EMBEDDING PROGRAMMING LANGUAGES: PROLOG IN HASKELL

by

Mehul Chandrakant Solanki

B.Eng, 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

April 2016

Abstract

This thesis focuses on combining the two most important and wide spread declarative programming paradigms, functional and logic programming. The proposed approach aims at adding logic programming features which are native to PROLOG onto HASKELL. We develop extensions which replicate the target language by utilizing advanced features of the host language for an efficient implementation.

The thesis aims to provide insights into merging two declarative languages namely, HASKELL and PROLOG by embedding the latter into the former and analyzing the results of doing so as the two languages have conflicting characteristics. The finished products will be something similar to a *haskellised* PROLOG which has logic programming-like capabilities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODE LISTINGS

Chapter 1

Introduction

This chapter introduces the scope of the thesis along with the preliminary arguments.

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.

A programming language must not only provide an easy to use environment but also adaptability towards the problem domain.

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 disadvantages. How does one choose which approach to adopt? Perhaps one does not need to choose! This document looks at the attempts, improvements and future possibilities of uniting HASKELL, a purely functional programming language and PROLOG, a logical programming language so that one is not forced to choose. The task at hand involves replicating PROLOG-like features in HASKELL such as unification and a single typed system. The thesis aims at leveraging the features of the host language in order to incorporate logic programming features resulting in extending HASKELL with capabilities like unification. We achieve this by adopting various aspects from approaches related to merging paradigms and embedding techniques for programming languages. This results in a hybrid approach which provides a library taking advantage of the host language features.

1.1 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 analyzing the result of doing so as they have conflicting characteristics. The finished product will be something like a *haskellised* PROLOG which has logic programming like capabilities.

1.2 Problem statement

Over the years the development of programming languages has become more and more rapid. Today the number of is in the thousands and counting [178, 46]. The successors attempt to introduce new concepts and features to simplify the process of coding a solution and assist the programmer by lessening the burden of carrying out standard tasks and procedures. A new one tries to capture the best of the old; learn from the mistakes, add new concepts and move on; which seems to be good enough from an evolutionary perspective. However, all is not that straight forward when shifting from one language to another. There are costs and incompatibilities to look at. A language might be simple to use and provide better performance than its predecessor but not always be worth the switch. Another approach would be to replicate target features exhibited by a language in the present one to avoid the hassle of jumping between the two. Commonly this results in an embedded language or a foreign function interface. A mixture of these ideologies results in a multi-paradigm / merged programming language. We try to encapsulated both the approaches of embedding and merging to develop a hybrid approach.

PROLOG is a language that has a hard time being adopted. Born in an era where procedural languages were receiving a lot of attention, it suffered from competing against another new kid on the block: C.

Some of the problems were due to its own limitations. Basic features like modules were not provided by all compilers. Practical features for real world problems were added in an ad hoc way resulting in the loss of its purely declarative nature. Some say that PROLOG is fading away, [91, 146, 145]. It is apparently not used for building large programs [160, 122, 66]. However there are a lot of good things about PROLOG: it is ideal for search problems; it has a simple syntax, and a strong underlying theory. It is a language that should not die away.

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

One idea is to make PROLOG an add-on to another language which is widely used and in demand. Here the choice is HASKELL; as both the languages are declarative they share a common background which can help to blend the two. HASKELL also has some support for logic programming [149] which enables encoding of search problems.

A domain specific language (DSL) is a concise micro language that offers tools and functionalities focused on a particular problem domain. In many cases, DSL programs are translated to calls to a subroutine library and the DSL can be viewed as a means to hide the details of that library [144]. A DSL is built specically to express the concepts of a particular domain. Generally, two types of DSL can be distinguished according to the underlying implementation strategy. An external DSL is constructed from scratch, starting from definition of the desired syntax to implementation of the associated parser for reading it. In contrast, internal or embedded DSLs (eDSL) are constructed on top of an existing host language [79].

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

Generally speaking, programming languages with a wide scope over problem domains do not provide bespoke support for accomplishing mundane tasks. Approaching towards the solution can be complicated and tiresome, but the programming language in question acts as the master key. A general purpose language, as the name suggests, provides a general set of tools to cover many problem domains. The downside is that such general purpose languages lack tools specific to certain problem domains.

Flipping the coin to the other side, we see, the more specific the language is to the problem domain, the easier it is to solve the problem since the solution need not be moulded according to the capability of the language. For example, a problem with a naturally recursive solution cannot take advantage of tail recursion in many imperative languages. Many domains require being able to prove that certain chunks of code are side effect free, but must deal with a language with uncontrolled side effects.

The solution is to develop a programming language with a split personality, in our case, sometimes functional, sometimes logical and sometimes both. Depending upon the problem, the language shapes itself accordingly and exhibits the desired characteristics. The ideal situation is a language with a rich feature set and the ability to mould itself according to the problem. A language with the ability to take the appropriate skill set and provide them to the programmer will reduce the hassle of jumping between languages or forcibly trying to solve a problem according to the limitations of a paradigm.

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

A HASKELL with PROLOG-like features will result in a set of characteristics which are from both the declarative paradigms.

