

EMBEDDING PROGRAMMING LANGUAGES: PROLOG IN HASKELL

A Master's Thesis by
Mehul Chandrakant Solanki
230108015 solanki@unbc.ca
29 10 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	10
3.1 Current Work	10
3.2 Contributions	11
3.3 Thesis Contributions	11
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	Meta Syntactic Variables	37
14	Related Terms or Keywords	39
15	Haskell or Why Haskell ?	40
16	Prolog or Why Prolog ?	43
17	Miscellaneous or Possibly Related Content	44
18	Prototype 1	45
18.1	About this chapter	45
18.2	How Prolog works ?	45
18.3	What we do in this Prototype	47
18.4	Creating a data type	48
18.5	Working with the language	49
18.6	Black box	49
19	Prototype 2.1	50
19.1	About this chapter	50
19.2	How prolog-0.2.0.1 works	50
19.3	What we do in this prototype?	50
19.4	Current implementation (prolog-0.2.0.1)	50
19.5	Modifications	51

19.6 Results	51
20 Prototype 2.2	52
21 Prototype 3	53
21.1 Unification	53
21.2 Resolution	53
21.3 Search strategies	53
21.4 Stack Engine	54
21.5 Current Unification	54
21.6 Syntax Modification	54
21.7 Monadic Unification	54
22 Prototype 4	55
23 Work Completed	56
23.1 What we are doing	56
23.2 Unifiable Data Structures	56
23.3 Why Fix is necessary?	57
23.4 Dr. Casperson's Explanation	57
23.5 The other fix	58
23.6 The Fix we use	58
24 Results	59
24.1 Types	59
24.2 Lazy Evaluation	59
24.3 Opening up the Language	59
24.4 Quasi Quotation	60
24.5 Template Haskell	60
24.6 Higher Order Functions	60
24.7 I/O	60
24.8 Mutability	60
24.9 Unification	60
24.10 Monads	60
25 Conclusion / Expected Outcomes	61
26 Editing to do	62
26.1 Editing suggestions from David	64
Bibliography	66

List of Figures

1	Trace for append [118]	47
2	Functor Hierarchy [138]	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

2 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.

25 1.2 Thesis Statement

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

30 1.3 Problem Statement

31 Over the years the development of programming languages has become more and
32 more rapid. Today the number of is in the thousands and counting. The successors
33 attempt to introduce new concepts and features to simplify the process of coding a
34 solution and assist the programmer by lessening the burden of carrying out standard
35 tasks and procedures. A new one tries to capture the best of the old; learn from the
36 mistakes, add new concepts and move on; which seems to be good enough from an
37 evolutionary perspective. But all is not that straight forward when shifting from one
38 language to another. There are costs and incompatibilities to look at. A language
39 might be simple to use and provide better performance than its predecessor but not
40 always be worth the switch.

41 PROLOG is a language that has a hard time being adopted. Born in an era where
42 procedural languages were receiving a lot of attention, it suered from competing
43 against another new kid on the block: C. Some of the problems were of its own
44 making. Basic features like modules were not provided by all compilers. Practical
45 features for real world problems were added in an ad hoc way resulting in the loss of
46 its purely declarative charm. Some say that PROLOG is fading away, [85, 131, 130].
47 It is apparently not used for building large programs [144, 111, 63]. However there
48 are a lot of good things about Prolog: it is ideal for search problems; it has a simple
49 syntax, and a strong underlying theory. It is a language that should not die away.
50 So the question is how to have all the good qualities of PROLOG without actually

51 using PROLOG?

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

55 Generally speaking, programming languages with a wide scope over problem do-
56 mains do not provide bespoke support for accomplishing even mundane tasks. Ap-
57 proaching towards the solution can be complicated and tiresome, but the program-
58 ming language in question acts as the master key.

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

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

70 The solution, a programming language with a split personality, in our case, some-
71 times functional, sometimes logical and sometimes both. Depending upon the prob-
72 lem, the language shapes itself accordingly and exhibits the desired characteristics.
73 The ideal situation is a language with a rich feature set and the ability to mould itself
74 according to the problem. A language with ability to take the appropriate skill set
75 and present it to the programmer, which will reduce the hassle of jumping between
76 languages or forcibly trying to solve a problem according to a paradigm.

77 The subject in question here is HASKELL and the split personality being PROLOG.

78 How far can HASKELL be pushed to dawn the avatar of PROLOG ? is the million
79 dollar question.

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

82 This can be achieved in two ways,

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

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

96 1.4 Proposal Organization

97 The next chapter, [Chapter 2](#) provides details about the short comings of the previ-
98 ous works and the road to a better future. [Chapter 3](#), the background talks about the
99 programming paradigms and languages in general and the ones in question. Then we
100 look at the question from different angles namely, [Chapter 4](#), Embedding a Program-
101 ming Language into another Programming Language and [Chapter 5](#), Multi Paradigm
102 Languages (Functional Logic Languages). Some of the indirectly related content

¹⁰³ [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 [68] 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, [55] 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 [156], 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 [101], 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 pre-terminated 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 languages are based on Hindley-Milner or Damas-Milner or Damas- Hindley-Milner [154] type system. The ability of the type system to give the most general type of a program without any help (annotation). The algorithm [20] 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, [113] 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 [144] 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 [73] functional languages around and is the first language to incorporate Monads [133] for safe *IO*. Monads can be described as composable computation descriptions [142]. 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 [31] in other languages like SCHEME [110], LISP [66,

99, 100], JAVA [144, 58], JAVASCRIPT [59] and the list [93] 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 [33, 12, 94, 165, 52, 43]. 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) [155], 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) [151] which is an implementation of the Common Language Infrastructure (CLI) standard [150].

Another important concept is meta programming [158], 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 [49] 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 [76, 136, 149], 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 be represented using HASKELL syntax and performing 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, going through the code, the first rule says that an 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 recurs on the tails. The same in HASKELL,

Consider the Object Functional Programming Language, SCALA [168], 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 which are discussed below along with the shortcomings.

1. Very few embedded implementations exist which offer a perspective into the job at hand. One of the earliest implementations [61] is for an older specification of HASKELL called HASKELL 98 **hugs**. It is more of a proof of concept providing a mechanism to include variable search strategies in order to produce a result. Another implementation [166] based of it simplifies the notation to a list format. Nonetheless, both implementations lack simplicity and support for basic PROLOG features such as *cuts*, *fails*, *assert* among others.
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 [129] 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.

3.3 Thesis Contributions

1. Prototype 1 does flattening language opening up the language (binding monad) adding custom variables monadic unification (stuff happens in a bubble) rec

- 260 type \rightarrow non rec type \rightarrow fix non rec type isomorphically $==$ rec type
- 261 2. Prototype 2 does extends current prolog-0.2.0.1 this is to show that we can plug
262 out approach into existing implementation and things work
- 263 3. Prototype 3 does variable search strategy what ever method you do for searching
264 at the point of unification you can do it with our approach
- 265 4. Prototype 4 does how can io be squeezed into this model where whenever the
266 resolver encounters an io operation it generates a thunk (sort of unsolved state-
267 ment) which when executed would result in a side effect but till that point every
268 thing is pure

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 [69], which lists a number of PROLOG implementations in a variety of languages like LISP, SCHEME, SCALA, JAVA, JAVASCRIPT, RACKET [110] 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 [61]. 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, [166].

