Embedding Programming Languages: Prolog in Haskell

A Master's Thesis by

Mehul Chandrakant Solanki

230108015 solanki@unbc.ca

6 8 2015

Submitted to the graduate faculty of the

MCPS

in partial fulfillment of the requirements

for the Master's Thesis and

subsequent MSc. in Computer Science

Committee Members:

Dr. David Casperson, Committee Chair

Dr. Alex Aravind

Dr. Mark Shegelski

Outline

Abstract **vi**

1 Introduction **1**
1.1 Beginnings 1
1.2 Thesis Statement 2
1.3 Problem Statement 2
1.4 Proposal Organization 4

2 Background **6**

3 Proposed Work **11**
3.1 Current Work 11
3.2 Contributions 12

4 Embedding a Programming Language into another Programming Language **13**
4.1 The Informal Content from Blogs, Articles and Internet Discussions . 13
4.2 Related Books 14
4.3 Related Papers 14
4.4 Related Libraries in Haskell 16

5 Multi Paradigm Languages (Functional Logic Languages) **18**
5.1 The Informal Content from Blogs, Articles and Internet Discussions . 18
5.2 Literature and Publications 19
5.3 Some Multi Paradigm Languages 21
5.4 Functional Logic Programming Languages 21

6 Related Work **23**

7 Embedding a Programming Language into another Programming Language **24**
7.1 Theory 25
7.2 Implementations 26
7.3 Important People 26
7.4 Miscellaneous / Possibly Related Content 26

8 Prolog in ____ **27**
8.1 Theory 27
8.2 Implementations 27
8.3 Important People 28
8.4 Miscellaneous / Possibly Related Content 28

9 Prolog in Haskell **29**
9.1 Theory . . . 29
9.2 Implementations . . . 30
9.3 Important People . . . 31
9.4 Miscellaneous / Possibly Related Content . . . 31

10 Unifying or Marrying or Merging or Combining Programming Paradigms or Theories **32**
10.1 Theory . . . 32
10.2 Implementations . . . 32
10.3 Miscellaneous / Possibly Related Content . . . 33

11 Functional Logic Programming Languages **34**
11.1 Theory . . . 34
11.2 Implementations . . . 35
11.3 Miscellaneous / Possibly Related Content . . . 35

12 Quasiquotation **36**
12.1 Theory . . . 36
12.2 Implementations . . . 36
12.3 Miscellaneous / Possibly Related Content . . . 36

13 Related Terms or Keywords **37**

14 Haskell or Why Haskell ? **38**

15 Prolog or Why Prolog ? **42**

16 Miscellaneous or Possibly Related Content **45**

17 Prototype 1 **46**
17.1 Creating a data type . . . 46
17.2 Working with the language . . . 49
17.3 Black box . . . 51

18 Prototype 2.1 **52**

19 Prototype 2.2 **53**

20 Prototype 3 **54**

21 Prototype 4 **55**

22 Work Completed **56**
22.1 What we are doing . . . 56
22.2 Unifiable Data Structures . . . 56
22.3 Why `fix` is necessary? . . . 57

22.4 Dr. Casperson's Explanation 58

23 Results **59**
23.1 Types 59
23.2 Lazy Evaluation 60
23.3 Opening up the Language 60
23.4 Quasi Quotation 60
23.5 Template Haskell 60
23.6 Higher Order Functions 60
23.7 I/O 61
23.8 Mutability 61
23.9 Unification 61
23.10Monads 61

24 Conclusion / Expected Outcomes **62**

Bibliography **63**

List of Figures

1 Functor Hierarchy [121] 56

List of Tables

Abstract

This document looks at the problem of combining programming languages with contrasting and conflicting characteristics which mostly belong to different programming paradigms. The purpose to be fulfilled here is that rather than moulding a problem to fit in the chosen language it must be the other way around that the language adapts to the problem at hand. Moreover, it reduces the need for jumping between different languages. The aim is achieved either by embedding a target language whose features are desirable or to be captured into the host language which is the base on to which the mapping takes place which can be carried out by creating a module or library as an extension to the host language or developing a hybrid programming language that accommodates the best of both worlds.

This research focuses on combining the two most important and wide spread declarative programming paradigms, functional and logical programming. This will include playing with languages from each paradigm, Haskell from the functional side and Prolog from the logical side. The proposed approach aims at adding logic programming features which are native to Prolog onto Haskell by developing an extension which replicates the target language and utilises the advanced features of the host for an efficient implementation.

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 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 programming paradigms, for example Object Oriented Languages. Over the last decade the declarative style of programming has gained popularity. The methodologies that have stood out are the Functional and Logical Approaches. The former is based on Functions and Lambda Calculus, while the latter is based on Horn Clause Logic. Each of them has its own advantages and aws. How does one choose which approach to adopt? Perhaps one does not need to choose! This document looks at the attempts, improvements and future possibilities of uniting Haskell, a Purely Functional Programming Language and Prolog, a Logical Programming Language so that one is not forced to choose.

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

Over the years the development of programming languages has become more and more rapid. Today the number of is in the thousands and counting. The successors attempt to introduce 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 procedures. A new one tries to capture the best of the old; learn from the mistakes, add new concepts and move on; which seems to be good enough from an evolutionary perspective. But all is not that straight forward when shifting from one language to another. There are costs and incompatibilities to look at. A language might be simple to use and provide better performance than its predecessor but not always be worth the switch.

PROLOG is a language that has a hard time being adopted. Born in an era where procedural languages were receiving a lot of attention, it suered from competing against another new kid on the block: C. Some of the problems were of its own making. Basic features like modules were not provided by all compilers. Practical features for real world problems were added in an ad hoc way resulting in the loss of its purely declarative charm. Some say that PROLOG is fading away, [72, 115, 114]. It is apparently not used for building large programs citewikiprolog,somogyi1995logic,website:prolog1000db. However there are a lot of good things about Prolog: it is ideal for search problems; it has a simple syntax, and a strong underlying theory. It is a language that should not die away.

So the question is how to have all the good qualities of PROLOG without actually using PROLOG?

Well one idea is to make PROLOG an add-on to another language which is widely used and in demand. Here the choice is HASKELL; as both the languages are declarative they share a common background which can help to blend the two.

Generally speaking, programming languages with a wide scope over problem domains do not provide bespoke support for accomplishing even mundane tasks. Approaching towards the solution can be complicated and tiresome, but the programming language in question acts as the master key.

Flipping the coin to the other side we see, the more specific the language is to the problem domain 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 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.

2 Background

Programming Languages fall into different categories also known as "paradigms". They exhibit different characteristics according to the paradigm they fall into. It has been argued [57] that rather than classifying a language into a particular paradigm, it is more accurate that a language exhibits a set of characteristics from a number of paradigms. Either way, the broader the scope of a language the more the expressibility or use it has.

Programming Languages that fall into the same family, in our case declarative programming languages, can be of different paradigms and can have very contrasting, conflicting characteristics and behaviours. The two most important ones in the family of declarative languages are the Functional and Logical style of programming.

Functional Programming, [50] gets its name as the fundamental concept is to apply mathematical functions to arguments to get results. A program itself consists of functions and functions only which when applied to arguments produce results without changing the state that is values on variables and so on. Higher order functions allow functions to be passed as arguments to other functions. The roots lie in λ-calculus [138], a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application using variable binding and substitution. It can be thought as the smallest programming language [88], a single rule and a single function definition scheme. In particular there are typed and untyped λ calculi. In the untyped λ calculus functions have no predetermined type whereas typed lambda calculus puts restriction on what sort(type) of data can a function work with. SCHEME is based on the untyped variant while ML and HASKELL are based on typed λ calculus. Most typed λ calculus langauges are based on Hindley-Milner or Damas-Milner or Damas- Hindley-Milner [136] type system. The ability of the type system to give the most general type of a program without any help (annotation). The algorithm [18] works by initially assigning un-

defined types to all inputs, next check the body of the function for operations that impose type constraints and go on mapping the types of each of the variables, lastly unifying all of the constraints giving the type of the result.

Logical Programming, [100] on the other hand is based on formal logic. A program is a set of rules and formulæ in symbolic logic that are used to derive new formulas from the old ones. This is done until the one which gives the solution is not derived.

The languages to be worked with being Haskell and Prolog respectively. Some differences include things like, Haskell uses Pattern Matching while Prolog uses Unification, Haskell is all about functions while Prolog is on Horn Clause Logic and so on.

Prolog [126] being one of the most dominant Logic Programming Languages has spawned a number of distributions and is present from academia to industry.

Haskell is one the most popular [62] functional languages around and is the first language to incorporate Monads [117] for safe *IO*. Monads can be described as composable computation descriptions [124] . Each monad consists of a description of what has action has to be executed, how the action has to be run and how to combine such computations. An action can describe an impure or side-effecting computation, for example, *IO* can be performed outside the language but can be brought together with pure functions inside in a program resulting in a separation and maintaining safety with practicality. Haskell computes results lazily and is strongly typed.

