Empty and then structured it to be a BST, *fig 1.1*, I got used the data type from David's Sinclair's notes [1].

I proceeded to form the five functions afterwards, each one with a similar layout. I completed the insert function with the aid of a youtube video and David Sinclair's notes [1] [2]. It operates by: If a tree is empty then create x as the node with two empty children. Otherwise, if the tree is not empty then do perform BST rules in order to insert an element, *fig 1.2.*

With the aid of online resources [1][3], the search function was formed. The search function is similar to the insert function in the sense that it recursively searches for an element. The only difference is that if a tree is empty, it returns false. See *fig 1.3.*

Lastly, I produced the three traversals. I started off developing inorder, see *fig 1.4,* which was supported by Sinclair's notes [1] and through the use of inorder, creating the other traversals was then a simple task.

Logic Programming Code Analysis

I approached my Prolog code after my Haskell and followed a similar procedure.

First I set a data type for a Binary Tree, *fig 2.1*.

Next, I implemented an insert *(fig 2.2*) and a search function (*fig 2.3*) with the aid of Sinclair's notes [4]. Insert performs by inserting a value into the tree and by recursively placing the element into the left or right tree.

Using the same notes [4], a search function was applied in order to return true or false if an element is found to be in the tree.

Lastly, likewise in Haskell, I proceeded to implement the inorder and preorder traversals with the support of [5]. Postorder was developed with the same method however it was a bit more complicated as it needed a new list to append the values to. With the aid of [6], this complication was then overcome. Each function operates by ordering the tree in the right order as this should be and then appending the order to a list that is displayed to the terminal. See *fig 2.4*, to see all traversals.

Functional vs Logic Programing

Data Types / Systems

One big difference between Prolog and Haskell is how their type systems are set out to be. To begin, Haskell has a usual set of type parameters such as numeric types (integers and floats), Bool and char [8], as well as a multi-parameter type class feature as you can call a type class and reuse it for other problems[7], E.g. *fig 1.3*. Haskell also supports a collection of composite types which include lists, functions, tuples and algebraic types. In addition, Haskell is one of the few languages which accepts general recursive types [11].

On the other hand, Prolog is dynamically typed [11][13] unlike Haskell which is a strictly-typed [13] language and its primitive types are atoms and numbers [9]. Atoms are composed of either string of letters, integers or underscores that begin with either a lowercase or any printable ASCII characters delimited by apostrophes [10]. Atoms are unable to be compared and have no extra further properties [9].

Now to compare them and mention the benefits and disadvantages between each other:

Haskell Cons and Pros in Terms of Data Types

- Haskell starts with a few huge advantages, the first being having a flexible type system that supports type inference and which is checked at compile time. This is helpful when it comes to reducing errors and producing documentation.
- Haskell is efficient in regards to type safety, which means the user is able to make global changes without having too many cautions in their code.
- Haskell is an expressive type system that allows for code reuse [12].
- On the downside, Haskell's data types are complex and would be difficult for a first-time user to comprehend if they were not already educated with Haskell types.
- Haskell separates the type declarations away from the function declarations which in turn makes it harder for a user to structure and then further debug their code.

Prolog Cons and Pros in Terms of Data Types

There is not much to type data/systems with Prolog however these are the few:

- Like many dynamically typed languages, Prolog exceeds with:
    - Easy and short code design.
    - Conciseness.
- Prolog is an easier language to divulge into with types as the parameters and the way types are approached have a much more comfortable and user-friendly approach.

Data Type Conclusion

In my opinion, overall, Haskell dominates over Prolog when it comes to the topic of data types. It offers plenty of more features as well to assist a user when dealing with types. Realistically, Prolog does not have much on the table to present apart from the fact that its data types are much easier to first get a grip on. To provide evidence that Haskell is much preferred, in the article "Escape from Zurg: An Exercise in Logic Programming" [7], students were asked to program a task in Haskell and

Prolog and the conclusions stated that overall they found Haskell to be more helpful with its type system when trying to reach a solution.

Performance / Efficiency

Both Haskell and Prolog rely on time and space in regards to the topic of how efficient they are.

Haskell's space/memory performance is poor. Haskell is lazily evaluated [8] which means if a data structure is repeatedly updated without being used then unevaluated thunks will use more memory than needed [15]. To combat this Haskell providers users' with libraries and functions to structure code and save memory which in turn optimizes the performance speed.

Prolog's space performance is measured by its compilers and interpreters. Each one operated through either trail, global or local stack [14]. In contrast, Prolog's time efficiency is also dependant on how quick the compiler and interpreter perform. Overall Prolog is efficient when dealing with concise, clean code suited to solve a small problem that doesn't require an overload of an algorithm.

Haskell Cons and Pros in Terms of Performance

- Deals with a lot of memory leaks due to being lazily evaluated.
- High-level language allows the compiler to perform outstanding optimizations without having the code break [16].
- Provides libraries which reduces memory and up's speed efficiency.
- Slow when operating on code which takes in a lot of data and optimization.

Prolog Cons and Pros in Terms of Performance

- Quick time dealing with significantly limited solutions as the performance from compilers are designed to provide a lot of development effort.
- Prolog supports in-built back-tracking which allows the user to order possible solutions from a code [14].
- Lacks performance when tackling problems that are not too logical/general based.
- The larger the algorithm and problem, the slower Prolog will perform.

Performance Conclusion

Overall Haskell and Prolog are pretty even within the topic of performance. Both have their downs and up's however in every case there is a programming language that will perform better in regards to space and time. They serve their purpose and what they're intended for. In terms of which is superior is opinionated. In my opinion, I would say Haskell due to possessing a much more modern compiler that can handle more chunks of code.