Adding fuel to fire, is the question on PROLOG's existence and survival [130, 85, 131, 111] 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 [167] of the same family, the family of Declarative Languages. Not to forget HASKELL also has some tricks [134] 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 [25], 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 [135]. A general out look towards implementing PROLOG has also been discussed by [67] 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 [115, 107, 108, 114, 105] 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 [116] 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, [19] 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 [36] 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. [51] 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 [120] being a preliminary or "mini Prolog" as such with not much in it to be able to be useful, [121] is all powerful but is an Foreign Function Interface so it is "Prolog in Haskell" but we need Prolog for it, [103] 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. [23] is a continuation-based, backtracking, logic programming monad which sort of depicts Prolog's backtracking behaviour. Prolog is heavily based on formal logic, [41] provides a powerful system for

370 Propositional Logic. Others include small hybrid languages [37] and Parallelis-
371 ing Logic Programming and Tree Exploration [22].

372 • Unification Libraries

373 The more specific the feature the lesser the support in Haskell. Moving on to
374 the other distinct feature of Prolog is Unification, two libraries exist [123], [95]
375 that unify two Prolog Terms and return the resulting substitution.

376 • Backtracking

377 Another important aspect of PROLOG is backtracking. To simulate it in HASKELL,
378 the libraries [38, 112] use monads. Moreover, there is a package for the EGISON
379 programming language [53] which supports non-linear pattern-matching with
380 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 [168], 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 [68], more over as Timothy Budd puts it [160] "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 [160, 86, 13] converge to roughly the same thing that of providing a framework to work with different styles with a list of languages [157, 30] 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 [148, 145] 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 [47] 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 [45] 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 [52] 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

430 as always since we are talking about HASKELL we have to include monads and
431 unifying theories using monads [43].

432 • Functional Logic Programming Languages

433 A recent survey [46] throws light on these hybrid languages.

434 One of the most prominent multi paradigm languages in HASKELL is CURRY
435 [4]. Th syntax is borrowed from the parent language and so are a lot of the
436 features. Taking a recap, a functional programming language works on the
437 notion of mathematical functions while a logic programming language is based
438 on predicate logic. The strong points of CURRY are that the features or basis
439 of the language are general and are visible in a number of languages like [27].
440 The language can play with problems from both worlds. In a problem where
441 there are no unknowns and/or variables the language behaves like a functional
442 language which is pattern matching the rules and execute the respective bodies.
443 In the case of missing information, it behaves like PROLOG; a sub-expression e is
444 evaluated on the conditions that it should satisfy which constraint the possible
445 values of e . This brings us to the first important feature of functional logic
446 languages *narrowing*. The expressions contain *free variables*; simply speaking
447 incomplete information that needs to be *unified* to a value depending on the
448 constraints of the problem. The language introduces only a few new constructs
449 to support non determinism and choice. Firstly, *narrowing* ($==$), which deals
450 with the expressions and unknown values and binds them with appropriate
451 values. The next one is the *choice* operator ($?$) for non-deterministic operations.
452 Lastly, for unifying variables and values under some conditions, ($\&$) operator
453 has been provided to add constraints to the equation. Putting it all together,
454 it gives us the feel of a logic language for something that looks very much like
455 HASKELL. Unification is like two way pattern matching and with a similar

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

459 **5.3 Some Multi Paradigm Languages**

460 The list of multi paradigm languages is huge, but in this thesis we will mostly stick
461 to Functional Logical programming languages. Beginning with functional hybrids, a
462 small project language called VIRGIL [128], combining objects to work with functions
463 and procedures. On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [146].
464 This can be justified as object oriented programming has been one of the most dom-
465 inant styles of programming and hence even HASKELL has one called O'HASKELL
466 [87] though it last saw a release back in 2001. Another prominent implementation is
467 OCAML [159, 90] which adds object oriented capabilities with a powerful type sys-
468 tem and module support. This is the case with most of the languages in this section
469 hardly a few have survived as the new ones incorporated the positives of the old. As
470 mentioned before one of the most popular [73] and widely usage both in academia and
471 industry is the SCALA [168] programming language stands out.

472 **5.4 Functional Logic Programming Languages**

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

have been three popular ones, beginning with NEU PROLOG, [74], OZ (MOZART PROGRAMMING SYSTEM) [21] and MERCURY [28]. 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 [75] comes very close to HASKELL with monads, higher order functions and lazy evaluation. Taking a look at PROLOG variants, CIAO [18]; a preprocessor to PROLOG for functional syntax support, λ PROLOG [84] aims at modular higher order programming with abstract data types in a logical setting, BABEL [50, 81, 80] combines pure PROLOG with a first order functional notation, LIFE [127] is for Logic, Inheritance, Functions and Equations in PROLOG syntax with currying and other features like functional languages and others [10, 77].

The functional language SCHEME is a very popular choice for this sort of a thing. With a book [25] and an implementation to accompany [26, 122] which seems to have translated into HASKELL, [57, 39, 132].

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 [125, 27].

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 [153] 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 [152], 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 [48, 147]. 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 [75], LIFE [127], NUE-PROLOG [74] and OZ [21]. 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 [125], BABEL [50], LPG [10] and CURRY [129].

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 [54], Foreign 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. Semantics
3. Standard Library
4. Runtime System
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 [140]. 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,

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, [96]
- (b) Building domain-specific embedded languages, [54]
- (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, [91]
- (g) Application-specific foreign-interface generation, [97]
- (h) Embedding S in other languages and environments, [72]

2. Books

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

3. Articles / Blogs / Discussions

- 569 (a) Embedding one language into another, [70]
- 570 (b) Application-specific foreign-interface generation, [71]
- 571 (c) Linguistic Abstraction, [88]
- 572 (d) LISP, Unification and Embedded Languages, [89]

573 4. Websites

- 574 (a) Embedding SWI-Prolog in other applications, [31]

575 **7.2 Implementations**

- 576 1. Lots of them I guess

577 **7.3 Important People**

- 578 1. ????

579 **7.4 Miscellaneous / Possibly Related Content**

- 580 1. ????

8 Prolog in ----

Prolog in -----

8.1 Theory

- Papers

1. QLog, [66]

2. LogLisp Motivation, design, and implementation, [99]

- Books

1. Warrens Abstract Machine A TUTORIAL RECONSTRUCTION, [1]

2. LOGLISP: an alternative to PROLOG, [100]

- Articles / Blogs / Discussions

1. Hello

- Websites

1. Hello

8.2 Implementations

1. Castor : Logic paradigm for C++, [83]

2. GNU Prolog for Java, [44]

3. JLog - Prolog in Java, [58]

4. JScriptLog - Prolog in Java, [59]

5. Quintus Prolog, [92]

600 6. Yield Prolog, [93]

601 7. Racklog, [110]

602 **8.3 Important People**

603 1. ???

604 **8.4 Miscellaneous / Possibly Related Content**

605 1. ???