This can be achieved in two ways:

Embedding (Chapter 5): This approach involves translating a complete language into the host language as an extension such as a library or module. The result is very shallow as all the positives as well as the negatives are brought into the host language. The negatives mentioned being, that languages from different paradigms usually have conflicting characteristics and result in inconsistent properties of the resulting embedding. Examples and further discussion on the same are provided in the chapters to come.

Paradigm Integration (Chapter 6): This approach goes much deeper as it does not involve a direct translation. An attempt is made by taking a particular characteristic of a language and merging it with the characteristic of the host language in order to eliminate conflicts resulting in a multi-paradigm language. It is more of weaving the two languages into one tight package with the best of both and maybe even the worst of both.

1.3 Thesis organization

The next chapter, Chapter 2 talks about the programming paradigms and languages in general and the ones in question. Chapter 3 provides details about the shortcomings of the previous works and an approach for improvements. Chapter 4 provides details on the conceptual contributions of this thesis and the prototypes. Then we look at the question from different angles namely, Chapter 5, embedding a programming language into another programming language and Chapter 6, multi-paradigm languages (functional logic languages). Some of the indirectly related content resides in Chapter 9. Chapters 7 and 8 discuss the languages used in this thesis; HASKELL and PROLOG. Chapters 10, 11, 12 and 13 describe the ideas and the implementation details of the prototypes. Chapter 14 gives a glimpse of the future and we conclude with Chapter 15.

Chapter 2

Background

2.1 About this chapter

This chapter consists of information on the subject of programming languages and their classification into paradigms such as functional and logic styles. Further on, we talk about the languages used for the implementations in this thesis, namely PROLOG and HASKELL. Moreover, this chapter provides a starting point for various approaches for bringing features of different languages into the same environment.

2.2 Languages and their classification

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 [70] 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. The broader the scope of a language, the broader is the versatility in solving problems.

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 logic style of programming.

Functional programming, [61] 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-calculi [170], a formal system in mathematical logic and computer science for expressing computations based on function abstraction and application using variable binding and substitution. It can be thought as the smallest programming language [112], a single rule and a single function definition scheme. In particular there are typed and untyped λ-calculi. In untyped λ-calculi functions have no predetermined type, whereas typed λ-calculi puts restriction on what sort (type) of data can a function work with. SCHEME is based on the untyped variant, while ML and HASKELL are based on typed λ-calculi. Most typed λ-calculi languages are based on the Hindley-Milner (or Damas-Milner or Damas-Hindley-Milner, [56, 84, 168]) type system. The Hindley-Milner-like type systems have the ability to give a most general type to a program without any annotations. The algorithm [20] works by initially assigning undefined types to all inputs, next check the body of the function for operations that impose type constraints and go on mapping the types of each of the variables, lastly unifying all of the constraints giving the type of the result. This is, in fact, an instance of the unification algorithm that we discuss in a different context in Chapter 10.

Logic programming, [124] on the other hand is based on formal logic. A program is a set of rules and formulæ in symbolic logic that are used to derive new formulas from the old ones. This is done until the one which gives the solution is not derived.

2.3 HASKELL and PROLOG

In this thesis we aim to merge two languages HASKELL and PROLOG together and produce a result which exhibits hybrid properties. The languages in question are HASKELL and PROLOG. These two languages come from the functional programming and logical programming branches of the declarative language group respectively. Some of the dissimilarities between the languages are:

1. HASKELL uses `pattern matching` while PROLOG uses `unification`.

2. HASKELL is all about functions while PROLOG is on Horn clause logic.

PROLOG [160], being one of the most dominant logic programming languages, has spawned a number of distributions and is present from academia to industry.

HASKELL is one the most popular [75] functional languages around and is the first language to incorporate monads [148] for safe input and output. Monads can be described as composable computation descriptions [158]. Each monad consists of a description of what action has to be executed, how the action has to be run and how to combine such computations. An action can describe an impure or side-effecting computation, for example, input and output can be performed outside the language but can be brought together with pure functions inside in a program resulting in a separation and maintaining safety with practicality. HASKELL computes results lazily and is strongly typed.

PROLOG and HASKELL are contrasting in nature, and bringing them into the same environment is tricky. The differences in typing, execution, working among others lead to an altogether mixed bag of properties.

The selection of the target language is not uncommon: PROLOG seems to be the all time favourite for "let's implement PROLOG in the language X for proving its power and expressibility". The PROLOG language has been partially implemented [31] in other languages such as SCHEME [121], LISP [68, 110, 111], JAVA [160, 62], JAVASCRIPT [63] and the list [104] goes on.

2.4 Approaches to integration

The technique of embedding is a shallow one. It is as if the embedded language floats over the host language. Other approaches provide deeper integration between the target and the host. Here we look at a few. Over time there has been an approach that branches out, which is paradigm integration. The First International Symposium on *Unifying the Theories of Programming* ([35]) produced a lot of work [14, 105, 180, 58, 44]. Hybrid languages that have characteristics from more than one paradigm are coming into the mainstream. One of the more successful attempts is SCALA [36]. Simply speaking it is like a *functional* JAVA providing side-effect free programming environment along with JAVA-like features.

