Ryan Hirscher
CSC4510 Programming Language Design
Assignment 5
10/15/2023

Functional and Logic Programming

Introduction:

The following sections Functional Paradigm and Logical Paradigm discuss the merits, ease of use, and downsides of their respective language representatives and paradigms through my personal experiences and examples. Additionally, the sections are concluded with documentation references that aided my program designs and how helpful they were for research.

Functional Paradigm (Haskell):

The functional programming paradigm is certainly a different perspective of programming, considering new programmers are generally introduced to object-oriented or imperative programming styles. The functional paradigm uses expressions and attempts to bind everything into a mathematical function style based on lambda calculus which focuses on “what to solve” rather than “how to solve” (GeeksForGeeks). An expression is evaluated to produce a value whereas a statement is executed to assign variables. This is a unique and intuitive thought to produce results. The paradigm becomes easier to understand when concepts of functional programming are realized, which is evident in Haskell. The use of pure functions prevents any side effects by using immutable variables. Meaning, they do not modify any arguments, local/global variables, or I/O streams. They are deterministic and the function’s only result is the value returned (GeeksForGeeks). This makes debugging very simple and easy to use. Furthermore, it supports concurrency with minimal memory risks. However, there is a steep learning curve to adjust to some other features. There are no “for” or “while” loops in functional programming. You use recursive functions which can be strange to wrap your head around at first, but eventually it starts to make sense. For example, the following Haskell code demonstrates a recursive function to calculate the factorial of n (Figure 1.1).

Haskell Factorial Function Figure 1.1 (Learn Haskell)

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The code is very straightforward and easy to read which really makes functions the star of the show. It will always be challenging at first to learn the syntax of a new language, but Haskell did a very good job of introducing the functional paradigm as an alternative option for appropriate situations.

The syntax and semantics of the language can often be confusing, but in many ways useful. Upon learning how to compile simple functional Haskell programs, I recognized that

there is a lot of clutter similar to Ada. A '.hi' file is created for headers and a '.o' to link all the modules together to create an executable. Additionally, the binary executables are extremely large. For my first simple program, the main source code was only around 1k bytes while the executable was over a million bytes. Similarly, my second program's source code was around 3k bytes while the executable was almost 3 million bytes. The language does not seem to be very intuitive for tertiary space as it creates several files and takes up a lot of space. One of the very interesting and flexible features of Haskell is type inference. Generally, there is no need to declare variables or their types, but you can do so with '::' if needed (Learn Haskell). Not indicating your variable types however, can make a program difficult to read because a user may find it difficult to track and know the variables in the code. This may lead the user to infer the wrong types of data which can be difficult to troubleshoot. As a programmer, I like to have full control over what is happening and not leave these decisions to the compiler. Fortunately, Haskell offers the ability to do so which can prevent headaches. Another feature that adds to the confusion is the indentations. Haskell is indent-significant similar to Python, it does not often use curly braces like we would see in other programming languages like Java or C++. I find it difficult to read a program and understand the scope of variables with large indentations and no clear start or end. A useful lesson here is to clearly label and identify variables in comments to understand what the code does.

Haskell is very well documented online. It was created in 1990 and rose in popularity in the early 2000s (Microsoft). GeeksForGeeks is great for introductions as they explain overarching concepts, features, and reasoning behind their implementations. The functional paradigm introduction explained pure functions, immutable data, recursion, and more (GeeksForGeeks). Syntax and semantics is made very easily readable on the official Haskell wiki. For example, the following code clearly demonstrates how to use a standard library and lists (Figure 1.2).

Standard Library Example Figure 1.2 (Learn Haskell)

```
module Main where

import qualified Data.Map as M

errorsPerLine = M.fromList
   [ ("Chris", 472), ("Don", 100), ("Simon", -5) ]

main = do putStrLn "Who are you?"
          name <- getLine
          case M.lookup name errorsPerLine of
             Nothing -> putStrLn "I don't know you"
             Just n  -> do putStr "Errors per line: "
                           print n
```

This website provides a great introduction to the language and offers similar examples for a rich suite of functional concepts and syntax. Another website, Basic Haskell, guided me in creating the command line manager. It goes into detail explaining line-by-line of a similar "to-do list"

(Basic Haskell). There are countless examples and documentation online that can aid someone to learn Haskell and the functional paradigm. Ideas, concepts, and code that aided my research and examples can be found from the short list below.

"Learn Haskell in 10 Minutes." *Learn Haskell in 10 Minutes - HaskellWiki*, wiki.haskell.org/Learn_Haskell_in_10_minutes

"Basic Haskell: An Examination of the Todo List." *Leveraging Holistic Synergies*, www.benlopatin.com/basic-haskell-todo/

A History of Haskell: Being Lazy with Class - Microsoft.Com, www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf

"Functional Programming Paradigm." *GeeksforGeeks*, GeeksforGeeks, 28 June 2022, www.geeksforgeeks.org/functional-programming-paradigm/

Gibiansky, Andrew. "Andrew Gibiansky :: Math → [Code]." *Intro to Haskell Syntax - Andrew Gibiansky*, www.gibiansky.com/blog/haskell/haskell-syntax/index.html

"Haskell Libraries." *Haskell Hierarchical Libraries*, downloads.haskell.org/ghc/latest/docs/libraries/

Logical Paradigm (Prolog):