9 Prolog in Haskell

Prolog in Haskell

9.1 Theory

• Papers

1. Embedding Prolog in Haskell / Functional Reading of Logic Programs,
[115]
2. Algebra of Logic Programming, [107]
3. The Algebra of Logic Programming, [105]
4. Optimisation Problems in Logic Programming : An Algebraic Approach,
[106]
5. Higher Order Transformation of Logic Programs, [108]
6. The Algebra of Searching, [114]
7. FUNCTIONAL PEARL Combinators for breadth-first search, [116]
8. Type Logic Variables, K Classen, [19]
9. A Type-Safe Embedding of Constraint Handling Rules into Haskell Wei-
Ngan Chin, Mar-tin Sulzmann and Meng Wang, [17]
10. Prological Features in a Functional Setting Axioms and Implementation,
R Hinze, [51]
11. Escape from Zurg: An Exercise in Logic Programming, [36]

• Books

1. The Reasoned Schemer, Daniel P. Friedman, William E. Byrd, Oleg Kise-
lyov, [25]

628 2. Programming Languages: Application and Interpretation, Shriram Kr-
629 ishnamurthi, Chapters 33-34 of PLAI discuss Prolog and implementing
630 Prolog, [67]

631 • Articles / Blogs / Discussions

632 1. Lambda the Ultimate, Programming Languages, [69]

633 2. Takashi's Workplace (Implementation), [166]

634 3. Haskell vs. Prolog Comparison, [117]

635 • Websites

636 1. Logic Programming in Haskell, [134]

637 **9.2 Implementations**

638 1. A Prolog in Haskell, Takashi's Workplace, [166]

639 2. Mini Prolog for Hugs 98, [61]

640 3. Nano Prolog, [120]

641 4. Prolog, [103]

642 5. cspm-To-Prolog, [40]

643 6. prolog-graph, [7]

644 7. prolog-graph-lib, [102]

645 8. hswip, [121]

9.3 Important People

1. Mike Spivey

2. Silvija Seres

9.4 Miscellaneous / Possibly Related Content

1. Unification Libraries

(a) unification-fd, [123]

(b) cmu, [95]

2. Logic Libraries

(a) logicct, [23], [24]

(b) logic-classes, [?]

(c) proplogic, [41]

(d) cflp, [37]

(e) logic-grows-on-trees, [22]

3. Concatenative Programming

(a) peg, [29]

4. Constraint Programming and Constraint Handling Rules

(a) monadiccp, [98]

(b) monadicccp-gecode, [124]

(c) csp, [6]

(d) liquid fix point, [104]

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, [43]
2. Symposium on Unifying Theories of Programming, 2006, [33].
3. Symposium on Unifying Theories of Programming, 2008, [12].
4. Symposium on Unifying Theories of Programming, 2010, [94].
5. Symposium on Unifying Theories of Programming, 2012, [165].

- Books

1. Unifying Theories of Programming, [52]

- Articles / Blogs / Discussions

1. ???

- Websites

1. ???

10.2 Implementations

1. Scala

2. Virgil

685 3. CLOS, Common Lisp Object System

686 4. Visual Prolog

687 5. ????

688 **10.3 Miscellaneous / Possibly Related Content**

689 1. ???

11 Functional Logic Programming Languages

Functional Logic Programming Languages

11.1 Theory

- Paper

1. FLPL Introduction 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

710 **11.2 Implementations**

711 1. Hello

712 **11.3 Miscellaneous / Possibly Related Content**

713 1. Hello

714 **12 Quasiquotation**

715 **12.1 Theory**

716 1. Papers

717 (a)

718 2. Books

719 (a)

720 3. Articles / Blogs / Discussions

721 (a)

722 4. Websites

723 (a) Quasiquotation Wikipedia, [149]

724 (b) Quasiquotation in Haskell, [136]

725 **12.2 Implementations**

726 1.

727 **12.3 Miscellaneous / Possibly Related Content**

728 1.

13 Meta Syntactic Variables

Some sources for the topic

[164] A metasyntactic variable is a placeholder name used in computer science, a word without meaning intended to be substituted by some objects pertaining to the context where it is used. The word `foo` as used in IETF Requests for Comments is a good example. By mathematical analogy, a metasyntactic variable is a word that is a variable for other words, just as in algebra letters are used as variables for numbers. Any symbol or word which does not violate the syntactic rules of the language can be used as a metasyntactic variable.

[15] A name used in examples and understood to stand for whatever thing is under discussion, or any random member of a class of things under discussion. The word `foo` is the canonical example. To avoid confusion, hackers never (well, hardly ever) use `foo` or other words like it as permanent names for anything. In filenames, a common convention is that any filename beginning with a metasyntactic-variable name is a scratch file that may be deleted at any time.

Metasyntactic variables are so called because they are variables in the metalanguage used to talk about programs etc; they are variables whose values are often variables (as in usages like the value of `f(foo,bar)` is the sum of `foo` and `bar`). However, it has been plausibly suggested that the real reason for the term metasyntactic variable is that it sounds good. To some extent, the list of one's preferred metasyntactic variables is a cultural signature. They occur both in series (used for related groups of variables or objects) and as singletons. Here are a few common signatures:

[56] In programming, a metasyntactic (which derives from meta and syntax) variable is a variable (a changeable value) that is used to temporarily represent a function . Examples of metasyntactic variables include (but are by no means limited to) `ack`, `bar` , `baz`, `blarg`, `wibble`, `foo` , `fum`, and `qux`. Metasyntactic variables are sometimes used in developing a conceptual version of a program or examples of programming

756 code written for illustrative purposes.

757 Any filename beginning with a metasyntactic variable denotes a scratch file. This
758 means the file can be deleted at any time without affecting the program.

759 [14]

760 A word, used in conversation or text that is meant as a variable. There is a fairly
761 standard set in the ComputerScience culture. People tend to create their own if they
762 are not exposed to others, which can be confusing. Of course, if you haven't seen
763 them before they can be quite confusing. They are, however, useful enough that this
764 is not enough reason to give them up. Standard set: foo, bar, baz, foobar/quux,
765 quuux, quuuux,

766 example: "Suppose I have a list, foo, with a node, bar, ..."

14 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

15 Haskell or Why Haskell ?

In this chapter we discuss the properties of HASKELL

This chapter discusses the properties of the host language HASKELL and mainly the feature set it provides for embedding domain specific languages(EDSLs).

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 [139]. It is one of the popular functional programming languages [73]. HASKELL is widely used in the industry [143].

Shifting a bit to Embedded Domain Specific Languages (EDSLs) such as Emacs LISP. Opting for embedding provides a "shortcut" to create a language which may be designed to provide specific functionality. Designing a language from scratch would require writing 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. All of the aforementioned are provided by the host language; in this case HASKELL. Examples for the same can be found here [62, 79] which talk about introducing combinator libraries for custom functionality.

The flip side of the coin is that the host language enforces certain aspects and properties of the eDSL and hence might not be exact to specification, all required constructs cannot be implemented due to constraints, programs could be difficult to debug since it happens at the host level and so on.

2. Looking at HASKELL as a tool for embedding domain specific languages[60]

(a) Monads

Control flow defines the order/ manner of execution of statements in a program[162]. The specification is set by the programming language. Generally, in the case of imperative languages the control flow is sequential while for a functional language is recursion [126]. For example, JAVA has a top down sequential execution approach. The declarative style consists of defining components of programs i.e. computations not a control flow[163].