Before moving on, let us take a look at some terms related to the content above. To begin with foreign function interfaces (FFI) [169] provide a mechanism by which a program written in one programming language can make use of services written in another programming language. For example, a function written in C can be called within a program written in HASKELL and vice versa through the FFI mechanism. Currently the HASKELL foreign function interface works only for C. Another notable example is the common foreign function interface (CFFI) [12] for LISP which provides fairly complete support for C functions and data. As yet another example, JAVA provides the JAVA native interface (JNI) for the working with other languages. Moreover there are services that provide a common platform for multiple languages to work with each other. They can be termed as multilingual runtimes which lay down a common layer for languages to use each others functions. An example for this is the Microsoft common language runtime (CLR) [167] which is an implementation of the common language infrastructure (CLI) standard [166].

Another important concept is meta programming [172], which involves writing computer programs that write or manipulate other programs. The language used to write meta programs is known as the meta language while the the language in which the program to be modified is written is the object language. Sometimes the meta language and the object language are the same.

HASKELL programs can be modified using Template HASKELL [53], an extension to the language which provides services to jump between the two types of programs. The abstract syntax tree used to define a grammar is in the form of a HASKELL data type. This allows us to play with the code and go back and forth between the meta program and the object program.

A specific tool used in meta programming is quasi-quotation [80, 151, 165], which 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 data type it must 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 code in Figure 2.1 in PROLOG to append two lists, going through

Listing 2.1 Code to append in PROLOG.

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

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 recurses on the tails. Listing 2.2 shows how a direct translation of Listing 2.1 into HASKELL while Listing 2.3 shows the same using quasi quotation.

Consider the object functional programming language, SCALA [36]. It is purely functional but with objects and classes. With the above in mind, coming back to the problem of implementing PROLOG in HASKELL and the possible methodologies to do so, there have been quite a few attempts to "merge" these two programming languages that are from different programming paradigms. The attempts fall into two categories as follows:

1. Embedding, where PROLOG is merely translated to the host language HASKELL or

Listing 2.2 PROLOG append function translated to HASKELL.

```
[ C (Clause (Struct "append"
              [(Struct "[]" []),
               VariableName 0 "X",
               VariableName 0 "X"]) []),
  C (Clause
     (Struct "append"
      [Struct "." [VariableName 0 "X",
                   VariableName 0 "Xs"],
       VariableName 0 "Ys",
       Struct "." [VariableName 0 "X",
                   VariableName 0 "Zs"]])
     [(Struct "append"
      [VariableName 0 "Xs",
       VariableName 0 "Ys",
       VariableName 0 "Zs"])])
]
```

Listing 2.3 PROLOG append function translated to HASKELL using quasi quotation.

```
[ C (Clause [pr| append([], X, X)|] []),
  C (Clause [pr| append([X|Xs], Ys, [X|Zs])|]
     [[pr| append(Xs, Ys, Zs)|]])
]
```

a foreign function interface.

2. Paradigm integration, developing a hybrid programming language that is a functional logic programming language with a set of characteristics derived from both the participating languages.

The approaches of embedding and paradigm integration are discussed in Chapter 5 and Chapter 6 respectively.

Chapter 3

Related Work

This chapter discusses the current implementations and publications for embedding PROLOG in languages and specifically in HASKELL. We take a look at their shortcomings and propose improvements.

3.1 Existing work by others

There have been several attempts at embedding PROLOG into HASKELL, which are discussed below, along with their shortcomings.

1. A total of three embedded implementations exist which offer a starting point for the task of embedding. One of the earliest implementations [64] 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, [181], simplifies the notation of PROLOG literals 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 for the proposed concepts (see [125, 118, 116, 126, 117]).

Moreover, none of them are complete and most lack many practical parts of PROLOG.

3. In the case of libraries, around a dozen exist. Most are not in active development. Almost all only provide a shell through which one must do all the work, which is synonymous with the embeddings mentioned above. Some are more feature rich than others; that is, some have practical PROLOG concepts, but are still not complete.

As for the idea of merging paradigms goes, it is not the main focus of this thesis and can be more of an "add-on". A handful of crossover hybrid languages based on HASKELL exist, CURRY [143] being the prominent one. Moving away from HASKELL and exploring other languages from different paradigms, a respectable number of crossover implementations exist but again most of them have faded out.

3.2 Proposed improvements

As discussed in Section 3.1, either an embedding or an integration approach is taken up for programming languages to work together. So, there is either a shallow approach that does not utilize the constructs available in the host language and results in a translation of the characteristics, or the other is a fairly complex process which results in tackling the conflicting nature of different programming paradigms and languages, resulting in a toned-down compromised language that takes advantages of neither paradigms. Mostly, the trend is towards the former.

From the problems mentioned in the sections above, here is a list that this thesis tackles:

1. The eDSL supports `fails` and `cuts`.

2. the implementation is more complete.

3. The implementation is not Read-Eval-Print Loop (REPL) based. You can write a program file and compile and run it like a normal HASKELL file.

