

Embedding Programming Languages: PROLOG in HASKELL

by

Mehul Chandrakant Solanki

Bachelor of Engineering in Computer Science and Engineering Mumbai University 2012

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

November 2015

© Mehul Chandrakant Solanki, 2015

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.

0.1 Thesis Statement

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

We explore embedding domain specific languages in HASKELL

TABLE OF CONTENTS

Abstract	ii
0.1 Thesis Statement	ii
Table of Contents	iii
1 Introduction	1
1.1 Beginnings	1
1.2 Thesis Statement	2
1.3 Problem Statement	2
1.4 Thesis Organization	4
2 Background	6
3 Accomplished Work	11
3.1 Current Work	11
3.2 Contributions	12
3.3 Improved Contributions	13
3.4 Thesis Contributions	14
3.5 What work was done in terms of points	14
4 Embedding a Programming Language into another Programming Language	16
4.1 The Informal Content from Blogs, Articles and Internet Discussions	16
4.2 Related Books	17
4.3 Related Papers	18
4.4 Related Libraries in Haskell	19
4.5 From chap 7	20
4.6 Theory	22
4.7 Implementations	23
4.8 Important People	23
4.9 Miscellaneous / Possibly Related Content	23
5 Multi Paradigm Languages (Functional Logic Languages)	24
5.1 The Informal Content from Blogs, Articles and Internet Discussions	25
5.2 Literature and Publications	26
5.3 Some Multi Paradigm Languages	27

5.4	Functional Logic Programming Languages	27
5.5	From chap 9	28
5.6	Theory	29
5.7	Implementations	29
5.8	Miscellaneous / Possibly Related Content	30
6	Related Concepts	31
7	Prolog in ---- other languages	33
7.1	Theory	33
7.2	Implementations	34
7.3	Important People	34
7.4	Miscellaneous / Possibly Related Content	34
8	Prolog in Haskell	35
8.1	Theory	35
8.2	Implementations	36
8.3	Important People	37
8.4	Miscellaneous / Possibly Related Content	37
9	Quasiquotation	39
9.1	Theory	39
9.2	Implementations	39
9.3	Miscellaneous / Possibly Related Content	40
9.4	What is Quasiquotation ?	40
9.5	Quasiquotaion in HASKELL	40
10	Meta Syntactic Variables	42
11	Haskell or Why Haskell ?	44
12	Prolog or Why Prolog ?	47
13	Prototype 1	49
13.1	About this chapter	49
13.2	How Prolog works ?	49
13.3	What we do in this Prototype	51
13.4	Creating a data type	52
13.5	Working with the language	54
13.6	Black box	55
13.7	Something about unification-fd and Monadic Unification	55
14	Prototype 2.1	62
14.1	About this chapter	62
14.2	How prolog-0.2.0.1 works	62
14.3	What we do in this prototype?	64

14.4	Current implementation (prolog-0.2.0.1)	65
14.5	Modifications	66
14.6	Results	71
15	Prototype 3	72
15.1	Unification	72
15.2	Resolution	72
15.3	Search strategies	73
15.4	Stack Engine	73
15.5	Pure Engine	74
15.6	Andorra Engine	77
15.7	Current Unification	79
15.8	Syntax Modification	82
15.9	Monadic Unification	91
16	Prototype 4	93
16.1	I/O is pure	93
16.2	Dr. Casperson Pure IO	102
16.3	Mehul Pure IO	102
17	Work Completed	105
17.1	What we are doing	105
17.2	Unifiable Data Structures	105
17.3	Why Fix is necessary?	105
17.4	Dr. Casperson's Explanation	107
17.5	The other fix	107
17.6	The Fix we use	108
17.7	Opening up or Extending language Explanation using Box Analogy	110
18	Results	113
18.1	Types	113
18.2	Lazy Evaluation	114
18.3	Opening up the Language	114
18.4	Quasi Quotation	114
18.5	Template Haskell	114
18.6	Higher Order Functions	114
18.7	I/O	115
18.8	Mutability	115
18.9	Unification	115
18.10	Monads	115
19	Future Scope	116
20	Conclusion / Expected Outcomes	117

21 Editing to do	118
21.1 Editing suggestions from David	120
Bibliography	123

List of Tables

List of Figures

13.1	Trace for append [122]	51
13.2	A sample Minted figure	61
14.1	A language-processing system [1]	63
14.2	Phases of Compiler [1]	64
17.1	Functor Hierarchy [146]	106
21.1	A sample Minted figure	121

Chapter 1

Introduction

1.1 Beginnings

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 awns. 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

1 forced to choose.

2 **1.2 Thesis Statement**

3 The thesis aims to provide insights into merging two declarative languages namely, HASKELL
4 and PROLOG by embedding the latter into the former and analysing the result of doing so as
5 they have conflicting characteristics. The finished product will be something like a *haskel-*
6 *lised* PROLOG which has logical programming like capabilities.

7 **1.3 Problem Statement**

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

17 PROLOG is a language that has a hard time being adopted. Born in an era where proce-
18 dural languages were receiving a lot of attention, it suered from competing against another
19 new kid on the block: C. Some of the problems were of its own making. Basic features
20 like modules were not provided by all compilers. Practical features for real world problems
21 were added in an ad hoc way resulting in the loss of its purely declarative charm. Some
22 say that PROLOG is fading away, [89, 138, 137]. It is apparently not used for building large
23 programs [152, 115, 67]. However there are a lot of good things about Prolog: it is ideal
24 for search problems; it has a simple syntax, and a strong underlying theory. It is a language

1 that should not die away.

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

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

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

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

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

22 The solution, a programming language with a split personality, in our case, sometimes
23 functional, sometimes logical and sometimes both. Depending upon the problem, the lan-
24 guage shapes itself accordingly and exhibits the desired characteristics. The ideal situation
25 is a language with a rich feature set and the ability to mould itself according to the problem.
26 A language with ability to take the appropriate skill set and present it to the programmer,
27 which will reduce the hassle of jumping between languages or forcibly trying to solve a

1 problem according to a paradigm.

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

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

6 This can be achieved in two ways,

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

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

20 **1.4 Thesis Organization**

21 The next chapter, [Chapter 2](#) provides details about the short comings of the previous works
22 and the road to a better future. [Chapter 3](#), the background talks about the programming
23 paradigms and languages in general and the ones in question. Then we look at the ques-
24 tion from different angles namely, [Chapter 4](#), Embedding a Programming Language into
25 another Programming Language and [Chapter 5](#), Multi Paradigm Languages (Functional

- 1 Logic Languages). Some of the indirectly related content [Chapter 6](#) and finishing off with
- 2 the [Chapter 7](#), the expected outcomes.

Chapter 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 [72] 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, [59] 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 [164], 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 [105], a single rule and a single function definition scheme.

1 In particular there are typed and untyped λ calculi. In the untyped λ calculus functions have
2 no predetermined type whereas typed lambda calculus puts restriction on what sort(type)
3 of data can a function work with. SCHEME is based on the untyped variant while ML
4 and HASKELL are based on typed λ calculus. Most typed λ calculus languages are based
5 on Hindley-Milner or Damas-Milner or Damas- Hindley-Milner [162] type system. The
6 ability of the type system to give the most general type of a program without any help
7 (annotation). The algorithm [22] works by initially assigning undefined types to all inputs,
8 next check the body of the function for operations that impose type constraints and go on
9 mapping the types of each of the variables, lastly unifying all of the constraints giving the
10 type of the result.

11 Logical Programming, [117] on the other hand is based on formal logic. A program is
12 a set of rules and formulæ in symbolic logic that are used to derive new formulas from the
13 old ones. This is done until the one which gives the solution is not derived.

14 The languages to be worked with being HASKELL and PROLOG respectively. Some
15 differences include things like, HASKELL uses Pattern Matching while PROLOG uses Uni-
16 fication, HASKELL is all about functions while PROLOG is on Horn Clause Logic and so
17 on.

18 PROLOG [152] being one of the most dominant Logic Programming Languages has
19 spawned a number of distributions and is present from academia to industry.

20 HASKELL is one the most popular [77] functional languages around and is the first
21 language to incorporate Monads [140] for safe *IO*. Monads can be described as composable
22 computation descriptions [150] . Each monad consists of a description of what has action
23 has to be executed, how the action has to be run and how to combine such computations.
24 An action can describe an impure or side-effecting computation, for example, *IO* can be
25 performed outside the language but can be brought together with pure functions inside in
26 a program resulting in a separation and maintaining safety with practicality. HASKELL
27 computes results lazily and is strongly typed.

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

4 The selection of languages is not uncommon and this not only the case with HASKELL,
5 PROLOG seems to be the all time favourite for "let's implement PROLOG in the language
6 X for proving it's power and expressibility". The PROLOG language has been partially
7 implemented [34] in other languages like SCHEME [114], LISP [70, 103, 104], JAVA [152,
8 62], JAVASCRIPT [63] and the list [97] goes on and on.

9 The technique of embedding is a shallow one, it is as if the embedded language floats
10 over the host. Over time there has been an approach that branches out, which is Paradigm
11 Integration. A lot of work has been done on Unifying the Theories of Programming [36,
12 14, 98, 174, 56, 46]. All sorts of hybrid languages which have characteristics from more
13 than one paradigm are coming into the mainstream.

14 Before moving on, let us take a look at some terms related to the content above. To
15 begin with Foreign Function Interfaces (FFI) [163], a mechanism by which a program
16 written in one programming language can make use of services written in another. For
17 example, a function written in C can be called within a program written in HASKELL and
18 vice versa through the FFI mechanism. Currently the HASKELL foreign function interface
19 works only for one language. Another notable example is the Common Foreign Function
20 Interface (CFFI) [13] for LISP which provides fairly complete support for C functions and
21 data. JAVA provides the Java Native Interface(JNI) for the working with other languages.
22 Moreover there are services that provide a common platform for multiple languages to
23 work with each other and run their programs. They can be termed as multi lingual run
24 times which lay down a common layer for languages to use each others functions. An
25 example for this is the Microsoft Common Language Runtime (CLR) [159] which is an
26 implementation of the Common Language Infrastructure (CLI) standard [158].

27 Another important concept is meta programming [166], which involves writing com-

puter 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 [52] 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 [80, 143, 157], 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,

```

1  append(Ps, Qs, Rs) = (Ps = [] & Qs = Rs) ||
2      ( X, Xs, Ys -> Ps = [X|Xs] &
3          Rs = [X|Ys] &
4          append(Xs, Qs, Ys))

```

Consider the Object Functional Programming Language, SCALA [177], 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 1. Embedding, where PROLOG is merely translated to the host language HASKELL or
2 a Foreign Function Interface.
- 3 2. Paradigm Integration, developing a hybrid programming language that is a Func-
4 tional Logic Programming Language with a set of characteristics derived from both
5 the participating languages.
- 6 The approaches listed above are next in line for discussions.

Chapter 3

Accomplished 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 [65] 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 [175] 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. In the case of 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

1 is synonymous with the embeddings mentioned above. Some are far more feature
2 rich than others that is with some practical PROLOG concepts, but are not complete.

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

4 And as far as the idea of merging paradigms goes, it is not the main focus of this
5 thesis and can be more of an "add-on". A handful of crossover hybrid languages based
6 on HASKELL exist, CURRY [136] being the prominent one. Moving away from HASKELL
7 and exploring other languages from different paradigms, a respectable number of crossover
8 implementations exist but again most of them have faded out.

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

16 3.2 Contributions

17 Taking into consideration above, there is quite some room for improvement and additions.
18 Moving onto what this thesis shall explore, first thing's first a complete, fully functional
19 library which comes close to a PROLOG like language and has practical abilities to carry
20 out real-world tasks. They include predicates like *cut*, *assert*, *fail*, *setOf*, *bagOf* among
21 others. This would form the first stage of the implementation. Secondly, exploring aspects
22 such as *assert* and database capabilities. A third question to address is the accommodation
23 of input and output, specifically dealing with the *IO Monad* in HASKELL with PROLOG *IO*.
24 Moreover, PROLOG is an untyped language which allows lists with elements of different
25 types to be created. Something like this is not by default in HASKELL. Hence syntactic

1 support for the same is the next question to address. Furthermore, experimenting with how
2 programs expressed with same declarative meaning differ operationally. Lastly, how would
3 characteristics of hybrid languages fit into and play a role in an embedded setting.

4 **3.3 Improved Contributions**

5 1. Most languages have a recursive abstract syntax which restricts the eDSL in terms of
6 its capability to *open up* the language i.e. to include meta syntactic variables, adding
7 custom quantifiers and logic. ([Prototype 1](#)) provides a methodology to convert a
8 language whose recursive abstract syntax is represented by a tree into a non-recursive
9 version whose fixed point is isomorphically equivalent to the original type. One of
10 the outcomes is a polymorphically typed embedded language within HASKELL

11 To test it out we adopt the closed PROLOG like language defined in [107] and open
12 it up. And for the unification part we use [128], which provides a generic unification
13 algorithm implementation encapsulated into a monad.

14 2. ([Prototype 2](#)) does the what a PROLOG query resolver would do given a query and a
15 knowledge base. The mechanism for the same is adopted from [107]. The embedded
16 language is modified as per the procedure in ([Prototype 1](#)) and the monadic unifica-
17 tion part is plugged into the existing architecture to demonstrate that it is independent
18 of the other components. Lastly the result is converted into the original language via
19 a translate function.

20 3. ([Prototype 3](#)) demonstrates the modularity of the unification process of the query
21 resolver with multiple search strategies.

22 4. ([Prototype 4](#)) throws light on how IO operations can be embedded into the abstract
23 syntax of a DSL which when interpreted would produce output consisting of a pure
24 set of instructions irrespective of the nature of the construct. The effects are only

1 produced only when the actions are executed.

2 **3.4 Thesis Contributions**

3 1. Prototype 1 does flattening language opening up the language (binding monad) adding
4 custom variables monadic unification (stuff happens in a bubble) $\text{rec type} \rightarrow \text{non rec}$
5 $\text{type} \rightarrow \text{fix non rec type isomorphically} == \text{rec type}$

6 You can make an Flatterm int
7 but you cannot make term int
8 adding quantifiers

9 2. Prototype 2 does extends current prolog-0.2.0.1 this is to show that we can plug out
10 approach into existing implementation and things work

11 3. Prototype 3 does variable search strategy what ever method you do for searching at
12 the point of unification you can do it with our approach

13 4. Prototype 4 does how can io be squeezed into this model where whenever the resolver
14 encounters an io operation it generates a thunk (sort of unsolved statement) which
15 when executed would result in a side effect but till that point every thing is pure

16 **3.5 What work was done in terms of points**

17 1. Literature review on eDSL's.

18 2. Short survey on multi paradigm declarative languages.

19 3. Accumulated and evaluated PROLOG in HASKELL.

20 4. Defined a procedure to open up a language starting from a generic recursive abstract
21 syntax.

- 1 5. Made a few libraries to work together.
- 2 6. Some stuff for monadic unification.
- 3 7. Something to show it was modular and independent of the original grammar.
- 4 8. Something to show that the unification part is independent of the search strategy and
5 hence multiple ones can be used, possibly simultaneously to find a solution.
- 6 9. Creating a micro language to represent and encapsulate IO operation in an eDSL so
7 that the it remains pure even after interpretation and only produces side effects when
8 the action is actually executed and hence in some way it can be controlled.

Chapter 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 let's take a look at some of the unofficially published content. 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

1 content.

2 A lot has been talked about embedding languages and also the techniques and methods
3 to do so. It might not seem such a hot topic as such but it has always been a part of any pro-
4 gramming language to work and integrate their code with other programming languages.
5 One of the top discussions are in, Lambda the Ultimate, The Programming Languages
6 Weblog [73], which lists a number of PROLOG implementations in a variety of languages
7 like LISP, SCHEME, SCALA, JAVA, JAVASCRIPT, RACKET [114] and so on. Moreover the
8 discussion focusses on a lot of critical points that should be considered in a translation of
9 PROLOG to the host language regarding types and modules among others.

10 One of the implementations discussed redirects us to one of the most earliest imple-
11 mentations of PROLOG in HASKELL for Hugs 98, called Mini PROLOG [65]. Although this
12 implementation takes as reference the working of the PROLOG Engine and other details,
13 it still is an unofficial implementation with almost no documentation, support or ongoing
14 development. Moreover, it comes with an option of three engines to play with but still lacks
15 complete list support and a lot of practical features that PROLOG has and this seems to be
16 a common problem with the only other implementation that exists, [175].

17 Adding fuel to fire, is the question on PROLOG's existence and survival [137, 89, 138,
18 115] since its use in industry is far scarce than the leading languages of other paradigms.
19 The purely declarative nature lacks basic requirements such as support for modules. And
20 then there is the ongoing comparison between the siblings [176] of the same family, the
21 family of Declarative Languages. Not to forget HASKELL also has some tricks [141] up its
22 sleeve which enables encoding of search problems.

23 **4.2 Related Books**

24 As HASKELL is relatively new in terms of being popular, its predecessors like SCHEME
25 have explored the territory of embedding quite profoundly [27], which aims at adding a

1 few constructs to the language to bring together both styles of Declarative Programming
2 and capture the essence of PROLOG. Moreover, HASKELL also claims for it to be suitable
3 for basic Logic Programming naturally using the List Monad [142]. A general out look
4 towards implementing PROLOG has also been discussed by [71] to push the ideas forward.

5 **4.3 Related Papers**

6 There is quite some literature that can be found and which consist of embedding detailed
7 parts of Prolog features like basic constructs, search strategies and data types. One of
8 the major works is covered by the subsection below consisting of a series of papers from
9 Mike Spivey and Silvija Seres aimed at bring Haskell and Prolog closer to each other. The
10 next subsection covers the literature based on the above with improvements and further
11 additions.

12 • Papers from Mike Spivey and Silvija Seres

13 The work presented in the series [119, 111, 112, 118, 109] attempts to encapsulate
14 various aspects of an embedding of PROLOG in HASKELL. Being the very first doc-
15 umented formal attempt, the work is influenced by similar embeddings of PROLOG
16 in other languages like SCHEME and LISP. Although the host language has distinct
17 characteristics such as lazy evaluation and strong type system the proposed scheme
18 tends to be general as the aim here is to achieve PROLOG like working not a multi
19 paradigm declarative language. PROLOG predicates are translated to HASKELL func-
20 tions which produce a stream of results lazily depicting depth first search with sup-
21 port for different strategies and practical operators such as *cut* and *fail* with higher
22 order functions. The papers provide a minimalistic extension to HASKELL with only
23 four new constructs. Though no implementation exists, the synthesis and transforma-
24 tion techniques for functional programs have been *logicalised* and applied to PRO-
25 LOG programs. Another related work [120] looks through conventional data types so

1 as to adapt to the problems at hand so as to accommodate and jump between search
2 strategies.

3 • Other works related or based on the above

4 Continuing from above, [21] taps into the advantages of the host language to em-
5 bed a typed functional logic programming language. This results in typed logical
6 predicates and a backtracking monad with support for various data types and search
7 strategies. Though not very efficient nor practical the method aims at a more ele-
8 gant translation of programs from one language to the other. While other papers [39]
9 attempt at exercising `HASKELL` features without adding anything new rather doing
10 something new with what is available. Specifically speaking, using `HASKELL` type
11 classes to express general structure of a problem while the solutions are instances.
12 [55] replicates `PROLOG`'s control operations in `HASKELL` suggesting the use of the
13 `HASKELL State Monad` to capture and maintain a global state. The main contribu-
14 tions are a Backtracking Monad Transformer that can enrich any monad with back-
15 tracking abilities and a monadic encapsulation to turn a `PROLOG` predicate into a
16 `HASKELL` function.