This is where HASKELL shines by providing something called a *monad*. Functional Programming Languages define computations which then need to be ordered in some way to form a combination[137]. A monad gives a bubble within the language to allow modification of control flow without affecting the rest of the universe. This is especially useful while handling side effects.

A related topic would be of persistence languages, architectures and data structures. Persistent programming is concerned with creating and manipulating data in a manner that is independent of its lifetime [82]. A persistent data structure supports access to multiple versions which may arise after modifications [32, 64]. A structure is partially persistent if all versions can be accessed but only the current can be modified and fully persistent if all of them can be modified.

Coming back to control flow; for example, implementing backtracking in an imperative language would mean undoing side effects which even PROLOG is not able to do since the asserts and retracts cannot be undone. In HASKELL, a monad defines a model for control flow and how side effects would propagate through a computation from step to step or modification

830 to modification. And HASKELL allows creation of custom monads relieving
831 the burden of dealing with a fixed model of the host language.

832 (b) Lazy Evaluation

833 Another property of HASKELL is laziness or lazy evaluation which means
834 that nothing is evaluated until it is necessary. This results in the ability
835 to define infinite data structures because at execution only a fragment is
836 used [141].

16 Prolog or Why Prolog ?

This chapter discusses the properties of the target language PROLOG and the feature set that will be translated to the host language to extend its capabilities.

1. PROLOG as a logic programming language.

PROLOG is a general purpose logic programming language mainly used in artificial intelligence and computational linguistics. It is a Declarative language i.e. a program is a set of facts and rules running a query on which will return a result. The relation between them is defined by clauses using *Horn Clauses*[144].

PROLOG is very popular and has a number of implementations [161] for different purposes.

2. Why embed PROLOG ?

848 **17 Miscellaneous or Possibly Related Content**

849 Miscellaneous / Possibly Related Content

850 1. ???

18 Prototype 1

18.1 About this chapter

This chapter throws light on what PROLOG does to resolve a given query via *unification* and this can be replicated in the host language along with the challenges.

This chapter discusses the aspects of opening a language while preserving the original structure of a closed recursive structure in HASKELL. Also discussed are the issues related to customizing certain aspects such as meta-syntactic variables.

18.2 How Prolog works ?

Looking at how PROLOG works [119].

Most PROLOG distributions have three types of terms:

1. Constants.
2. Variables.
3. Complex terms.

Two terms can be unified if they are the same or the variables can be assigned to terms such that the resulting terms are equal.

The possibilities could be,

1. If term1 and term2 are constants, then term1 and term2 unify if and only if they are the same atom, or the same number.
2. If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 unify, and term2 is instantiated to term1 . (So if they are both variables, theyre both instantiated to each other, and we say that they share values.)

3. If term1 and term2 are complex terms, then they unify if and only if:

- 875 (a) They have the same functor and arity, and
- 876 (b) all their corresponding arguments unify, and
- 877 (c) the variable instantiations are compatible.
- 878 4. Two terms unify if and only if it follows from the previous three clauses that
- 879 they unify.

880 For example, consider the append function

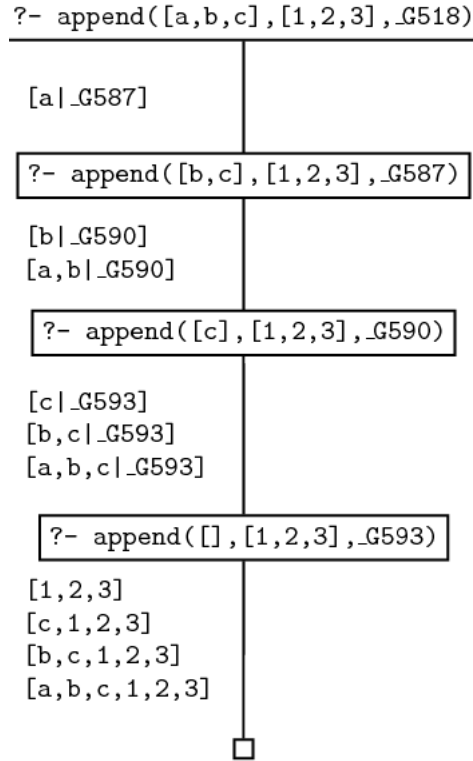


Figure 1: Trace for append [118]

881 18.3 What we do in this Prototype

882 This prototype throws light on the process of tackling the issues involved in creating
883 a data type to replicate the target language type system while conforming to the host
884 language restrictions and also utilizing the benefits.

885 We have a PROLOG like language in HASKELL defined via *data*.

886 The language defined is recursive in nature.

887 We convert it into a non recursive data type.

888 Basically we do Unification monadically.

18.4 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 [78]

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 An ideal case would would be something that is
dont know what to write

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 predicates, clauses etc.), don't cares, cuts, variables and so on.

912 Consider the language below,

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

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

916 One of the issues with the above is that it is not possible to distinguish the
917 structure of the data from the data type itself [109]. Consider the following, a reduced
918 version of the above data type,

919 Also one cannot create Quantifiers plus logic

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

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

925 and a generic fuinction like,

926 **18.5 Working with the language**

927 Creating instances,

928 After flattening do fixing,

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

930 **18.6 Black box**

19 Prototype 2.1

19.1 About this chapter

This chapter attempts to infuse the generic methodology from 18 in a current PROLOG implementation [103] and make the unification "monadic".

19.2 How prolog-0.2.0.1 works

The original syntax used by the library,

The above language suffers from most of the problems discussed in the previous chapter.

The above is used to construct PROLOG "terms" which are of a "single type".

A database is used to store the terms which can then be used to resolve a query.

An interpreter to solve a query and lastly the unifier,

There are a few other components such as the REPL, Parser.

19.3 What we do in this prototype?

In the first prototype we just did unification of two terms not query resolution.

We do complete PROLOG query resolution like stuff.

18 provides a generic procedure / methodology to convert a language into monadic unifiable form

19.4 Current implementation (prolog-0.2.0.1)

The current unification uses basic pattern matching to unify the terms

950 **19.5 Modifications**

951 The first modification is to the language is to make it compatible with the library
952 which provides this nice generic mechanism a perform unification in a monadic manner.
953 Fixing, flattening, creating necessary instances

954 some translation and helper functions

955 and finally the unification

956 **19.6 Results**

957 It works,

958 **20 Prototype 2.2**

959 nothing to do here

21 Prototype 3

When two terms are to be unified we can use 18 ,

term1 and term2 are matched and an assignment is the result

now this may be a part of a query resolution procedure

to reach the point where two terms need to unified will happen through some sort
of search strategy

and our approach is independent of that, and this prototype is a proof of concept
to implementing query resolution using unification with variable search strategy

21.1 Unification

The first, "unification," regards how terms are matched and variables assigned to
make terms match. [35]

21.2 Resolution

this where the complete procedure takes place after the query is passed along with
the knowledge

the resolver searches to create and a list of sub goals and then tries to achieve
each one.

[34]

21.3 Search strategies

The base implementation used for this prototype is [61] and below are the search
strategies

980 **21.4 Stack Engine**

981 **21.4.1 Pure Engine**

982 **21.4.2 Andorra Engine**

983 **21.5 Current Unification**

984 **21.6 Syntax Modification**

985 **21.7 Monadic Unification**

22 Prototype 4

Our aim to embedd IO into the DSL

So something like a "data" declaration for IO operations

So when the program is getting interpreted the interpreter encounters an IO operation which then gets "interpreted" to the above and it continues normally.