3.3 Summary

This chapter reviewed current work and proposed improvements. The next chapter provides the contributions of this thesis which are not necessarily built upon existing work.

Chapter 4

Other Contributions

Having looked at the existing work for embedding PROLOG in HASKELL along with the suggestions for improvements in the previous chapter, this chapter describes further contributions. The thesis reviews literature on combining two languages as described in Section 4.1. Section 4.2 describes the minimalistic implementations for the ideas presented in this thesis.

4.1 Possible directions for PROLOG in HASKELL

This thesis provides a literature review on embedded eDSLs in Chapter 5 and a survey on multi-paradigm declarative languages in Chapter 6. The current chapter and Chapter 3 evaluate and assess the current work for embedding PROLOG in HASKELL.

This thesis also provides an environment for multiple HASKELL libraries which provide parts of PROLOG-like functionality.

4.2 The prototypes

A large part of this thesis consists of prototypes of a PROLOG-like language in HASKELL. The conceptual advances of these prototypes are:

1. provide a modular methodology to functorize a recursive abstract grammar used to define a language,
2. modular monadic unification to leverage imperative unification algorithms,
3. integrate the procedures mentioned above into a working `Prolog` implementation,
4. support for variable search strategies,
5. approach for embedding IO in an eDSL.

The details of these prototypes follows.

4.2.1 Prototype 1

Most languages have a recursive abstract syntax which restricts the eDSL (eDSL, see p. 3) in terms of its capability to *open up* the language, i.e, to include meta syntactic variables, custom quantifiers and logic. Prototype 1 provides a methodology to convert a language whose recursive abstract syntax is represented by a tree into a non-recursive version whose fixed point is isomorphically equivalent to the original type. One of the outcomes is producing a polymorphically typed embedded language within HASKELL.

To test it out we adopt the closed PROLOG-like language defined in [114] and open it up. As for the unification part we use [137], which provides a generic unification algorithm implementation encapsulated into a monad.

4.2.2 Prototype 2

Prototype 2 does what a PROLOG query resolver would do given a query and a knowledge base. The mechanism for the same is adopted from [114]. The embedded language is modified as per the procedure in Prototype 1 and the monadic unification part is plugged into the existing architecture to demonstrate that it is independent of the other components. Lastly the result is converted into the original language via a translate function.

4.2.3 Prototype 3

Prototype 3 demonstrates the modularity of the unification process of the query resolver with multiple search strategies.

4.2.4 Prototype 4

Prototype 4 throws light on how IO operations can be embedded into the abstract syntax of a DSL which when interpreted would produce output consisting of a pure set of instructions irrespective of the nature of the construct. The effects are produced only when the actions are executed.

The next two chapters are the literature review.

Chapter 5

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 problems in the domain. 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.

5.1 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 content.

A lot has been talked about embedding languages and also the techniques and methods to do so. It might not seem such a hot topic as such but it has always been a part of any programming language to work and integrate their code with other programming languages. One of the top discussions are in, Lambda the Ultimate, The Programming Languages Weblog [71] lists a number of PROLOG implementations in a variety of languages like LISP, SCHEME, SCALA, JAVA, JAVASCRIPT, RACKET [121] and so on. Moreover the discussion focuses on a lot of critical points that should be considered in a translation of PROLOG to the host language in terms of types and modules among others.

One of the implementations discussed redirects us to one of the earliest implementations of PROLOG in HASKELL for `Hugs 98`, called `Mini` PROLOG [64]. Although this implementation aims in the right direction, it is not supported by documentation or literature. It comes with three engines for query resolution, but still lacks practical PROLOG features. This seems to be a common problem with the other implementations, [181]. Other informal discussions of PROLOG have already been mentioned in Chapter 1.

5.2 Literature related to implementing logic programming

Some books related to implementing logic programming are [23, 150, 69]. [23] aims at adding a few constructs to SCHEME to bring together the functional and logic styles of programming and capture the essence of PROLOG in SCHEME. Moreover, [150] claims that HASKELL is suitable for basic logic programming using the `List Monad`. [69] provides a general outlook towards embedding PROLOG in other languages.

Abundant literature can be found on embedding detailed parts of PROLOG features such as basic constructs, search strategies and data types. One of the major works is covered by the subsection below consisting of a series of papers from Mike Spivey and Silvija Seres aimed at bring HASKELL and PROLOG closer to each other. The next subsection covers this literature with improvements and further additions.

5.2.1 Papers from Mike Spivey and Silvija Seres

The work presented in the series [125, 118, 119, 127, 116] attempts to encapsulate various aspects of an embedding of PROLOG in HASKELL. Being one of the very first attempts, the work is influenced by embeddings of PROLOG in other languages like SCHEME and LISP. Although the host language HASKELL has distinct characteristics such as lazy evaluation and a strong type system, the proposed scheme tends to be general as the aim here is to achieve a PROLOG-like eDSL and not a multi-paradigm declarative language. PROLOG `predicates` are translated to HASKELL functions which produce a stream of results lazily depicting depth first search with support for different strategies and practical operators such as `cut` and `fail` with higher order functions. The papers provide a minimalistic extension to HASKELL with only four new constructs. Though no implementation exists, the synthesis and transformation techniques for functional programs have been *logicalised* and applied to PROLOG programs. PROLOG is based on relations and HASKELL is based on functions; [126] takes this into consideration and attempts to model relations using functions.