17 4.4 Related Libraries in Haskell

18 • Prolog Libraries

19 To replicate Prolog like capabilities Haskell seems to be already in the race with a
20 host of related libraries. First we begin with the libraries about Prolog itself, a few
21 exist [124] being a preliminary or "mini Prolog" as such with not much in it to be able
22 to be useful, [125] is all powerful but is an Foreign Function Interface so it is "Prolog
23 in Haskell" but we need Prolog for it, [107] which is the only implementation that
24 comes the closest to something like an actual practical Prolog. But all they give is a
25 small interpreter, none or a few practical features, incomplete support for lists, minor

1 or no monadic support and an REPL without the ability to "write a Prolog Program
2 File".

3 • Logic Libraries

4 The next category is about the logical aspects of Prolog, again a handful of libraries
5 do exist and provide a part of the functionality which is related propositional logic
6 and backtracking. [25] is a continuation-based, backtracking, logic programming
7 monad which sort of depicts Prolog's backtracking behaviour. Prolog is heavily
8 based on formal logic, [44] provides a powerful system for Propositional Logic.
9 Others include small hybrid languages [40] and Parallelising Logic Programming
10 and Tree Exploration [24].

11 • Unification Libraries

12 The more specific the feature the lesser the support in Haskell. Moving on to the other
13 distinct feature of Prolog is Unification, two libraries exist [128], [99] that unify two
14 Prolog Terms and return the resulting substitution.

15 • Backtracking

16 Another important aspect of PROLOG is backtracking. To simulate it in HASKELL,
17 the libraries [41, 116] use monads. Moreover, there is a package for the EGISON
18 programming language [57] which supports non-linear pattern-matching with back-
19 tracking.

20 **4.5 From chap 7**

21 Embedding a language into another language has been explored with a variety of languages.
22 Attempts have been made to build Domain Specific Languages from the host languages
23 [58], Foreign Function Interfaces [10]

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

- 4 1. Syntax
- 5 2. Semantics
- 6 3. Standard Library
- 7 4. Runtime System
- 8 5. Parsers
- 9 6. Code Generators
- 10 7. Interpreters
- 11 8. Debuggers

12 A lot of the above can be skipped or taken from the base language if an embedding
13 approach is chosen. For an embedded domain specific language the functionality is trans-
14 lated and written as an add on. The result can be thought of as a library. But the difference
15 between an ordinary library and an eDSL is the feature set provided and the degree of em-
16 bedding [148]. For example, reading a file and parsing its contents to perform certain
17 operations to return *string* results is a shallow form of embedding as the generation of
18 code, results is not native nor are the functions processing them dealing with embedded
19 data types as such. On the other hand, building data structures in the base language which
20 represent the target language expression would be called a deep embedding approach.

21 The snippet of HASKELL code below describes PROLOG entities,

```
1  data Term = Struct Atom [Term]
2           | Var VariableName
3           | Wildcard
```

```

4         | PString   !String
5         | PInteger  !Integer
6         | PFloat    !Double
7         | Flat [FlatItem]
8         | Cut Int
9     deriving (Eq, Data, Typeable)

```

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

3 As discussed in the

4 4.6 Theory

5 1. Papers

- 6 (a) Embedding an interpreted language using higher-order functions, [100]
- 7 (b) Building domain-specific embedded languages, [58]
- 8 (c) Embedded interpreters, [11]
- 9 (d) Cayenne – a Language With Dependent Types, [6]
- 10 (e) Foreign interface for PLT Scheme, [10]
- 11 (f) Dot-Scheme: A PLT Scheme FFI for the .NET framework, [95]
- 12 (g) Application-specific foreign-interface generation, [101]
- 13 (h) Embedding S in other languages and environments, [76]

14 2. Books

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

16 3. Articles / Blogs / Discussions

- 17 (a) Embedding one language into another, [74]

1 (b) Application-specific foreign-interface generation, [75]

2 (c) Linguistic Abstraction, [92]

3 (d) LISP, Unification and Embedded Languages, [93]

4 4. Websites

5 (a) Embedding SWI-Prolog in other applications, [34]

6 **4.7 Implementations**

7 1. Lots of them I guess

8 **4.8 Important People**

9 1. ????

10 **4.9 Miscellaneous / Possibly Related Content**

11 1. ????

1 Chapter 5

2 Multi Paradigm Languages (Functional 3 Logic Languages)

4 Over the years another approach has branched off from embedding languages, to merge
5 and/or integrate programming languages from different paradigms. Let us take an example
6 of the SCALA Programming Language [177], a hybrid Object-Functional Programming
7 Language which takes a leaf from each of the two books. In this thesis, the languages in
8 question are HASKELL and PROLOG. This section takes a look at the literature on Multi
9 Paradigm Languages, mainly Functional Logic Programming Languages that combine two
10 of the most widespread Declarative Programming Styles.

11 A peak into language classification reveals that it is not always a straight forward task to
12 segregate languages according to their features and/or characteristics. Turns out that there
13 are a number of notions which play a role in deciding where the language belongs. Many
14 a times a language ends up being a part of almost all paradigms due extensive libraries.
15 Simply speaking, a multi-paradigm programming language is a programming language
16 that supports more than one programming paradigm [72], more over as Timothy Budd
17 puts it [168] ”The idea of a multi paradigm language is to provide a framework in which
18 programmers can work in a variety of styles, freely intermixing constructs from different

1 paradigms.”

2 **5.1 The Informal Content from Blogs, Articles and Inter-** 3 **net Discussions**

4 • Multi Paradigm Languages

5 A lot has been talked and discussed on coming to clear grounds about the classifica-
6 tion of programming languages. If the conventional ideology is considered then the
7 scope of each language is pretty much infinite as small extension modules replicate
8 different feature sets which are not naturally native to the language itself. The defini-
9 tions of multi paradigm languages across the web [168, 90, 15] converge to roughly
10 the same thing that of providing a framework to work with different styles with a list
11 of languages [165, 33] that ticks the boxes. Generally speaking, it does not feel all
12 that hot or popular in programming circles; one reason could be that it is a very broad
13 topic and specifying details can clear the fog.

14 • Functional Logic Programming Languages

15 Continuing from the previous section, narrowing down the search by considering
16 only multi paradigm declarative languages namely, Functional Logical programming
17 languages. By doing so a large amount of information pops up, from articles that
18 give brief description and mentions [156, 153] to the implementing techniques [3]
19 which give a brief overview of the aim and also the backdrop of publications.

20 The jackpot however is the fact that there is a dedicated website [50] for the history,
21 research and development, existing languages, the literature, the contacts and every-
22 thing else that one can think of for functional logic languages. As a matter of fact the
23 holy grail of information is maintained by two of the most important people in the
24 field Michael Hanus [48] and Sergio Antoy [4].

1 5.2 Literature and Publications

2 • Multi Paradigm Languages

3 Possibly one of the most important works towards bringing programming styles to-
4 gether is the book by C.A.R. Hoare [56] which points out that among the large num-
5 ber of programming paradigms and/or theories the unification theory serves as a com-
6 plementary rather than a replacement to relate the universe. As as always since we
7 are talking about HASKELL we have to include monads and unifying theories using
8 monads [46].

9 • Functional Logic Programming Languages

10 A recent survey [49] throws light on these hybrid languages.

11 One of the most prominent multi paradigm languages in HASKELL is CURRY [5].
12 Th syntax is borrowed from the parent language and so are a lot of the features.
13 Taking a recap, a functional programming language works on the notion of mathe-
14 matical functions while a logic programming language is based on predicate logic.
15 The strong points of CURRY are that the features or basis of the language are general
16 and are visible in a number of languages like [29]. The language can play with prob-
17 lems from both worlds. In a problem where there are no unknowns and/or variables
18 the language behaves like a functional language which is pattern matching the rules
19 and execute the respective bodies. In the case of missing information, it behaves
20 like PROLOG; a sub-expression e is evaluated on the conditions that it should satisfy
21 which constraint the possible values of e . This brings us to the first important fea-
22 ture of functional logic languages *narrowing*. The expressions contain *free variables*;
23 simply speaking incomplete information that needs to be *unified* to a value depending
24 on the constraints of the problem. The language introduces only a few new constructs
25 to support non determinism and choice. Firstly, *narrowing* ($==$), which deals with
26 the expressions and unknown values and binds them with appropriate values. The

1 next one is the *choice* operator (?) for non-deterministic operations. Lastly, for uni-
2 fying variables and values under some conditions, (&) operator has been provided to
3 add constraints to the equation. Putting it all together, it gives us the feel of a logic
4 language for something that looks very much like HASKELL. Unification is like two
5 way pattern matching and with a similar analogy CURRY is a HASKELL that works
6 both ways and hence variables can be on either sides. Although the language can do
7 a lot but gaps do exist such as the improvement of narrowing techniques.

8 **5.3 Some Multi Paradigm Languages**

9 The list of multi paradigm languages is huge, but in this thesis we will mostly stick to Func-
10 tional Logical programming languages. Beginning with functional hybrids, a small project
11 language called VIRGIL [135], combining objects to work with functions and procedures.
12 On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [154]. This can be justified
13 as object oriented programming has been one of the most dominant styles of programming
14 and hence even HASKELL has one called O'HASKELL [91] though it last saw a release
15 back in 2001. Another prominent implementation is OCAML [167, 94] which adds object
16 oriented capabilities with a powerful type system and module support. This is the case with
17 most of the languages in this section hardly a few have survived as the new ones incorpo-
18 rated the positives of the old. As mentioned before one of the most popular [77] and widely
19 usage both in academia and industry is the SCALA [177] programming language stands
20 out.

21 **5.4 Functional Logic Programming Languages**

22 Knowing that there is quite some amount of literature out there on these type of languages,
23 it is fairly easy to say that there have been numerous attempts at specifications and/or imple-
24 mentations. Sadly though not many have survived leave alone being successful as a result of

1 the competition. Only the ones that are easily available or have an implementation or have
2 been cited or referred by other attempts have been included as the list is long and does not
3 reflect the main intention of the document. Beginning with the ones from Australia, which
4 seems to be a popular destination for fiddling with PROLOG and merging paradigms. As of
5 now there have been three popular ones, beginning with NEU PROLOG, [78], OZ (MOZART
6 PROGRAMMING SYSTEM) [23] and MERCURY [30]. Delving deeper the languages feel
7 more like extensions of PROLOG rather than hybrids. Starting with MERCURY which a
8 boundary between deterministic and non-deterministic programs, similarly NUE PROLOG
9 has special support for functions while OZ gives concurrent constraint programming plus
10 distributed support, with different function types for goal solving and expression rewrit-
11 ing. ESCHER [79] comes very close to HASKELL with monads, higher order functions and
12 lazy evaluation. Taking a look at PROLOG variants, CIAO [20]; a preprocessor to PROLOG
13 for functional syntax support, λ PROLOG [88] aims at modular higher order programming
14 with abstract data types in a logical setting, BABEL [53, 85, 84] combines pure PROLOG
15 with a first order functional notation, LIFE [134] is for Logic, Inheritance, Functions and
16 Equations in PROLOG syntax with currying and other features like functional languages
17 and others [12, 81].

18 The functional language SCHEME is a very popular choice for this sort of a thing. With
19 a book [27] and an implementation to accompany [28, 127] which seems to have translated
20 into HASKELL, [61, 42, 139].

21 Finally talking about CURRY, one of the most popular HASKELL based multi paradigm
22 languages with support for deterministic and non-deterministic computations. Contributing
23 to the same there have been some predecessors [132, 29].

24 **5.5 From chap 9**

25 Unifying / Marrying / Merging / Combining Programming Paradigms / Theories

1 **5.6 Theory**

2 • Papers

- 3 1. Unifying Theories of Programming with Monads, [46]
- 4 2. Symposium on Unifying Theories of Programming, 2006, [36].
- 5 3. Symposium on Unifying Theories of Programming, 2008, [14].
- 6 4. Symposium on Unifying Theories of Programming, 2010, [98].
- 7 5. Symposium on Unifying Theories of Programming, 2012, [174].

8 • Books

- 9 1. Unifying Theories of Programming, [56]

10 • Articles / Blogs / Discussions

- 11 1. ???

12 • Websites

- 13 1. ???

14 **5.7 Implementations**

- 15 1. Scala
- 16 2. Virgil
- 17 3. CLOS, Common Lisp Object System
- 18 4. Visual Prolog
- 19 5. ????

¹ **5.8 Miscellaneous / Possibly Related Content**

- ² 1. ???

Chapter 6

Related Concepts

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 [2] 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 [161] 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 [160], 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 [51, 155]. Residuation involves delaying of functions calls until they are deterministic, that is, deterministic reduction of functions with partial data. This princi-

1 ple is used in languages like ESCHER [79], LIFE [134], NUE-PROLOG [78] and Oz [23].
2 Narrowing on the other hand is a mixture of reduction in functional languages and unifi-
3 cation in logic languages. In narrowing, a variable is bound a value within the specified
4 constraints and try to find a solution, values are generated while searching rather than just
5 for testing. The languages based on this approach are ALF [132], BABEL [53], LPG [12]
6 and CURRY [136].

7 F-Algebras

8 We are now ready to define F-algebras in the most general terms. First I'll use the
9 language of category theory and then quickly translate it to HASKELL.

10 An F-algebra consists of:

- 11 1. an endofunctor F in a category C,
- 12 2. an object A in that category, and
- 13 3. a morphism from F(A) to A.

14 An F-algebra in HASKELL is defined by a functor f, a carrier type a, and a function
15 from (f a) to a. (The underlying category is Hask.)

16 Right about now the definition with which I started this post should start making sense:

```
type Algebra f a = f a -> a
```

17 For a given functor f and a carrier type a the algebra is defined by specifying just one
18 function. Often this function itself is called the algebra, hence my use of the name alg in
19 previous examples.

Chapter 7

Prolog in ____ other languages

Prolog in -----

7.1 Theory

• Papers

1. QLog, [70]

2. LogLisp Motivation, design, and implementation, [103]

• Books

1. Warrens Abstract Machine A TUTORIAL RECONSTRUCTION, [2]

2. LOGLISP: an alternative to PROLOG, [104]

• Articles / Blogs / Discussions

1. Hello

• Websites

1. Hello

1 **7.2 Implementations**

- 2 1. Castor : Logic paradigm for C++, [87]
- 3 2. GNU Prolog for Java, [47]
- 4 3. JLog - Prolog in Java, [62]
- 5 4. JScriptLog - Prolog in Java, [63]
- 6 5. Quintus Prolog, [96]
- 7 6. Yield Prolog, [97]
- 8 7. Racklog, [114]

9 **7.3 Important People**

- 10 1. ???

11 **7.4 Miscellaneous / Possibly Related Content**

- 12 1. ???

1 Chapter 8

2 Prolog in Haskell

3 Prolog in Haskell

4 8.1 Theory

5 • Papers

6 1. Embedding Prolog in Haskell / Functional Reading of Logic Programs, [119]

7 2. Algebra of Logic Programming, [111]

8 3. The Algebra of Logic Programming, [109]

9 4. Optimisation Problems in Logic Programming : An Algebraic Approach, [110]

10 5. Higher Order Transformation of Logic Programs, [112]

11 6. The Algebra of Searching, [118]

12 7. FUNCTIONAL PEARL Combinators for breadth-first search, [120]

13 8. Type Logic Variables, K Classen, [21]

14 9. A Type-Safe Embedding of Constraint Handling Rules into Haskell Wei-Ngan
15 Chin, Mar-tin Sulzmann and Meng Wang, [19]

1 10. Prological Features in a Functional Setting Axioms and Implementation, R
2 Hinze, [55]

3 11. Escape from Zurg: An Exercise in Logic Programming, [39]

4 • Books

5 1. The Reasoned Schemer, Daniel P. Friedman, William E. Byrd, Oleg Kiselyov,
6 [27]

7 2. Programming Languages: Application and Interpretation, Shriram Krishna-
8 murthi, Chapters 33-34 of PLAI discuss Prolog and implementing Prolog, [71]

9 • Articles / Blogs / Discussions

10 1. Lambda the Ultimate, Programming Languages, [73]

11 2. Takashi's Workplace (Implementation), [175]

12 3. Haskell vs. Prolog Comparison, [121]

13 • Websites

14 1. Logic Programming in Haskell, [141]

15 **8.2 Implementations**

16 1. A Prolog in Haskell, Takashi's Workplace, [175]

17 2. Mini Prolog for Hugs 98, [65]

18 3. Nano Prolog, [124]

19 4. Prolog, [107]

20 5. cspm-To-Prolog, [43]

- 1 6. prolog-graph, [9]
- 2 7. prolog-graph-lib, [106]
- 3 8. hswip, [125]

4 **8.3 Important People**

- 5 1. Mike Spivey
- 6 2. Silvija Seres

7 **8.4 Miscellaneous / Possibly Related Content**

- 8 1. Unification Libraries
 - 9 (a) unification-fd, [128]
 - 10 (b) cmu, [99]
- 11 2. Logic Libraries
 - 12 (a) logicct, [25], [26]
 - 13 (b) logic-classes, [?]
 - 14 (c) proplogic, [44]
 - 15 (d) cflp, [40]
 - 16 (e) logic-grows-on-trees, [24]
- 17 3. Concatenative Programming
 - 18 (a) peg, [31]
- 19 4. Constraint Programming and Constraint Handling Rules

- 1 (a) monadiccp, [102]
- 2 (b) monadicccp-gecode, [131]
- 3 (c) csp, [7]
- 4 (d) liquid fix point, [108]

Chapter 9

Quasiquotation

9.1 Theory

1. Papers

(a)

2. Books

(a)

3. Articles / Blogs / Discussions

(a)

4. Websites

(a) Quasiquotation Wikipedia, [157]

(b) Quasiquotation in Haskell, [143]

9.2 Implementations

1.

1 9.3 Miscellaneous / Possibly Related Content

2 1.

3 9.4 What is Quasiquotation ?

4 1. [54]

5 When language is used to attribute properties to language or otherwise theorize about
6 it, a linguistic device is needed that turns language on itself. Quotation is one such
7 device. It is our primary meta-linguistic tool.

8 2. [32]

9 a metalinguistic device for referring to the form of an expression containing variables
10 without referring to the symbols for those variables. Thus while "not p" refers to the
11 expression consisting of the word not followed by the letter p, the quasi-quotation
12 \ulcorner not p \urcorner refers to the form of any expression consisting of the word not followed by
13 any value of the variable p.

14 3. Quasiquotation Wikipedia, [157]

15 Quasi-quotation or Quine quotation is a linguistic device in formal languages that
16 facilitates rigorous and terse formulation of general rules about linguistic expressions
17 while properly observing the usemention distinction.

18 [173] The usemention distinction is a foundational concept of analytic philosophy,[1]
19 according to which it is necessary to make a distinction between using a word (or
20 phrase) and mentioning it

21 9.5 Quasiquotaion in HASKELL

22 [143, 80]

1 Quasiquoting allows programmers to use custom, domain-specific syntax to construct
2 fragments of their program. Along with HASKELL's existing support for domain specific
3 languages, you are now free to use new syntactic forms for your EDSLs.

4 Working with complex data types can impose a significant syntactic burden; extensive
5 applications of nested data constructors are often required to build values of a given data
6 type, or, worse yet, to pattern match against values.

7 Allow HASKELL expressions and patterns to be constructed using domain specific,
8 programmer-defined concrete syntax.

Chapter 10

Meta Syntactic Variables

Some sources for the topic

[172] 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.