The interpreted program is still pure since the IO actions have not been executed if the running is done inside a monad then the IO still is pure.

23 Work Completed

23.1 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 [123].

23.2 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 2.

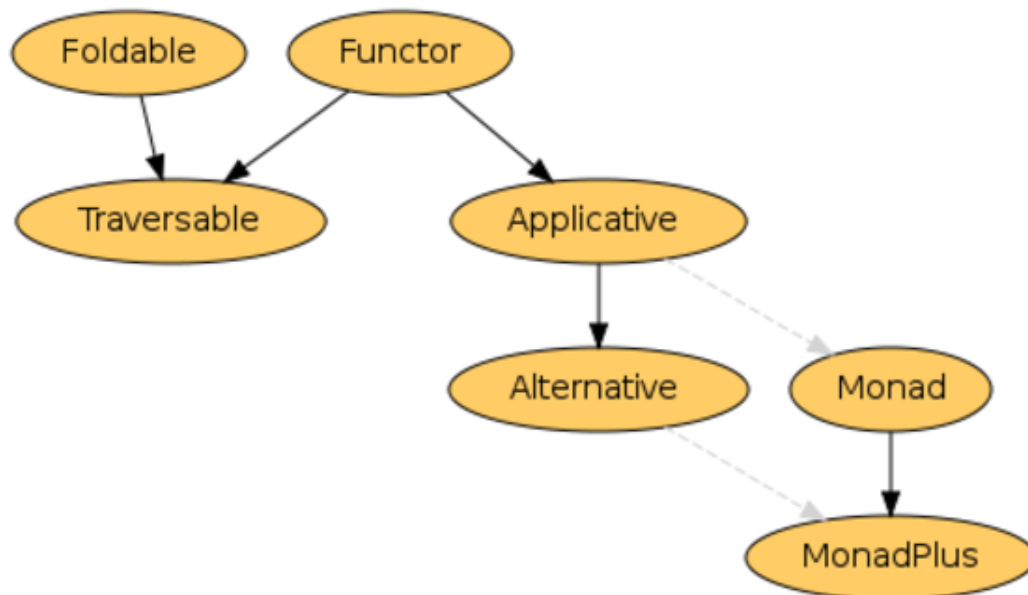


Figure 2: Functor Hierarchy [138]

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

1004 **23.3 Why `Fix` is necessary?**

1005 Since `HASKELL` is a lazy language it can work with infinite data structures. *Type*
1006 *Synonyms* in `HASKELL` cannot be self referential.

1007 In our case consider the following example,

1008 A `FlatTerm` can be of infinite depth which due to the reason stated above cannot
1009 be accounted for during application function. The resulting type signature would be
1010 of the form,

1011 Enter the `Fix` same as the function as a data type. The above would be simply
1012 reduced to,

1013 resulting in the `PROLOG` Data Type

1014 **23.4 Dr. Casperson's Explanation**

1015 A recursive data type in `HASKELL` is where one value of some type contains values of
1016 that type, which in turn contain more values of the same type and so on. Consider
1017 the following example.

1018 A sample `Tree` would be,

1019 The above structure can be infinitely deep since `HASKELL` is a *lazy* programming
1020 language. But working with an infinitely deep / nested structure is not possible
1021 and will result in a *occurs check* error. This is because writing a type signature for
1022 a function to deal with such a parameter is not possible. One option would be to
1023 *flatten* the data type by the introduction of a type variable. Consider the following,

1024 A sample `FlatTerm` would be similar to `Tree`.

1025 The `FlatTree` is recursive but does not reference itself. But it too can be
1026 infinitely deep and hence writing a function to work on the structure is not possible.

1027 **23.5 The other fix**

1028 The `fix` function in the `Control.Monad.Fix` module allows for the definition of
1029 recursive functions in HASKELL. Consider the following scenario,

1030 The above function results in an infinite application stream,

1031 A fixed point of a function `f` is a value `a` such that `f a == a`. This is where the
1032 name of `fix` comes from: it finds the least-defined fixed point of a function.

1033 **23.6 The Fix we use**

1034 Fix-point type allows to define generic recursion schemes [65].

1035 **24 Results**

1036 **24.1 Types**

1037 One of the major differences between PROLOG and HASKELL is how each language
1038 handles types. PROLOG is an untyped language meaning any operation can be per-
1039 formed on the data irrespective of its type. HASKELL on the other hand is strongly
1040 typed i.e. each operation requires a signature stating what types of data it can work
1041 with. Moreover, the HASKELL type system is static.

1042 PROLOG like any other language can work with some basic data types like num-
1043 bers, characters, strings among others. Using these one can make terms like *Atoms*,
1044 *Clauses*, *Constants*, *Strings*, *Characters*, *Predicates*, *Structures*, *Special Characters*
1045 and so on. These need to be incorporated into the implementation so as to give a
1046 palette for writing programs.

1047 Our preliminary implementation is as follows,

1048 which in PROLOG would look like,

1049 **24.2 Lazy Evaluation**

1050 **24.3 Opening up the Language**

1051 **Flattening**

1052 **Fixing**

1053 **MetaSyntactic Variables**

- 1054 **24.4 Quasi Quotation**
- 1055 **24.5 Template Haskell**
- 1056 **24.6 Higher Order Functions**
- 1057 **24.7 I/O**
- 1058 **24.8 Mutability**
- 1059 **24.9 Unification**
- 1060 **24.10 Monads**

1061 **25 Conclusion / Expected Outcomes**

1062 The aim of this study is to experiment with two different languages working to-
1063 gether and/or contributing in providing a solution. Mixing and matching conflicting
1064 characteristics may lead to a behaviour similar to that of a multi paradigm language.
1065 The points to be looked at are efficiency of the emulation, semantics of the resulting
1066 embedding.

1067 Moreover, this will be an attempt to answer the question how practical PROLOG
1068 fits into HASKELL.

26 Editing to do

This Chapter needs to be removed from the final work.

Either

1. Rename “proposal.*” to “thesis-solanki.*”.
2. Switch the thesis style to UNBC thesis style. (Not urgent, if this breaks other tools, we can do this last, but it would be nice to have a sense of what the thesis is going to look like.)
3. Check the rules for spacing in the bibliography to ensure that we have them right.

Mehul

4. **Rewrite (Section) Chapter 3.2.** You are now in a position to state what your contributions are. In some sense everything else flows around this.
5. Fix the reference at the bottom of page 2:
`citewikipro- log,somogyi1995logic,website:prolog1000db.` **SOLVED**
6. Write enough of Chapters 18–22 that we can decide what material is needed in Chapters 12, 13, and 14.
7. [T_EXnical] Remove the `\paragraph{ }`s from the running text. L^AT_EX ends a paragraph every time that it encounters two end-of-lines with only whitespace between them. `\par` does the same thing.

The `\paragraph` command is in the same family as `chapter`, `\section`, and so on. For its correct use, see later in this file.

If you don't like the shape of the paragraphs that you get without `paragraph`, use something like

```
\setlength{\parindent}{3em}
\setlength{\parskip}{2\baselineskip}
```

to adjust either the initial paragraph indent, or the inter-paragraph space.