The languages taken up are contrasting in nature and bringing them onto the same plate is tricky. The differences in typing, execution, working among others lead to an altogether mixed bag of properties.

The selection of languages is not uncommon and this not only the case with Haskell, Prolog seems to be the all time favourite for "let's implement Prolog in the language X for proving it's power and expressibility". The Prolog language has been partially implemented [29] in other languages like Scheme [97], Lisp [55, 86, 87],

Java [126, 52], JavaScript [53] and the list [80] goes on and on.

The technique of embedding is a shallow one, it is as if the embedded language floats over the host. Over time there has been an approach that branches out, which is Paradigm Integration. A lot of work has been done on Unifying the Theories of Programming [30, 12, 81, 143, 47, 38]. All sorts of hybrid languages which have characteristics from more than one paradigm are coming into the mainstream.

Before moving on, let us take a look at some terms related to the content above. To begin with Foreign Function Interfaces (FFI) [137], a mechanism by which a program written in one programming language can make use of services written in another. For example, a function written in C can be called within a program written in Haskell and vice versa through the FFI mechanism. Currently the Haskell foreign function interface works only for one language. Another notable example is the Common Foreign Function Interface (CFFI) [11] for Lisp which provides fairly complete support for C functions and data. Java provides the Java Native Interface(JNI) for the working with other languages. Moreover there are services that provide a common platform for multiple languages to work with each other and run their programs. They can be termed as multi lingual run times which lay down a common layer for languages to use each others functions. An example for this is the Microsoft Common Language Runtime (CLR) [133] which is an implementation of the Common Language Infrastructure (CLI) standard [132].

Another important concept is meta programming [140], which involves writing computer programs that write or manipulate other programs. The language used to write meta programs is known as the meta language while the the language in which the program to be modified is written is the object language. If both of them are the same then the language is said to be reflective. Haskell programs can be modified using Template Haskell [44] an extension to the language which provides services to jump between the two types of programs. The abstract syntax trees in the form

of HASKELL data types can be modified at compile time which playing with the code and going back and forth.

A specific tool used in meta programming is quasi quotation [65, 120, 131], permits HASKELL expressions and patterns to be constructed using domain specific, programmer-defined concrete syntax. For example, consider a particular application that requires a complex data type. To accommodate the same it has to represented using HASKELL syntax and preforming pattern matching may turn into a tedious task. So having the option of using specific syntax reduces the programmer from this burden and this is where a quasi-quoter comes into the picture. Template HASKELL provides the facilities mentioned above. For example, consider the following code in PROLOG to append two lists,

```
append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

going through the code, the first rule says that and empty list appended with any list results in the list itself. The second predicate matches the head of the first and the resulting lists and then re-curses on the tails. The same in HASKELL,

