1 Introduction

1.1 Beginnings

Computers have become a part of everyone's life. From the ones in our pockets to the ones on desks or in our school bags; working or in fact living without them is difficult if not impossible. All the more reason to know how to use one. Simply speaking just using a computer these days is not enough. To be able to utilise their true potential, one must go deeper and communicate with them. This is where the art of programming steps in.

Programming has become an integral part of working and interacting with computers and day by day more and more complex problems are being tackled using the power of programming technologies. It is possibly the only way to talk to computers and hence the need for a robust and multi purpose programming language has never been more urgent. The desirability of a programming language depends on a lot of factors such as the ease of use, the features and/or functionalities that it provides, adaptability and what sort of problems can it solve. One is spoilt for choice with a number of options for a wide variety of problem domains, for example Object Oriented Languages. In the last few years or nearly a decade, the Declarative Style of programming has gained popularity in being more suited for solving problems and also in the way that it has easily adapted to a number of domains. Declarative Programming Languages have not only challenged but also have proved as a better more richer alternative than the conventional Procedural Imperative Object Oriented way of doing things. The methodologies that have stood out are the Functional and Logical Approaches. The former based on Functions and Lambda Calculus while the latter on Horn Clause Logic. With each of them having their own advantages and flaws, one has to make a choice or may be not? This document looks at the attempts, improvements and future possibilities of bringing Haskell, a Purely Functional Programming Language and Prolog, a Logical Programming Language, one step closer to make the choice easier if not eliminate it.

1.2 Thesis Statement

The thesis aims to provide insights into merging two declarative languages namely, Haskell and Prolog by embedding the latter into the former and analysing the result of doing so as they have conflicting characteristics. The finished product will be something like a *haskellised* Prolog which has logical programming like capabilities.

1.3 Problem Statement

Programming languages are emerging more frequently than ever before. They include new concepts and features to simplify the process of coding a solution and assist the programmer by lessening the burden of carrying out standard tasks and/or procedures. A new one tries to capture the best of the old; learn from the mistaken, add new concepts and move on; which seems to be the mantra. But all is not that

straight forward shifting from one language to another is not always easy. There are costs, incompatibilities to look at. A language might be simple to use and provide better performance than its predecessor but is not always worth the switch. The reasons may vary but the effort required to learn and use the new technologies along with the cost of doing so when it is compared to the upsides.

PROLOG has a similar story. It was born in an era where procedural programming had made everyone notice their presence. Talking about competition, it was against something radical; the C programming language. The languages C has influenced is off the chart and so is the performance. It had paved the way for structured procedural programming and had given birth to the Unix operating system. Though the original version of PROLOG has given rise to a large number of different flavours but a few drawbacks remain through the bloodline and as a result it did become the first choice. Some basic requirements such as modules are not provided by all compilers. To make it do real world stuff, a set of practical features are pushed in now and then which results in the loss of the purely declarative charm. The problem is that PROLOG is fading away, [**?**, **?**, **?**], not many people use it and most of the times when it is used, the variant is usually *practical* PROLOG and the area being academia. It is not used for building large programs [**?**, **?**, **?**]. But there are a lot of good things about PROLOG that should not die away. Moreover, PROLOG is ideal for search problems.
So the question is how to have all the good qualities of PROLOG without actually using PROLOG?

Well one idea is to make PROLOG as an add-on to another language which is widely used and in demand. Here the choice is HASKELL; as both the languages are from the same family tree there are some commons which can help to blend the two.

Programming languages pop up from time to time. The number of languages today is in the hundreds or even thousands. Not all of them survive or end up being scarcely used. But many a times the case is that even though a language has a lot going for it, the reluctancy to change makes that difference ever so slightly in its adoption and wide spread. Other reasons could be that the need dies out or the language is unable to adapt to the changing requirements.

Flipping the coin to the other side we see, the more specific the language the easier it is to solve the problem. The simple reason being that, the problem need not be moulded according to the capability of the language. For example a problem with a naturally recursive solution cannot take advantage of tail recursion in many imperative languages. Many problems require the system to be mutation free, but have to deal with uncontrolled side-effects and so on.

Putting all of the above together, Domain Specific Languages are pretty good in doing what they are designed to do, but nothing else, resulting in choosing a different language every time. On the other hand, a general purpose language can be used for solving a wide variety of problems but many a times, the programmer ends up writing some code dictated by the language rather than the problem.

The solution, a programming language with a split personality, in our case, sometimes functional, sometimes logical and sometimes both. Depending upon the problem, the language shapes itself accordingly and exhibits the desired characteristics.

The ideal situation is a language with a rich feature set and the ability to mould itself according to the problem. A language with ability to take the appropriate skill set and present it to the programmer, which will reduce the hassle of jumping between languages and/or forcibly trying to solve a problem according to a paradigm.

The subject in question here is HASKELL and the split personality being PROLOG. How far can HASKELL be pushed to dawn the avatar of PROLOG ? is the million dollar question.

The above will result in a set of characteristics which are from both the declarative paradigms.

This can be achieved in two ways,

Embedding (Chapter 4): This approach involves, translating a complete language into the host language as an extension such as a library and/ or module . The result is very shallow as all the positives as well as the negatives are brought into the host language. The negatives mentioned being, that languages from different paradigms usually have conflicting characteristics and result in inconsistent properties of the resulting embedding. Examples and further discussion on the same is provided in the chapters to come.

Paradigm Integration (Chapter 5): This approach goes much deeper as it does not involve a direct translation. An attempt is made by taking a particular characteristic of a language and merging it with the characteristic of the host language in order to eliminate conflicts resulting in a multi paradigm language. It is more of weaving the two languages into one tight package with the best of both and maybe even the worst of both.

1.4 Proposal Organization

The next chapter, Chapter 2 provides details about the short comings of the previous works and the road to a better future. Chapter 3, the background talks about the programming paradigms and languages in general and the ones in question. Then we look at the question from different angles namely, Chapter 4, Embedding a Programming Language into another Programming Language and Chapter 5, Multi Paradigm Languages (Functional Logic Languages). Some of the indirectly related content Chapter 6 and finishing off with the Chapter 7, the expected outcomes.