8. Rewrite (Section) Chapter 3 in formal English.
9. Bump the sectioning levels up by one. That is, what is currently a section should become a chapter, what is currently a subsection should become a section, and so on. It may not make sense to do this until you have switch to `thesis.sty`.
10. “re-curses” means to swear again (*p* 9). **Changed to recurs**
11. I am not sure that I agree with the use of “reflective” on *p* 8 (*l* 25). Reflection often means run-time introspection (for instance the Java `.getClass()` method). In computer science, reflection is the ability of a computer program to examine (see type introspection) and modify its own structure and behavior (specifically the values, meta-data, properties and functions) at runtime.
12. Supply your credentials in the front material (what degrees do you have?).
(Search for `%% Supply your credentials in proposal1.tex.`)

David

13. Review Chapter 1
14. Review Chapter 2
15. Review Chapter 3
16. Review Chapter 4
17. Review Chapter 5
18. Review Chapter 6

19. Review Chapter 7

20. Review Chapter 8

21. Review Chapter 18

26.1 Editing suggestions from David

Thoughts on 1.1 We need to firmly fix in mind who the target audience is. Some possibilities

1. Undergraduate Physics students
2. Undergraduate Computer Science students
3. Future graduate students of Casperson who have just begun their thesis work.
4. Simon Peyton-Jones.

If we assume (3), then the material in the first paragraph and part of the second are unnecessary.

Thoughts on 1.3 I am unsure that I can summarize this subsection in two sentences. I don't know what the problem statement is at the end of it.

Thoughts on 1.4 Rename to "Thesis Organization".

Thoughts on Chapter 2 Here are some potential keywords from Chapter 2:

• Hindley-Milner type systems • Horn clauses • λ -calculi • HASKELL • SCALA
• declarative programming languages • foreign function interfaces • functional programming • implementing Prolog in other languages • language embedding • language families • language paradigms • logic programming • meta-programming • monads

• paradigm integration • quasi-quotation • the typed λ -calculus • the untyped λ -calculus .

What is the overall message?

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 *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] c2 wiki. Metasyntactic variables, September 2011.
- [15] Catb. Metasynatactic variables.
- [16] Prolog Development Center. Visual prolog, June 2013.
- [17] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A type-safe embedding of constraint handling rules into haskell. *Technical report School of Computing, National University of Singapore, Boston, MA, USA*, 2003.

- 1102 [18] Ciao. Ciao programming language, August 2011.
- 1103 [19] Koen Claessen and Peter Ljunglöf. Typed logical variables in haskell. *Electr.*
1104 *Notes Theor. Comput. Sci.*, 41(1):37, 2000.
- 1105 [20] Code Commit. Hindley milner type system, December 2008.
- 1106 [21] Mozart Consortium. Oz / mozart, March 2013.
- 1107 [22] Gregory Crosswhite. The logicgrowsontrees package, September 2013. <http://hackage.haskell.org/package/LogicGrowsOnTrees>.
1108
- 1109 [23] DanDoel. The logict package, August 2013. <http://hackage.haskell.org/package/logict>.
1110
- 1111 [24] DanDoel. The logict package example, August 2013. <http://okmij.org/ftp/Computation/monads.html>.
1112
- 1113 [25] Oleg Kiselyov Daniel P. Friedman, William E. Byrd. *The Reasoned Schemer*.
1114 The MIT Press, Cambridge Massachusetts, London England, 2005.
- 1115 [26] William E. Byrd Daniel P. Friedman and Oleg Kiselyov. Kanren, March 2009.
- 1116 [27] Universidad Complutense de Madrid. Toy, Decmeber 2006.
- 1117 [28] University Of Melbourne Computer Science department. Mercury programming
1118 language, February 2014.
- 1119 [29] Dustin DeWeese. The peg package, April 2012. <http://hackage.haskell.org/package/peg>.
1120
- 1121 [30] Open Directory Project dmoz. Multi paradigm, November 2013.
- 1122 [31] SWI Prolog Documentation. Embedding swi-prolog in other applications,
1123 June 2013. [http://www.swi-prolog.org/pldoc/man?section=](http://www.swi-prolog.org/pldoc/man?section=embedded)
1124 [embedded](http://www.swi-prolog.org/pldoc/man?section=embedded).
- 1125 [32] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tar-
1126 jan. Making data structures persistent. In *Proceedings of the eighteenth annual*
1127 *ACM symposium on Theory of computing*, pages 109–121. ACM, 1986.
- 1128 [33] Steve Dunne and Bill Stoddart, editors. *Unifying Theories of Programming,*
1129 *First International Symposium, UTP 2006, Walworth Castle, County Durham,*
1130 *UK, February 5-7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes*
1131 *in Computer Science*. Springer, 2006.
- 1132 [34] Joshua Eckroth. Prolog resolution, April 2014.
- 1133 [35] Joshua Eckroth. Prolog unification, April 2014.

- 1134 [36] Martin Erwig. Escape from zurg: an exercise in logic programming. *Journal of*
1135 *Functional Programming*, 14(03):253–261, 2004.
- 1136 [37] Sebastian Fischer. The cflp package, June 2009. [http://hackage.](http://hackage.haskell.org/package/cflp)
1137 [haskell.org/package/cflp](http://hackage.haskell.org/package/cflp).
- 1138 [38] Sebastian Fischer. stream-monad, September 2012.
- 1139 [39] Adam C. Foltzer. Molog, March 2013.
- 1140 [40] Marc Fontaine. The cspm-toprolog package, August 2013. [http://hackage.](http://hackage.haskell.org/package/CSPM-ToProlog)
1141 [haskell.org/package/CSPM-ToProlog](http://hackage.haskell.org/package/CSPM-ToProlog).
- 1142 [41] David Fox. The proplogic package, April 2012. [http://hackage.haskell.](http://hackage.haskell.org/package/PropLogic)
1143 [org/package/PropLogic](http://hackage.haskell.org/package/PropLogic).
- 1144 [42] David Fox. The logic-classes package, October 2013. [http://hackage.](http://hackage.haskell.org/package/logic-classes)
1145 [haskell.org/package/logic-classes](http://hackage.haskell.org/package/logic-classes).
- 1146 [43] Jeremy Gibbons. Unifying theories of programming with monads. In *Unifying*
1147 *Theories of Programming*, pages 23–67. Springer, 2013.
- 1148 [44] GNU. Gnu prolog for java, August 2010. [http://www.gnu.org/](http://www.gnu.org/software/gnuprologjava/)
1149 [software/gnuprologjava/](http://www.gnu.org/software/gnuprologjava/).
- 1150 [45] Michael Hanus. Michael hanus home page.
- 1151 [46] Michael Hanus. Multi-paradigm declarative languages. In *Logic Programming*,
1152 pages 45–75. Springer, 2007.
- 1153 [47] Michael Hanus. Functional logic programming, February 2009.
- 1154 [48] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A
1155 truly functional logic language. In *Proc. ILPS*, volume 95, pages 95–107, 1995.
- 1156 [49] Haskellwiki. Template haskell, October 2013.
- 1157 [50] Juan Jose Moreno Navarro Herbert Kuchen. Babel programming language,
1158 January 1988.
- 1159 [51] Ralf Hinze et al. Prological features in a functional setting axioms and imple-
1160 mentation. In *Fuji International Symposium on Functional and Logic Program-*
1161 *ming*, pages 98–122. Citeseer, 1998.
- 1162 [52] Charles Anthony Richard Hoare and Jifeng He. *Unifying theories of program-*
1163 *ming*, volume 14. Prentice Hall Englewood Cliffs, 1998.
- 1164 [53] Satoshi Egi Ryo Tanaka Takahisa Watanabe Kentaro Honda. Egison package,
1165 March 2014.

