CA341 - Comparing Logic and Functional Programming

Group 10: Benjamin Olojo (19500599) and Przemyslaw Majda (20505049)

Introduction

This assignment consisted of implementing programs to perform operations on ordered binary trees in both a logic and functional programming language. The programs include operations for inserting and searching user-defined values as well as inorder, postorder and preorder listing of nodes within the tree. The languages chosen for the logic and functional implementations of the program were Prolog and Haskell respectively.
These choices were made due to the contrasting nature of the Prolog and Haskell programming languages, with key differences in terms of data types, control flow, abstraction and the mathematical underpinnings of each language.

Upon initial analysis, we identified potential differences in implementation such as how functions and predicates could be used to create the tree operations, how referential transparency and general resolution would define the behaviour of the tree operations and the structure of the tree operations based on the data types available within each language.

Referential Transparency & General Resolution

Referential Transparency

Programs written in a functional language are constructed using expressions rather than statements, such as the inorder traversal operation shown in Figure 1.3 of Appendix 1. Each expression has a meaning or value associated with it based on what the expression evaluates to. In Haskell, there is no sequencing and no concept of state in the sense of variable, meaning that there are no side-effects within the language.
In a programming language without side-effects, an expression can be considered referentially transparent if its value can be replaced without changing the behaviour of the program [1]. Referential transparency places an emphasis on abstraction in functional languages such as Haskell, allowing programmers to focus on modelling a particular style of solution rather than focusing on the utilisation of hardware through concepts such as variables and sequencing [1].

General Resolution

The basic unit in a logic program is a relation. In languages such as Prolog, a relation is generally defined by smaller relations and programs tend to contain a vast number of relations. These relations can also be utilised within horn clauses which outline logical formulae of a particular rule-like form [2], such as the inorder traversal rules shown in Figure 1.4 of Appendix 1.
A key aspect of the Prolog programming language is resolution. Resolution is an inference rule which states that given two clauses containing complementary unifiable literals, produce a new clause using the literals of both clauses except for the complementary ones, then apply a unifier to them [2]. Resolution in prolog utilises backtracking to search for the most general unifier based on closed-world assumptions.

Static Typing & Dynamic Typing

Static Typing

Haskell is statically typed, meaning that function inputs and output types are defined by the programmer. This is different from dynamically typed languages like Python, Javascript and Prolog.
An advantage of static typing is that errors will be found at compile time rather than at runtime [3]. For example, in a dynamically typed language, the programmer could test their code and run it without error, but then a user could encounter an edge case and their program would crash. In Haskell, this would not happen as the error would not make it to the running program as it would be caught during compilation [3].
This consequently leads to a lot less unit testing about type agreement between functions.
In Haskell, functions are expressions just like integers and strings and so they also have type (this is the type of the value it returns).

Dynamic Typing
In contrast, Prolog is dynamically typed, meaning that all types (atoms, numbers and structures) can be used interchangeably.
If a term is provided and it's of a different type than stated in the predicate, Prolog will simply evaluate it to false. The advantage of this is that development of larger Prolog programs can be quick with less errors during compilation. However, this can also lead to unexpected behaviour in certain scenarios where the programmer is not certain about what type of term will be used at runtime.

Data Types & Operations

Haskell
Since Haskell is a statically typed language, everything has a type. The common basic types are those shown in Figure 1.1 below.
There are also compound types which are made up of multiple values of basic types. For example, a list of Integers defined as `[Int]`, which is also used in the function declaration depicted in Figure 1. 3 of Appendix 1.
When a function is defined, its parameters and the value which it evaluates to must also have their types defined. The syntax looks like this:
```
rectangleArea :: Float -> Float -> Float
rectangleArea x y = x * y
```
The first line defines what type the function evaluates to, and the ones before it are the types of the inputs, in order. The second line defines the actual behaviour of the function and what it should do with the inputs.
Functions are defined in sequential order, where different lines define different behaviour in different cases. Recursion plays a major part in Haskell, and defining base cases is a core concept in recursive programming [4], which can be seen in the inorder operation shown in Figure 1.3 of Appendix 1.
Our Haskell program uses recursion in every function as it allows us to perform operations on binary trees easily by working on smaller and smaller subtrees until it reaches the goal.
Haskell also has '`if`' and '`else`' statements which allow for further variance in function behaviour.