```
append(Ps, Qs, Rs) = (Ps = [] & Qs = Rs) ||
(∃ X, Xs, Ys → Ps = [X|Xs] &
Rs = [X|Ys] &
append(Xs, Qs, Ys))
```

Consider the Object Functional Programming Language, SCALA [146], it is purely functional but with objects and classes. With the above in mind, coming back to the problem of implementing PROLOG in HASKELL. There have been quite a few attempts to "merge" the two programming languages from different programming paradigms. The attempts fall into two categories as follows,

1. Embedding, where PROLOG is merely translated to the host language HASKELL or a Foreign Function Interface.

2. Paradigm Integration, developing a hybrid programming language that is a Functional Logic Programming Language with a set of characteristics derived from both the participating languages.

The approaches listed above are next in line for discussions.

3 Proposed Work

3.1 Current Work

There have been several attempts at embedding PROLOG into HASKELL , a few shortcomings are very clear,

1. Only two embeddings exist, one of them is old and made for **hugs** a functional programming system based on the HASKELL 98 specification. It is complex and also lacks a lot of PROLOG like features including *cuts, fails, assert* among others. The second one is based off the first one to make it simple but it loses the variable search strategy support which allows the programmer to choose the manner in which a solution is produced.

2. The papers that try to take the above further are also few in number and do not have any implementations with the proposed concepts. Moreover, none of them are complete and most lack many practical parts of PROLOG.

3. Libraries, a few exist, most are old and are not currently maintained or updated. Many provide only a shell through which one has to do all the work, which is synonymous with the embeddings mentioned above. Some are far more feature rich than others that is with some practical PROLOG concepts, but are not complete.

4. Moreover, none of the above have full list support that exist in PROLOG.

And as far as the idea of merging paradigms goes, it is not the main focus of this thesis and can be more of an "add-on". A handful of crossover hybrid languages based on HASKELL exist, CURRY [113] being the prominent one. Moving away from HASKELL and exploring other languages from different paradigms, a respectable number of crossover implementations exist but again most of them have faded out.

As discussed in the sections above, either an embedding or an integration approach is taken up for programming languages to work together. So, there is either a very shallow approach that does not utilize the constructs available in the host language and results in a mere translation of the characteristics, or the other is a fairly complex process which results in tackling the conflicting nature of different programming paradigms and languages, resulting in a toned-down compromised language that takes advantages of neither paradigms. Mostly the trend is to build a library for extension to replicate the features as an add on.

3.2 Contributions

Taking into consideration above, there is quite some room for improvement and additions. Moving onto what this thesis shall explore, first thing's first a complete, fully functional library which comes close to a PROLOG like language and has practical abilities to carry out real-world tasks. They include predicates like *cut, assert, fail, setOf, bagOf* among others. This would form the first stage of the implementation. Secondly, exploring aspects such as*assert* and database capabilities. A third question to address is the accommodation of input and output, specifically dealing with the *IO Monad* in HASKELL with PROLOG *IO*. Moreover, PROLOG is an untyped language which allows lists with elements of different types to be created. Something like this is not by default in HASKELL. Hence syntactic support for the same is the next question to address. Furthermore, experimenting with how programs expressed with same declarative meaning differ operationally. Lastly, how would characteristics of hybrid languages fit into and play a role in an embedded setting.

4 Embedding a Programming Language into another Programming Language

The art of embedding a programming language into another one has been explored a number of times in the form of building libraries or developing Foreign Function Interfaces and so on. This area mainly aims at an environment and setting where two or more languages can work with each other harmoniously with each one able to play a part in solving the problem at hand. This chapter mainly reviews the content related to embedding Prolog in Haskell but also includes information on some other implementations and embedding languages in general.

4.1 The Informal Content from Blogs, Articles and Internet Discussions

Before moving on to the formal content such as publications, modules and libraries it is time to get *street smart*. This subsection takes a look at the information, thoughts and discussions that are currently taking place from time to time on the internet. A lot of interesting content is generated which has often led to some formal content.

A lot has been talked about embedding languages and also the techniques and methods to do so. It might not seem such a hot topic as such but it has always been a part of any programming language to work and integrate their code with other programming languages. One of the top discussions are in, Lambda the Ultimate, The Programming Languages Weblog [58], which lists a number of Prolog implementations in a variety of languages like Lisp, Scheme, Scala, Java, Javascript, Racket [97] and so on. Moreover the discussion focusses on a lot of critical points that should be considered in a translation of Prolog to the host language regarding types and modules among others.

One of the implementations discussed redirects us to one of the most earliest imple-

mentations of PROLOG in HASKELL for Hugs 98, called Mini PROLOG [54]. Although this implementation takes as reference the working of the PROLOG Engine and other details, it still is an unofficial implementation with almost no documentation, support or ongoing development. Moreover, it comes with an option of three engines to play with but still lacks complete list support and a lot of practical features that PROLOG has and this seems to be a common problem with the only other implementation that exists, [144].

Adding fuel to fire, is the question on PROLOG's existence and survival [114, 72, 115, 98] since its use in industry is far scarce than the leading languages of other paradigms. The purely declarative nature lacks basic requirements such as support for modules. And then there is the ongoing comparison between the siblings [145] of the same family, the family of Declarative Languages. Not to forget HASKELL also has some tricks [118] up its sleeve which enables encoding of search problems.

4.2 Related Books

As HASKELL is relatively new in terms of being popular, its predecessors like SCHEME have explored the territory of embedding quite profoundly [23], which aims at adding a few constructs to the language to bring together both styles of Declarative Programming and capture the essence of PROLOG. Moreover, HASKELL also claims for it to be suitable for basic Logic Programming naturally using the List Monad [119]. A general out look towards implementing PROLOG has also been discussed by [56] to push the ideas forward.

4.3 Related Papers

There is quite some literature that can be found and which consist of embedding detailed parts of Prolog features like basic constructs, search strategies and data types. One of the major works is covered by the subsection below consisting of a

series of papers from Mike Spivey and Silvija Seres aimed at bring Haskell and Prolog closer to each other. The next subsection covers the literature based on the above with improvements and further additions.

- Papers from Mike Spivey and Silvija Seres

  The work presented in the series [102, 94, 95, 101, 92] attempts to encapsulate various aspects of an embedding of PROLOG in HASKELL. Being the very first documented formal attempt, the work is influenced by similar embeddings of PROLOG in other languages like SCHEME and LISP. Although the host language has distinct characteristics such as lazy evaluation and strong type system the proposed scheme tends to be general as the aim here is to achieve PROLOG like working not a multi paradigm declarative language. PROLOG predicates are translated to HASKELL functions which produce a stream of results lazily depicting depth first search with support for different strategies and practical operators such as *cut* and *fail* with higher order functions. The papers provide a minimalistic extension to HASKELL with only four new constructs. Though no implementation exists, the synthesis and transformation techniques for functional programs have been *logicalised* and applied to PROLOG programs. Another related work [103] looks through conventional data types so as to adapt to the problems at hand so as to accommodate and jump between search strategies.

- Other works related or based on the above

  Continuing from above, [17] taps into the advantages of the host language to embed a typed functional logic programming language. This results in typed logical predicates and a backtracking monad with support for various data types and search strategies. Though not very efficient nor practical the method aims at a more elegant translation of programs from one language to the other. While other papers [31] attempt at exercising HASKELL features without adding

anything new rather doing something new with what is available. Specifically speaking, using HASKELL type classes to express general structure of a problem while the solutions are instances. [46] replicates PROLOG's control operations in HASKELL suggesting the use of the HASKELL *State Monad* to capture and maintain a global state. The main contributions are a Backtracking Monad Transformer that can enrich any monad with backtracking abilities and a monadic encapsulation to turn a PROLOG predicate into a HASKELL function.

4.4 Related Libraries in Haskell

- Prolog Libraries

  To replicate Prolog like capabilities Haskell seems to be already in the race with a host of related libraries. First we begin with the libraries about Prolog itself, a few exist [105] being a preliminary or "mini Prolog" as such with not much in it to be able to be uselul, [106] is all powerful but is an Foreign Function Interface so it is "Prolog in Haskell" but we need Prolog for it, [90] which is the only implementation that comes the closest to something like an actual practical Prolog. But all they give is a small interpreter, none or a few practical features, incomplete support for lists, minor or no monadic support and an REPL without the ability to "write a Prolog Program File".

- Logic Libraries

  The next category is about the logical aspects of Prolog, again a handful of libraries do exist and provide a part of the functionality which is related propositional logic and backtracking. [21] is a continuation-based, backtracking, logic programming monad which sort of depicts Prolog's backtracking behaviour. Prolog is heavily based on formal logic, [36] provides a powerful system for Propositional Logic. Others include small hybrid languages [32] and Parallelis-

ing Logic Programming and Tree Exploration [20].

- Unification Libraries

  The more specific the feature the lesser the support in Haskell. Moving on to the other distinct feature of Prolog is Unification, two libraries exist [108], [82] that unify two Prolog Terms and return the resulting substitution.

- Backtracking

  Another important aspect of PROLOG is backtracking. To simulate it in HASKELL, the libraries [33, 99] use monads. Moreover, there is a package for the EGISON programming language [48] which supports non-linear pattern-matching with backtracking.

5 Multi Paradigm Languages (Functional Logic Languages)

Over the years another approach has branched off from embedding languages, to merge and/or integrate programming languages from different paradigms. Let us take an example of the Scala Programming Language [146], a hybrid Object-Functional Programming Language which takes a leaf from each of the two books. In this thesis, the languages in question are Haskell and Prolog. This section takes a look at the literature on Multi Paradigm Languages, mainly Functional Logic Programming Languages that combine two of the most widespread Declarative Programming Styles.

A peak into language classification reveals that it is not always a straight forward task to segregate languages according to their features and/or characteristics. Turns out that there are a number of notions which play a role in deciding where the language belongs. Many a times a language ends up being a part of almost all paradigms due extensive libraries. Simply speaking, a multi-paradigm programming language is a programming language that supports more than one programming paradigm [57], more over as Timothy Budd puts it [142] "The idea of a multi paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms."

5.1 The Informal Content from Blogs, Articles and Internet Discussions

- Multi Paradigm Languages

  A lot has been talked and discussed on coming to clear grounds about the classification of programming languages. If the conventional ideology is considered then the scope of each language is pretty much infinite as small extension

modules replicate different feature sets which are not naturally native to the language itself. The definitions of multi paradigm languages across the web [142, 73, 13] converge to roughly the same thing that of providing a framework to work with different styles with a list of languages [139, 28] that ticks the boxes. Generally speaking, it does not feel all that hot or popular in programming circles; one reason could be that it is a very broad topic and specifying details can clear the fog.

- Functional Logic Programming Languages

  Continuing from the previous section, narrowing down the search by considering only multi paradigm declarative languages namely, Functional Logical programming languages. By doing so a large amount of information pops up, from articles that give brief description and mentions [130, 127] to the implementing techniques [2] which give a brief overview of the aim and also the backdrop of publications.

  The jackpot however is the fact that there is a dedicated website [42] for the history, research and development, existing languages, the literature, the contacts and everything else that one can think of for functional logic languages. As a matter of fact the holy grail of information is maintained by two of the most important people in the field Michael Hanus [40] and Sergio Antoy [3].

5.2 Literature and Publications

- Multi Paradigm Languages

  Possibly one of the most important works towards bringing programming styles together is the book by C.A.R. Hoare [47] which points out that among the large number of programming paradigms and/or theories the unification theory serves as a complementary rather than a replacement to relate the universe. As

as always since we are talking about HASKELL we have to include monads and unifying theories using monads [38].

- Functional Logic Programming Languages

  A recent survey [41] throws light on these hybrid languages.

  One of the most prominent multi paradigm languages in HASKELL is CURRY [4]. Th syntax is borrowed from the parent language and so are a lot of the features. Taking a recap, a functional programming language works on the notion of mathematical functions while a logic programming language is based on predicate logic. The strong points of CURRY are that the features or basis of the language are general and are visible in a number of languages like [25]. The language can play with problems from both worlds. In a problem where there are no unknowns and/or variables the language behaves like a functional language which is pattern matching the rules and execute the respective bodies. In the case of missing information, it behaves like PROLOG; a sub-expression e is evaluated on the conditions that it should satisfy which constraint the possible values of e. This brings us to the first important feature of functional logic languages *narrowing*. The expressions contain *free variables*; simply speaking incomplete information that needs to be *unified* to a value depending on the constraints of the problem. The language introduces only a few new constructs to support non determinism and choice. Firstly, *narrowing* (=:=), which deals with the expressions and unknown values and binds them with appropriate values. The next one is the *choice* operator (?) for non-deterministic operations. Lastly, for unifying variables and values under some conditions, (&) operator has been provided to add constraints to the equation. Putting it all together, it gives us the feel of a logic language for something that looks very much like HASKELL. Unification is like two way pattern matching and with a similar

analogy CURRY is a HASKELL that works both ways and hence variables can be on either sides. Although the language can do a lot but gaps do exist such as the improvement of narrowing techniques.

5.3 Some Multi Paradigm Languages

The list of multi paradigm languages is huge, but in this thesis we will mostly stick to Functional Logical programming languages. Beginning with functional hybrids, a small project language called VIRGIL [112], combining objects to work with functions and procedures. On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [128]. This can be justified as object oriented programming has been one of the most dominant styles of programming and hence even HASKELL has one called O'HASKELL [74] though it last saw a release back in 2001. Another prominent implementation is OCAML [141, 77] which adds object oriented capabilities with a powerful type system and module support. This is the case with most of the languages in this section hardly a few have survived as the new ones incorporated the positives of the old. As mentioned before one of the most poplar [62] and widely usage both in academia and industry is the SCALA [146] programming language stands out.

5.4 Functional Logic Programming Languages

Knowing that there is quite some amount of literature out there on these type of languages, it is fairly easy to say that there have been numerous attempts at specifications and/or implementations. Sadly though not many have survived leave alone being successful as a result of the competition. Only the ones that are easily available or have an implementation or have been cited or referred by other attempts have been included as the list is long and does not reflect the main intention of the document. Beginning with the ones from Australia, which seems to be a popular destination for fiddling with PROLOG and merging paradigms. As of now there

have been three popular ones, beginning with Neu Prolog, [63], Oz (Mozart Programming System) [19] and Mercury [26]. Delving deeper the languages feel more like extensions of Prolog rather than hybrids. Starting with Mercury which a boundary between deterministic and non-deterministic programs, similarly Nue Prolog has special support for functions while Oz gives concurrent constraint programming plus distributed support, with different function types for goal solving and expression rewriting. Escher [64] comes very close to Haskell with monads, higher order functions and lazy evaluation. Taking a look at Prolog variants, Ciao [16]; a preprocessor to Prolog for functional syntax support, λ Prolog [71] aims at modular higher order programming with abstract data types in a logical setting, Babel [45, 69, 68] combines pure Prolog with a first order functional notation, LIFE [111] is for Logic, Inheritance, Functions and Equations in Prolog syntax with currying and other features like functional languages and others [10, 66].

The functional language Scheme is a very popular choice for this sort of a thing. With a book [23] and an implementation to accompany [24, 107] which seems to have translated into Haskell, [51, 34, 116].

Finally talking about Curry, one of the most popular Haskell based multi paradigm languages with support for deterministic and non-deterministic computations. Contributing to the same there have been some predecessors [110, 25].

6 Related Work

There are some technicalities which are indirectly related to the problem but do not bare a point of contact. The underpinnings of the languages throw some more light on the how different languages work to solve a problem. Different programming paradigms incorporate different operational mechanisms. For example, PROLOG programs execute on the Warren Abstract Machine [1] which has three different storage usages; a global stack for compound terms, for environment frames and choice points and lastly the trail to record which variables bindings ought to be undone on backtracking.

Constraint programming [135] is closely related to the declarative programming paradigm in the sense that the relations between variables is specified in the form of constraints. For example, consider a program to solve a simultaneous equation, now adding on to that restricting the range of the values that the variables can possible take, thus adding constraints to the possible solutions. Related to the same are Constraint Handling Rules [134], which are extensions to a language, simply speaking adding constraints to a language like PROLOG.

Lastly some details on the working of functional logic programming languages, residuation and narrowing [43, 129]. Residuation involves delaying of functions calls until they are deterministic, that is, deterministic reduction of functions with partial data. This principle is used in languages like ESCHER [64], LIFE [111], NUE-PROLOG [63] and OZ [19]. Narrowing on the other hand is a mixture of reduction in functional languages and unification in logic languages. In narrowing, a variable is bound a value within the specified constraints and try to find a solution, values are generated while searching rather than just for testing. The languages based on this approach are ALF [110], BABEL [45], LPG [10] and CURRY [113].

7 Embedding a Programming Language into another Programming Language

Embedding a language into another language has been explored with a variety of languages. Attempts have been made to build Domain Specific Languages from the host languages [49], Foriegn Function Interfaces [8]

Creating a programming language from scratch is a tedious task requiring ample amount of programming, not to mention the effort required in designing. A typical procedure would consist of formulating characteristics and properties based on the following points,

1. Syntax
2. Semamtics
3. Standard Library
4. Runtime Sytsem
5. Parsers
6. Code Generators
7. Interpreters
8. Debuggers

A lot of the above can be skipped or taken from the base language if an embedding approach is chosen. For an embedded domain specific language the functionality is translated and written as an add on. The result can be thought of as a library. But the difference between an ordinary library and an eDSl is the feature set provided and the degree of embedding [123]. For example, reading a file and parsing its contents

to perform certain operations to return *string* results is a shallow form of embedding as the generation of code, results is not native nor are the functions processing them dealing with embedded data types as such. On the other hand, building data structures in the base language which represent the target language expression would be called a deep embedding approach.

The snippet of HASKELL code below describes PROLOG entities,