5.2.2 Other works related or based on the above

Continuing from above, [19] taps into the advantages of the host language to embed a typed functional logic programming language. This results in typed logical predicates and a backtracking `monad` with support for various data types and search strategies. Though not very efficient or practical, the method aims at a more elegant translation of programs from one language to the other. Publications such as [37] attempt to exercise HASKELL features without adding any new constructs. It uses HASKELL's type class to express a general structure for the problem while its instances represent the solutions. [57] replicates PROLOG's control operations in HASKELL suggesting the use of the HASKELL `State`

`Monad` to capture and maintain a global state. The main contributions are a Backtracking Monad Transformer that can enrich any `monad` with backtracking abilities and a `monadic encapsulation` to turn a PROLOG predicate into a HASKELL function.

5.3 Related libraries in HASKELL

5.3.1 PROLOG libraries

To replicate PROLOG-like capabilities HASKELL seems to be a popular choice with a host of related libraries. First we begin with the libraries about PROLOG itself. A few exist. [133] is a preliminary or "mini PROLOG" with not much in it to be able to be useful. [134] is all powerful, but is a foreign function interface so it is "PROLOG in HASKELL" but we need PROLOG for it; [114] is the implementation that comes the closest to something like an actual practical PROLOG. All these libraries provide is a small interpreter, none or a few practical features, incomplete support for lists, minor or no `monadic` support and a REPL without the ability to "write a PROLOG program file".

5.3.2 Logic libraries

The next category is about the logical aspects of PROLOG; again a handful of libraries provide functionality related to propositional logic and backtracking. [32] is a continuation-based, backtracking, logic programming `monad` which attempts to replicate PROLOG's backtracking behaviour. PROLOG is heavily based on formal logic. [42] provides a powerful system for propositional logic. Other libraries include small hybrid languages [38] and "Parallelising Logic Programming and Tree Exploration" [22].

5.3.3 Unification libraries

HASKELL has minimal support for unification. There are two libraries ([106, 137]) that unify two terms and return the resulting substitution.

5.3.4 Backtracking libraries

Another important aspect of PROLOG is backtracking. To simulate it in HASKELL, the libraries [39, 123] use `monads`. Moreover, there is a package for the EGISON programming language [59] which supports non-linear pattern-matching with backtracking.

5.4 Summary

Recapitulating, this chapter surveys the approach to embedding programming languages, especially PROLOG in HASKELL. Moreover, it describes tools, some of which are utilized in the prototype implementations.

Chapter 6

Multi-Paradigm Languages (Functional Logic Languages)

This chapter describes the approach for integrating properties of multiple languages from different programming paradigms, especially, functional logic programming languages.

6.1 Multi-paradigm languages

Over the years another approach has branched off from embedding languages, merging and integrating programming languages from different paradigms. Let us take the SCALA programming language [36] as an example. It is a hybrid object-functional programming language which takes features from each of the two paradigms. This section looks at the literature on multi-paradigm languages, mainly functional logic programming languages. This is of interest because in this thesis, the languages in question are HASKELL and PROLOG which are of functional and logic programming paradigms respectively.

A peak into language classification reveals that it is not always a straight forward task to segregate languages according to their features and characteristics. It turns out that there are a number of notions which play a role in deciding where the language belongs. Often a language ends up being a part of almost all paradigms due to extensive libraries.

Simply speaking, a multi-paradigm programming language is a programming language that supports more than one programming paradigm [70]. Moreover, as Timothy Budd puts it [174] "The idea of a multi-paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms."

6.2 The informal content from blogs, articles and internet discussions

6.2.1 Multi-paradigm languages

A lot has been worked upon before coming to clear grounds about the classification of programming languages. One approach is to consider that the scope of each language is pretty much infinite, as small extension modules that replicate different feature sets not naturally native to the language itself can be implemented on top of the base language. The descriptions of multi-paradigm languages across the web [174, 92, 15] converge to roughly of a framework providing tools to work with different styles of programming [171, 30]. Generally speaking, the above approach to language classification is not very popular in programming circles; one reason is that this approach fails to identify the essence of a language.

6.2.2 Functional logic programming languages

Continuing from the previous section, we narrow down the scope by considering only multi-paradigm declarative languages, i.e., functional logical programming languages. By doing so a large amount of information pops up, from articles that give brief descriptions [164, 161] to articles that provide implementation techniques (for instance [3], which also gives an overview of implementing functional logic languages and a list of publications).

An important resource for this topic is [49] which is maintained by Michael Hanus [47] and Sergio Antoy [4] who developed CURRY [50].

6.3 Literature

6.3.1 Multi-paradigm languages

Possibly one of the most important works towards bringing programming styles together is the book [58] by C.A.R. Hoare which points out that among the large number of programming paradigms and theories, the unification theory serves as a complement rather than a replacement for classification. Since we are talking about HASKELL we should mention [44], which includes monads and unifying theories using monads.