Figure 1.1 - Common Haskell Data Types

Keyword	***Type***	***Sample Value***
Int	integer	12
Float	Floating point number	-1.23
Char	Character	"A"
Bool	Boolean	True

Prolog

Prolog does not have functions but rather a set of rules or predicates that define the code's behaviour. While Haskell has its data types, Prolog technically is typeless as it only has one data type, a 'term'. A term however can be an atom, number, variable, or compound term. An atom is the closest thing to a String type that Prolog has.
An atom can be a string of characters or a symbol such as `ben` in
`likes(ben, reading).`
It is similar to Haskell in that predicates are defined line by line with different behaviour specified on each line. Prolog also makes heavy use of recursion, like Haskell.
It differs however as the predicates do not have any defined types, and do not return a specified type. Instead, the output is dependent on the values used with the predicate [4]. If specified values are given, the predicate could return `true` or `false`, but if a value is left blank, Prolog can display possible values that would make the statement true.
This offers a unique advantage as other programming languages do not offer this feature.

Figure 1.2 - Comparison of features of Haskell and Prolog programming languages

Haskell Programming Language (Functional)	Prolog Programming Language (Logic)
Advantages	Advantages
- Lazy evaluation engine doesn't allocate memory for data structures until needed - Contains runtime type checking and built-in list handling operations - Automatic memory allocation and garbage collection - Supports concurrent programs	- Contains built-in list handling operations - Simple to build databases within programs - Supports finding multiple solutions to queries through backtracking engine - Cuts can improve the efficiency of Prolog programs
Disadvantages	Disadvantages
- Relatively difficult to perform Input/Output operations - Unable to store variables within programs - Hard to predict memory usage of expressions such as lazy lists	- Difficult to test programs due to logical structure of programs - No built-in runtime type checking - Unable to build mutable data structures such as arrays due to immutability of Prolog data

References

[1] R.W. Sebesta, “Functional Programming Languages”, in Concepts of Programming Languages, 11 ed. Boston: Pearson, 2015, pp. 662

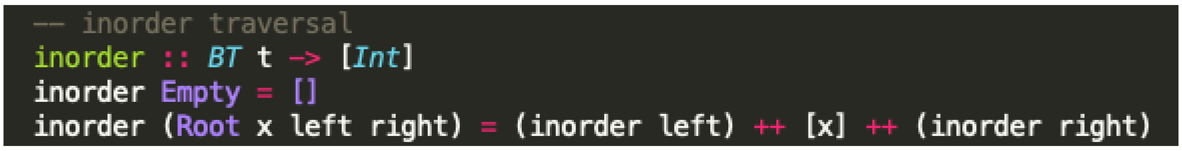
[2] R.W. Sebesta, “Logic Programming Languages”, in Concepts of Programming Languages, 11 ed. Boston: Pearson, 2015, pp. 718-720

[3] Monday Morning Haskell, (2016, Dec. 06), *Understanding Haskell's Type System* [Online], Available: https://mmhaskell.com/blog/2016/12/5/7mkljzq7zy97d66zm4yvtn8v1ph502

[4] GeeksForGeeks, (2022, Feb. 24), *Difference Between Functional and Logical Programming* [Online], Available: https://www.geeksforgeeks.org/difference-between-functional-and-logical-programming/

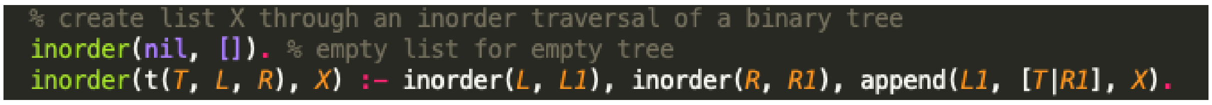
Appendix 1

Figure 1.3 - Inorder traversal expression in Haskell



```haskell
-- inorder traversal
inorder :: BT t -> [Int]
inorder Empty = []
inorder (Root x left right) = (inorder left) ++ [x] ++ (inorder right)
```

Figure 1.4 - Inorder traversal rule in Prolog



```prolog
% create list X through an inorder traversal of a binary tree
inorder(nil, []). % empty list for empty tree
inorder(t(T, L, R), X) :- inorder(L, L1), inorder(R, R1), append(L1, [T|R1], X).
```