```
1 data Term = Struct Atom [Term]
2            | Var VariableName
3            | Wildcard
4            | PString   !String
5            | PInteger  !Integer
6            | PFloat    !Double
7            | Flat [FlatItem]
8            | Cut Int
9         deriving (Eq, Data, Typeable)
```

The above can be described as concrete syntax for the "new" language and can be used to write a program.

As discussed in the

7.1 Theory

1. Papers

   (a) Embedding an interpreted language using higher-order functions, [83]

   (b) Building domain-specific embedded languages, [49]

   (c) Embedded interpreters, [9]

   (d) Cayenne – a Language With Dependent Types, [5]

(e) Foreign interface for PLT Scheme, [8]

(f) Dot-Scheme: A PLT Scheme FFI for the .NET framework, [78]

(g) Application-specific foreign-interface generation, [84]

(h) Embedding S in other languages and environments, [61]

2. Books

(a) ?????????

3. Articles / Blogs / Discussions

(a) Embedding one language into another, [59]

(b) Application-specific foreign-interface generation, [60]

(c) Linguistic Abstraction, [75]

(d) LISP, Unification and Embedded Languages, [76]

4. Websites

(a) Embedding SWI-Prolog in other applications, [29]

7.2 Implementations

1. Lots of them I guess

7.3 Important People

1. ????

7.4 Miscellaneous / Possibly Related Content

1. ????

8 Prolog in ____

Prolog in _____

8.1 Theory

- Papers
    1. QLog, [55]
    2. LogLisp Motivation, design, and implementation, [86]
- Books
    1. Warrens Abstract Machine A TUTORIAL RECONSTRUCTION, [1]
    2. LOGLISP: an alternative to PROLOG, [87]
- Articles / Blogs / Discussions
    1. Hello
- Websites
    1. Hello

8.2 Implementations

1. Castor : Logic paradigm for C++, [70]
2. GNU Prolog for Java, [39]
3. JLog - Prolog in Java, [52]
4. JScriptLog - Prolog in Java, [53]
5. Quintus Prolog, [79]

6. Yield Prolog, [80]

7. Racklog, [97]

8.3 Important People

1. ???

8.4 Miscellaneous / Possibly Related Content

1. ???

9 Prolog in Haskell

Prolog in Haskell

9.1 Theory

- Papers

    1. Embedding Prolog in Haskell / Functional Reading of Logic Programs, [102]
    2. Algebra of Logic Programming, [94]
    3. The Algebra of Logic Programming, [92]
    4. Optimisation Problems in Logic Programming : An Algebraic Approach, [93]
    5. Higher Order Transformation of Logic Programs, [95]
    6. The Algebra of Searching, [101]
    7. FUNCTIONAL PEARL Combinators for breadth-first search, [103]
    8. Type Logic Variables, K Classen, [17]
    9. A Type-Safe Embedding of Constraint Handling Rules into Haskell Wei-Ngan Chin, Mar-tin Sulzmann and Meng Wang, [15]
    10. Prological Features in a Functional Setting Axioms and Implementation, R Hinze, [46]
    11. Escape from Zurg: An Exercise in Logic Programming, [31]

- Books

    1. The Reasoned Schemer, Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, [23]

2. Programming Languages: Application and Interpretation, Shriram Krishnamurthi, Chapters 33-34 of PLAI discuss Prolog and implementing Prolog, [56]

- Articles / Blogs / Discussions

1. Lambda the Ultimate, Programming Languages, [58]
2. Takashi's Workplace (Implementation), [144]
3. Haskell vs. Prolog Comparison, [104]

- Websites

1. Logic Programming in Haskell, [118]

9.2 Implementations

1. A Prolog in Haskell, Takashi's Workplace, [144]
2. Mini Prolog for Hugs 98, [54]
3. Nano Prolog, [105]
4. Prolog, [90]
5. cspm-To-Prolog, [35]
6. prolog-graph, [7]
7. prolog-graph-lib, [89]
8. hswip, [106]

9.3 Important People

1. Mike Spivey
2. Silvija Seres

9.4 Miscellaneous / Possibly Related Content

1. Unification Libraries
   (a) unification-fd, [108]
   (b) cmu, [82]
2. Logic Libraries
   (a) logicct, [21], [22]
   (b) logic-classes, [**?**]
   (c) proplogic, [36]
   (d) cflp, [32]
   (e) logic-grows-on-trees, [20]
3. Concatenative Programming
   (a) peg, [27]
4. Constraint Programming and Constraint Handling Rules
   (a) monadiccp, [85]
   (b) monadicccp-gecode, [109]
   (c) csp, [6]
   (d) liquid fix point, [91]

10 Unifying or Marrying or Merging or Combining Programming Paradigms or Theories

Unifying / Marrying / Merging / Combining Programming Paradigms / Theories

10.1 Theory

- Papers
    1. Unifying Theories of Programming with Monads, [38]
    2. Symposium on Unifying Theories of Programming, 2006, [30].
    3. Symposium on Unifying Theories of Programming, 2008, [12].
    4. Symposium on Unifying Theories of Programming, 2010, [81].
    5. Symposium on Unifying Theories of Programming, 2012, [143].
- Books
    1. Unifying Theories of Programming, [47]
- Articles / Blogs / Discussions
    1. ???
- Websites
    1. ???

10.2 Implementations

1. Scala
2. Virgil

3. CLOS, Common Lisp Object System

4. Visual Prolog

5. ????

10.3 Miscellaneous / Possibly Related Content

1. ???

11 Functional Logic Programming Languages

Functional Logic Programming Languages

11.1 Theory

- Paper
    1. FLPL Introdunction Theory
        (a) Hello
    2. FLPL Surveys
        (a) Hello
    3. Narrowing in FLPL
        (a) Hello
    4. Residuation in FLPL
        (a) Hello
    5. Computation Model for FLPL
        (a) Hello
- Books
    1. Hello
- Articles / Blogs / Discussions
    1. Hello
- Websites
    1. Hello

11.2 Implementations

1. Hello

11.3 Miscellaneous / Possibly Related Content

1. Hello

12 Quasiquotation

12.1 Theory

1. Papers
    (a)
2. Books
    (a)
3. Articles / Blogs / Discussions
    (a)
4. Websites
    (a) Quasiquotation Wikipedia, [131]
    (b) Quasiquotation in Haskell, [120]

12.2 Implementations

1.

12.3 Miscellaneous / Possibly Related Content

1.

13 Related Terms or Keywords

Related Terms / Keywords

1. Prolog in Other Languages
2. Prolog in Haskell
3. Embedding One language into another language
4. Constraint Programming
5. Constraint Handling Rules
6. Concatenative Programming
7. Functional Logic Programming Languages
8. Residuation
9. Narrowing
10. Warren Abstraction Machine
11. Foreign Function Interfaces
12. Quasiquotation
13. Programming Theory Unification

14 Haskell or Why Haskell ?

Haskell / Why Haskell ?

1. Haskell as a functional programming language

   Haskell is an advanced purely-functional programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language [122]. It is one of the popular functional programming languages [62]. Haskell is widely used in the industry [125].

2. Haskell as a tool for embedding domain specific languages

   (a) Monads

   Each language has a flow of control, a model how programs are executed. Many languages for example Java have a top-down sequential execution approach. But very few languages allow control flow modelling, for example in a functional language how the propagation of side effects is handled. A monad provides the ability to customize and develop your own model from one step to another, how side effects are handled.

http://paulspontifications.blogspot.ca/2008/01/why-haskell-is-good-for-embedded-domain.html

Why Haskell is Good for Embedded Domain Specific Languages Domain Specific Languages (DSLs) are attracting some attention these days. They have always been around, of course: Emacs Lisp is a DSL, as are the various dialects of Visual Basic embedded in MS Office applications. And of course Unix hands know YACC (now Bison) and Lex (now Flex).

However creating a full-blown language is a lot of work: you have to write a parser, code generator / interpreter and possibly a debugger, not to mention

all the routine stuff that every language needs like variables, control structures and arithmetic types. An embedded DSL (eDSL) is basically a short cut if you can't afford to do that. Instead you write the domain-specific bits as a library in some more general purpose "host" language. The uncharitable might say that "eDSL" is just another name for "library module", and its true there is no formal dividing line. But in a well designed eDSL anything you might say in domain terms can be directly translated into code, and a domain expert (i.e. a non-programmer) can read the code and understand what it means in domain terms. With a bit of practice they can even write some code in it.

This paper describes an eDSL for financial contracts built in Eiffel which worked exactly that way. It doesn't talk about "domain specific language" because the term hadn't been invented back then, but the software engineers defined classes for different types of contracts that the financial analysts could plug together to create pricing models. Its interesting to compare it with this paper about doing the same thing in Haskell.

But eDSLs have problems. The resulting programs are often hard to debug because a bug in the application logic has to be debugged at the level of the host language; the debugger exposes all the private data structures, making it hard for application programmers to connect what they see on the screen with the program logic. The structure of the host language also shows through, requiring application programmers to avoid using the eDSL functions with certain constructs in the host language.

This is where Haskell comes in. Haskell has three closely related advantages over other languages:

Monads. The biggest way that a host language messes up an eDSL is by imposing a flow of control model. For example, a top-down parser library is effectively

an eDSL for parsing. Such a library can be written in just about any language. But if you want to implement backtracking then its up to the application programmer to make sure that any side effects in the abandoned parse are undone, because most host languages do not have backtracking built in (and even Prolog doesn't undo "assert" or "retract" when it backtracks). But the Parsec library in Haskell limits side effects to a single user-defined state type, and can therefore guarantee to unwind all side effects. More generally, a monad defines a model for flow of control and the propagation of side effects from one step to the next. Because Haskell lets you define your own monad, this frees the eDSL developer from the model that all impure languages have built in. The ultimate expression of this power is the Continuation monad, which allows you to define any control structure you can imagine.

Laziness. Haskell programmers can define infinite (or merely very large) data structures because at at any given point in the execution only the fragment being processed will actually be held in memory. This also frees up the eDSL developer from having to worry about the space required by the evaluation model. (update: this isn't actually true. As several people have pointed out, while laziness can turn O(n) space into O(1), it can also turn O(1) into O(n). So the developers do have to worry about memory, but lazy evaluation does give them more options for dealing with it.)

The type system allows very sophisticated constraints to be placed on the use of eDSL components and their relationships with other parts of the language. The Parsec library mentioned above is a simple example. All the library functions return something of type "Parser foo", so an action from any other monad (like an IO action that prints something out) is prohibited by the type system. Hence when the parser backtracks it only has to unwind its internal state, and not the rest of the universe.

There are other programming languages that are good for writing eDSLs, of course. Lisp and Scheme have callCC and macros, which together can cover a lot of the same ground. Paul Graham's famous "Beating the Averages" paper talks about using lots of macros, and together with his patent for continuation-based web serving it is pretty clear that what he and Robert Morris actually created was an eDSL for web applications, hosted in Lisp.

But I still think that Haskell has the edge. I'm aware of the Holy War between static and dynamic type systems, but if I you put a Haskell eDSL in front of a domain expert then you only have to explain a compiler type mismatch message that points to the offending line. This is much easier to grasp than some strange behaviour at run time, especially if you have to explain how the evaluation model of your eDSL is mapped down to the host language. Non-programmers are not used to inferring dynamic behaviour from a static description, so anything that helps them out at compile time has to be a Good Thing. And its pretty useful for experienced coders too.

(Update: I should point out that monads can be done in any language with lambdas and closures, and this is pretty cool. But only in Haskell are they really a native idiom)

15 Prolog or Why Prolog ?

Prolog / Why Prolog ?

1. PROLOG as a logic programming language.

   http://eliminatingwork.blogspot.ca/2010/02/why-prolog-is-by-far-best-most.html

   First of all I will only advocate the use of pure prolog - that means no recursion, lists, forall's, and any other features. Extra features that were added destroy the whole point of the elegance of prolog.

   A prolog equivalent is pervasively used everywhere right under everyone's nose - sql. Pure prolog is almost exactly the same as relational database sql, except that sql has a much worse syntax and requires declaring column names. Column names are a necessary thing for sql's use case (use by many programmers/dba's over the years), but there is no excuse for the sql syntax (attempts were made in the past to get relational databbases to get prolog syntax in the form of datalog but to no avail).

   Business rules engines used in many "enterprise" application servers are also shoddy versions of prolog (when they're backward chaining. Forward chaining is inferior to backward chaining, which implies that all these rules engines should be embedded prolog's if the implementors had bothered to study up history).

   Sparql is also prolog, except that you can only have facts (predicates) with three arguments.

   Disregard procedural and object oriented languages - there's plenty of other blogs/essays/textbooks/papers that tell why those are wrong. Some good exlanations are in Paul Graham's and Peter Norvig's writings (google it - why lisp, dynamic languages, on lisp, paradigms of ai).

   So the question is, why is prolog better than functional languages lisp,haskell,ml,ruby

etc.

Here's why, in order of increasing importance

1) The syntax is incredibly simple (like lisp), and incredibly elegant (unlike lisp and any other language). ( ) . :- , " ! ; fail repeat write read assert retract those are all the researved characters/words you need (if you're working in pure prolog which I'm advocating)

I won't bother explaining it here for newbie's - the following is a good start

A prolog introduction for hackers http://www.kuro5hin.org/story/2004/2/25/124713/784

2) You don't need to know recursion, lambda's, closures, folds, monads, side effects, pattern matching, map/reduce's, flatten, cyclic this and that, blah blah blah rocket science. You don't even need to know data structures - forget linked lists,arrays,trees,graphs In fact, forget algorithms. If you're working in prolog the entirety of computer science is irrelevant (for programming in domains other than computer science itself of course. If you're implementing machine learning/computer vision /database systems/operating systems you would have to know computer science - but you could more elagantly do those tasks in prolog than other languages as well).

3) Prolog programs can usually be translated to/from readable english with a simple regular expression s/(/ is /g s/)././g s/:-/ implies /g s/,/ and /g s/;/ or /g s/fail/try the next choice/g s/!/abandon this line of reasoning/g s/[capital letter X]/[unknown X]/g (I can't be bothered to figure out how to do this in regex, if it's possible at all)

You could even write your program in a spreadsheet, export as csv, and convert to prolog with equally simple as above regexp (and the reverse prolog-¿spreadsheet).

What this means is that most/all of your program can be written by a non-

programmer (similar to how cobol and sql were intended, except that it has a better chance of working this time because the greater inherent simplicity of the syntax).

4) When you program in prolog, you're almost always just creating a description of the world in small "orthogonal" chunks without any concious effort to do so. In functional programming there are builtin "orthogonals", like map/filter etc, but most of the time you have to work very hard to make sure you're writing elegant concise code.

Good functional programmers keep refactoring their code sitting in their repl. Prolog programs just seem to pour out in a concise form that is the only way it can be written. If this seems miraculous - just imagine that you're actually writing sql code. There is only one obvious way to write a sql query (forget performance concerns - you're not managing millions of rows of data as you are in an actual sql database). In sql all you're ever writing is queries and views. In prolog all you're ever writing is queries (views are just queries in prolog).

Sql programmers don't think about code refactoring - there's only one way to do it, and it's the most concise way as well, and orthogonal to all other sql code. As do prolog programmers.