6.3.2 Functional logic programming languages

A recent survey [48] throws light on this category of hybrid languages.

One of the most prominent multi-paradigm languages in HASKELL is CURRY [5]. The syntax is borrowed from the parent language and so are a lot of the features. Recapitulating, a functional programming language works on the notion of mathematical functions while a logic programming language is based on predicate logic. The strong points of CURRY are that the features of the language are general, and are visible in a number of languages like [25]. The language can play with problems from both worlds. In a problem where there are no unknowns (variables) the language behaves like a functional language which is pattern matching the rules and executing the respective bodies. In the case of missing information, it behaves like PROLOG: a sub-expression e is evaluated on the conditions that it should satisfy, which constrain the possible values of e. This brings us to the first important feature of functional logic languages, *narrowing*. In narrowing, the expressions contain *free variables*, simply speaking incomplete information that needs to be *unified* to a value depending on the constraints of the problem. The language introduces only a few new constructs to support non-determinism and choice. Firstly, narrowing (=:=), which deals with the expressions and unknown values and binds them with appropriate values. The next one is the *choice* operator (?) for non-deterministic operations. Lastly, for unifying variables and values under some conditions, (&) operator has been provided to add constraints to the equation. Putting it all together, it gives us the feel of a logic language for something that looks very much like HASKELL. Unification is like two-way pattern matching, and by a similar analogy CURRY is a HASKELL that works both ways and hence variables can be on either side of an assignment. Although the language can do a lot; gaps do exist, such as the need to improve narrowing techniques.

6.4 Some multi-paradigm languages

The list of multi-paradigm languages is huge, but in this thesis we will mostly stick to functional logic programming languages. Beginning with functional hybrids, a small project language called VIRGIL, [142], combines objects to work with functions and procedures. On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [162]. Combining objects with functions is important as object-oriented programming has become one of the most common styles of programming. As object-oriented programming has been a dominant style of programming, even HASKELL has one called O'HASKELL [93], though it last saw a release back in 2001. Another prominent language is OCAML [173, 98], which adds object-oriented capabilities to a powerful type system and module support. Many of the multi-paradigm languages were developed as a proof of concept and are replaced by new ones based on different approaches. As mentioned before, one of the most popular (see [75]) and widely used both in academia and industry is the SCALA [36] programming language.

6.5 Some functional logic programming languages

Knowing the amount of literature on these type of languages, it is fairly easy to say that there have been numerous attempts at specification and implementation. Sadly though, not many have survived. Only the ones that (a) are easily available, or (b) have an implementation, or (c) have been cited or referred by other attempts have been included below. We begin with variants of PROLOG. As of now, there have been three popular ones, beginning with NUE PROLOG, [77], OZ (MOZART PROGRAMMING SYSTEM) [21] and MERCURY [27]. These languages represent themselves as extensions of PROLOG rather than hybrids. To start with MERCURY represents a boundary between deterministic and non-deterministic programs. Similarly, NUE PROLOG has special support for functions, while OZ provides concurrent constraint programming plus distributed support, with different function types for goal solving and expression rewriting. ESCHER [78] comes very close to HASKELL with monads, higher order functions and lazy evaluation. We take a look at PROLOG variants: CIAO ([18]) is a preprocessor to PROLOG for functional syntax support; λ PROLOG ([90]) aims at modular higher order programming with abstract data types in a logical setting; BABEL ([54, 86, 85]) combines pure PROLOG with a first order functional notation; LIFE ([141]) is for Logic, Inheritance, Functions and Equations in PROLOG syntax with currying and other features like functional languages; and there are others ([11, 81]).

The functional language SCHEME is a very popular choice for adding logic programming functionality to a functional language as described in the book [23] along with the accompanying implementation [24, 136]. [89, 40, 147] provide a similar approach, but for HASKELL.

Finally talking about CURRY, one of the most popular HASKELL based multi-paradigm languages with support for deterministic and non-deterministic computations. Contributing to the same there have been some predecessors [139, 25].

6.6 Summary

Recapitulating, this chapter surveys the approach to merging different programming paradigms to result in a hybrid programming language. Moreover, we talked about multi-paradigm declarative languages along with PROLOG and HASKELL hybrids respectively.

Chapter 7

HASKELL

This chapter discusses HASKELL as a functional language and its features which assist in embedding DSLs. HASKELL as a functional programming language, is an advanced purely-functional programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language [156]. It is one of the popular functional programming languages [75]. HASKELL is widely used in the industry [159].

7.1 Functional programming languages

Functional programming revolves around the concept of functions being applied to arguments to get results. In functional programming functions are first class citizens and a main program itself is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. Programs contain no assignment statements, so variables, once given a value, never change and hence contain no side-effects [61].

7.2 Embedded domain specific languages (eDSLs)

Shifting a bit to 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 such as 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 [65, 83] which talk about introducing combinator libraries for custom functionality.