- 1166 [54] Paul Hudak. Building domain-specific embedded languages. *ACM Comput.*
1167 *Surv.*, 28(4es):196, 1996.
- 1168 [55] John Hughes. Why functional programming matters. *The computer journal*,
1169 32(2):98–107, 1989.
- 1170 [56] What is Tech Target. Metasynatactic variables, September 2005.
- 1171 [57] JaimieMurdock. Haskell kanren, March 2012.
- 1172 [58] JLogic. Jlog - prolog in java, September 2012. [http://jlogic.](http://jlogic.sourceforge.net/index.html)
1173 [sourceforge.net/index.html](http://jlogic.sourceforge.net/index.html).
- 1174 [59] JLogic. Jscriptlog - prolog in javascript, September 2012. [http://jlogic.](http://jlogic.sourceforge.net/index.html)
1175 [sourceforge.net/index.html](http://jlogic.sourceforge.net/index.html).
- 1176 [60] Paul Johnson. Why haskell is good for embedded domain specific languages,
1177 January 2008.
- 1178 [61] Mark P Jones. Mini-prolog for hugs 98, June 1996. [http://darcs.](http://darcs.haskell.org/hugs98/demos/prolog/)
1179 [haskell.org/hugs98/demos/prolog/](http://darcs.haskell.org/hugs98/demos/prolog/).
- 1180 [62] Simon L Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing con-
1181 tracts: An adventure in financial engineering. In *FME*, volume 2021, page 435,
1182 2001.
- 1183 [63] Mark Kantrowitz. The prolog 1000 database, August 2012.
- 1184 [64] David Karger. Persistent data structures, September 2005.
- 1185 [65] Anton Kholomiov. data-fix, February 2013.
- 1186 [66] H Jan Komorowski. Qlog: The programming environment for prolog in lisp.
1187 *Logic Programming*, pages 315–324, 1982.
- 1188 [67] Shriram Krishnamurthi. *Programming languages: Application and interpreta-*
1189 *tion*, chapter 33-34, pages 295–305, 307–311. Brown Univ., 2007.
- 1190 [68] Shriram Krishnamurthi. Teaching programming languages in a post-linnaean
1191 age. *SIGPLAN Not.*, 43(11):81–83, November 2008.
- 1192 [69] The Programming Languages Weblog Lambda The Ultimate. Embedding pro-
1193 log in haskell, July 2004. [http://lambda-the-ultimate.org/node/](http://lambda-the-ultimate.org/node/112)
1194 [112](http://lambda-the-ultimate.org/node/112).
- 1195 [70] The Programming Languages Weblog Lambda The Ultimate. Embedding
1196 one language into another, March 2005. [http://lambda-the-ultimate.](http://lambda-the-ultimate.org/node/578)
1197 [org/node/578](http://lambda-the-ultimate.org/node/578).

- [71] The Programming Languages Weblog Lambda The Ultimate. Application-specific foreign-interface generation, October 2006. <http://lambda-the-ultimate.org/node/2304>.
- [72] Duncan Temple Lang. Embedding s in other languages and environments. In *Proceedings of DSC*, volume 2, page 1, 2001.
- [73] LangPop.com. Programming language popularity, October 2013.
- [74] University of Melbourne Lee Naish. Neu prolog, February 1991.
- [75] John W Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3(1-49):68–69, 1999.
- [76] 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.
- [77] 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.
- [78] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer, 2004.
- [79] Bertrand Meyer. Eiffel as a framework for verification. In *Verified Software: Theories, Tools, Experiments*, pages 301–307. Springer, 2008.
- [80] Juan José Moreno-Navarro and Mario Rodríguez-Artalejo. Babel: A functional and logic programming language based on constructor discipline and narrowing. In *Algebraic and Logic Programming*, pages 223–232. Springer, 1988.
- [81] Juan Jose Moreno-Navarro and Mario Rodríguez-Artalejo. Logic programming with functions and predicates: The language babel. *The Journal of Logic Programming*, 12(3):191–223, 1992.
- [82] R Morrison and MP Atkinson. Persistent languages and architectures. In *Security and Persistence*, pages 9–28. Springer, 1990.
- [83] MPprogramming.com. Castor : Logic paradigm for c++, August 2010. <http://www.mpprogramming.com/cpp/>.
- [84] Gopalan Nadathur. λ prolog, September 2013.
- [85] Mark J Nelson. Why did prolog lose steam?, August 2010. http://www.kmjn.org/notes/prolog_lost_steam.html.
- [86] Mozilla Developer Network. Multi paradigm language, February 2014.

- [87] Johan Nordlander. O'haskell, January 2001.
- [88] Kurt Nrmak 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.
- [89] University of Maryland Medical Center. Lisp, unification and embedded languages, October 2012. <http://www.cs.unm.edu/~luger/ai-final2/LISP/>.
- [90] Ocaml Org. Ocaml programming language, March 2014.
- [91] Pedro Pinto. Dot-scheme: A plt scheme ffi for the .net framework. In *Workshop on Scheme and Functional Programming*. Citeseer, 2003.
- [92] Quintus Prolog. Embeddability, December 2003. <http://quintus.sics.se/isl/quintuswww/site/embed.html>.
- [93] Yield Prolog. Yield prolog, October 2011. <http://yieldprolog.sourceforge.net/>.
- [94] 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.
- [95] John Ramsdell. The cmu package, February 2013. <http://hackage.haskell.org/package/cmu>.
- [96] 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.
- [97] 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.
- [98] Maik Riechert. The monadiccp package, July 2013. <http://hackage.haskell.org/package/monadiccp>.
- [99] J Alan Robinson and Ernest E Sibert. Loglisp: Motivation, design, and implementation, 1982.
- [100] John Alan Robinson and EE Silbert. *LOGLISP: an alternative to PROLOG*. School of Computer and Information SCience, Syracuse University, 1980.
- [101] Raúl Rojas. A tutorial introduction to the lambda calculus. DOI= <http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf>, 2004.