```
conclusion :- prolog_rules.
prolog_rules :- write("Prolog is by far the best, most productive,
```

2. Why embed Prolog ?

16 Miscellaneous or Possibly Related Content

Miscellaneous / Possibly Related Content

1. ???

17 Prototype 1

This chapter looks into solving the issue of conflicting type systems of the languages in question. HASKELL is a strong statically typed language requiring type signature for programming constructs at compile time while PROLOG is strong dynamically typed which lets through untyped programs. This prototype throws light on the process of tackling the issues involved in creating a data type to replicate the target language type system while conforming to the host language restrictions and also utilizing the benefits.

17.1 Creating a data type

A type system consists of a set of rules to define a "type" to different constructs in a programming language such as variables, functions and so on. A static type system requires types to be attached to the programming constructs before hand which results in finding errors at compile time and thus increase the reliability of the program. The other end is the dynamic type system which passes through code which would not have worked in former environment, it comes of as less rigid.

The advantages of static typing [67]

1. Earlier detection of errors
2. Better documentation in terms of type signatures
3. More opportunities for compiler optimizations
4. Increased run-time efficiency
5. Better developer tools

For dynamic typing

1. Less rigid

2. Ideal for prototyping / unknown / changing requirements or unpredictable behaviour

3. Re-usability

Transitional paragraph hello

To start with, replicating the single type "term" in PROLOG one must consider the distinct constructs it can be associated to such as complex structures (for example, predicated clauses etc.), don't cares, cuts, variables and so on.