At first, logical programming seemed intuitive and easy to understand. You have a knowledge base of facts and then rules to establish more advanced relationships to build a virtual network or a graph. It is very straightforward and mechanical. In Prolog, you set up an environment that consults your knowledge base and you post queries to it. Then the shell will respond with the solution(s). For example, if you set up a rule to output the parent of a child in a family tree knowledge base, you should expect the output from parent(X,frank) to be all the parents of frank and parent(frank,X) to be all the children of frank. However, it will do exactly what you program it to do, if your logic is inconsistent it will not correct you or let you know. So if you build your family tree upside down or sideways, it can still be a valid network. The same goes for spelling mistakes within your facts. This is very hard to troubleshoot, especially if you do not employ good programming practices. For example, not taking it step-by-step and testing along the way will make your life very difficult. Rules often build upon each other and themselves. It seems simple at first, but it can quickly get complicated. Making the cousin(X,Y) rule was challenging. First you have to decide what you want the cousin to be in plain English and ask yourself if you want to include second cousins, third cousins, or more. If you do not do this for most rule creation, your logic will get turned around and cause headaches, especially if you are calling other rules. Next you have to implement the planned and desired logic. During testing, you may realize that there are multiple paths that arrive at the same solution which leads to duplicate results. Thankfully, the language allows you to build sets with queries. If you use rules nested within other rules, it can propagate errors unintentionally. For a simple

hypothetical example, if you define mother(X, Y) :- male(X), parent(X,Y), this would return the father to all the rules that would call mother(X,Y). In my case, I accidentally changed the logic of parent(X,Y) in the rules. This quickly became a difficult problem to recognize since it ruined the logic of most of my rules for grandparents, cousins, and other relatives. I initially defined all my facts to be parent(Y,X) and later incongruently built my rules off of parent(X,Y). If used correctly, issues like error propagation, slow troubleshooting, and logical misuse should not be an issue. However, first introductions can be rough. Logic programming provides a flexible and robust solution for building networks and graphs to identify relationships between nodes, but if misused it can cause many unintended consequences.

The Prolog interpreter had several difficult design decisions to make simply because catching logic errors is a non-trivial process. Meaning, you cannot easily save a programmer from making incorrect, but valid, code as seen previously. This is precisely why some of the warnings and error messages can be misleading and hinder the learning process. Generally speaking, I agree with this implementation for logic-based programming, but it does make for a challenging experience if you have not planned out all your features beforehand or if you are unfamiliar with the language semantics. It makes sense that a compiler should not stop a programmer from doing something legal. There are some decisions with the interpreter that I do not agree with. The language is extremely case sensitive and sometimes space sensitive. Moreover, the compiler will not recognize many of these cases as errors. For example, I defined my facts starting with capital letters like male(John). In this specific example, the compiler did catch this issue, but it did not address it in a way that explained the issue. Upon compilation, it 'warned' me that each of my facts were singleton variables. I was slightly concerned since I did not know what that meant, but the compilation successfully completed and an executable was created which gave me a false sense of security. My family tree was empty and my rules returned nonsense which was very confusing to me since I did not know my family tree was unpopulated at the time. It took a very long time to figure out my situation, especially since I thought that "there was no possible way I am NOT allowed to have capital characters in my own facts." I immediately ruled that out and thought something else was wrong. After several hours, it turns out, a predicate name must begin with a lower case letter (Basics of Prolog). I believe that there should at least be a mention of capitalized character misuse alongside singleton variable warnings. That would have saved me hours of thumb-twiddling and tracing rules to figure out why I was being returned nonsense. Another issue that gave me a headache was a simple spelling mistake in one of the facts. It just so happened that the only name I mistyped was supposed to be the result of my first queries after fixing my logic issues. For my specific case, male(lenoard) has no connection to parent(brian,leonard). In the end, the compiler allows an executable to be created even if you have declared singleton variables because your rules may still have a purpose if you consult another database or knowledge base. With an unpopulated tree, hidden spelling mistakes, and backwards logic, you are in for a very rough debugging experience. My introduction to Prolog was not pleasant to say the least.

Documentation online was very useful for the most part. It sometimes failed to mention very specific and minute details like its case sensitivity which I did not discover until debugging. Nevertheless, Prolog became a very powerful tool because of how straightforward the coding becomes once the syntax is learned. The basics of Prolog and an introduction to family tree

exercise are well documented on the website, Basics of Prolog. For example, the website posts an example of how to define a rule for a grandparent (Figure 2.1).

Grandparent Example Figure 2.1 (Basics of Prolog)
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

Here the code is showing that a grandparent is defined as a parent of a parent. Syntax and query building information was very easy and intuitive to understand after reading JavaTpoint. This website not only gives you an excellent introduction, it also has modules to further expand on its capabilities. The assignment to build the family tree was not solely dependent on syntax knowledge and it focused more on how to return the correct solution during runtime. TutorialsPoint highlights the importance of relations in Prolog. It dives deep into a family tree example that helped guide me to appropriate implementation. Additionally, it illustrates in diagrams what the nodes look like and how relationships are defined. Prolog would have been infinitely more challenging to learn without these useful online documentations. My experience shows that it is difficult to decipher the difference between a valid logic error and language misuse, but after learning how the language and paradigm works, it becomes a simple but powerful tool.

The following list of references aided my knowledge, examples, and solution.

"Prolog - Relations." *Online Courses and eBooks Library*, www.tutorialspoint.com/prolog/prolog_relations.htm

"The Basics of Prolog." *The Basics of Prolog*, cinuresearch.tripod.com/ai/www-cee-hw-ac-uk/_alison/ai3notes/section2_3_2.html#:~:text=Facts%20consist%20of%3A,with%20a%20lower%20case%20letter

"Prolog Syntax - Javatpoint." *Www.Javatpoint.Com*, www.javatpoint.com/prolog-syntax

"Steve Harlow." *Steve Harlow, University of York*, www-users.york.ac.uk/~sjh1/courses/L334css/

"Prolog: An Introduction." *GeeksforGeeks*, GeeksforGeeks, 28 June 2022, www.geeksforgeeks.org/prolog-an-introduction/