[17] 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

1 some extent, the list of one's preferred metasyntactic variables is a cultural signature. They
2 occur both in series (used for related groups of variables or objects) and as singletons. Here
3 are a few common signatures:

4 [60] In programming, a metasyntactic (which derives from meta and syntax) variable is
5 a variable (a changeable value) that is used to temporarily represent a function . Examples
6 of metasyntactic variables include (but are by no means limited to) ack, bar , baz, blarg,
7 wibble, foo , fum, and qux. Metasyntactic variables are sometimes used in developing a
8 conceptual version of a program or examples of programming code written for illustrative
9 purposes.

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

12 [16]

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

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

Chapter 11

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. Why a Functional Language?

2. 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 [147]. It is one of the popular functional programming languages [77]. HASKELL is widely used in the industry [151].

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 [66, 83] which talk about introducing combinator libraries for custom functionality.

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

5 3. Looking at HASKELL as a tool for embedding domain specific languages[64]

6 (a) Monads

7 Control flow defines the order/ manner of execution of statements in a pro-
8 gram[170]. The specification is set by the programming language. Generally,
9 in the case of imperative languages the control flow is sequential while for a
10 functional language is recursion [133]. For example, JAVA has a top down
11 sequential execution approach. The declarative style consists of defining com-
12 ponents of programs i.e. computations not a control flow[171].

13 This is where HASKELL shines by providing something called a *monad*. Func-
14 tional Programming Languages define computations which then need to be or-
15 dered in some way to form a combination[144]. A monad gives a bubble within
16 the language to allow modification of control flow without affecting the rest of
17 the universe. This is especially useful while handling side effects.

18 A related topic would be of persistence languages, architectures and data struc-
19 tures. Persistent programming is concerned with creating and manipulating
20 data in a manner that is independent of its lifetime [86]. A persistent data struc-
21 ture supports access to multiple versions which may arise after modifications
22 [35, 68]. A structure is partially persistent if all versions can be accessed but
23 only the current can be modified and fully persistent if all of them can be mod-
24 ified.

25 Coming back to control flow; for example, implementing backtracking in an
26 imperative language would mean undoing side effects which even PROLOG is

1 not able to do since the asserts and retracts cannot be undone. In HASKELL, a
2 monad defines a model for control flow and how side effects would propagate
3 through a computation from step to step or modification to modification. And
4 HASKELL allows creation of custom monads relieving the burden of dealing
5 with a fixed model of the host language.

6 (b) Lazy Evaluation

7 Another property of HASKELL is laziness or lazy evaluation which means that
8 nothing is evaluated until it is necessary. This results in the ability to define
9 infinite data structures because at execution only a fragment is used [149].

Chapter 12

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. Why a Logic Programming Language ?

2. 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*[152]. PROLOG is very popular and has a number of implementations [169] for different purposes.

3. Why embed PROLOG ?

(a) Existing Implementations

As a starting point a few publications and implementations helped in exploring the topic

(b) Simple Syntax

(c) Simple Semantics

1 (d) Universal Horn Clauses

2 (e) Unification

Chapter 13

Prototype 1

13.1 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.

13.2 How Prolog works ?

Looking at how PROLOG works [123].

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.

1 The possibilities could be,

2 1. If term1 and term2 are constants, then term1 and term2 unify if and only if they are
3 the same atom, or the same number.

```
1  ?-  =(mia,mia) .  
2  yes
```

4 2. If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and
5 term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type
6 of term, then term1 and term2 unify, and term2 is instantiated to term1 . (So if they
7 are both variables, theyre both instantiated to each other, and we say that they share
8 values.)

```
1  ?-  mia  =  X .  
2  X  =  mia  
3  yes
```

```
1  ?-  X  =  Y .  
2  yes
```

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

10 (a) They have the same functor and arity, and

11 (b) all their corresponding arguments unify, and

12 (c) the variable instantiations are compatible.

```
1  ?-  k(s(g),Y)  =  k(X,t(k)) .  
2  X  =  s(g)  
3  Y  =  t(k)  
4  yes
```

13 4. Two terms unify if and only if it follows from the previous three clauses that they
14 unify.

1 For example, consider the append function

```
1 append([],L,L).  
2 append([H|T],L2,[H|L3]) :- append(T,L2,L3).
```

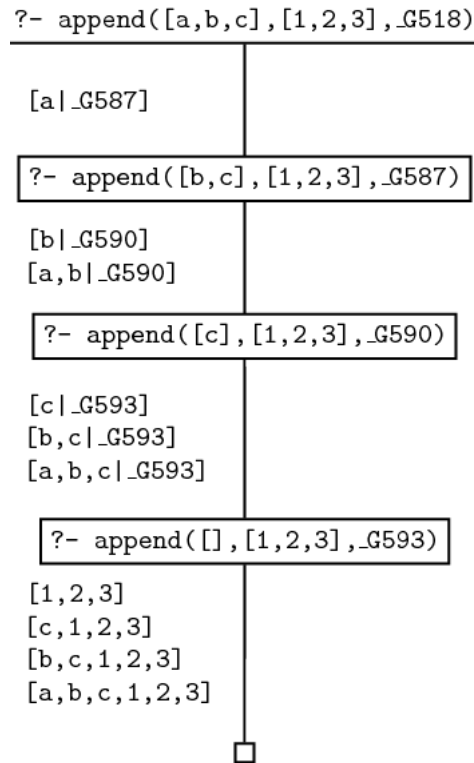


Figure 13.1: Trace for append [122]

2 13.3 What we do in this Prototype

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

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

7 The language defined is recursive in nature.

8 We convert it into a non recursive data type.

1 Basically we do Unification monadically.

2 **13.4 Creating a data type**

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

9 The advantages of static typing [82]

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

15 For dynamic typing

- 16 1. Less rigid
- 17 2. Ideal for prototyping / unknown / changing requirements or unpredictable behaviour
- 18 3. Re-usability

19 **Transitional paragraph** An ideal case would would be something that is dont
20 know what to write

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

4 Consider the language below,

```

1 data VariableName = VariableName Int String
2     deriving (Eq, Data, Typeable, Ord)
3 data Atom         = Atom         !String
4                   | Operator    !String
5     deriving (Eq, Ord, Data, Typeable)
6 data Term = Struct Atom [Term]
7           | Var VariableName
8           | Wildcard
9           | PString    !String
10          | PInteger   !Integer
11          | PFloat     !Double
12          | Flat [FlatItem]
13          | Cut Int
14     deriving (Eq, Data, Typeable)
15 data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
16              | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
17     deriving (Data, Typeable)
18 type Program = [Sentence]
19 type Body    = [Goal]
20 data Sentence = Query    Body
21              | Command Body
22              | C Clause
23     deriving (Data, Typeable)

```

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

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

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

```
Struct Atom [Term]
```

8 One of the issues with the above is that it is not possible to distinguish the structure of
 9 the data from the data type itself [113]. Consider the following, a reduced version of the
 10 above data type,

```

1 type Atom          = String
2 data VariableName = VariableName Int String
3     deriving (Eq, Data, Typeable, Ord)
4 data Term = Struct Atom [Term]
5           | Var VariableName
6           | Wildcard -- Don't cares
7           | Cut Int
8     deriving (Eq, Data, Typeable)

```

1 Also one cannot create Quantifiers plus logic
 2 To split a data type into two levels, a single recursive data type is replaced by two related
 3 data types. Consider the following,

```

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

```

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

```
FlatTerm Bool
```

7 and a generic fuinction like,

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

8 13.5 Working with the language

9 Creating instances,

```

1 instance Functor (FlatTerm) where
2     fmap = T.fmapDefault
3 instance Foldable (FlatTerm) where
4     foldMap = T.foldMapDefault
5 instance Traversable (FlatTerm) where
6     traverse f (Struct atom x) = Struct atom <$>
7                               sequenceA (Prelude.map f x)

```

```

8         traverse _ (Var v)           = pure (Var v)
9         traverse _ Wildcard          = pure (Wildcard)
10        traverse _ (Cut i)            = pure (Cut i)
11 instance Unifiable (FlatTerm) where
12     zipMatch (Struct al ls) (Struct ar rs) =
13         if (al == ar) && (length ls == length rs)
14             then Struct al <$>
15                 pairWith (\l r -> Right (l,r)) ls rs
16             else Nothing
17     zipMatch Wildcard _ = Just Wildcard
18     zipMatch _ Wildcard = Just Wildcard
19     zipMatch (Cut i1) (Cut i2) = if (i1 == i2)
20         then Just (Cut i1)
21         else Nothing
22 instance Applicative (FlatTerm) where
23     pure x = Struct "" [x]
24     _ <*> Wildcard          = Wildcard
25     _ <*> (Cut i)           = Cut i
26     _ <*> (Var v)           = (Var v)
27     (Struct a fs) <*> (Struct b xs) = Struct (a ++ b) [f x | f <- fs, x <- xs]

```

1 After flattening do fixing,

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

3 13.6 Black box

4 13.7 Something about unification-fd and Monadic Unifi- 5 cation

6 Library [128]

7 Tutorial 1 [129]

8 Tutorial 2 [130]

9 1. What library provides ?

10 This module provides first-order structural unification over general structure types.

1 It also provides the standard suite of functions accompanying unification (applying
2 bindings, getting free variables, etc.).

3 The implementation makes use of numerous optimization techniques. First, we use
4 path compression everywhere (for weighted path compression see `Control.Unification.Ranked`).
5 Second, we replace the occurs-check with visited-sets. Third, we use a technique for
6 aggressive opportunistic observable sharing; that is, we track as much sharing as
7 possible in the bindings (without introducing new variables), so that we can compare
8 bound variables directly and therefore eliminate redundant unifications.

9 2. Unifiable stuff

10 The basic class for generating, reading, and writing to bindings stored in a monad.
11 These three functionalities could be split apart, but are combined in order to simplify
12 contexts. Also, because most functions reading bindings will also perform path com-
13 pression, there's no way to distinguish "true" mutation from mere path compression.

14 The superclass constraints are there to simplify contexts, since we make the same
15 assumptions everywhere we use `BindingMonad`.

16 In order to use our `T` data type with the rest of the API, we'll need to give a Unifi-
17 able instance for it. Before we do that we'll have to give `Functor`, `Foldable`, and
18 `Traversable` instances. These are straightforward and can be automatically derived
19 with the appropriate language pragmas.

20 The `Unifiable` class gives one step of the unification process. Just as we only need
21 to specify one level of the ADT (i.e., `T`) and then we can use the library's `UTerm` to
22 generate the recursive ADT, so too we only need to specify one level of the unification
23 (i.e., `zipMatch`) and then we can use the library's operators to perform the recursive
24 unification, subsumption, etc.

25 The `zipMatch` function takes two arguments of type `t a`. The abstract `t` will be our
26 concrete `T` type. The abstract `a` is polymorphic, which ensures that we can't mess

1 around with more than one level of the term at once. If we abandon that guarantee,
2 then you can think of it as if a is $\text{UTerm } T \ v$. Thus, $t \ a$ means $T \ (\text{UTerm } T \ v)$; and
3 $T \ (\text{UTerm } T \ v)$ is essentially the type $\text{UTerm } T \ v$ with the added guarantee that the
4 values aren't in fact variables. Thus, the arguments to `zipMatch` are non-variable
5 terms.

6 The `zipMatch` method has the rather complicated return type: `Maybe (t (Either a`
7 `(a,a)))`. Let's unpack this a bit by thinking about how unification works. When we
8 try to unify two terms, first we look at their head constructors. If the constructors
9 are different, then the terms aren't unifiable, so we return `Nothing` to indicate that
10 unification has failed. Otherwise, the constructors match, so we have to recursively
11 unify their subterms. Since the T structures of the two terms match, we can return
12 `Just t0` where $t0$ has the same T structure as both input terms. Where we still have to
13 recursively unify subterms, we fill $t0$ with `Right(l,r)` values where l is a subterm of the
14 left argument to `zipMatch` and r is the corresponding subterm of the right argument.
15 Thus, `zipMatch` is a generalized zipping function for combining the shared structure
16 and pairing up substructures. And now, the implementation:

```
1 instance Unifiable T where
2     zipMatch (T m ls) (T n rs)
3         | m /= n      = Nothing
4         | otherwise =
5             T n <$> pairWith (\l r -> Right(l,r)) ls rs
```

17 Where `list-extras>Data.List.Extras.Pair.pairWith` is a version of `zip` which returns
18 `Nothing` if the lists have different lengths. So, if the names m and n match, and
19 if the two arguments have the same number of subterms, then we pair those subterms
20 off in order; otherwise, either the names or the lengths don't match, so we return
21 `Nothing`.

22 3. UTerm stuff

1 The type of terms generated by structures `t` over variables `v`. The structure type should
2 implement `Unifiable` and the variable type should implement `Variable`.

3 The `Show` instance doesn't show the constructors, in order to improve legibility for
4 large terms.

5 All the category theoretic instances (`Functor`, `Foldable`, `Traversable`,...) are provided
6 because they are often useful; however, beware that since the implementations must
7 be pure, they cannot read variables bound in the current context and therefore can
8 create incoherent results. Therefore, you should apply the current bindings before
9 using any of the functions provided by those classes.

10 4. `STVar` stuff

11 This module defines an implementation of unification variables using the `ST` monad.

12 5. `IntVar` stuff

13 This module defines a state monad for functional pointers represented by integers as
14 keys into an `IntMap`. This technique was independently discovered by Dijkstra et al.
15 This module extends the approach by using a state monad transformer, which can
16 be made into a backtracking state monad by setting the underlying monad to some
17 `MonadLogic` (part of the `logict` library, described by Kiselyov et al.).

18 Atze Dijkstra, Arie Middelkoop, S. Doaitse Swierstra (2008) Efficient Functional
19 Unification and Substitution, Technical Report UU-CS-2008-027, Utrecht Univer-
20 sity.

21 Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry (2005) Back-
22 tracking, Interleaving, and Terminating Monad Transformers, ICFP

23 A "mutable" unification variable implemented by an integer. This provides an en-
24 tirely pure alternative to truly mutable alternatives (like `STVar`), which can make
25 backtracking easier.

1 N.B., because this implementation is pure, we can use it for both ranked and unranked
2 monads.

3 6. Binding Monad Stuff

4 A monad for handling STVar bindings.

5 Run the ST ranked binding monad. N.B., because STVar are rank-2 quantified, this
6 guarantees that the return value has no such references. However, in order to remove
7 the references from terms, you'll need to explicitly apply the bindings and ground
8 the term.

9 7. U.unify stuff

10 Unify two terms, or throw an error with an explanation of why unification failed.
11 Since bindings are stored in the monad, the two input terms and the output term
12 are all equivalent if unification succeeds. However, the returned value makes use of
13 aggressive opportunistic observable sharing, so it will be more efficient to use it in
14 future calculations than either argument.

15 8. U.unifyOccurs

16 A variant of unify which uses occursIn instead of visited-sets. This should only
17 be used when eager throwing of occursFailure errors is absolutely essential (or for
18 testing the correctness of unify). Performing the occurs-check is expensive. Not only
19 is it slow, it's asymptotically slow since it can cause the same subterm to be traversed
20 multiple times.

¹ 9. Translation stuff

```

1  monadicUnification :: (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
2      ErrorT (UT.UFailure (FlatTerm) (ST.STVar s (FlatTerm)))
3          (ST.STBinding s) (UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm))),
4          Map VariableName (ST.STVar s (FlatTerm))))
5  monadicUnification t1 t2 = do
6      -- let
7      --     t1f = termFlattener t1
8      --     t2f = termFlattener t2
9      (x1,d1) <- lift . translateToUTerm $ t1
10     (x2,d2) <- lift . translateToUTerm $ t2
11     x3 <- U.unify x1 x2
12     --get state from somewhere, state -> dict
13     return $! (x3, d1 'Map.union' d2)
14
15
16  goUnify ::
17      (forall s. (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
18      =>
19          (ErrorT
20              (UT.UFailure FlatTerm (ST.STVar s FlatTerm))
21              (ST.STBinding s)
22              (UT.UTerm FlatTerm (ST.STVar s FlatTerm),
23              Map VariableName (ST.STVar s FlatTerm)))
24          )
25      -> [(VariableName, Prolog)]
26  goUnify test = ST.runSTBinding $ do
27      answer <- runErrorT $ test --ERROR
28      case answer of
29          (Left _)          -> return []
30          (Right (_, dict)) -> f1 dict
31
32
33  f1 ::
34      (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
35      => (forall s. Map VariableName (STVar s FlatTerm)
36          -> (ST.STBinding s [(VariableName, Prolog)])
37          )
38  f1 dict = do
39      let ld1 = Map.toList dict
40      ld2 <- Control.Monad.Error.sequence [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v
41      let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 'zip' ld2]
42          ld4 = [ (k,v) | (k,v2) <- ld3, let v = translateFromUTerm dict v2 ]
43      return ld4

```

Figure 13.2: A sample Minted figure

1 Chapter 14

2 Prototype 2.1

3 14.1 About this chapter

4 This chapter attempts to infuse the generic methodology from 13 in a current PROLOG
5 implementation [107] and make the unification "monadic".

6 14.2 How prolog-0.2.0.1 works

7 As described in the previous chapter about extending languages to incorporate functional-
8 ity, this prototype applies the procedure to the eDSL in [107].

9 The original abstract syntax used by the library,

```
1 data VariableName = VariableName Int String
2     deriving (Eq, Data, Typeable, Ord)
3
4 type Atom          = String
5
6 data Term = Struct Atom [Term]
7     | Var VariableName
8     | Wildcard -- Don't cares
9     | Cut Int
10     deriving (Eq, Data, Typeable)
11
12 data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
```

```

13         | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
14     deriving (Data, Typeable)
15
16 type Goal      = Term
17 type Program   = [Clause]

```

1 From the above we will focus on the *Term* since the others just add wrappers around
2 expressions which can be created by it. The above language suffers from most of the prob-
3 lems discussed in the previous chapter. The above is used to construct PROLOG "terms"
4 which are of a "single type".

5 The implementation consists of components that one would find in a Language Pro-
6 cessing System 14.1,

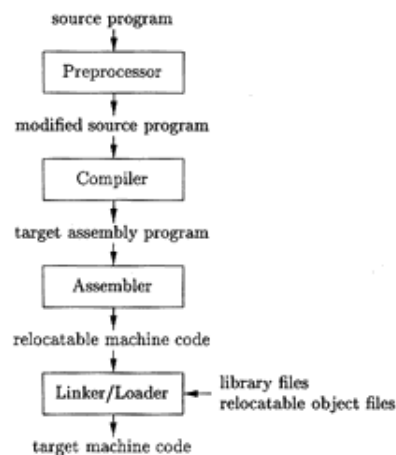


Figure 1.5: A language-processing system

Figure 14.1: A language-processing system [1]

7 specifically speaking, parts of a compiler 14.2,

8 The architecture for a compiler as described in 14.2 would not be needed since HASKELL
9 provides most of them. Nonetheless, the library has the following major components,

- 10 1. Syntax, defining the language.
- 11 2. Database, to create a storage for the expressions.