```
1  --david-0.2.0.2
2
3  data VariableName = VariableName Int String
4         deriving (Eq, Data, Typeable, Ord)
5
6  data Atom          = Atom      !String
7                     | Operator  !String
8         deriving (Eq, Ord, Data, Typeable)
9
10 data Term = Struct Atom [Term]
11           | Var VariableName
12           | Wildcard
13           | PString   !String
14           | PInteger  !Integer
15           | PFloat    !Double
16           | Flat [FlatItem]
17           | Cut Int
18        deriving (Eq, Data, Typeable)
19
```

```
20 data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
21              | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
22       deriving (Data, Typeable)
23
24 type Program = [Sentence]
25
26 type Body    = [Goal]
27
28 data Sentence = Query   Body
29                | Command Body
30                | C Clause
31       deriving (Data, Typeable)
```

Even though *Term* has a number of constructors the resulting construct has a single type. Hence, a function would still be untyped / singly typed,

```
append :: [Term] -> [Term] -> [Term]
```

The above data type is recursive as seen in the constructor,

```
Struct Atom [Term]
```

One of the issues with the above is that it is not possible to distinguish the structure of the data from the data type itself [96]. Consider the following, a reduced version of the above data type,

```
1 type Atom         = String
2
3 data VariableName = VariableName Int String
4       deriving (Eq, Data, Typeable, Ord)
5
```

```
6  data Term = Struct Atom [Term]
7            | Var VariableName
8            | Wildcard -- Don't cares
9            | Cut Int
10       deriving (Eq, Data, Typeable)
```

To split a data type into two levels, a single recursive data type is replaced by two related data types. Consider the following,

```
1  data FlatTerm a =
2                    Struct Atom [a]
3           |        Var VariableName
4           |        Wildcard
5           |        Cut Int deriving (Show, Eq, Ord)
```

One result of the approach is that the non-recursive type *FlatTerm* is modular and generic as the structure "FlatTerm" is separate from it's type which is "a". Simply speaking we can have something like

```
FlatTerm Bool
```

and a generic fuinction like,

```
map :: (a -> b) -> FlatTerm a -> FlatTerm b
```

17.2 Working with the language

Creating instances,

```
1  instance Functor (FlatTerm) where
2          fmap = T.fmapDefault
3
```

```
4  instance Foldable (FlatTerm) where
5           foldMap = T.foldMapDefault
6
7  instance Traversable (FlatTerm) where
8            traverse f (Struct atom x)        =        Struct atom <$>
9                                          sequenceA (Prelude.map f x)
10           traverse _ (Var v)          =         pure (Var v)
11           traverse _ Wildcard          =          pure (Wildcard)
12           traverse _ (Cut i)          =          pure (Cut i)
13
14 instance Unifiable (FlatTerm) where
15         zipMatch (Struct al ls) (Struct ar rs) =
16                 if (al == ar) && (length ls == length rs)
17                         then Struct al <$>
18                                 pairWith (\l r -> Right (l,r)) ls rs
19                         else Nothing
20         zipMatch Wildcard _ = Just Wildcard
21         zipMatch _ Wildcard = Just Wildcard
22         zipMatch (Cut i1) (Cut i2) = if (i1 == i2)
23                 then Just (Cut i1)
24                 else Nothing
25
26 instance Applicative (FlatTerm) where
27         pure x = Struct "" [x]
28         _ <*> Wildcard          =          Wildcard
29         _ <*> (Cut i)           =          Cut i
30         _ <*> (Var v)          =         (Var v)
```

```
31          (Struct a fs) <*> (Struct b xs) = Struct (a ++ b) [f x | f <- f
```

After flattening do fixing,

Opening up the language somehow so as to accommodate your own variables .

17.3 Black box

18 Prototype 2.1

19 Prototype 2.2

20 Prototype 3

21 Prototype 4

22 Work Completed

22.1 What we are doing

A partial implementation of the logic programming language PROLOG is provided by the library `prolog-0.2.0.1`. One of the objectives is to implement monadic unification using the library [108].

22.2 Unifiable Data Structures

For a data type to be Unifiable, it must have instances of Functor, Foldable and Traversable. The interaction between different classes is depicted in figure 1.

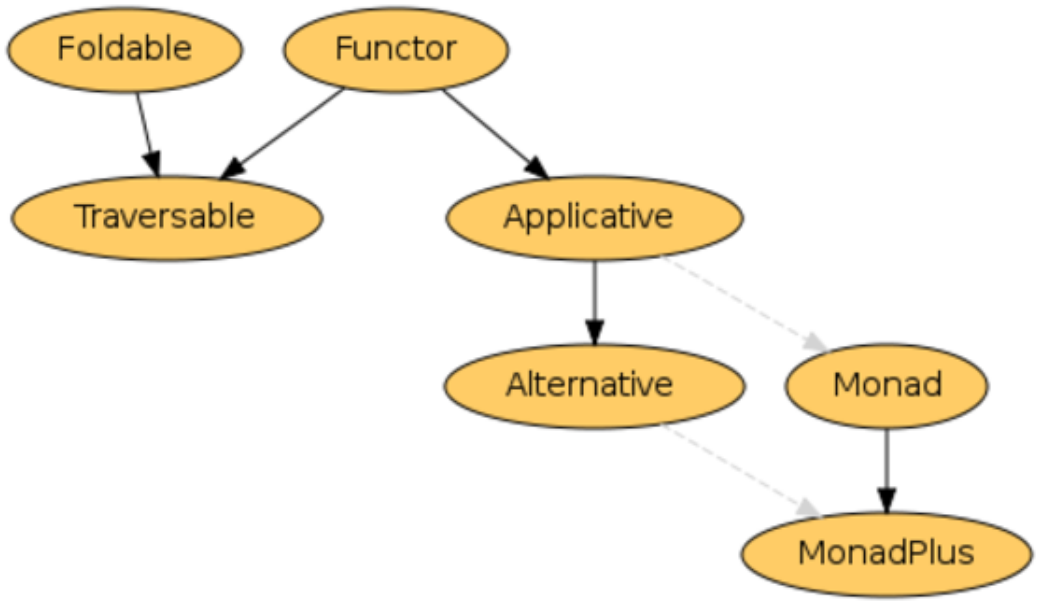

Figure 1: Functor Hierarchy [121]

The Functor class provides the `fmap` function which applies a particular operation to each element in the given data structure. The Foldable class *folds* the data structure by recursively applying the operation to each element and

22.3 Why `fix` is necessary?

Since HASKELL is a lazy language it can work with infinite data structures. *Type Synonyms* in HASKELL cannot be self referential.

In our case consider the following example,

```
-- The Prolog Syntax

type Atom = String

data VariableName = VariableName Int String  deriving (Show,Eq,Ord)

data FlatTerm a =
                  Struct Atom [a]
          |       Var VariableName
          |       Wildcard
          |       Cut Int deriving (Show,Eq,Ord)
```

A `FlatTerm` can be of infinite depth which due to the reason stated above cannot be accounted for during application function. The resulting type signature would be of the form,

```
FlatTerm (FlatTerm (FlatTerm (FlatTerm (.....))))
```

Enter the `Fix` same as the function as a data type. The above would be simply reduced to,

```
Fix FlatTerm
```

resulting in the PROLOG Data Type

```
data Prolog = P (Fix FlatTerm) deriving (Show,Eq,Ord)
```

22.4 Dr. Casperson's Explanation

A recursive data type in HASKELL is where one value of some type contains values of that type, which in turn contain more values of the same type and so on. Consider the following example.