- [102] Daniel Seidel. The prolog-graph-lib package, June 2012. <http://hackage.haskell.org/package/prolog-graph-lib>.
- [103] Daniel Seidel. The prolog package, June 2012. <http://hackage.haskell.org/package/prolog>.
- [104] Eric Seidel. The liquid-fixpoint package, September 2013. <http://hackage.haskell.org/package/liquid-fixpoint>.
- [105] Silvija Seres. *The algebra of logic programming*. PhD thesis, 2001.
- [106] Silvija Seres and Shin-Cheng Mu. Optimisation problems in logic programming: an algebraic approach. 2000.
- [107] Silvija Seres, J Michael Spivey, and CAR Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.
- [108] Silvija Seres and Michael Spivey. Higher-order transformation of logic programs. In *Logic Based Program Synthesis and Transformation*, pages 57–68. Springer, 2001.
- [109] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(05):547–587, 2004.
- [110] Dorai Sitaram. Racklog: Prolog-style logic programming, January 2014.
- [111] 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.
- [112] Andy Sonnenburg. logicst, April 2013.
- [113] JM Spivey. An introduction to logic programming through prolog, 1995.
- [114] JM Spivey and Silvija Seres. The algebra of searching. *Festschrift in honour of CAR Hoare*, 1999.
- [115] JM Spivey and Silvija Seres. Embedding prolog in haskell. In *Proceedings of Haskell*, volume 99, pages 1999–28. Citeseer, 1999.
- [116] Michael Spivey. Functional pearls combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000.
- [117] Stackoverflow. Haskell vs. prolog comparison, December 2009. <http://stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison>.
- [118] Patrick Blackburn Johan Bos Kristina Striegnitz. Learn prolog now, January 2012.

- 1300 [119] Patrick Blackburn Johan Bos Kristina Striegnitz. Learn prolog now, January
1301 2012.
- 1302 [120] Jurrien Stutterheim. The nanoprolog package, December 2011. [http://](http://hackage.haskell.org/package/NanoProlog)
1303 hackage.haskell.org/package/NanoProlog.
- 1304 [121] Evgeny Tarasov. The hswip package, August 2010. [http://hackage.](http://hackage.haskell.org/package/hswip)
1305 [haskell.org/package/hswip](http://hackage.haskell.org/package/hswip).
- 1306 [122] William E. Byrd The Reasoned Schemer' (MIT Press, 2005) by Daniel P. Fried-
1307 man and Oleg Kiselyov. minikanren.
- 1308 [123] Wren Thornton. The unification-fd package, July 2012. [http:](http://hackage.haskell.org/package/unification-fd)
1309 [//hackage.haskellhttp://yieldprolog.sourceforge.net/](http://hackage.haskell.org/package/unification-fd)
1310 [.org/package/unification-fd](http://hackage.haskell.org/package/unification-fd).
- 1311 [124] Jan Tikovsky. The monadiccp-gecode package, January 2014. [http://](http://hackage.haskell.org/package/monadiccp-gecode)
1312 hackage.haskell.org/package/monadiccp-gecode.
- 1313 [125] Carnegie Mellon University. Algebraic logic functional programming language,
1314 February 1995.
- 1315 [126] Dalhousie University. Control flow, January 2012.
- 1316 [127] Simon Fraiser University. Life programming language, March 1998.
- 1317 [128] Los Angeles University of California. Virgil programming language, March 2012.
- 1318 [129] Germany University of Kiel. Curry programming language, September 2013.
- 1319 [130] Maarten van Emden. Who killed prolog?, August 2010. [http://vanemden.](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/)
1320 [wordpress.com/2010/08/21/who-killed-prolog/](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/).
- 1321 [131] Andre Vellino. Prolog's death, August 2010. [http://synthese.](http://synthese.wordpress.com/2010/08/21/prologs-death/)
1322 [wordpress.com/2010/08/21/prologs-death/](http://synthese.wordpress.com/2010/08/21/prologs-death/).
- 1323 [132] Job Vranish. minikanrent, March 2013.
- 1324 [133] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer*
1325 *Science*, 2(04):461–493, 1992.
- 1326 [134] Haskell Website. Logic programming example, February 2010. [http://www.](http://www.haskell.org/haskellwiki/Logic_programming_example)
1327 [haskell.org/haskellwiki/Logic_programming_example](http://www.haskell.org/haskellwiki/Logic_programming_example).
- 1328 [135] Haskell Website. Logic programming example in haskell, February 2010.
- 1329 [136] Haskell Website. Quasiquote in haskell, January 2014. [http://www.](http://www.haskell.org/haskellwiki/Quasiquote)
1330 [haskell.org/haskellwiki/Quasiquote](http://www.haskell.org/haskellwiki/Quasiquote).
- 1331 [137] Haskell Wiki. Monads as computation, December 2011.

- 1332 [138] Haskell Wiki. Foldable and traversable, January 2013.
- 1333 [139] Haskell Wiki. The haskell programming language, October 2013.
- 1334 [140] Haskell Wiki. Embedded domain specific languages, September 2014.
- 1335 [141] Haskell Wiki. Haskell/laziness, November 2014.
- 1336 [142] Haskell Wiki. Monads in haskell, January 2014.
- 1337 [143] Haskell Wiki. Haskell in industry, June 2015.
- 1338 [144] Wikipedia. Prolog wikipedia, March 2004.
- 1339 [145] Wikipedia. Functional logic programming languages, February 2005.
- 1340 [146] Wikipedia. Common lisp object system, December 2013.
- 1341 [147] Wikipedia. Curry programming language, December 2013.
- 1342 [148] Wikipedia. Functional logic programming, May 2013.
- 1343 [149] Wikipedia. Quasiquotation, Novemeber 2013. [http://en.wikipedia.](http://en.wikipedia.org/wiki/Quasi-quotation)
1344 [org/wiki/Quasi-quotation](http://en.wikipedia.org/wiki/Quasi-quotation).
- 1345 [150] Wikipedia. Common language infrastructure, February 2014.
- 1346 [151] Wikipedia. Common language runtime, March 2014.
- 1347 [152] Wikipedia. Constraint handling rules, March 2014.
- 1348 [153] Wikipedia. Constraint programming, March 2014.
- 1349 [154] Wikipedia. Damas-hindley-milner type system, February 2014.
- 1350 [155] Wikipedia. Foreign function interface, January 2014.
- 1351 [156] Wikipedia. Lambda calculus, March 2014.
- 1352 [157] Wikipedia. List of multi paradigm languages, March 2014.
- 1353 [158] Wikipedia. Meta programming, March 2014.
- 1354 [159] Wikipedia. Ocaml, March 2014.
- 1355 [160] Wikipedia. Programming paradigm, March 2014.
- 1356 [161] Wikipedia. Comparison of prolog implementations, August 2015.
- 1357 [162] Wikipedia. Control flow, August 2015.
- 1358 [163] Wikipedia. Declarative programming, September 2015.

- 1359 [164] Wikipedia. Metasyntactic variable, October 2015.
- 1360 [165] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors.
1361 *Unifying Theories of Programming, 4th International Symposium, UTP 2012,*
1362 *Paris, France, August 27-28, 2012, Revised Selected Papers*, volume 7681 of
1363 *Lecture Notes in Computer Science*. Springer, 2013.
- 1364 [166] Takashi's Workplace. A prolog in haskell, April 2009. [http://propella.](http://propella.blogspot.in/2009/04/prolog-in-haskell.html)
1365 [blogspot.in/2009/04/prolog-in-haskell.html](http://propella.blogspot.in/2009/04/prolog-in-haskell.html).
- 1366 [167] xkcd. Haskell vs prolog, or giving haskell a choice, February 2009. [http:](http://echochamber.me/viewtopic.php?f=11&t=35369)
1367 [//echochamber.me/viewtopic.php?f=11&t=35369](http://echochamber.me/viewtopic.php?f=11&t=35369).
- 1368 [168] Switzerland cole Polytechnique Fdrale de Lausanne (EPFL) Lausanne. Scala
1369 programming language, 2002-2014.