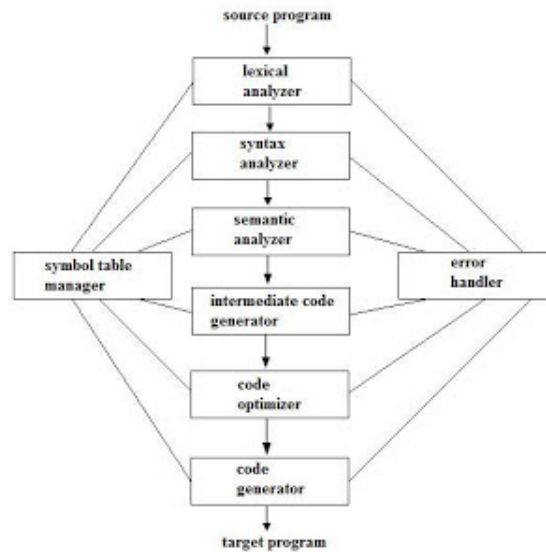


Fig 1.5 Phases of a compiler

Figure 14.2: Phases of Compiler [1]

1 3. Parser.

2 4. Interpreter.

3 5. Unifier.

4 6. REPL.

5 To prove the modularity of the approach for language modification and monadic unifi-
 6 cation only the abstract syntax and unifier will be customized.

7 14.3 What we do in this prototype?

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

9 We do complete PROLOG query resolution like stuff.

10 13 provides a generic procedure / methodology to convert a language into monadic
 11 unifiable form

1 14.4 Current implementation (prolog-0.2.0.1)

2 The current unification uses basic pattern matching to unify the terms

```
1 unify, unify_with_occurs_check :: MonadPlus m => Term -> Term
2   -> m Unifier
3
4 unify = fix unify'
5
6 unify_with_occurs_check =
7   fix $ \self t1 t2 -> if (t1 'occursIn' t2 || t2 'occursIn' t1)
8     then fail "occurs check"
9     else unify' self t1 t2
10  where
11    occursIn t = everything (||) (mkQ False (==t))
12
13 unify' :: MonadPlus m => (Term -> Term -> m Unifier) -> Term ->
14   Term -> m [(VariableName, Term)]
15
16 -- If either of the terms are don't cares then no unifiers exist
17 unify' _ Wildcard _ = return []
18 unify' _ _ Wildcard = return []
19
20 -- If one is a variable then equate the term to its value which
21 -- forms the unifier
22 unify' _ (Var v) t = return [(v,t)]
23 unify' _ t (Var v) = return [(v,t)]
24
25 -- Match the names and the length of their parameter list and
26 -- then match the elements of list one by one.
27 unify' self (Struct a1 ts1) (Struct a2 ts2)
28   | a1 == a2 && same length ts1 ts2 =
29     unifyList self (zip ts1 ts2)
30
31 unify' _ _ _ = mzero
32
33 same :: Eq b => (a -> b) -> a -> a -> Bool
34 same f x y = f x == f y
35
36 -- Match the elements of each of the tuples in the list.
37 unifyList :: Monad m => (Term -> Term -> m Unifier) ->
38   [(Term, Term)] -> m Unifier
39 unifyList _ [] = return []
40 unifyList unify ((x,y):xys) = do
```

```

41     u <- unify x y
42     u' <- unifyList unify (Prelude.map (both (apply u)) xys)
43     return (u++u')

```

1 14.5 Modifications

2 The resulting language is not far from what we did in 13 apart from the fact that the *Term*
3 expressions are encapsulated to form *Clauses* which in turn form a *Program*.

4 Moreover, the required instances make the language compatible with the unification
5 procedure.

```

1  data FTS a = FS Atom [a] | FV VariableName | FW | FC Int
2              deriving (Show, Eq, Typeable, Ord)
3
4  newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)
5
6  unP :: Prolog -> Fix FTS
7  unP (P x) = x
8
9  instance Functor (FTS) where
10     fmap          = T.fmapDefault
11
12  instance Foldable (FTS) where
13     foldMap        = T.foldMapDefault
14
15  instance Traversable (FTS) where
16     traverse f (FS atom xs)      = FS atom <$>
17     sequenceA (Prelude.map f xs)
18     traverse _ (FV v)            = pure (FV v)
19     traverse _ FW                = pure (FW)
20     traverse _ (FC i)            = pure (FC i)
21
22  instance Unifiable (FTS) where
23     zipMatch (FS al ls) (FS ar rs) =
24         if (al == ar) && (length ls == length rs)
25         then FS al <$> pairWith (\l r -> Right (l,r)) ls rs
26         else Nothing
27     zipMatch FW _ = Just FW
28     zipMatch _ FW = Just FW
29     zipMatch (FC i1) (FC i2) = if (i1 == i2)

```

```

30     then Just (FC i1)
31     else Nothing
32
33 instance Applicative (FTS) where
34     pure x                = FS "" [x]
35     _ <*> FW              = FW
36     _ <*> (FC i)          = FC i
37     _ <*> (FV v)          = (FV v)
38     (FS a fs) <*> (FS b xs) = FS (a ++ b) [f x | f <- fs, x <- xs]

```

1 Additionally helper functions for converting expressions between the two domains and
2 translation to *UTerm*.

```

1  termFlattener :: Term -> Fix FTS
2  termFlattener (Var v)           = DFF.Fix $ FV v
3  termFlattener (Wildcard)       = DFF.Fix FW
4  termFlattener (Cut i)          = DFF.Fix $ FC i
5  termFlattener (Struct a xs)    = DFF.Fix $ FS a (Prelude.map termFlattener xs)
6
7  unFlatten :: Fix FTS -> Term
8  unFlatten (DFF.Fix (FV v))     = Var v
9  unFlatten (DFF.Fix FW)         = Wildcard
10 unFlatten (DFF.Fix (FC i))      = Cut i
11 unFlatten (DFF.Fix (FS a xs))   = Struct a (Prelude.map unFlatten xs)
12
13
14 variableExtractor :: Fix FTS -> [Fix FTS]
15 variableExtractor (Fix x) = case x of
16     (FS _ xs)  -> Prelude.concat $ Prelude.map variableExtractor xs
17     (FV v)     -> [Fix $ FV v]
18     _         -> []
19
20 variableNameExtractor :: Fix FTS -> [VariableName]
21 variableNameExtractor (Fix x) = case x of
22     (FS _ xs) -> Prelude.concat $ Prelude.map variableNameExtractor xs
23     (FV v)    -> [v]
24     _        -> []
25
26 variableSet :: [Fix FTS] -> S.Set (Fix FTS)
27 variableSet a = S.fromList a
28
29 variableNameSet :: [VariableName] -> S.Set (VariableName)
30 variableNameSet a = S.fromList a
31
32 varsToDictM :: (Ord a, Unifiable t) =>

```

```

33     S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
34 varsToDictM set = foldrM addElt Map.empty set where
35     addElt sv dict = do
36         iv <- freeVar
37         return $! Map.insert sv iv dict
38
39
40 uTermify
41   :: Map VariableName (ST.STVar s (FTS))
42   -> UTerm FTS (ST.STVar s (FTS))
43   -> UTerm FTS (ST.STVar s (FTS))
44 uTermify varMap ux = case ux of
45     UT.UVar _           -> ux
46     UT.UTerm (FV v)     -> maybe (error "bad map") UT.UVar $ Map.lookup v varMap
47     -- UT.UTerm t        -> UT.UTerm $! fmap (uTermify varMap) t
48     UT.UTerm (FS a xs)  -> UT.UTerm $ FS a $! fmap (uTermify varMap) xs
49     UT.UTerm (FW)       -> UT.UTerm FW
50     UT.UTerm (FC i)     -> UT.UTerm (FC i)
51
52 translateToUTerm ::
53     Fix FTS -> ST.STBinding s
54     (UT.UTerm (FTS) (ST.STVar s (FTS)),
55      Map VariableName (ST.STVar s (FTS)))
56 translateToUTerm e1Term = do
57     let vs = variableNameSet $ variableNameExtractor e1Term
58     varMap <- varsToDictM vs
59     let t2 = uTermify varMap . unfreeze $ e1Term
60     return (t2, varMap)
61
62
63 -- / vTermify recursively converts @UVar x@ into @UTerm (VarA x).
64 -- This is a subroutine of @ translateFromUTerm @. The resulting
65 -- term has no (UVar x) subterms.
66
67 vTermify :: Map Int VariableName ->
68     UT.UTerm (FTS) (ST.STVar s (FTS)) ->
69     UT.UTerm (FTS) (ST.STVar s (FTS))
70 vTermify dict t1 = case t1 of
71     UT.UVar x -> maybe (error "logic") (UT.UTerm . FV) $ Map.lookup (UT.getVarID x)
72     UT.UTerm r ->
73         case r of
74             FV iv -> t1
75             _     -> UT.UTerm . fmap (vTermify dict) $ r
76
77 translateFromUTerm ::

```

```

78     Map VariableName (ST.STVar s (FTS)) ->
79     UT.UTerm (FTS) (ST.STVar s (FTS)) -> Prolog
80 translateFromUTerm dict uTerm =
81     P . maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm where
82     forKV dict initial fn = Map.foldlWithKey' (\a k v -> fn k v a) initial dict
83     varIdDict = forKV dict Map.empty $ \ k v -> Map.insert (UT.getVarID v) k
84
85
86 -- / Unify two (E1 a) terms resulting in maybe a dictionary
87 -- of variable bindings (to terms).
88 --
89 -- NB !!!!
90 -- The current interface assumes that the variables in t1 and t2 are
91 -- disjoint. This is likely a mistake that needs fixing
92
93 unifyTerms :: Fix FTS -> Fix FTS -> Maybe (Map VariableName (Prolog))
94 unifyTerms t1 t2 = ST.runSTBinding $ do
95     answer <- runExceptT $ unifyTermsX t1 t2
96     return $! either (const Nothing) Just answer
97
98 -- / Unify two (E1 a) terms resulting in maybe a dictionary
99 -- of variable bindings (to terms).
100 --
101 -- This routine works in the unification monad
102
103 unifyTermsX ::
104     (Fix FTS) -> (Fix FTS) ->
105     ExceptT (UT.UFailure (FTS) (ST.STVar s (FTS)))
106     (ST.STBinding s)
107     (Map VariableName (Prolog))
108 unifyTermsX t1 t2 = do
109     (x1,d1) <- lift . translateToUTerm $ t1
110     (x2,d2) <- lift . translateToUTerm $ t2
111     _ <- U.unify x1 x2
112     makeDicts $ (d1,d2)
113
114 mapWithKeyM :: (Ord k,Applicative m,Monad m)
115     => (k -> a -> m b) -> Map k a -> m (Map k b)
116 mapWithKeyM = Map.traverseWithKey
117
118
119 makeDict ::
120     Map VariableName (ST.STVar s (FTS)) -> ST.STBinding s (Map VariableName
121 makeDict sVarDict =
122     flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do

```

```

123         Just xx <- UT.lookupVar $ iKey
124         return $! (translateFromUTerm sVarDict) xx
125
126
127     -- / recover the bindings for the variables of the two terms
128     -- unified from the monad.
129
130 makeDicts ::
131     (Map VariableName (ST.STVar s (FTS)), Map VariableName (ST.STVar s (FTS))) ->
132     ExceptT (UT.UFailure (FTS) (ST.STVar s (FTS)))
133     (ST.STBinding s) (Map VariableName (Prolog))
134 makeDicts (svDict1, svDict2) = do
135     let svDict3 = (svDict1 `Map.union` svDict2)
136     let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
137     applyBindingsAll ivs
138     -- the interface below is dangerous because Map.union is left-biased.
139     -- variables that are duplicated across terms may have different
140     -- bindings because 'translateToUTerm' is run separately on each
141     -- term.
142     lift . makeDict $ svDict3

```

1 Take original expressions flatten fix convert unify run it STBinding monad to extract
2 substitutions.

```

1 monadicUnification :: (BindingMonad FTS (STVar s FTS)
2   (ST.STBinding s))
3 => (forall s. ((Fix FTS) -> (Fix FTS) ->
4   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
5     (ST.STBinding s) (UT.UTerm (FTS) (ST.STVar s (FTS)),
6     Map VariableName (ST.STVar s (FTS)))))
7 monadicUnification t1 t2 = do
8   -- let
9   --   t1f = termFlattener t1
10  --   t2f = termFlattener t2
11  (x1,d1) <- lift . translateToUTerm $ t1
12  (x2,d2) <- lift . translateToUTerm $ t2
13  x3 <- U.unify x1 x2
14  --get state from somewhere, state -> dict
15  return $! (x3, d1 `Map.union` d2)
16
17
18 goUnify ::
19   (forall s. (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
20   =>

```

```

21         (ErrorT
22           (UT.UFailure FTS (ST.STVar s FTS))
23           (ST.STBinding s)
24           (UT.UTerm FTS (ST.STVar s FTS),
25             Map VariableName (ST.STVar s FTS)))
26       )
27   -> [(VariableName, Prolog)]
28 goUnify test = ST.runSTBinding $ do
29   answer <- runErrorT $ test --ERROR
30   case answer of
31     (Left _)           -> return []
32     (Right (_, dict)) -> f1 dict
33
34
35 f1 ::
36   (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
37   => (forall s. Map VariableName (STVar s FTS)
38     -> (ST.STBinding s [(VariableName, Prolog)]))
39   )
40 f1 dict = do
41   let ld1 = Map.toList dict
42   ld2 <- Control.Monad.Error.sequence
43   [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v]
44   let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 'zip' ld2]
45       ld4 = [ (k,v) | (k,v2) <- ld3,
46         let v = translateFromUTerm dict v2 ]
47   return ld4
48 unifierConvertor :: [(VariableName, Prolog)] -> Unifier
49 unifierConvertor xs = Prelude.map (\(v, p) -> (v, (unFlatten $ unP $ p))) xs
50
51 unify :: MonadPlus m => Term -> Term -> m Unifier
52 unify t1 t2 = unifierConvertor (goUnify (monadicUnification (termFlattener t1) (te

```

1 14.6 Results

2 It works,

Chapter 15

Prototype 3

When two terms are to be unified we can use 13 ,
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

15.1 Unification

The first, "unification," regards how terms are matched and variables assigned to make
terms match. [38]

15.2 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 goals and then tries to achieve each one.

1 [37]

2 15.3 Search strategies

3 The base implementation used for this prototype is [65] and below are the search strategies

4 15.4 Stack Engine

```
1  -- Stack based Prolog inference engine
2  -- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
3  -- and for Hugs 1.3 June 1996.
4  --
5  -- Suitable for use with Hugs 98.
6  --
7
8  module StackEngine( version, prove ) where
9
10 import Prolog
11 import st
12 import Interact
13
14 version = "stack based"
15
16 --- Calculation of solutions:
17
18 -- the stack based engine maintains a stack of triples (s,goal,alts)
19 -- corresponding to backtrack points, where s is the stitution at that
20 -- point, goal is the outstanding goal and alts is a list of possible ways
21 -- of extending the current proof to find a solution. Each member of alts
22 -- is a pair (tp,u) where tp is a new goal that must be proved and u is
23 -- a unifying stitution that must be combined with the stitution s.
24 --
25 -- the list of relevant clauses at each step in the execution is produced
26 -- by attempting to unify the head of the current goal with a suitably
27 -- renamed clause from the database.
28
29 type Stack = [ (st, [Term], [Alt]) ]
30 type Alt   = ([Term], st)
31
32 alts      :: Database -> Int -> Term -> [Alt]
```

```

33  alts db n g = [ (tp,u) | (tm:-tp) <- renClauses db n g, u <- unify g tm ]
34
35  -- The use of a stack enables backtracking to be described explicitly,
36  -- in the following 'state-based' definition of prove:
37
38  prove      :: Database -> [Term] -> [st]
39  prove db gl = solve 1 nullst gl []
40  where
41    solve :: Int -> st -> [Term] -> Stack -> [st]
42    solve n s []      ow      = s : backtrack n ow
43    solve n s (g:gs) ow
44        | g==theCut = solve n s gs (cut ow)
45        | otherwise = choose n s gs (alts db n (app s g)) ow
46
47    choose :: Int -> st -> [Term] -> [Alt] -> Stack -> [st]
48    choose n s gs []      ow = backtrack n ow
49    choose n s gs ((tp,u):rs) ow = solve (n+1) (u@@s) (tp++gs) ((s,gs,rs):ow)
50
51    backtrack      :: Int -> Stack -> [st]
52    backtrack n []      = []
53    backtrack n ((s,gs,rs):ow) = choose (n-1) s gs rs ow
54
55
56  --- Special definitions for the cut predicate:
57
58  theCut      :: Term
59  theCut      = Struct "!" []
60
61  cut          :: Stack -> Stack
62  cut ss      = []
63
64  --- End of Engine.hs

```

1 15.5 Pure Engine

```

1  -- The Pure Prolog inference engine (using explicit proof trees)
2  -- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
3  -- and for Hugs 1.3 June 1996.
4  --
5  -- Suitable for use with Hugs 98.
6  --
7

```

```

8  module PureEngine( version, prove ) where
9
10 import Prolog
11 import st
12 import Interact
13 import Data.List(nub)
14
15 version = "tree based"
16
17 --- Calculation of solutions:
18
19 -- Each node in a proof tree corresponds to:
20 -- either: a solution to the current goal, represented by Done s, where s
21 --          is the required stitution
22 -- or:      a choice between a number of trees ts, each corresponding to a
23 --          proof of a goal of the current goal, represented by Choice ts.
24 --          The proof tree corresponding to an unsolvable goal is Choice []
25
26 data ProofTree = Done st | Choice [ProofTree]
27
28 -- proof tree uses the rules of Prolog to construct a suitable proof tree for
29 -- a specified goal
30 proofTree :: Database -> Int -> st -> [Term] -> ProofTree
31 proofTree db = pt
32   where pt :: Int -> st -> [Term] -> ProofTree
33         pt n s [] = Done s
34         pt n s (g:gs) = Choice [ pt (n+1) (u@@s) (map (app u) (tp++gs))
35                                | (tm:-tp)<-renClauses db n g, u<-unify g tm ]
36   {--
37   pt 1 nullst [] = Done (nullst)
38
39   pt n s (g:gs)
40
41   renClauses :- Rename variables in a clause, the parameters are the database, an
42                  (head of list) resulting in a clause.
43
44   unify :- take the head of the list and and match with head of clause from renCla
45
46   app :- function for applying (st) to (Terms)
47   the new list is formed by replacing the cluase head with its body and applying t
48
49   so the new parameters for pt are
50
51   (n+1) (the old stitution + the new one from unify) (the list formed after applyi
52

```

```

53
54 Working of a small example
55
56 The database,
57 (foldl addClause emptyDb [((:-) (Struct "hello" []) []), ((:-) (Struct "hello" [
58 hello.
59 hello(world).
60 hello:-world.
61 hello(X_1).
62
63 The other parameters are 1 nullst(as mentioned in the prove function).
64
65 For the list of goals, [(Struct "hello" []), (Struct "hello" [(Struct "world" [
66
67 1. [Struct "hello" []] :: [Term]
68
69 * Rule 1 does not apply
70
71 * Rule 2 does apply,
72
73 (tm:- tp) <- renClauses db 1 (Struct "hello" [])
74
75 tm ==> "hello , hello(world) , hello , hello(X_1) , "
76 tp ==> "[] , [] , [world] , [] , "
77
78
79
80
81
82
83
84
85
86 --}
87
88
89
90 -- DFS Function
91 -- search performs a depth-first search of a proof tree, producing the list
92 -- of solution stitutions as they are encountered.
93 search :: ProofTree -> [st]
94 search (Done s) = [s]
95 search (Choice pts) = [ s | pt <- pts, s <- search pt ]
96
97

```

```

98 prove    :: Database -> [Term] -> [st]
99 prove db  = search . prooftree db 1 nullst
100
101 --- End of PureEngine.hs