```
data Tree = Leaf Int | Node Int (Tree) (Tree)
```

A sample `Tree` would be,

```
(Node 0 (Leaf 1) (Node 2 (Leaf 3) (Leaf 4)))
```

The above structure can be infinitely deep since HASKELL is a *lazy* programming language. But working with an infinitely deep / nested structure is not possible and will result in a *occurs check* error. This is because writing a type signature for a function to deal with such a parameter is not possible. One option would be to *flatten* the data type by the introduction of a type variable. Consider the following,

```
data FlatTree a = Leaf Int | Node Int a a
```

A sample `FlatTerm` would be similar to `Tree`.

The `FlatTree` is recursive but does not reference itself. But it too can be infinitely deep and hence writing a function to work on the structure is not possible.

The `fix` function in the `Control.Monad.Fix` module allows for the definition of recursive functions in HASKELL. Consider the following scenario,

```
fix :: (a -> a) -> a
```

The above function results in an infinite application stream,

```
f s : f (f (f (... )))
```

A fixed point of a function f is a value a such that f a == a. This is where the name of fix comes from: it finds the least-defined fixed point of a function.

23 Results

23.1 Types

One of the major differences between PROLOG and HASKELL is how each language handles types. PROLOG is an untyped language meaning any operation can be performed on the data irrespective of its type. HASKELL on the other hand is strongly typed i.e. each operation requires a signature stating what types of data it can work with. Moreover, the HASKELL type system is static.

PROLOG like any other language can work with some basic data types like numbers, characters, strings among others. Using these one can make terms like *Atoms*, *Clauses*, *Constants*, *Strings*, *Characters*, *Predicates*, *Structures*, *Special Characters* and so on. These need to be incorporated into the implementation so as to give a palette for writing programs.

Our preliminary implementation is as follows,

```
type Atom = String

data VariableName = VariableName Int String  deriving (Show,Eq,Ord)

data FlatTerm a =
                 Struct Atom [a]
          |        Var VariableName
          |       Wildcard
          |       Cut Int deriving (Show,Eq,Ord)

{--
Output :-
```

```
Struct "a" [Var (VariableName 0 "x"),Cut 0,Wildcard,Struct "b" []]

--}
```

which in PROLOG would look like,

```
a(X, !, b).
```

23.2 Lazy Evaluation

23.3 Opening up the Language

Flattening

Fixing

MetaSyntactic Variables

23.4 Quasi Quotation

23.5 Template Haskell

23.6 Higher Order Functions

```
% Mehul Solanki.

% Higher Order Functions.

% The following library contains the maplist function.
:- use_module(library(apply)).

% The maplist function takes a function and a list to apply the
```

```
% function.
% The function write is passes which will print out the elements
% of the list.
higherOrder(X) :- maplist(write,X).