Appendix

bst.hs

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

```

fig 1.1

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

fig 1.2

```haskell
search :: Ord t => t -> BinTree t -> Bool
search x Empty = False -- if tree is empty ret
search x (Node y left right)
    | x == y                = True
    | x < y                 = search x left
    | otherwise             = search x right
```

fig 1.3

```haskell
inorder :: BinTree t -> [t]
inorder Empty = [] -- if the bst is empty
inorder (Node x left right) = inorder left ++ [x] ++ inorder right
```

fig 1.4

```prolog
BinTree(left, x, right).
```

fig 2.1

```prolog
insert(Empty, x, BinTree(Empty, x, Empty)).
insert(BinTree(left, Root, right), x, BinTree(new_left, Root, right)) :-
    x =< Root, insert(left, x, new_left.
insert(BinTree(left, Root, right), x, BinTree(left, Root, new_right)) :-
    x =< Root, insert(right, x, new_right).
```

fig 2.2

```
13  search(x, BinTree(left, x, right)).
14  search(x, BinTree(left, Root, right)) :-
15      X =< Root, search(x, left).
16  search(x, BinTree(Root, left right)) :-
17      search(x, right).
```

fig 2.3

```
22  inorder(Empty, []).
23  inorder(BinTree(x, left, right), List) :-
24      inorder(left, new_left),
25      inorder(right, new_right),
26      append(new_left, [x|new_right], List).
27
28  preorder(Empty, []).
29  preorder(BinTree(x, left, right), [x|List]) :-
30      preorder(left, new_left),
31      preorder(right, new_right),
32      append(new_left, new_right, List).
33
34
35  postorder(Empty, []).
36  postorder(BinTree(x, left, right), List) :-
37      postorder(left, new_left),
38      postorder(right, new_right),
39      append(new_right, [x], newer_right),
40      append(new_left, newer_right, List).
```

fig 2.4

References

[1] D. D. Sinclair, “Tree of Ints,” *Dcu.ie*. [Online]. Available: https://www.computing.dcu.ie/~davids/courses/CA320/exercise5a.hs. [Accessed: 21-Nov-2021].

[2] “Binary search trees,” 07-Sep-2015. [Online]. Available: https://www.youtube.com/watch?v=dlHHflOEDpk. [Accessed: 21-Nov-2021].

[3] S. P. Suresh, “Programming in Haskell: Lecture 22,” *Cmi.ac.in*, 30-Oct-2019. [Online]. Available: https://www.cmi.ac.in/~spsuresh/teaching/prgh19/lectures/lecture22.pdf. [Accessed: 21-Nov-2021].

[4] D. Sincalir, “CA208 Logic Introduction to Prolog,” *Dcu.ie*. [Online]. Available: https://www.computing.dcu.ie/~davids/courses/CA208/CA208_Prolog_2p.pdf. [Accessed: 22-Nov-2021].

[5] “Solutions-4 - Prolog Site,” *Google.com*. [Online]. Available: https://sites.google.com/site/prologsite/prolog-problems/4/solutions-4. [Accessed: 22-Nov-2021].

[6] Lecture Module, “Advanced Features in Prolog,” *Iitd.ac.in*. [Online]. Available: https://www.cse.iitd.ac.in/~saroj/LFP/LFP_2013/L10.pdf. [Accessed: 24-Nov-2021].

[7] M. Erwig, “Escape from Zurg: An Exercise in Logic Programming,” *J. Funct. Prog.*, vol. 14, no. 3, pp. 253–261, 2004.

[8] D. Sinclair, “CA341 - Comparative Programming Languages Functional Programming Paradigm,” *Dcu.ie*. [Online]. Available: https://loop.dcu.ie/pluginfile.php/4194840/mod_resource/content/4/CA341_Functional_Programming_Paradigm.pdf. [Accessed: 22-Nov-2021].

[9] D. Sinclair, “CA341 - Comparative Programming Languages Logic Programming Paradigm,” *Dcu.ie*. [Online]. Available: https://loop.dcu.ie/pluginfile.php/4183133/mod_resource/content/1/CA341_Logic_Programming_Paradigm.pdf. [Accessed: 22-Nov-2021].

[10] R. V. Sebesta, “Logic Programming Languages,” *Dcu.ie*, 2015. [Online]. Available: https://loop.dcu.ie/pluginfile.php/4183145/mod_resource/content/2/Concepts%20of%20Programming%20Languages-Chapter16.pdf. [Accessed: 22-Nov-2021].

[11] D. Sinclair, “CA341 - Comparative Programming Languages Data Types and Scope,” *Dcu.ie*. [Online]. Available: https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Data_Types_and_Scope_2p.pdf. [Accessed: 22-Nov-2021].

[12] A. S. Mena, *Practical Haskell: A Real World Guide to Programming*, 2nd ed. Berlin, Germany: Apress, 2015.

[13] “CA341 - Comparative Programming Languages Data Types and Scope,” *Dcu.ie*. [Online]. Available: https://loop.dcu.ie/pluginfile.php/3969626/mod_resource/content/1/Data%20Types%20and%20Scope.pdf. [Accessed: 23-Nov-2021].

[14] “Efficiency of Prolog,” *Metalevel.at*. [Online]. Available: https://www.metalevel.at/prolog/efficiency. [Accessed: 23-Nov-2021].

[15] W. Sewell, “Making Efficient use of memory in Haskell,” *Pusher.com*, 02-Oct-2015. [Online]. Available: https://blog.pusher.com/making-efficient-use-of-memory-in-haskell/. [Accessed: 23-Nov-2021].

[16] “Why is Haskell (GHC) so darn fast?,” *Stackoverflow.com*. [Online]. Available: https://stackoverflow.com/questions/35027952/why-is-haskell-ghc-so-darn-fast. [Accessed: 25-Nov-2021].