```

1 15.6 Andorra Engine

```

1  {-
2  By Donald A. Smith, December 22, 1994, based on Mark Jones' PureEngine.
3
4  This inference engine implements a variation of the Andorra Principle for
5  logic programming. (See references at the end of this file.) The basic
6  idea is that instead of always selecting the first goal in the current
7  list of goals, select a relatively deterministic goal.
8
9  For each goal g in the list of goals, calculate the resolvents that would
10 result from selecting g. Then choose a g which results in the lowest
11 number of resolvents. If some g results in 0 resolvents then fail.
12 (This would occur for a goal like: ?- append(A,B,[1,2,3]),equals(1,2).)
13 Prolog would not perform this optimization and would instead search
14 and backtrack wastefully. If some g results in a single resolvent
15 (i.e., only a single clause matches) then that g will get selected;
16 by selecting and resolving g, bindings are propagated sooner, and useless
17 search can be avoided, since these bindings may prune away choices for
18 other clauses. For example: ?- append(A,B,[1,2,3]),B=[].
19 -}
20
21 module AndorraEngine( version, prove ) where
22
23 import Prolog
24 import st
25 import Interact
26
27 version = "Andorra Principle Interpreter (select deterministic goals first)"
28
29 solve    :: Database -> Int -> st -> [Term] -> [st]
30 solve db = slv where
31     slv    :: Int -> st -> [Term] -> [st]
32     slv n s [] = [s]
33     slv n s goals =
34         let allResolvents = resolve_selecting_each_goal goals db n in
35         let (gs,gres) = findMostDeterministic allResolvents in

```

```

36         concat [slv (n+1) (u@@s) (map (app u) (tp++gs)) | (u,tp) <- gres]
37
38 resolve_selecting_each_goal::
39     [Term] -> Database -> Int -> [[Term],[[st,[Term]]]]
40 -- For each pair in the list that we return, the first element of the
41 -- pair is the list of unresolved goals; the second element is the list
42 -- of resolvents of the selected goal, where a resolvent is a pair
43 -- consisting of a stitution and a list of new goals.
44 resolve_selecting_each_goal goals db n = [(gs, gResolvents) |
45     (g,gs) <- delete goals, let gResolvents = resolve db g n]
46
47 -- The unselected goals from above are not passed in.
48 resolve :: Database -> Term -> Int -> [(st,[Term])]
49 resolve db g n = [(u,tp) | (tm:-tp)<-renClauses db n g, u<-unify g tm]
50 -- u is not yet applied to tp, since it is possible that g won't be selected.
51 -- Note that unify could be nondeterministic.
52
53 findMostDeterministic:: [[Term],[[st,[Term]]]] -> ([Term],[[st,[Term]]])
54 findMostDeterministic allResolvents = minF comp allResolvents where
55     comp:: (a,[b]) -> (a,[b]) -> Bool
56     comp (_,gs1) (_,gs2) = (length gs1) < (length gs2)
57 -- It seems to me that there is an opportunity for a clever compiler to
58 -- optimize this code a lot. In particular, there should be no need to
59 -- determine the total length of a goal list if it is known that
60 -- there is a shorter goal list in allResolvents ... ?
61
62 delete :: [a] -> [(a,[a])]
63 delete l = d l [] where
64     d :: [a] -> [a] -> [(a,[a])]
65     d [g]sofar = [ (g,sofar) ]
66     d (g:gs)sofar = (g,sofar++gs) : (d gs (g:sofar))
67
68 minF :: (a -> a -> Bool) -> [a] -> a
69 minF f (h:t) = m h t where
70 -- m :: a -> [a] -> a
71     m sofar [] = sofar
72     m sofar (h:t) = if (f h sofar) then m h t else m sofar t
73
74 prove :: Database -> [Term] -> [st]
75 prove db = solve db 1 nullst
76
77 {- An optimized, incremental version of the above interpreter would use
78    a data representation in which for each goal in "goals" we carry around
79    the list of resolvents. After each resolution step we update the lists.
80 -}

```

```

81
82 {- References
83
84   Seif Haridi & Per Brand, "Andorra Prolog, an integration of Prolog
85   and committed choice languages" in Proceedings of FGCS 1988, ICOT,
86   Tokyo, 1988.
87
88   Vitor Santos Costa, David H. D. Warren, and Rong Yang, "Two papers on
89   the Andorra-I engine and preprocessor", in Proceedings of the 8th
90   ICLP. MIT Press, 1991.
91
92   Steve Gregory and Rong Yang, "Parallel Constraint Solving in
93   Andorra-I", in Proceedings of FGCS'92. ICOT, Tokyo, 1992.
94
95   Sverker Janson and Seif Haridi, "Programming Paradigms of the Andorra
96   Kernel Language", in Proceedings of ILPS'91. MIT Press, 1991.
97
98   Torkel Franzen, Seif Haridi, and Sverker Janson, "An Overview of the
99   Andorra Kernel Language", In LNAI (LNCS) 596, Springer-Verlag, 1992.
100 -}

```

1 15.7 Current Unification

```

1  {-# LANGUAGE DeriveDataTypeable,
2      ViewPatterns,
3      ScopedTypeVariables,
4      DefaultSignatures,
5      TypeOperators,
6      TypeFamilies,
7      DataKinds,
8      DataKinds,
9      PolyKinds,
10     OverlappingInstances,
11     TypeOperators,
12     LiberalTypeSynonyms,
13     TemplateHaskell,
14     AllowAmbiguousTypes,
15     ConstraintKinds,
16     Rank2Types,
17     MultiParamTypeClasses,
18     FunctionalDependencies,
19     FlexibleContexts,

```

```

20         FlexibleInstances,
21         UndecidableInstances
22     #-}
23
24 -- Institutions and Unification of Prolog Terms
25 -- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
26 -- and for Hugs 1.3 June 1996.
27 --
28 -- Suitable for use with Hugs 98.
29 --
30
31 module st where
32
33 import Prolog
34 import CustomSyntax
35 import Data.Map as Map
36 import Data.Maybe
37 import Data.Either
38
39 --Unification
40 import Control.Unification.IntVar
41 import Control.Unification.STVar as ST
42
43 import Control.Unification.Ranked.IntVar
44 import Control.Unification.Ranked.STVar
45
46 import Control.Unification.Types as UT
47
48 import Control.Monad.State.UnificationExtras
49 import Control.Unification as U
50
51 -- Monads
52 import Control.Monad.Error
53 import Control.Monad.Trans.Except
54
55 import Data.Functor.Fixedpoint as DFF
56
57 --State
58 import Control.Monad.State.Lazy
59 import Control.Monad.ST
60 import Control.Monad.Trans.State as Trans
61
62 infixr 3 @@
63 infix 4 ->-
64

```



```

65  --- substitutions:
66
67  type st = Id -> Term
68
69  newtype stP = stP { unstP :: st }
70
71  -- instance Show stP where
72  --   show (i) = show £ Var i
73  -- substitutions are represented by functions mapping identifiers to terms.
74  --
75  -- app s      extends the stitution s to a function mapping terms to terms
76  {--
77  Looks like an apply function that applies a stitution function tho the variables
78  --}
79
80
81  -- nullst is the empty stitution which maps every identifier to the same identifi
82
83
84
85  -- i ->- t    is the stitution which maps the identifier i to the term t, but oth
86
87
88  -- s1@@ s2    is the composition of stitutions s1 and s2
89  --           N.B. app is a monoid homomorphism from (st,nullst,(@@))
90  --           to (Term -> Term, id, (.) ) in the sense that:
91  --           app (s1 @@ s2) = app s1 . app s2
92  --           s @@ nullst = s = nullst @@ s
93
94  app                :: st -> Term -> Term
95  app s (Var i)      = s i
96  app s (Struct a ts) = Struct a (Prelude.map (app s) ts)
97  {--
98  app (stFunction) (Struct "hello" [Var (0, "Var")])
99  hello(Var_2) :: Term
100
101  --}
102
103
104  nullst              :: st
105  nullst i            = Var i
106  {--
107  nullst (0, "Var")
108  Var :: Term
109  --}

```

```

110
111
112  --
113  (->-) :: Id -> Term -> st
114  (i ->- t) j | j==i      = t
115              | otherwise = Var j
116  {--
117  :t (->-) (1,"X") (Struct "hello" [])
118  (1,"X") ->- Struct "hello" [] :: (Int,[Char]) -> Term
119  --}
120
121
122  -- Function composition for applying two stitution functions.
123  (@@) :: st -> st -> st
124  s1 @@ s2      = app s1 . s2

```

1 15.8 Syntax Modification

```

1  {-# LANGUAGE DeriveDataTypeable,
2      ViewPatterns,
3      ScopedTypeVariables,
4      FlexibleInstances,
5      DefaultSignatures,
6      TypeOperators,
7      FlexibleContexts,
8      TypeFamilies,
9      DataKinds,
10     OverlappingInstances,
11     DataKinds,
12     PolyKinds,
13     TypeOperators,
14     LiberalTypeSynonyms,
15     TemplateHaskell,
16     RankNTypes,
17     AllowAmbiguousTypes,
18     MultiParamTypeClasses,
19     FunctionalDependencies,
20     ConstraintKinds,
21     ExistentialQuantification
22     #-}
23
24  module CustomSyntax where

```

```

25
26 import Data.Generics (Data(..), Typeable(..))
27 import Data.List (intercalate)
28 import Data.Char (isLetter)
29
30 import Control.Monad.State.UnificationExtras
31 import Control.Unification as U
32
33
34 import Data.Functor.Fixedpoint as DFF
35
36
37 import Control.Unification.IntVar
38 import Control.Unification.STVar as ST
39
40 import Control.Unification.Ranked.IntVar
41 import Control.Unification.Ranked.STVar
42
43 import Control.Unification.Types as UT
44
45
46
47 import Data.Traversable as T
48 import Data.Functor
49 import Data.Foldable
50 import Control.Applicative
51
52
53 import Data.List.Extras.Pair
54 import Data.Map as Map
55 import Data.Set as S
56
57
58 import Control.Monad.Error
59 import Control.Monad.Trans.Except
60
61
62 import Prolog
63
64 data FTS a = forall a . FV Id | FS Atom [a] deriving (Eq, Show, Ord, Typeable)
65
66 newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)
67
68 unP :: Prolog -> Fix FTS
69 unP (P x) = x

```

```

70
71 instance Functor FTS where
72     fmap = T.fmapDefault
73
74 instance Foldable FTS where
75     foldMap = T.foldMapDefault
76
77 instance Traversable FTS where
78     traverse f (FS atom xs) = FS atom <$> sequenceA (Prelude.map f xs)
79     traverse _ (FV v) = pure (FV v)
80
81 instance Unifiable FTS where
82     zipMatch (FS al ls) (FS ar rs) = if (al == ar) && (length ls == length rs)
83                                     then FS al <$> pairWith (\l r -> Right (l,r))
84                                     else Nothing
85     zipMatch (FV v1) (FV v2) = if (v1 == v2) then Just (FV v1)
86                               else Nothing
87     zipMatch _ _ = Nothing
88
89 instance Applicative FTS where
90     pure x = FS "" [x]
91     (FS a fs) <*> (FS b xs) = FS (a ++ b) [f x | f <- fs, x <- xs]
92     --other cases
93     {--
94     instance Monad FTS where
95         func =
96     instance Variable FTS where
97         func =
98
99     instance BindingMonad FTS where
100         func =
101     --}
102
103 data VariableName = VariableName Int String
104
105 idToVariableName :: Id -> VariableName
106 idToVariableName (i, s) = VariableName i s
107
108 variablenameToId :: VariableName -> Id
109 variablenameToId (VariableName i s) = (i,s)
110
111 termFlattener :: Term -> Fix FTS
112 termFlattener (Var v) = DFF.Fix $ FV v
113 termFlattener (Struct a xs) = DFF.Fix $ FS a (Prelude.map termFlattener xs)
114

```

```

115 unFlatten :: Fix FTS -> Term
116 unFlatten (DFF.Fix (FV v))      = Var v
117 unFlatten (DFF.Fix (FS a xs))   = Struct a (Prelude.map unFlatten xs)
118
119
120 variableExtractor :: Fix FTS -> [Fix FTS]
121 variableExtractor (Fix x) = case x of
122   (FS _ xs)   -> Prelude.concat $ Prelude.map variableExtractor xs
123   (FV v)      -> [Fix $ FV v]
124   -- _        -> []
125
126 variableIdExtractor :: Fix FTS -> [Id]
127 variableIdExtractor (Fix x) = case x of
128   (FS _ xs) -> Prelude.concat $ Prelude.map variableIdExtractor xs
129   (FV v)    -> [v]
130
131 {--
132   variableNameExtractor :: Fix FTS -> [VariableName]
133   variableNameExtractor (Fix x) = case x of
134     (FS _ xs) -> Prelude.concat & Prelude.map variableNameExtractor xs
135     (FV v)    -> [v]
136     _        -> []
137   --}
138
139 variableSet :: [Fix FTS] -> S.Set (Fix FTS)
140 variableSet a = S.fromList a
141
142 variableNameSet :: [Id] -> S.Set (Id)
143 variableNameSet a = S.fromList a
144
145
146 varsToDictM :: (Ord a, Unifiable t) =>
147   S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
148 varsToDictM set = foldrM addElt Map.empty set where
149   addElt sv dict = do
150     iv <- freeVar
151     return $! Map.insert sv iv dict
152
153
154 uTermify
155   :: Map Id (ST.STVar s (FTS))
156   -> UTerm FTS (ST.STVar s (FTS))
157   -> UTerm FTS (ST.STVar s (FTS))
158 uTermify varMap ux = case ux of
159   UT.UVar _      -> ux

```

```

160   UT.UTerm (FV v)          -> maybe (error "bad map") UT.UVar $ Map.lookup v varMap
161   -- UT.UTerm t            -> UT.UTerm £! fmap (uTermify varMap) t
162   UT.UTerm (FS a xs)      -> UT.UTerm $ FS a $! fmap (uTermify varMap) xs
163
164
165   translateToUTerm ::
166       Fix FTS -> ST.STBinding s
167       (UT.UTerm (FTS) (ST.STVar s (FTS)),
168        Map Id (ST.STVar s (FTS)))
169   translateToUTerm e1Term = do
170       let vs = variableNameSet $ variableIdExtractor e1Term
171       varMap <- varsToDictM vs
172       let t2 = uTermify varMap . unfreeze $ e1Term
173       return (t2,varMap)
174
175
176   -- / vTermify recursively converts @UVar x@ into @UTerm (VarA x).
177   -- This is a routine of @ translateFromUTerm @. The resulting
178   -- term has no (UVar x) terms.
179
180   vTermify :: Map Int Id ->
181       UT.UTerm (FTS) (ST.STVar s (FTS)) ->
182       UT.UTerm (FTS) (ST.STVar s (FTS))
183   vTermify dict t1 = case t1 of
184       UT.UVar x -> maybe (error "logic") (UT.UTerm . FV) $ Map.lookup (UT.getVarID x)
185       UT.UTerm r ->
186           case r of
187               FV iv -> t1
188               _ -> UT.UTerm . fmap (vTermify dict) $ r
189
190   translateFromUTerm ::
191       Map Id (ST.STVar s (FTS)) ->
192       UT.UTerm (FTS) (ST.STVar s (FTS)) -> Prolog
193   translateFromUTerm dict uTerm =
194       P . maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm where
195       forKV dict initial fn = Map.foldlWithKey' (\a k v -> fn k v a) initial dict
196       varIdDict = forKV dict Map.empty $ \ k v -> Map.insert (UT.getVarID v) k
197
198
199   -- / Unify two (E1 a) terms resulting in maybe a dictionary
200   -- of variable bindings (to terms).
201   --
202   -- NB !!!!
203   -- The current interface assumes that the variables in t1 and t2 are
204   -- disjoint. This is likely a mistake that needs fixing

```

```

205
206 unifyTerms :: Fix FTS -> Fix FTS -> Maybe (Map Id (Prolog))
207 unifyTerms t1 t2 = ST.runSTBinding $ do
208   answer <- runExceptT $ unifyTermsX t1 t2
209   return $! either (const Nothing) Just answer
210
211 -- / Unify two (E1 a) terms resulting in maybe a dictionary
212 -- of variable bindings (to terms).
213 --
214 -- This routine works in the unification monad
215
216 unifyTermsX ::
217   Fix FTS -> Fix FTS ->
218   ExceptT (UT.UFailure (FTS) (ST.STVar s (FTS)))
219     (ST.STBinding s)
220     (Map Id (Prolog))
221 unifyTermsX t1 t2 = do
222   (x1,d1) <- lift . translateToUTerm $ t1
223   (x2,d2) <- lift . translateToUTerm $ t2
224   _ <- unify x1 x2
225   makeDicts $ (d1,d2)
226
227
228
229 mapWithKeyM :: (Ord k,Applicative m,Monad m)
230              => (k -> a -> m b) -> Map k a -> m (Map k b)
231 mapWithKeyM = Map.traverseWithKey
232
233
234 makeDict ::
235   Map Id (ST.STVar s (FTS)) -> ST.STBinding s (Map Id (Prolog))
236 makeDict sVarDict =
237   flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do
238     Just xx <- UT.lookupVar $ iKey
239     return $! (translateFromUTerm sVarDict) xx
240
241
242 -- / recover the bindings for the variables of the two terms
243 -- unified from the monad.
244
245 makeDicts ::
246   (Map Id (ST.STVar s (FTS)), Map Id (ST.STVar s (FTS))) ->
247   ExceptT (UT.UFailure (FTS) (ST.STVar s (FTS)))
248     (ST.STBinding s) (Map Id (Prolog))
249 makeDicts (svDict1, svDict2) = do

```

```

250   let svDict3 = (svDict1 'Map.union' svDict2)
251   let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
252   applyBindingsAll ivs
253   -- the interface below is dangerous because Map.union is left-biased.
254   -- variables that are duplicated across terms may have different
255   -- bindings because 'translateToUTerm' is run separately on each
256   -- term.
257   lift . makeDict $ svDict3
258
259   instance (UT.Variable v, Functor t) => Error (UT.UFailure t v) where {}
260
261   test1 ::
262     ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
263       (ST.STBinding s)
264       (UT.UTerm (FTS) (ST.STVar s (FTS)),
265        Map Id (ST.STVar s (FTS)))
266   test1 = do
267     let
268       t1a = (Fix $ FV $ (0, "x"))
269       t2a = (Fix $ FV $ (1, "y"))
270       (x1,d1) <- lift . translateToUTerm $ t1a --error
271       (x2,d2) <- lift . translateToUTerm $ t2a
272       x3 <- U.unify x1 x2
273       return (x3, d1 'Map.union' d2)
274
275
276   test2 ::
277     ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
278       (ST.STBinding s)
279       (UT.UTerm (FTS) (ST.STVar s (FTS)),
280        Map Id (ST.STVar s (FTS)))
281   test2 = do
282     let
283       t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
284       t2a = (Fix $ FV $ (1, "y"))
285       (x1,d1) <- lift . translateToUTerm $ t1a --error
286       (x2,d2) <- lift . translateToUTerm $ t2a
287       x3 <- U.unify x1 x2
288       return (x3, d1 'Map.union' d2)
289
290
291   test3 ::
292     ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
293       (ST.STBinding s)
294       (UT.UTerm (FTS) (ST.STVar s (FTS)),