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

7.3 Monads

Control flow defines the order/ manner of execution of statements in a program [176]. The specification is set by the programming language. Generally, in the case of imperative languages the control flow is sequential while for a functional language is recursion [140]. For example, JAVA has a top down sequential execution approach. The declarative style consists of defining components of programs i.e., computations not a control flow [177].

This is where HASKELL shines by providing something called a *monad*. Functional programming languages define computations which then need to be ordered in some way to form a combination [152]. A monad gives a bubble within the language to allow modification of control flow without affecting the rest of the universe. This is especially useful while handling side effects.

A related topic would be of persistent languages, architectures and data structures. Persistent programming is concerned with creating and manipulating data in a manner that is independent of its lifetime [87]. A persistent data structure supports access to multiple versions which may arise after modifications [34, 67]. A structure is partially persistent if all versions can be accessed but only the current can be modified and fully persistent if all of them can be modified.

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

7.4 Monads by example: state monad

In this section we try and replicate a dictionary of variables. The operations such as adding and removing entries to and from the dictionary are implemented so as to replicate modification. Listing 7.1 shows the structure of the `IntegerDictionary` for storing and modifying the `Variables`. Each `Variable` is a `VariableName`, `Value` pair.

Listing 7.1 HASKELL Monad Working: Data Types

```haskell
type VariableName      = String
type Value             = Int
data Variable          = Variable (VariableName, Value)
                         deriving (Eq)
data IntegerDictionary = ID [Variable]

(<--) = curry Variable
infix 8 <--

init_dictionary :: IntegerDictionary
init_dictionary = ID [
  ("x0" <-- 0), ("x1" <-- 1),
  ("x2" <-- 2), ("x3" <-- 3),
  ("x4" <-- 4), ("x5" <-- 5)     ]
```

Listing 7.2 shows the insertion and removal in a variable dictionary and the `run` function for applying the operation on the dictionary.

Listing 7.2 HASKELL Monad Working: Functions

```
variableName :: Variable -> VariableName
variableName (Variable (v,_)) = v

vNameEqual :: Variable -> Variable -> Bool
vNameEqual = (==) 'on' variableName where
  (f2 'on' f1) a b = f2 (f1 a) (f1 b)

insertVariable :: Variable -> State IntegerDictionary Variable
insertVariable variable = do
   Control.Monad.State.modify (insertVariableHelper variable)
   return variable

insertVariableHelper :: Variable -> IntegerDictionary -> IntegerDictionary
insertVariableHelper variable (ID dictionary) =
  ID (variable : filter (not . (vNameEqual variable)) dictionary)

runInsertVariable :: IntegerDictionary -> Variable -> IntegerDictionary
runInsertVariable init_dictionary variable = snd $
    runState (insertVariable variable) init_dictionary

removeVariable :: Variable -> State IntegerDictionary Variable
removeVariable variable = do
  Control.Monad.State.modify (removeVariableHelper variable)
  return variable

removeVariableHelper :: Variable -> IntegerDictionary -> IntegerDictionary
removeVariableHelper variable (ID dictionary) =
    ID $ filter (not . (vNameEqual variable)) dictionary

runRemoveVariable :: IntegerDictionary -> Variable -> IntegerDictionary
runRemoveVariable init_dictionary variable = snd $ runState (removeVariable
  variable) init_dictionary

extractVariableValue :: Variable -> Value
extractVariableValue (Variable (_, value)) = value
```

Listing 7.3 shows a combination of the operation of finding a product of the two variables and storing back the result in the dictionary.

Listing 7.3 HASKELL Monad Working: Examples

```haskell
exampleOperation :: Variable -> Variable -> State IntegerDictionary Variable
exampleOperation variableX variableY = do
   insertVariable variableX
   let vx = extractVariableValue variableX
   insertVariable variableY
   let vy = extractVariableValue variableY
       product = Variable ("product", vx * vy)
   insertVariable product
   return product

runExampleOperation :: IntegerDictionary -> Variable -> Variable ->
   IntegerDictionary
runExampleOperation init_dictionary variableX variableY = snd $ runState (
   exampleOperation variableX variableY) init_dictionary
```

Listing 7.4 shows the output of `runExampleOperation` function.

Listing 7.4 HASKELL Monad Working: Example output

```haskell
runExampleOperation init_dictionary ("x" <-- 10) ("y" <-- 20)
-- output
{[product <-- 200,y <-- 20,x <-- 10,x0 <-- 0,x1 <-- 1,x2 <-- 2,
    x3 <-- 3,x4 <-- 4,x5 <-- 5]}
```

7.5 Lazy evaluation

Another property of HASKELL is laziness or lazy evaluation which means that nothing is evaluated until it is necessary. This results in the ability to define infinite data structures because at execution only a fragment is used [157].

Consider the infinite list example:

```haskell
let x = 1:x in x
```

results in:

```
1 : 1 : 1 . . .
```

Lazy evaluation is part of operational semantics, i.e., how a HASKELL program is evaluated. This semantics allows one to bypass undefined values (e.g., results of infinite loops) and in this way it also allows one to process formally infinite data.