/*
higherOrder([1,2,3,4]).
1234
true
*/
```

23.7 I/O

```
data Result = Ordinary _____________ --No I/O required
| SideEffect (IO ____________)          -- Requiring Output
| ReadEffect (IO _____ -> Result)          -- Requiring Input
```

23.8 Mutability

23.9 Unification

23.10 Monads

24 Conclusion / Expected Outcomes

The aim of this study is to experiment with two different languages working together and/or contributing in providing a solution. Mixing and matching conflicting characteristics may lead to a behaviour similar to that of a multi paradigm language. The points to be looked at are efficiency of the emulation, semantics of the resulting embedding.

Moreover, this will be an attempt to answer the question how practical Prolog fits into Haskell.

Bibliography

[1] Hassan Aït-Kaci and Forêt Des Flambertins. Warrens abstract machine a tutorial reconstruction. 1999.

[2] Sergio Antoy. Implementing functional logic programming languages.

[3] Sergio Antoy. Sergio antoy home page.

[4] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[5] Lennart Augustsson. Cayenne – a language with dependent types. In *IN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING*, pages 239–250. ACM Press, 1998.

[6] Andrei Barbu. The csp package, August 2013. `http://hackage.haskell.org/package/csp`.

[7] Matthias Bartsch. The prolog-graph package, September 2011. `http://hackage.haskell.org/package/prolog-graph`.

[8] Eli Barzilay and Dmitry Orlovsky. Foreign interface for plt scheme. *on Scheme and Functional Programming*, page 63, 2004.

[9] Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

[10] Didier Bert, Pascal Drabik, and Rachid Echahed. Lpg: A generic, logic and functional programming language. In *STACS 87*, pages 468–469. Springer, 1987.

[11] James Bielman and Lus Oliveira. Common lisp foreign function interface, March 2014.

[12] Andrew Butterfield, editor. *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*. Springer, 2010.

[13] C2. Multi paradigm programming language, September 2012.

[14] Prolog Development Center. Visual prolog, June 2013.

[15] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A type-safe embedding of constraint handling rules into haskell. *Technical reportSchool of Computing, National University of Singapore, Boston, MA, USA*, 2003.

[16] Ciao. Ciao programming language, August 2011.

[17] Koen Claessen and Peter Ljunglöf. Typed logical variables in haskell. *Electr. Notes Theor. Comput. Sci.*, 41(1):37, 2000.

[18] Code Commit. Hindley milner type system, December 2008.

[19] Mozart Consortium. Oz / mozart, March 2013.

[20] Gregory Crosswhite. The logicgrowsontrees package, September 2013. http://hackage.haskell.org/package/LogicGrowsOnTrees.

[21] DanDoel. The logict package, August 2013. http://hackage.haskell.org/package/logict.

[22] DanDoel. The logict package example, August 2013. http://okmij.org/ftp/Computation/monads.html.

[23] Oleg Kiselyov Daniel P. Friedman, William E. Byrd. *The Reasoned Schemer*. The MIT Press, Cambridge Massachusetts, London England, 2005.

[24] William E. Byrd Daniel P. Friedman and Oleg Kiselyov. Kanren, March 2009.

[25] Universidad Complutense de Madrid. Toy, Decmeber 2006.

[26] University Of Melbourne Computer Science department. Mercury programming language, February 2014.

[27] Dustin DeWeese. The peg package, April 2012. http://hackage.haskell.org/package/peg.

[28] Open Directory Project dmoz. Multi paradigm, November 2013.

[29] SWI Prolog Documentation. Embedding swi-prolog in other applications, June 2013. http://www.swi-prolog.org/pldoc/man?section=embedded.

[30] Steve Dunne and Bill Stoddart, editors. *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*. Springer, 2006.

[31] Martin Erwig. Escape from zurg: an exercise in logic programming. *Journal of Functional Programming*, 14(03):253–261, 2004.

[32] Sebastian Fischer. The cflp package, June 2009. http://hackage.haskell.org/package/cflp.

[33] Sebastian Fischer. stream-monad, September 2012.

[34] Adam C. Foltzer. Molog, March 2013.

[35] Marc Fontaine. The cspm-toprolog package, August 2013. http://hackage.haskell.org/package/CSPM-ToProlog.

[36] David Fox. The proplogic package, April 2012. http://hackage.haskell.org/package/PropLogic.

[37] David Fox. The logic-classes package, October 2013. http://hackage.haskell.org/package/logic-classes.

[38] Jeremy Gibbons. Unifying theories of programming with monads. In *Unifying Theories of Programming*, pages 23–67. Springer, 2013.

[39] GNU. Gnu prolog for java, August 2010. http://www.gnu.org/software/gnuprologjava/.

[40] Michael Hanus. Michael hanus home page.

[41] Michael Hanus. Multi-paradigm declarative languages. In *Logic Programming*, pages 45–75. Springer, 2007.

[42] Michael Hanus. Functional logic programming, February 2009.

[43] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS*, volume 95, pages 95–107, 1995.

[44] Haskellwiki. Template haskell, October 2013.

[45] Juan Jose Moreno Navarro Herbert Kuchen. Babel programming language, January 1988.

[46] Ralf Hinze et al. Prological features in a functional setting axioms and implementation. In *Fuji International Symposium on Functional and Logic Programming*, pages 98–122. Citeseer, 1998.

[47] Charles Anthony Richard Hoare and Jifeng He. *Unifying theories of programming*, volume 14. Prentice Hall Englewood Cliffs, 1998.

[48] Satoshi Egi Ryo Tanaka Takahisa Watanabe Kentaro Honda. Egison package, March 2014.

[49] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[50] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[51] JaimieMurdock. Haskell kanren, March 2012.

[52] JLogic. Jlog - prolog in java, September 2012. http://jlogic.sourceforge.net/index.html.

[53] JLogic. Jscriptlog - prolog in javascript, September 2012. http://jlogic.sourceforge.net/index.html.

[54] Mark P Jones. Mini-prolog for hugs 98, June 1996. http://darcs.haskell.org/hugs98/demos/prolog/.

[55] H Jan Komorowski. Qlog: The programming environment for prolog in lisp. *Logic Programming*, pages 315–324, 1982.

[56] Shriram Krishnamurthi. *Programming languages: Application and interpretation*, chapter 33-34, pages 295–305, 307–311. Brown Univ., 2007.

[57] Shriram Krishnamurthi. Teaching programming languages in a post-linnaean age. *SIGPLAN Not.*, 43(11):81–83, November 2008.

[58] The Programming Languages Weblog Lambda The Ultimate. Embedding prolog in haskell, July 2004. http://lambda-the-ultimate.org/node/112.

[59] The Programming Languages Weblog Lambda The Ultimate. Embedding one language into another, March 2005. http://lambda-the-ultimate.org/node/578.

[60] The Programming Languages Weblog Lambda The Ultimate. Application-specific foreign-interface generation, October 2006. http://lambda-the-ultimate.org/node/2304.

[61] Duncan Temple Lang. Embedding s in other languages and environments. In *Proceedings of DSC*, volume 2, page 1, 2001.

[62] LangPop.com. Programming language popularity, October 2013.

[63] University of Melbourne Lee Naish. Neu prolog, February 1991.

[64] John W Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3(1-49):68–69, 1999.

[65] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[66] Yonathan Malachi, Zohar Manna, and Richard Waldinger. Tablog: The deductive-tableau programming language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 323–330. ACM, 1984.

[67] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer, 2004.

[68] Juan José Moreno-Navarro and Mario Rodriguez-Artalejo. Babel: A functional and logic programming language based on constructor discipline and narrowing. In *Algebraic and Logic Programming*, pages 223–232. Springer, 1988.

[69] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo. Logic programming with functions and predicates: The language babel. *The Journal of Logic Programming*, 12(3):191–223, 1992.

[70] MPprogramming.com. Castor : Logic paradigm for c++, August 2010. http://www.mpprogramming.com/cpp/.

[71] Gopalan Nadathur. λ prolog, September 2013.

[72] Mark J Nelson. Why did prolog lose steam?, August 2010. http://www.kmjn.org/notes/prolog_lost_steam.html.

[73] Mozilla Developer Network. Multi paradigm language, February 2014.

[74] Johan Nordlander. O'haskell, January 2001.

[75] Kurt Nrmark Department of Computer Science Aalborg University Denmark. Linguistic abstraction, July 2013. http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_intro-sec_section-title_1.

[76] University of Maryland Medical Center. Lisp, unification and embedded languages, October 2012. http://www.cs.unm.edu/~luger/ai-final2/LISP/.

[77] Ocaml Org. Ocaml programming language, March 2014.

[78] Pedro Pinto. Dot-scheme: A plt scheme ffi for the .net framework. In *Workshop on Scheme and Functional Programming*. Citeseer, 2003.

[79] Quintus Prolog. Embeddability, December 2003. http://quintus.sics.se/isl/quintuswww/site/embed.html.

[80] Yield Prolog. Yield prolog, October 2011. http://yieldprolog.sourceforge.net/.

[81] Shengchao Qin, editor. *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings*, volume 6445 of *Lecture Notes in Computer Science*. Springer, 2010.

[82] John Ramsdell. The cmu package, February 2013. http://hackage.haskell.org/package/cmu.

[83] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 6–14. ACM, 2003.

[84] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58. ACM, 2006.

[85] Maik Riechert. The monadiccp package, July 2013. http://hackage.haskell.org/package/monadiccp.

[86] J Alan Robinson and Ernest E Sibert. Loglisp: Motivation, design, and implementation, 1982.

[87] John Alan Robinson and EE Silbert. *LOGLISP: an alternative to PROLOG*. School of Computer and Information SCience, Syracuse University, 1980.

[88] Raúl Rojas. A tutorial introduction to the lambda calculus. *DOI= http://www.utdallas. edu/˜ gupta/courses/apl/lambda. pdf*, 2004.

[89] Daniel Seidel. The prolog-graph-lib package, June 2012. http://hackage.haskell.org/package/prolog-graph-lib.

[90] Daniel Seidel. The prolog package, June 2012. http://hackage.haskell.org/package/prolog.

[91] Eric Seidel. The liquid-fixpoint package, September 2013. http://hackage.haskell.org/package/liquid-fixpoint.

[92] Silvija Seres. *The algebra of logic programming*. PhD thesis, 2001.

[93] Silvija Seres and Shin-Cheng Mu. Optimisation problems in logic programming: an algebraic approach. 2000.

[94] Silvija Seres, J Michael Spivey, and CAR Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.

[95] Silvija Seres and Michael Spivey. Higher-order transformation of logic programs. In *Logic Based Program Synthesis and Transformation*, pages 57–68. Springer, 2001.

[96] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(05):547–587, 2004.

[97] Dorai Sitaram. Racklog: Prolog-style logic programming, January 2014.

[98] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard OKeefe. Logic programming for the real world. In *Proceedings of the ILPS*, volume 95, pages 83–94, 1995.

[99] Andy Sonnenburg. logicst, April 2013.

[100] JM Spivey. An introduction to logic programming through prolog, 1995.

[101] JM Spivey and Silvija Seres. The algebra of searching. *Festschritf in hounour of CAR Hoare*, 1999.

[102] JM Spivey and Silvija Seres. Embedding prolog in haskell. In *Proceedings of Haskell*, volume 99, pages 1999–28. Citeseer, 1999.

[103] Michael Spivey. Functional pearls combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000.

[104] Stackoverflow. Haskell vs. prolog comparison, December 2009. http://stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison.

[105] Jurrien Stutterheim. The nanoprolog package, December 2011. http://hackage.haskell.org/package/NanoProlog.

[106] Evgeny Tarasov. The hswip package, August 2010. http://hackage.haskell.org/package/hswip.

[107] William E. Byrd The Reasoned Schemer' (MIT Press, 2005) by Daniel P. Friedman and Oleg Kiselyov. minikanren.

[108] Wren Thornton. The unification-fd package, July 2012. http://hackage.haskellhttp://yieldprolog.sourceforge.net/.org/package/unification-fd.

[109] Jan Tikovsky. The monadiccp-gecode package, January 2014. http://hackage.haskell.org/package/monadiccp-gecode.

[110] Carnegie Mellon University. Algebraic logic functional programming language, February 1995.

[111] Simon Fraiser University. Life programming language, March 1998.

[112] Los Angeles University of California. Virgil programming language, March 2012.

[113] Germany University of Kiel. Curry programming language, September 2013.

[114] Maarten van Emden. Who killed prolog?, August 2010. http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/.

[115] Andre Vellino. Prolog's death, August 2010. http://synthese.wordpress.com/2010/08/21/prologs-death/.

[116] Job Vranish. minikanrent, March 2013.

[117] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

[118] Haskell Website. Logic programming example, February 2010. http://www.haskell.org/haskellwiki/Logic_programming_example.

[119] Haskell Website. Logic programming example in haskell, February 2010.

[120] Haskell Website. Quasiquotation in haskell, January 2014. http://www.haskell.org/haskellwiki/Quasiquotation.

[121] Haskell Wiki. Foldable and traversable, January 2013.

[122] Haskell Wiki. The haskell programming language, October 2013.

[123] Haskell Wiki. Embedded domain specific languages, September 2014.

[124] Haskell Wiki. Monads in haskell, January 2014.

[125] Haskell Wiki. Haskell in industry, June 2015.

[126] Wikipedia. Prolog wikipedia, March 2004.

[127] Wikipedia. Functional logic programming languages, February 2005.

[128] Wikipedia. Common lisp object system, December 2013.

[129] Wikipedia. Curry programming language, December 2013.

[130] Wikipedia. Functional logic programming, May 2013.

[131] Wikipedia. Quasiquotation, Novemeber 2013. http://en.wikipedia.org/wiki/Quasi-quotation.

[132] Wikipedia. Common language infrastructure, February 2014.

[133] Wikipedia. Common language runtime, March 2014.

[134] Wikipedia. Constraint handling rules, March 2014.

[135] Wikipedia. Constraint programming, March 2014.

[136] Wikipedia. Damas-hindley-milner type system, February 2014.

[137] Wikipedia. Foreign function interface, January 2014.

[138] Wikipedia. Lambda calculus, March 2014.

[139] Wikipedia. List of multi paradigm languages, March 2014.

[140] Wikipedia. Meta programming, March 2014.

[141] Wikipedia. Ocaml, March 2014.

[142] Wikipedia. Programming paradigm, March 2014.

[143] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors. *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers*, volume 7681 of *Lecture Notes in Computer Science*. Springer, 2013.

[144] Takashi's Workplace. A prolog in haskell, April 2009. http://propella.blogspot.in/2009/04/prolog-in-haskell.html.

[145] xkcd. Haskell vs prolog, or giving haskell a choice, February 2009. http://echochamber.me/viewtopic.php?f=11&t=35369.

[146] Switzerland cole Polytechnique Fdrale de Lausanne (EPFL) Lausanne. Scala programming language, 2002-2014.