```



```

295         Map Id (ST.STVar s (FTS)))
296 test3 = do
297     let
298         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
299         t2a = (Fix $ FV $ (0, "x"))
300         (x1,d1) <- lift . translateToUTerm $ t1a --error
301         (x2,d2) <- lift . translateToUTerm $ t2a
302         x3 <- U.unify x1 x2
303         return (x3, d1 'Map.union' d2)
304     {--
305     goTest test3
306     "ok:      STVar -9223372036854775807
307     [(VariableName 0 \"x\",STVar -9223372036854775808)]"
308     --}
309
310 test4 ::
311     ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
312           (ST.STBinding s)
313           (UT.UTerm (FTS) (ST.STVar s (FTS))),
314           Map Id (ST.STVar s (FTS)))
315 test4 = do
316     let
317         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
318         t2a = (Fix $ FV $ (0, "x"))
319         (x1,d1) <- lift . translateToUTerm $ t1a --error
320         (x2,d2) <- lift . translateToUTerm $ t2a
321         x3 <- U.unifyOccurs x1 x2
322         return (x3, d1 'Map.union' d2)
323     {--
324     goTest test4
325     "ok:      STVar -9223372036854775807
326     [(VariableName 0 \"x\",STVar -9223372036854775808)]"
327     --}
328
329 test5 ::
330     ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
331           (ST.STBinding s)
332           (UT.UTerm (FTS) (ST.STVar s (FTS))),
333           Map Id (ST.STVar s (FTS)))
334 test5 = do
335     let
336         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
337         t2a = (Fix $ FS "b" [Fix $ FV $ (0, "y")])
338         (x1,d1) <- lift . translateToUTerm $ t1a --error
339         (x2,d2) <- lift . translateToUTerm $ t2a

```

```

340     x3 <- U.unify x1 x2
341     return (x3, d1 'Map.union' d2)
342
343 goTest :: (Show b) => (forall s .
344     (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
345       (ST.STBinding s)
346       (UT.UTerm (FTS) (ST.STVar s (FTS)),
347         Map Id (ST.STVar s (FTS)))))) -> String
348 goTest test = ST.runSTBinding $ do
349     answer <- runErrorT $ test
350     return $! case answer of
351         (Left x)  -> "error: " ++ show x
352         (Right y) -> "ok:    " ++ show y
353
354
355 -----
356 -----
357 -----GLUE-CODE-----
358 {--
359 monadicUnify :: Term -> Term -> ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
360             (ST.STBinding s)
361             (UT.UTerm (FTS) (ST.STVar s (FTS)),
362             Map Id (ST.STVar s (FTS)))
363 monadicUnify t1 t2 = do
364     let
365         t1f = termFlattener t1
366         t2f = termFlattener t2
367         (x1,d1) <- lift . translateToUTerm £ t1f
368         (x2,d2) <- lift . translateToUTerm £ t2f
369         x3 <- U.unify x1 x2
370         return (x3, d1 'Map.union' d2)
371
372 --}
373
374 -- type st = Id -> Term
375
376 -- Convert result from monadicUnify to [st]
377 {--
378 goMonadicTest :: (Show b) => (forall s .
379     (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
380       (ST.STBinding s)
381       (UT.UTerm (FTS) (ST.STVar s (FTS)),
382         Map Id (ST.STVar s (FTS)))))) -> [st]
383 goMonadicTest test = ST.runSTBinding £ do
384     answer <- runErrorT £ test

```

```

385     return £! case answer of
386       (Left x)  -> [nullst]
387       (Right y) -> convertTost y
388   --}
389
390   --(Id, STVar s FTS)
391   --convertTost :: Map Id (ST.STVar s FTS) -> [(Id, ST.STVar s FTS)]
392   {--
393   convertTost m = convertTost1 Map.toAscList m
394
395   convertTost1 (id, ST.STVar _ fts):xs = (id, (unFlatten fts)) : convertTost1 xs
396   --}

```

1 15.9 Monadic Unification

```

1  monadicUnification :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s)) => (forall
2      (ST.STBinding s) (UT.UTerm (FTS) (ST.STVar s (FTS)),
3      Map Id (ST.STVar s (FTS))))
4  monadicUnification t1 t2 = do
5      let
6          t1f = termFlattener t1
7          t2f = termFlattener t2
8          (x1,d1) <- lift . translateToUTerm $ t1f
9          (x2,d2) <- lift . translateToUTerm $ t2f
10         x3 <- U.unify x1 x2
11         --get state from somewhere, state -> dict
12         return $! (x3, d1 'Map.union' d2)
13
14
15  goUnify ::
16      (forall s. (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
17      =>
18          (ErrorT
19              (UT.UFailure FTS (ST.STVar s FTS))
20              (ST.STBinding s)
21              (UT.UTerm FTS (ST.STVar s FTS),
22              Map Id (ST.STVar s FTS)))
23          )
24      -> [(Id, Prolog)]
25  goUnify test = ST.runSTBinding $ do
26      answer <- runErrorT $ test --ERROR
27      case answer of

```

```

28     (Left _)          -> return []
29     (Right (_, dict)) -> f1 dict
30
31
32 f1 ::
33   (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
34   => (forall s. Map Id (STVar s FTS)
35       -> (ST.STBinding s [(Id, Prolog)]))
36       )
37 f1 dict = do
38   let ld1 = Map.toList dict
39   ld2 <- sequence [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v]
40   let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 'zip' ld2]
41   ld4 = [ (k,v) | (k,v2) <- ld3, let v = translateFromUTerm dict v2 ]
42   return ld4
43
44
45 --unify :: Term -> Term -> [st]
46 unify t1 t2 = stConvertor (goUnify (monadicUnification t1 t2))
47
48
49 varX :: Term
50 varX = Var (0,"x")
51
52 varY :: Term
53 varY = Var (1,"y")
54
55
56 stConvertor :: [(Id, Prolog)] -> [st]
57 stConvertor xs = Prelude.map (\(varId, p) -> (->-) varId (unFlatten $ unP $ p)) xs

```

1 Chapter 16

2 Prototype 4

3 Our aim to embedd IO into the DSL

4 So something like a "data" declaration for IO operations

5 16.1 I/O is pure

6 [126]

7 A common question amongst people learning Haskell is whether I/O is pure or not.
8 Haskell advertises itself as a purely functional programming language, but I/O looks like
9 its inherently impure - for example, the function `getLine`, which gets a line from `stdin`,
10 returns a different result depending on what the user types:

```
1 Prelude> x <- getLine
2 Hello
3 Prelude> x
4 "Hello"
```

11 How can this possibly be pure?

12 In this post I want to explain exactly why I/O in Haskell is pure. Ill do it by building up
13 data structures that represent blocks of code. These data structures can later be executed,
14 and they cause effects to occur - but until that point well always work with pure functions,

1 never with effects.

2 Lets look at a simplified form of I/O, where we only care about reading from stdin,
3 writing to stdout and returning a value. We can model this with the IOAction data type.

4 That is, an IOAction is one of the following three things:

- 5 1. A container for a value of type a,
- 6 2. A container holding a String to be printed to stdout, followed by another IOAction a,
7 or
- 8 3. A container holding a function from String -> IOAction a, which can be applied to
9 whatever String is read from stdin.

10 Notice that the only terminal constructor is Return that means that any IOAction must
11 be a combination of Get and Put constructors, finally ending in a Return.

12 Some simple actions include the one that prints to stdout before returning ():

```
put s = Put s (Return ())
```

13 and the action that reads from stdin and returns the string unchanged:

```
get = Get (\s -> Return s)
```

14 To build up a language for doing I/O we need to be able to combine and sequence
15 actions. We want the ability to perform an IOAction a followed by an IOAction b, and
16 return some result.

17 In fact, we could have the second IOAction depend on the return value of the first one -
18 that is, we need a sequencing combinator of the following type:

```
seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
```

19 We want to take the IOAction a supplied in the first argument, get its return value (which
20 is of type a) and feed that to the function in the second argument, getting an IOAction b
21 out, which can be sequenced with the first IOAction a.

1 That's a bit of a mouthful, but writing this combinator isn't too hard. When we reach the
2 final `Return`, we apply the function `f` to get a new action. For the other constructors, we
3 keep the form of the action the same, and just thread `seqio` through the `seqio` constructor.

4 Using `seqio` we can define the action that gets input from `stdin` and immediately prints
5 it to the screen:
6 or even more complicated actions:

```
1  hello = put "What is your name?"      'seqio' \_    ->
2         get                          'seqio' \name ->
3         put "What is your age?"        'seqio' \_    ->
4         get                          'seqio' \age   ->
5         put ("Hello " ++ name ++ "!") 'seqio' \_    ->
6         put ("You are " ++ age ++ " years old")
```

7 Although this looks like imperative code (admittedly with pretty unpleasant syntax), it's
8 really a value of type `IOAction ()`. In Haskell, code can be data and data can be code.

9 In the gist I've defined a function to convert an `IOAction` to a `String`, which allows them
10 to be printed, so you can load the file into `GHCi` and verify that `hello` is in fact just data:

```
1  *Main> print hello
2  Put "What is your name?" (
3    Get ($0 ->
4      Put "What is your age?" (
5        Get ($1 ->
6          Put "Hello $0!" (
7            Put "You are $1 years old" (
8              Return ()
9            )
10           )
11         )
12       )
13     )
14   )
```

11 It will surprise no one to learn that `IOAction` is a monad. In fact we've already defined the
12 necessary `bind` operation in `seqio`, so getting the `Monad` instance is trivial:

```
1  instance Monad IOAction where
2      return = Return
3      (>>=)  = seqio
```

1 The main benefit of doing this is that we can now sequence actions using Haskell's `do`
2 notation, which desugars into calls to `(\l=>)`, and hence to `seqIO`. Our earlier hello example
3 can now be written as:

```
1  hello2 = do put "What is your name?"
2              name <- get
3              put "What is your age?"
4              age <- get
5              put ("Hello, " ++ name ++ "!")
6              put ("You are " ++ age ++ " years old!")
```

4 Remember though, that this is still just defining a value of type `IOAction ()` - no code is
5 executed, and no effects occur! So far, this post is 100 % pure.

6 To see the effects, we need to define a function that takes an `IOAction a` and converts
7 it into a value of type `IO a`, which can then be executed by the interpreter or the runtime
8 system. It's easy to write such a function just by turning it into the appropriate calls to
9 `putStrLn` and `getLine`.

```
1  run :: IOAction a -> IO a
2  run (Return a) = return a
3  run (Put s io) = putStrLn s >> run io
4  run (Get g)    = getLine >>= \s -> run (g s)
```

10 You can now load up GHCi and apply `run` to any action - a value of type `IO a` will be
11 returned, and then immediately executed by the interpreter:

```
1  *Main> run hello
2  What is your name?
3  Chris
4  What is your age?
5  29
6  Hello Chris!
7  You are 29 years old
```

12 Is there any practical use to this?

13 Yes - an `IOAction` is a mini-language for doing I/O. In this mini language you are
14 restricted to only reading from `stdin` and writing to `stdout` - there is no accessing files,

1 spawning threads or network I/O.

2 In effect we have a safe domain-specific language. If a user of your program or library
3 supplies a value of type `IOAction a`, you know that you are free to convert it to an `IO a` using
4 `run` and execute it, and it will never do anything except reading from `stdin` and writing to
5 `stdout` (not that those things aren't potentially dangerous in themselves, but)

```
1  -- http://chris-taylor.github.io/blog/2013/02/09/io-is-not-a-side-effect/
2
3  data IOAction a =
4    -- A container for a value of type a.
5      Return a
6    -- A container holding a String to be printed to stdout, followed by another IOAction
7      | Put String (IOAction a)
8    -- A container holding a function from String -> IOAction a, which can be applied
9      | Get (String -> IOAction a)
10  {--
11
12  Return 1
13
14  Put "hello" (Return ())
15  Put "hello" (
16    Return ()
17  )
18
19  Put "hello" (Return 1)
20  Put "hello" (
21    Return 1
22  )
23
24  Put "hello" (get)
25  Put "hello" (
26    Get (£0 ->
27      Return "£0"
28    )
29  )
30
31  Get put
32  Get (£0 ->
33    Put "£0" (
34      Return ()
35    )
36  )
```

```

37
38  --}
39
40  -- Read and return
41  get :: IOAction String
42  get  = Get Return
43  {--
44
45  Get (f0 ->
46      Return "f0"
47  )
48
49  --}
50
51  -- Print and return.
52  put :: String -> IOAction ()
53  put s = Put s (Return ())
54  {--
55
56  put "hello"
57  Put "hello" (
58      Return ()
59  )
60
61  --}
62
63  -- (>=) Action sequencer and combiner :- read -> write -> read -> write -> ....
64  seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
65  --      (First action      (Take and perform
66  --      which generates next action)
67  --      value a)
68  seqio (Return a) f = f a
69  seqio (Put s io) f = Put s (seqio io f)
70  seqio (Get g)     f = Get (\s -> seqio (g s) f)
71
72  --Take input and print.
73  echo :: IOAction ()
74  echo = get 'seqio' put
75  {--
76
77  Get (f0 ->
78      Put "f0" (
79          Return ()
80      )
81  )

```

```

82
83 --}
84
85 hello :: IOAction ()
86 hello = put "What is your name?"      'seqio' \_ ->
87         get                          'seqio' \name ->
88         put "What is your age?"      'seqio' \_ ->
89         get                          'seqio' \age ->
90         put ("Hello " ++ name ++ "!") 'seqio' \_ ->
91         put ("You are " ++ age ++ " years old")
92 {--
93
94   Put "What is your name?" (
95     Get (f0 ->
96       Put "What is your age?" (
97         Get (f1 ->
98           Put "Hello f0!" (
99             Put "You are f1 years old" (
100               Return ()
101             )
102           )
103         )
104       )
105     )
106   )
107
108   run hello
109   What is your name?
110   Mehul
111   What is your age?
112   25
113   Hello Mehul!
114   You are 25 years old
115
116 --}
117
118 -- hello in "do" block since IOAction is a Monad
119 hello2 :: IOAction ()
120 hello2 = do put "What is your name?"
121            name <- get
122            put "What is your age?"
123            age <- get
124            put ("Hello, " ++ name ++ "!")
125            put ("You are " ++ age ++ " years old!")
126 {--

```

```

127
128 Put "What is your name?" (
129     Get (l0 ->
130         Put "What is your age?" (
131             Get (l1 ->
132                 Put "Hello, l0!" (
133                     Put "You are l1 years old!" (
134                         Return ()
135                     )
136                 )
137             )
138         )
139     )
140 )
141
142 run hello2
143 What is your name?
144 Mehul
145 What is your age?
146 25
147 Hello, Mehul!
148 You are 25 years old!
149
150 --}
151
152 -- where the effects happen.
153 -- "Real" IO functions like return, putStrLn, getLine.
154 run :: IOAction a -> IO a
155 run (Return a) = return a
156 run (Put s io) = putStrLn s >> run io
157 run (Get f)    = getLine >>= run . f
158 {--
159
160 run (Return 1)
161 1
162
163 run (Put "hello" get)
164 hello
165 1
166 "1"
167
168 run (Get put)
169 1
170 1
171

```

```

172 --}
173
174
175 -- Glue code that makes everything play nice --
176
177 instance Monad IOAction where
178     return = Return
179     (>=) = seqio
180
181 instance Show a => Show (IOAction a) where
182     show io = go 0 0 io
183     where
184         go m n (Return a) = ind m "Return " ++ show a
185         go m n (Put s io) = ind m "Put " ++ show s ++ " (\n" ++ go (m+2) n io ++ "\n"
186         go m n (Get g)     = let i = "$" ++ show n
187                               in ind m "Get (" ++ i ++ " -> \n" ++ go (m+2) (n+1) (g i)
188
189         ind m s = replicate m ' ' ++ s
190
191 -- IOAction is also a Functor --
192
193 mapio :: (a -> b) -> IOAction a -> IOAction b
194 mapio f (Return a) = Return (f a)
195 mapio f (Put s io) = Put s (mapio f io)
196 mapio f (Get g)    = Get (\s -> mapio f (g s))
197 {--
198
199 mapio (+1) (Return 1)
200 Return 2
201
202 mapio (id) (Put "hello" get)
203 Put "hello" (
204     Get (£0 ->
205         Return "£0"
206     )
207 )
208
209 mapio (id) (Get put)
210 Get (£0 ->
211     Put "£0" (
212         Return ()
213     )
214 )
215
216 --}

```

```

217
218 instance Functor IOAction where
219     fmap = mapio

```

1 16.2 Dr. Casperson Pure IO

```

1  -- Prolog IO
2
3  {--
4  FREE MONADS
5  In general, a structure is called free when it is left-adjoint to a forgetful functor.
6  In this specific instance, the Term data type is a higher-order functor that maps
7  a functor f to the monad Term f ; this is illustrated by the above two instance
8  definitions. This Term functor is left-adjoint to the forgetful functor from monads
9  to their underlying functors.
10 --}
11
12 data Term f a = Pure a
13                | Impure (f (Term f a))
14
15 main = undefined
16
17 instance Functor f => Functor (Term f) where
18     fmap f (Pure x )      = Pure (f x )
19     fmap f (Impure t)     = Impure (fmap (fmap f ) t)
20
21 instance Functor f => Monad (Term f) where
22     return x              = Pure x
23     (Pure x )      >>=    f      = f x
24     (Impure t)     >>=    f      = Impure (fmap (>>= f ) t)

```

2 16.3 Mehul Pure IO

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

5 The interpreted program is still pure since the IO actions have not been executed

6 if the running is done inside a monad then the IO still is pure.

```

1  import Data.Traversable
2  import Control.Monad
3  import Data.Functor
4  import Control.Applicative
5  import System.IO
6
7  data PrologResult
8      = NoResult
9      | Cons OneBinding PrologResult
10     | IOIn (IO String) (String -> PrologResult)
11     | IOOut (IO ()) PrologResult
12
13
14
15  data OneBinding = Pair VariableName VariableName
16
17
18  --data MiniLang a = MyData a | Empty | Input
19
20  --runInIO :: PrologResult -> IO [OneBinding]
21
22
23  data PrologIO a = Input (IO a) | Output (a -> IO ()) | PrologData a | Empty
24                  deriving (Show, Eq, Ord)
25  {--
26  instance Functor (PrologIO) where
27      fmap f Empty                = Empty
28      fmap f (Input (IO a))       = Input (IO (f a))
29      fmap f (Output (a -> IO ())) = Output (a -> IO ())
30      fmap f (PrologData a)       = PrologData (f a)
31  --}
32
33  instance Monad PrologIO where
34      return a = PrologData a
35      --      (Input i) >>= (Output o) = i >>= (\a -> (o a))
36
37  instance (Show a) => Show (PrologIO a) where
38      show (Empty)                = show "No result"
39      show (PrologData a)         = show a
40      --      show (Input f)        = show (f ++ "")
41      --      show (Output )
42
43
44  -- (>>=) Action sequencer and combiner :- read -> write -> read -> write -> ....
45  seqio :: PrologIO a -> (a -> PrologIO b) -> PrologIO b

```

```

46  --      (First action   (Take and perform
47  --      which generates next action)
48  --      value a)
49  seqio (PrologData a)          f          = f a
50  --seqio (Output o)           f          = \a -> o a
51  --seqio (Input i)            f          = \s -> (seqio (i s) f) --
52
53
54
55  {--
56  instance Applicative PrologIO where
57      func =
58
59  instance Traversable PrologIO where
60      traverse f Empty                                = Empty
61      traverse f (Input (IO a))                      = Input (IO (f a))
62      traverse f (Output (a -> IO ()))                = Output ((a) -> IO (f a))
63      traverse f (PrologData a)                      = PrologData (f a)
64  --}
65
66
67  concat :: PrologIO t -> PrologIO t -> IO ()
68  concat (Input f1) (Output f2) = do
69      x <- f1
70      f2 x
71  {--
72  concat (Input getLine) (Output putStrLn)
73  Loading package list-extras-0.4.1.4 ... linking ... done.
74  Loading package syb-0.5.1 ... linking ... done.
75  Loading package array-0.5.0.0 ... linking ... done.
76  Loading package deepseq-1.3.0.2 ... linking ... done.
77  Loading package containers-0.5.5.1 ... linking ... done.
78  Loading package transformers-0.4.3.0 ... linking ... done.
79  Loading package mtl-2.2.1 ... linking ... done.
80  Loading package logict-0.6.0.2 ... linking ... done.
81  Loading package unification-fd-0.10.0.1 ... linking ... done.
82  1
83  1
84  --}

```


1 Chapter 17

2 Work Completed

3 17.1 What we are doing

4 A partial implementation of the logic programming language PROLOG is provided by the
5 library `prolog-0.2.0.1`. One of the objectives is to implement monadic unification using
6 the library [128].

7 17.2 Unifiable Data Structures

8 For a data type to be Unifiable, it must have instances of Functor, Foldable and Traversable.
9 The interaction between different classes is depicted in figure 17.1.

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

13 17.3 Why Fix is necessary?

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

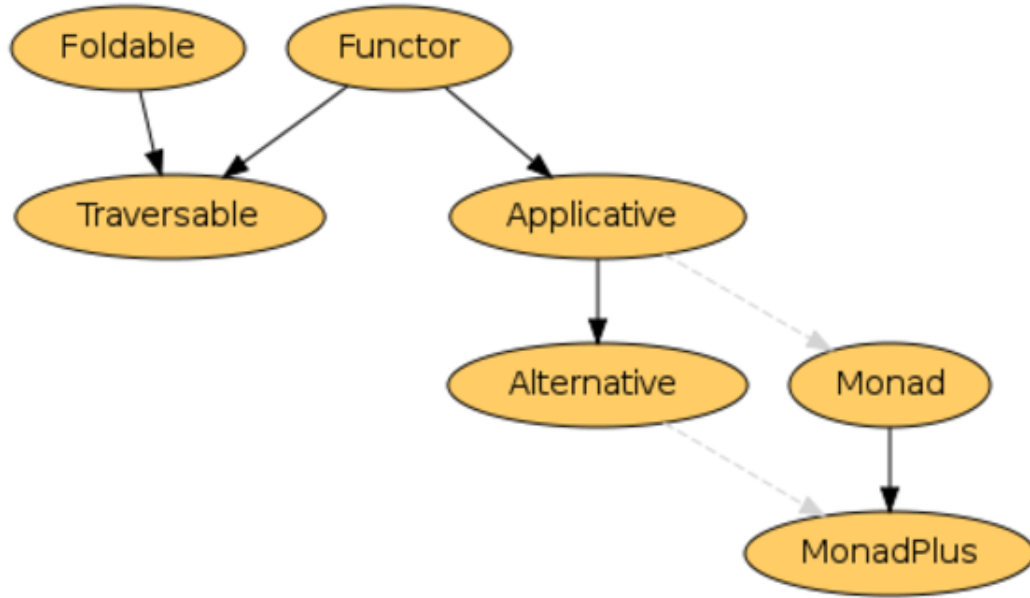


Figure 17.1: Functor Hierarchy [146]

1 In our case consider the following example,

```

-- The Prolog Syntax
type Atom = String
data VariableName = VariableName Int String deriving (Show,Eq,Ord)
data FlatTerm a =
    | Struct Atom [a]
    | Var VariableName
    | Wildcard
    | Cut Int deriving (Show,Eq,Ord)
  
```

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

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

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

```
Fix FlatTerm
```

1 resulting in the PROLOG Data Type

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

2 **17.4 Dr. Casperson's Explanation**

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

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

6 A sample Tree would be,

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

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

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

12 A sample FlatTerm would be similar to Tree.

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

15 **17.5 The other fix**

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

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

1 The above function results in an infinite application stream,

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

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

4 17.6 The Fix we use

5 Fix-point type allows to define generic recursion schemes [69]. [8]

6 **What is Algebra** Naively speaking algebra gives us the ability to perform calculations
7 with numbers and symbols.

8 **What can algebra do** The ability to form and evaluate expressions.

9 **How to generate expressions** Using grammars, for example

```
1 data Expr = Const Int
2           | Add Expr Expr
3           | Mul Expr Expr
```

10 **How to uncover primitives from a recursive type** Make it non-recursive by defining a
11 type function, otherwise known as type constructor,

```
1 ExprF a = Const Int
2         | Add a a
3         | Mul a a
```

12 **How to create a nested structure from the above** The fractally recursive structure of `Expr`
13 can be generated by repeatedly applying `ExprF` to itself.

```
1 (ExprF (ExprF (ExprF a)))
```

14 **How to generate really deep expressions** Keep on applying

```
ExprF
```

1 **Is there a better way** After infinitely many iterations we should get to a fix point where
2 further iterations make no difference. It means that applying one more ExprF would
3 not change anything – a fix point does not move under ExprF. It's like adding one to
4 infinity: you get back infinity.

5 **How do that in HASKELL** In HASKELL, we can express the fix point of a type construc-
6 tor f as a type:

```
1 newtype Fix f = f (Fix f)
```

7 With that, we can redefine Expr as a fixed point of ExprF:

```
1 type Expr = Fix ExprF
```

8 **Any other benefits** Writing functions is simpler. You can have the terms of all depths
9 encapsulated under the same type, i.e.

```
Fix ExprF
```

10 So rather than writing separate functions for,

```
1 (ExprF a)
2
3 (ExprF (ExprF a))
4
5 (ExprF (ExprF (ExprF a)))
6
7 (ExprF (ExprF (ExprF ...)))
```

11 We write a function from,

```
func :: Fix ExprF -> Fix ExprF
```

17.7 Opening up or Extending language Explanation using Box Analogy

This section will describe what it means to "open up or extend a language".

1. Let us start with a sample language with a recursive abstract syntax,

```
1 type Atom = String
2
3 data VariableName = VariableName Int String
4   deriving (Eq, Data, Typeable, Ord)
5
6 data Term = Struct Atom [Term]
7           | Var VariableName
8           | Wildcard
9           | Cut Int
10    deriving (Eq, Data, Typeable)
```

The above language represent a stripped down version of PROLOG from [107]. The pool of the expressions that can be generated from *Term* are restricted to the constructors,

```
1 Struct "hello" [Struct "a" []]      -- hello(a).
2 Var (VariableName 125 "X")          -- X = 125.
3 Wildcard                            -- _
4 Cut 0                               -- !.
```

It does not allow the ability to have a "typed" *Term*, for example a *Term* of type *int* or *string* and so on.

2. So we **flatten** the language by introducing a type variable,

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

```

8         | Wildcard
9         | Cut Int deriving (Show, Eq, Ord)

```

1 The above language can be of any type a . A more accurate way of saying it would
2 be that a can be a *kind* in HASKELL.

3 In type theory, a kind is the type of a type constructor or, less commonly, the type
4 of a higher-order type operator. A kind system is essentially a simply typed lambda
5 calculus 'one level up,' endowed with a primitive type, denoted $*$ and called 'type,'
6 which is the kind of any (monomorphic) data type for example [145],

```

1 Int :: *
2 Maybe :: * -> *
3 Maybe Bool :: *
4 a -> a :: *
5 [] :: * -> *
6 (->) :: * -> * -> *

```

7 Simply speaking the a can be changed.

8 3. It gives the language the capability to be expanded. Adding some functionality to the
9 original language could be done in a no.of ways

10 (a) Manually modifying the structure of the language,

```

1  type Atom                = String
2
3  data VariableName        = VariableName Int String
4      deriving (Eq, Data, Typeable, Ord)
5
6  data Term                = Struct Atom [Term]
7                          | Var VariableName
8                          | Wildcard
9                          | Cut Int
10                         | New_Constructor_1 .....
11                         | New_Constructor_2 .....
12      deriving (Eq, Data, Typeable)

```

11 This would then trigger a ripple effect throughout the architecture because ac-
12 comodations need to be made for the new functionality.

1 (b) The other option would be to *functorize* language like we did by adding a type
 2 variable which can be used to plug something that provides the functionality
 3 into the language. Consider the following example,

```
1 data Box f = Abox | T f (Box f) deriving (.....)
```

4 then something like,

```
1 T (Struct 'atom' [Abox, T (Cut 0)])
```

5 is possible. Since we needed the fixed point of the language we used *Fix* but
 6 generically one could add multiple custom functionality.

7 4. If we have a grammar that support an expressions like,

8 $x \cdot y + x \cdot z$

9 Once the language is 'functorized' one can add quantifiers and logic to the language
 10 to do something like,

11 $\forall x \forall y \forall z \quad x \cdot y + x \cdot z$

12 $= x \cdot (y + z)$

13 5. Multiple modifications

14 6. As is with the original language it can be wrapped with multiple other data structures,

```
1 Just (Strcut ..... ) -- A Maybe Term
2 [Cut 0]                -- A List of Terms
```

15 and so on. But the core expression can only be of type *Term*.

16 Whereas a *FlatTerm* expression can not only have an outer wrapper but also its type
 17 is 'open'.

Chapter 18

Results

18.1 Types

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

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

Our preliminary implementation is as follows,

```
type Atom = String

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

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

```
{--
Output :-

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

--}
```

1 which in PROLOG would look like,

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

2 **18.2 Lazy Evaluation**

3 **18.3 Opening up the Language**

4 **Flattening**

5 **Fixing**

6 **MetaSyntactic Variables**

7 **18.4 Quasi Quotation**

8 **18.5 Template Haskell**

9 **18.6 Higher Order Functions**

```
% Mehul Solanki.
```

```
% Higher Order Functions.
```

```
% The following library contains the maplist function.
```

```

:- use_module(library(apply)).

% The maplist function takes a function and a list to apply the
% function.
% The function write is passes which will print out the elements
% of the list.
higherOrder(X) :- maplist(write,X).

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

```

1 18.7 I/O

```

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

```

2 18.8 Mutability

3 18.9 Unification

4 18.10 Monads

1 Chapter 19

2 Future Scope

- 3 1. Quasi quoter to get something like,

1 [prolog|a(**X**) :- b(y)|]

4 where X is a PROLOG variable and y is a HASKELL variable injected into the expres-
5 sion

- 6 2. We already have variable search strategies, what if the query resolver could be in-
7 structed to use a particular search strategy to get the result.

1 queryResolver searchStrategy query knowledgeBase

- 8 3. Add database operations

- 9 4. Multi type variable Language

Chapter 20

Conclusion / Expected Outcomes

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

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

Chapter 21

Editing to do

This Chapter needs to be removed from the final work.

2015-10-29

1. Abstract is too long and incorrect.
2. Remove first ¶ from intro.
3. Thesis statement is close to being an abstract.

Either

4. We need a convention for what words to capitalize in chapter and section titles.

Mehul

5. Chapter 13.5 needs fleshing out.
6. **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.
7. Fix the reference at the bottom of page 2:
`citewikipro- log,somogyi1995logic,website:prolog1000db.` **SOLVED**

8. Write enough of Chapters 13–16 that we can decide what material is needed in Chapters 9, 10, and ??.
9. [mainly done] 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.
10. Rewrite (Section) Chapter 3 in formal English.
11. “re-curses” means to swear again (*p* 9). **Changed to recurs**
12. 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.
13. Supply your credentials in the front material (what degrees do you have?). (Search for `%% Supply your credentials in proposal1.tex`.)
14. The abstract is too long. UNBC guidelines limit Masters' theses abstracts to 150 words.
15. Citation `logic-classes` is not defined (in `./prologinhaskell.tex`).

David

16. Clean up the non-exclusive license page in `unbcthesis.cls`
17. Incorporate `unbcthesis.cls` into Mehul's work.

18. Review Chapter 2
19. Review Chapter 3
20. Review Chapter 4
21. Review Chapter 5
22. Review Chapter 6
23. Review Chapter 7
24. Review Chapter 8
25. Review Chapter 18

21.1 Editing suggestions from David

Thoughts on Chapter 14

- Do not use naked `\refs`: “*the generic methodology from 13*” should be “*the generic methodology from **Chapter 13***”.
- You should say more about [107], either here or in an earlier section and reference that discussion here. For instance, it isn’t clear that `prolog-0.2.0.1` comes from [107].
- See my comments below. I suspect that longer listings should be separate figures.
- Line 7 on p 55 is not a complete sentence.
- I suspect that § 14.2 should start with a sentence like

The `prolog-0.2.0.1` ([107]) was written by Indira Ghandi and consists of 718 HASKELL files. It implements data base assertions and cuts but lacks any IO facilities...

```

1 data VariableName = VariableName Int String
2   deriving (Eq, Data, Typeable, Ord)
3 data Atom         = Atom         !String
4                   | Operator    !String
5   deriving (Eq, Ord, Data, Typeable)
6 data Term = Struct Atom [Term]
7           | Var VariableName
8           | Wildcard
9           | PString    !String
10          | PInteger   !Integer
11          | PFloat     !Double
12          | Flat [FlatItem]
13          | Cut Int
14   deriving (Eq, Data, Typeable)
15 data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
16             | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
17   deriving (Data, Typeable)
18 type Program = [Sentence]
19 type Body    = [Goal]
20 data Sentence = Query    Body
21              | Command Body
22              | C Clause
23   deriving (Data, Typeable)

```

Figure 21.1: A sample Minted figure

and then go on to discuss the syntax.

Thoughts on Chapter 13 I am looking at what are currently lines 145–*on* in `proto1.tex`, and I am not sure whether

1. the text should be loose—as you have it, or floated to a figure, as shown in Figure 21.1.
 2. I am also not sure whether I like the inlined code, or whether I would prefer to have it `\inputminted` from a `HASKELL` file. I suppose that this depends on your work-flow.
- Thoughts?

I am not sure what conventions you are following with respect to code in text. At some point you have `FlatTerm` in italics (à la *FlatTerm*); at other points you have it typeset in

straight double quotes ("FlatTerm") and I don't know what the different typesetting implies.

Just above Section 13.5 you mention a generic function `map`, which for STANDARD ML and HASKELL readers likely means the function with signature $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$.

Why not `fmap`?

I am not sure what the point of the ¶ before Section 13.5 is.

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?

1 Bibliography

- 2 [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, tech-*
3 *niques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA,
4 USA, 1986.
- 5 [2] Hassan Aït-Kaci and Forêt Des Flambertins, *Warrens abstract machine a tutorial*
6 *reconstruction*, (1999).
- 7 [3] Sergio Antoy, *Implementing functional logic programming languages*.
- 8 [4] ———, *Sergio antoy home page*.
- 9 [5] Sergio Antoy and Michael Hanus, *Functional logic programming*, Communications
10 of the ACM **53** (2010), no. 4, 74–85.
- 11 [6] Lennart Augustsson, *Cayenne – a language with dependent types*, IN INTER-
12 NATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, ACM Press,
13 1998, pp. 239–250.
- 14 [7] Andrei Barbu, *The csp package*, August 2013, [http://hackage.haskell.org/](http://hackage.haskell.org/package/csp)
15 [package/csp](http://hackage.haskell.org/package/csp).
- 16 [8] FP Complete Bartosz Milewski, *Understanding algebras*, October 2013.
- 17 [9] Matthias Bartsch, *The prolog-graph package*, September 2011, [http://hackage.](http://hackage.haskell.org/package/prolog-graph)
18 [haskell.org/package/prolog-graph](http://hackage.haskell.org/package/prolog-graph).
- 19 [10] Eli Barzilay and Dmitry Orlovsky, *Foreign interface for plt scheme*, on Scheme and
20 Functional Programming (2004), 63.
- 21 [11] Nick Benton, *Embedded interpreters*, Journal of Functional Programming **15** (2005),
22 no. 4, 503–542.
- 23 [12] Didier Bert, Pascal Drabik, and Rachid Echahed, *Lpg: A generic, logic and func-*
24 *tional programming language*, STACS 87, Springer, 1987, pp. 468–469.
- 25 [13] James Bielman and Lus Oliveira, *Common lisp foreign function interface*, March
26 2014.

- 1 [14] Andrew Butterfield (ed.), *Unifying theories of programming, second international*
2 *symposium, utp 2008, dublin, ireland, september 8-10, 2008, revised selected pa-*
3 *pers*, Lecture Notes in Computer Science, vol. 5713, Springer, 2010.
- 4 [15] C2, *Multi paradigm programming language*, September 2012.
- 5 [16] c2 wiki, *Metasyntactic variables*, September 2011.
- 6 [17] Catb, *Metasynatactic variables*.
- 7 [18] Prolog Development Center, *Visual prolog*, June 2013.
- 8 [19] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang, *A type-safe embedding of con-*
9 *straint handling rules into haskell*, Technical report School of Computing, National
10 University of Singapore, Boston, MA, USA (2003).
- 11 [20] Ciao, *Ciao programming language*, August 2011.
- 12 [21] Koen Claessen and Peter Ljunglöf, *Typed logical variables in haskell.*, Electr. Notes
13 Theor. Comput. Sci. **41** (2000), no. 1, 37.
- 14 [22] Code Commit, *Hindley milner type system*, December 2008.
- 15 [23] Mozart Consortium, *Oz / mozart*, March 2013.
- 16 [24] Gregory Crosswhite, *The logicgrowsontrees package*, September 2013, [http://](http://hackage.haskell.org/package/LogicGrowsOnTrees)
17 hackage.haskell.org/package/LogicGrowsOnTrees.
- 18 [25] DanDoel, *The logict package*, August 2013, [http://hackage.haskell.org/](http://hackage.haskell.org/package/logict)
19 [package/logict](http://hackage.haskell.org/package/logict).
- 20 [26] ———, *The logict package example*, August 2013, [http://okmij.org/ftp/](http://okmij.org/ftp/Computation/monads.html)
21 [Computation/monads.html](http://okmij.org/ftp/Computation/monads.html).
- 22 [27] Oleg Kiselyov Daniel P. Friedman, William E. Byrd, *The reasoned schemer*, The
23 MIT Press, Cambridge Massachusetts, London England, 2005.
- 24 [28] William E. Byrd Daniel P. Friedman and Oleg Kiselyov, *Kanren*, March 2009.
- 25 [29] Universidad Complutense de Madrid, *Toy*, Decmeber 2006.
- 26 [30] University Of Melbourne Computer Science department, *Mercury programming lan-*
27 *guage*, February 2014.
- 28 [31] Dustin DeWeese, *The peg package*, April 2012, [http://hackage.haskell.org/](http://hackage.haskell.org/package/peg)
29 [package/peg](http://hackage.haskell.org/package/peg).
- 30 [32] Free Dictionary, *Quasi-quotation*.
- 31 [33] Open Directory Project dmoz, *Multi paradigm*, November 2013.

- 1 [34] SWI Prolog Documentation, *Embedding swi-prolog in other applications*, June
2 2013, <http://www.swi-prolog.org/pldoc/man?section=embedded>.
- 3 [35] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan,
4 *Making data structures persistent*, Proceedings of the eighteenth annual ACM sym-
5 posium on Theory of computing, ACM, 1986, pp. 109–121.
- 6 [36] Steve Dunne and Bill Stoddart (eds.), *Unifying theories of programming, first in-
7 ternational symposium, utp 2006, walworth castle, county durham, uk, february 5-
8 7, 2006, revised selected papers*, Lecture Notes in Computer Science, vol. 4010,
9 Springer, 2006.
- 10 [37] Joshua Eckroth, *Prolog resolution*, April 2014.
- 11 [38] ———, *Prolog unification*, April 2014.
- 12 [39] Martin Erwig, *Escape from zurg: an exercise in logic programming*, Journal of Func-
13 tional Programming **14** (2004), no. 03, 253–261.
- 14 [40] Sebastian Fischer, *The cflp package*, June 2009, [http://hackage.haskell.org/
15 package/cflp](http://hackage.haskell.org/package/cflp).
- 16 [41] ———, *stream-monad*, September 2012.
- 17 [42] Adam C. Foltzer, *Molog*, March 2013.
- 18 [43] Marc Fontaine, *The cspm-toprolog package*, August 2013, [http://hackage.
19 haskell.org/package/CSPM-ToProlog](http://hackage.haskell.org/package/CSPM-ToProlog).
- 20 [44] David Fox, *The proplogic package*, April 2012, [http://hackage.haskell.org/
21 package/PropLogic](http://hackage.haskell.org/package/PropLogic).
- 22 [45] ———, *The logic-classes package*, October 2013, [http://hackage.haskell.
23 org/package/logic-classes](http://hackage.haskell.org/package/logic-classes).
- 24 [46] Jeremy Gibbons, *Unifying theories of programming with monads*, Unifying Theories
25 of Programming, Springer, 2013, pp. 23–67.
- 26 [47] GNU, *Gnu prolog for java*, August 2010, [http://www.gnu.org/software/
27 gnuprologjava/](http://www.gnu.org/software/gnuprologjava/).
- 28 [48] Michael Hanus, *Michael hanus home page*.
- 29 [49] Michael Hanus, *Multi-paradigm declarative languages*, Logic Programming,
30 Springer, 2007, pp. 45–75.
- 31 [50] Michael Hanus, *Functional logic programming*, February 2009.
- 32 [51] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro, *Curry: A truly
33 functional logic language*, Proc. ILPS, vol. 95, 1995, pp. 95–107.

- 1 [52] Haskellwiki, *Template haskell*, October 2013.
- 2 [53] Juan Jose Moreno Navarro Herbert Kuchen, *Babel programming language*, January
3 1988.
- 4 [54] Ernest Lepore Herman Cappelen, *Quotation*, January 2012.
- 5 [55] Ralf Hinze et al., *Prological features in a functional setting axioms and implemen-*
6 *tation.*, Fuji International Symposium on Functional and Logic Programming, Cite-
7 seer, 1998, pp. 98–122.
- 8 [56] Charles Anthony Richard Hoare and Jifeng He, *Unifying theories of programming*,
9 vol. 14, Prentice Hall Englewood Cliffs, 1998.
- 10 [57] Satoshi Egi Ryo Tanaka Takahisa Watanabe Kentaro Honda, *Egison package*, March
11 2014.
- 12 [58] Paul Hudak, *Building domain-specific embedded languages*, ACM Comput. Surv.
13 **28** (1996), no. 4es, 196.
- 14 [59] John Hughes, *Why functional programming matters*, The computer journal **32**
15 (1989), no. 2, 98–107.
- 16 [60] What is Tech Target, *Metasynatactic variables*, September 2005.
- 17 [61] JaimieMurdock, *Haskell kanren*, March 2012.
- 18 [62] JLogic, *Jlog - prolog in java*, September 2012, <http://jlogic.sourceforge.net/index.html>.
- 19 [63] ———, *Jscriptlog - prolog in javascript*, September 2012, <http://jlogic.sourceforge.net/index.html>.
- 20 [64] Paul Johnson, *Why haskell is good for embedded domain specific languages*, January
21 2008.
- 22 [65] Mark P Jones, *Mini-prolog for hugs 98*, June 1996, <http://darcs.haskell.org/hugs98/demos/prolog/>.
- 23 [66] Simon L Peyton Jones, Jean-Marc Eber, and Julian Seward, *Composing contracts:*
24 *An adventure in financial engineering*, FME, vol. 2021, 2001, p. 435.
- 25 [67] Mark Kantrowitz, *The prolog 1000 database*, August 2012.
- 26 [68] David Karger, *Persistent data structures*, September 2005.
- 27 [69] Anton Kholomiov, *data-fix*, February 2013.
- 28 [70] H Jan Komorowski, *Qlog: The programming environment for prolog in lisp*, Logic
29 Programming (1982), 315–324.
- 30
31
32

- 1 [71] Shriram Krishnamurthi, *Programming languages: Application and interpretation*,
2 ch. 33-34, pp. 295–305, 307–311, Brown Univ., 2007.
- 3 [72] ———, *Teaching programming languages in a post-linnaean age*, SIGPLAN Not.
4 **43** (2008), no. 11, 81–83.
- 5 [73] The Programming Languages Weblog Lambda The Ultimate, *Embedding prolog in*
6 *haskell*, July 2004, <http://lambda-the-ultimate.org/node/112>.
- 7 [74] ———, *Embedding one language into another*, March 2005, [http://](http://lambda-the-ultimate.org/node/578)
8 lambda-the-ultimate.org/node/578.
- 9 [75] ———, *Application-specific foreign-interface generation*, October 2006, [http://](http://lambda-the-ultimate.org/node/2304)
10 lambda-the-ultimate.org/node/2304.
- 11 [76] Duncan Temple Lang, *Embedding s in other languages and environments*, Proceed-
12 ings of DSC, vol. 2, 2001, p. 1.
- 13 [77] LangPop.com, *Programming language popularity*, October 2013.
- 14 [78] University of Melbourne Lee Naish, *Neu prolog*, February 1991.
- 15 [79] John W Lloyd, *Programming in an integrated functional and logic language*, Journal
16 of Functional and Logic Programming **3** (1999), no. 1-49, 68–69.
- 17 [80] Geoffrey Mainland, *Why it's nice to be quoted: quasiquoting for haskell*, Proceed-
18 ings of the ACM SIGPLAN workshop on Haskell workshop, ACM, 2007, pp. 73–82.
- 19 [81] Yonathan Malachi, Zohar Manna, and Richard Waldinger, *Tablog: The deductive-*
20 *tableau programming language*, Proceedings of the 1984 ACM Symposium on LISP
21 and functional programming, ACM, 1984, pp. 323–330.
- 22 [82] Erik Meijer and Peter Drayton, *Static typing where possible, dynamic typing when*
23 *needed: The end of the cold war between programming languages*, Citeseer, 2004.
- 24 [83] Bertrand Meyer, *Eiffel as a framework for verification*, Verified Software: Theories,
25 Tools, Experiments, Springer, 2008, pp. 301–307.
- 26 [84] Juan José Moreno-Navarro and Mario Rodriguez-Artalejo, *Babel: A functional and*
27 *logic programming language based on constructor discipline and narrowing*, Alge-
28 braic and Logic Programming, Springer, 1988, pp. 223–232.
- 29 [85] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo, *Logic programming with*
30 *functions and predicates: The language babel*, The Journal of Logic Programming
31 **12** (1992), no. 3, 191–223.
- 32 [86] R Morrison and MP Atkinson, *Persistent languages and architectures*, Security and
33 Persistence, Springer, 1990, pp. 9–28.

- 1 [87] MPprogramming.com, *Castor : Logic paradigm for c++*, August 2010, [http://](http://www.mpprogramming.com/cpp/)
2 www.mpprogramming.com/cpp/.
- 3 [88] Gopalan Nadathur, *λ prolog*, September 2013.
- 4 [89] Mark J Nelson, *Why did prolog lose steam?*, August 2010, [http://www.kmjn.org/](http://www.kmjn.org/notes/prolog_lost_steam.html)
5 [notes/prolog_lost_steam.html](http://www.kmjn.org/notes/prolog_lost_steam.html).
- 6 [90] Mozilla Developer Network, *Multi paradigm language*, February 2014.
- 7 [91] Johan Nordlander, *O'haskell*, January 2001.
- 8 [92] Kurt Nrmak Department of Computer Science Aalborg University Denmark,
9 *Linguistic abstraction*, July 2013, [http://people.cs.aau.dk/~normark/](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_)
10 [prog3-03/html/notes/languages_themes-intro-sec.html#languages_](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_)
11 [intro-sec_section-title_1](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_).
- 12 [93] University of Maryland Medical Center, *Lisp, unification and embedded languages*,
13 October 2012, <http://www.cs.unm.edu/~luger/ai-final2/LISP/>.
- 14 [94] Ocaml Org, *Ocaml programming language*, March 2014.
- 15 [95] Pedro Pinto, *Dot-scheme: A plt scheme ffi for the .net framework*, Workshop on
16 Scheme and Functional Programming, Citeseer, 2003.
- 17 [96] Quintus Prolog, *Embeddability*, December 2003, [http://quintus.sics.se/isl/](http://quintus.sics.se/isl/quintuswww/site/embed.html)
18 [quintuswww/site/embed.html](http://quintus.sics.se/isl/quintuswww/site/embed.html).
- 19 [97] Yield Prolog, *Yield prolog*, October 2011, [http://yieldprolog.sourceforge.](http://yieldprolog.sourceforge.net/)
20 [net/](http://yieldprolog.sourceforge.net/).
- 21 [98] Shengchao Qin (ed.), *Unifying theories of programming - third international symposium, utp 2010, shanghai, china, november 15-16, 2010. proceedings*, Lecture Notes
22 in Computer Science, vol. 6445, Springer, 2010.
- 23 [99] John Ramsdell, *The cmu package*, February 2013, [http://hackage.haskell.](http://hackage.haskell.org/package/cmu)
24 [org/package/cmu](http://hackage.haskell.org/package/cmu).
- 25 [100] Norman Ramsey, *Embedding an interpreted language using higher-order functions*
26 *and types*, Proceedings of the 2003 workshop on Interpreters, virtual machines and
27 emulators, ACM, 2003, pp. 6–14.
- 28 [101] John Reppy and Chunyan Song, *Application-specific foreign-interface generation*,
29 Proceedings of the 5th international conference on Generative programming and
30 component engineering, ACM, 2006, pp. 49–58.
- 31 [102] Maik Riechert, *The monadiccp package*, July 2013, [http://hackage.haskell.](http://hackage.haskell.org/package/monadiccp)
32 [org/package/monadiccp](http://hackage.haskell.org/package/monadiccp).
- 33

- 1 [103] J Alan Robinson and Ernest E Sibert, *Loglisp: Motivation, design, and implementa-*
2 *tion*, 1982.
- 3 [104] John Alan Robinson and EE Silbert, *Loglisp: an alternative to prolog*, School of
4 Computer and Information Science, Syracuse University, 1980.
- 5 [105] Raúl Rojas, *A tutorial introduction to the lambda calculus*, DOI= [http://www. utdal-](http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf)
6 [las. edu/~ gupta/courses/apl/lambda. pdf](http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf) (2004).
- 7 [106] Daniel Seidel, *The prolog-graph-lib package*, June 2012, [http://hackage.](http://hackage.haskell.org/package/prolog-graph-lib)
8 [haskell.org/package/prolog-graph-lib](http://hackage.haskell.org/package/prolog-graph-lib).
- 9 [107] ———, *The prolog package*, June 2012, [http://hackage.haskell.org/](http://hackage.haskell.org/package/prolog)
10 [package/prolog](http://hackage.haskell.org/package/prolog).
- 11 [108] Eric Seidel, *The liquid-fixpoint package*, September 2013, [http://hackage.](http://hackage.haskell.org/package/liquid-fixpoint)
12 [haskell.org/package/liquid-fixpoint](http://hackage.haskell.org/package/liquid-fixpoint).
- 13 [109] Silvija Seres, *The algebra of logic programming*, Ph.D. thesis, 2001.
- 14 [110] Silvija Seres and Shin-Cheng Mu, *Optimisation problems in logic programming: an*
15 *algebraic approach*, (2000).
- 16 [111] Silvija Seres, J Michael Spivey, and CAR Hoare, *Algebra of logic programming.*,
17 ICLP, 1999, pp. 184–199.
- 18 [112] Silvija Seres and Michael Spivey, *Higher-order transformation of logic programs*,
19 Logic Based Program Synthesis and Transformation, Springer, 2001, pp. 57–68.
- 20 [113] Tim Sheard and Emir Pasalic, *Two-level types and parameterized modules*, Journal
21 of Functional Programming **14** (2004), no. 05, 547–587.
- 22 [114] Dorai Sitaram, *Racklog: Prolog-style logic programming*, January 2014.
- 23 [115] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard OKeefe, *Logic*
24 *programming for the real world*, Proceedings of the ILPS, vol. 95, 1995, pp. 83–94.
- 25 [116] Andy Sonnenburg, *logicst*, April 2013.
- 26 [117] JM Spivey, *An introduction to logic programming through prolog*, 1995.
- 27 [118] JM Spivey and Silvija Seres, *The algebra of searching*, Festschrift in honour of
28 CAR Hoare (1999).
- 29 [119] ———, *Embedding prolog in haskell*, Proceedings of Haskell, vol. 99, Citeseer,
30 1999, pp. 1999–28.
- 31 [120] Michael Spivey, *Functional pearls combinators for breadth-first search*, Journal of
32 Functional Programming **10** (2000), no. 4, 397–408.

- 1 [121] Stackoverflow, *Haskell vs. prolog comparison*, December 2009, [http://](http://stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison)
2 stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison.
- 3 [122] Patrick Blackburn Johan Bos Kristina Striegnitz, *Learn prolog now*, January 2012.
- 4 [123] ———, *Learn prolog now*, January 2012.
- 5 [124] Jurrien Stutterheim, *The nanoprolog package*, December 2011, [http://hackage.](http://hackage.haskell.org/package/NanoProlog)
6 [haskell.org/package/NanoProlog](http://hackage.haskell.org/package/NanoProlog).
- 7 [125] Evgeny Tarasov, *The hswip package*, August 2010, [http://hackage.haskell.](http://hackage.haskell.org/package/hswip)
8 [org/package/hswip](http://hackage.haskell.org/package/hswip).
- 9 [126] Chris Taylor, *Io is pure*, February 2013.
- 10 [127] William E. Byrd The Reasoned Schemer’ (MIT Press, 2005) by Daniel P. Friedman
11 and Oleg Kiselyov, *minikanren*.
- 12 [128] Wren Thornton, *The unification-fd package*, July 2012, [http://hackage.](http://hackage.haskell.org/package/yieldprolog)
13 [haskellhttp://yieldprolog.sourceforge.net/.org/package/](http://hackage.haskell.org/package/yieldprolog)
14 [unification-fd](http://hackage.haskell.org/package/yieldprolog).
- 15 [129] ———, *Unification tutorial 1*, October 2015.
- 16 [130] ———, *Unification tutorial 2*, October 2015.
- 17 [131] Jan Tikovsky, *The monadiccp-gecode package*, January 2014, [http://hackage.](http://hackage.haskell.org/package/monadiccp-gecode)
18 [haskell.org/package/monadiccp-gecode](http://hackage.haskell.org/package/monadiccp-gecode).
- 19 [132] Carnegie Mellon University, *Algebraic logic functional programming language*,
20 February 1995.
- 21 [133] Dalhousie University, *Control flow*, January 2012.
- 22 [134] Simon Fraiser University, *Life programming language*, March 1998.
- 23 [135] Los Angeles University of California, *Virgil programming language*, March 2012.
- 24 [136] Germany University of Kiel, *Curry programming language*, September 2013.
- 25 [137] Maarten van Emden, *Who killed prolog?*, August 2010, [http://vanemden.](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/)
26 [wordpress.com/2010/08/21/who-killed-prolog/](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/).
- 27 [138] Andre Vellino, *Prolog’s death*, August 2010, [http://synthese.wordpress.com/](http://synthese.wordpress.com/2010/08/21/prologs-death/)
28 [2010/08/21/prologs-death/](http://synthese.wordpress.com/2010/08/21/prologs-death/).
- 29 [139] Job Vranish, *minikanrent*, March 2013.
- 30 [140] Philip Wadler, *Comprehending monads*, Mathematical Structures in Computer Sci-
31 *ence* **2** (1992), no. 04, 461–493.

- 1 [141] Haskell Website, *Logic programming example*, February 2010, http://www.haskell.org/haskellwiki/Logic_programming_example.
- 2
- 3 [142] ———, *Logic programming example in haskell*, February 2010.
- 4 [143] ———, *Quasiquote in haskell*, January 2014, <http://www.haskell.org/haskellwiki/Quasiquote>.
- 5
- 6 [144] Haskell Wiki, *Monads as computation*, December 2011.
- 7 [145] ———, *Kind*, August 2012.
- 8 [146] ———, *Foldable and traversable*, January 2013.
- 9 [147] ———, *The haskell programming language*, October 2013.
- 10 [148] ———, *Embedded domain specific languages*, September 2014.
- 11 [149] ———, *Haskell/laziness*, November 2014.
- 12 [150] ———, *Monads in haskell*, January 2014.
- 13 [151] ———, *Haskell in industry*, June 2015.
- 14 [152] Wikipedia, *Prolog wikipedia*, March 2004.
- 15 [153] ———, *Functional logic programming languages*, February 2005.
- 16 [154] ———, *Common lisp object system*, December 2013.
- 17 [155] ———, *Curry programming language*, December 2013.
- 18 [156] ———, *Functional logic programming*, May 2013.
- 19 [157] ———, *Quasiquote*, November 2013, <http://en.wikipedia.org/wiki/Quasiquote>.
- 20
- 21 [158] ———, *Common language infrastructure*, February 2014.
- 22 [159] ———, *Common language runtime*, March 2014.
- 23 [160] ———, *Constraint handling rules*, March 2014.
- 24 [161] ———, *Constraint programming*, March 2014.
- 25 [162] ———, *Damas-hindley-milner type system*, February 2014.
- 26 [163] ———, *Foreign function interface*, January 2014.
- 27 [164] ———, *Lambda calculus*, March 2014.
- 28 [165] ———, *List of multi paradigm languages*, March 2014.

- 1 [166] ———, *Meta programming*, March 2014.
- 2 [167] ———, *Ocaml*, March 2014.
- 3 [168] ———, *Programming paradigm*, March 2014.
- 4 [169] ———, *Comparison of prolog implementations*, August 2015.
- 5 [170] ———, *Control flow*, August 2015.
- 6 [171] ———, *Declarative programming*, September 2015.
- 7 [172] ———, *Metasyntactic variable*, October 2015.
- 8 [173] ———, *Usemention distinction*, October 2015.
- 9 [174] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi (eds.), *Unifying*
10 *theories of programming, 4th international symposium, utp 2012, paris, france, au-*
11 *gust 27-28, 2012, revised selected papers*, Lecture Notes in Computer Science, vol.
12 7681, Springer, 2013.
- 13 [175] Takashi's Workplace, *A prolog in haskell*, April 2009, [http://propella.](http://propella.blogspot.in/2009/04/prolog-in-haskell.html)
14 [blogspot.in/2009/04/prolog-in-haskell.html](http://propella.blogspot.in/2009/04/prolog-in-haskell.html).
- 15 [176] xkcd, *Haskell vs prolog, or giving haskell a choice*, February 2009, [http://](http://echochamber.me/viewtopic.php?f=11&t=35369)
16 echochamber.me/viewtopic.php?f=11&t=35369.
- 17 [177] Switzerland cole Polytechnique Fdrale de Lausanne (EPFL) Lausanne, *Scala pro-*
18 *gramming language*, 2002-2014.