7.6 Quasiquotation and HASKELL

7.6.1 Quasiquotation

Quotation is a device for exactly specifying some text [55]. Thus “not p” refers to the expression consisting of the word not followed by the letter p. Quasi-quotation, or Quine quotation, is a metalinguistic device for referring to the form of an expression containing variables without referring to the symbols for those variables. Thus $\ulcorner \text{not}\, p \urcorner$ refers to the form of any expression consisting of the word not followed by any value of the variable p [29]. The variable p is sometimes called a meta-syntactic variable. Quasi-quotation facilitates rigorous but terse formulation of general rules about expressions [165].

7.6.2 Quasiquotaion in HASKELL

Quasiquoting allows programmers to use custom, domain specific syntax to construct fragments of their program. Along with HASKELL’s existing support for DSLs, you are now free to use new syntactic forms for your eDSLs. Working with complex data types can impose a significant syntactic burden; extensive applications of nested data constructors are often required to build values of a given data type, or, worse yet, to pattern match against values. Quasiquotation allows HASKELL expressions and patterns to be constructed using domain specific, programmer-defined concrete syntax [151, 80]. Listing 2.3 shows us the advantages of quasi quotation (in this example there are no metasynctactic variables).

Chapter 8

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. 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* [160]. PROLOG is very popular and has a number of implementations [175] for different purposes. PROLOG comes from the same family as HASKELL i.e., the declarative paradigm. One of the reasons for selecting these languages is that both HASKELL, the base language and PROLOG, the target language are from the same paradigm. This provides a platform to play around with the conflicting characteristics of the two languages. PROLOG seems to be a very popular choice as a target language. Also for the specific topic of embedding PROLOG in HASKELL, implementations and publications exist which provide a starting point.

8.1 Syntax

PROLOG is dynamically typed. It has a single data type, the term, which has several subtypes: atoms, numbers, variables and compound terms [160].

An atom is a general-purpose name with no inherent meaning. It is composed of a sequence of characters that is parsed by the PROLOG reader as a single unit.

Numbers can be floats or integers. Many PROLOG implementations also provide unbounded integers and rational numbers.

Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms. A variable can become instantiated (bound to equal a specific term) via unification.

A compound term is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero.

A PROLOG program is a description of relations, defined by the use of clauses. Pure PROLOG is restricted to Horn clauses, a Turing-complete subset of first-order predicate logic. The clauses can be one of two types: facts and rules [96].

8.2 Semantics

Since the commutative nature of logical disjunction and conjunction, declaratively speaking the order of rules and their sub goals is irrelevant. However, the procedural aspect must be taken into account to determine the execution strategy of PROLOG since it has to deal with impure predicates. Moreover, a particular order of execution can lead to infinite recursion.

8.3 Universal Horn clauses

A Horn clause is a logical formula of a particular rule-like form which gives it useful properties for use in logic programming, formal specification, and model theory [13].

A literal is an atomic formula or its negation. A clause is a disjunction of literals. A Horn clause is a clause with exactly one positive literal. A Horn formula is a conjunctive normal form formula whose clauses are all Horn [76].

Consider the clauses in Listing 8.1:

Listing 8.1 PROLOG clause

```
c:- a, b.
a.
b.
```

and the Horn clause equivalent:

$[c \vee \neg a \vee \neg b] \wedge a \wedge b$

8.4 Unification

To replicate PROLOG we look into how it works (see, for instance, [132]). As any other language we start with the syntax and semantics. We begin with the programming constructs of the language.

PROLOG has three types of terms: constants, variables and complex terms.

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

The possibilities are:

1. If *term1* and *term2* are constants, then *term1* and *term2* unify if and only if they are the same atom, or the same number. Consider the example in Listing 8.2

Listing 8.2 Unification with constants.

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

2. If *term1* is an uninstantiated variable and *term2* is any type of term, then *term1* and *term2* unify, and *term1* is instantiated to *term2*. Similarly, if *term2* is a variable and *term1* is any type of term, then *term1* and *term2* unify, and *term2* is instantiated to *term1*. Consider the examples in Listing 8.3 and Listing 8.4. (So if they are both variables, they're both instantiated to each other, and we say that they share values.)

Listing 8.3 Unification with a single variable.

```
?-  mia  =  X.
X  =  mia
yes
```

Listing 8.4 Unification with variables.

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

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

   (a) they have the same functor and arity, and

   (b) all their corresponding arguments unify, and

   (c) the variable instantiations are compatible.

   Consider the example in Listing 8.5.

Listing 8.5 Unification of complex terms.

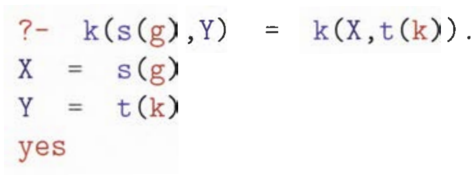
```
?-  k(s(g),Y)  =  k(X,t(k)).
X  =  s(g)
Y  =  t(k)
yes
```

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

Unification is just a part of the process where the language attempts to find a solution for the given query using the rules provided in the knowledge base. The other part (searching) is to reach a point where two terms are required to be unified. Together they form the query resolver in PROLOG.

For example, consider the append function shown in Listing 8.6.

Listing 8.6 append function in PROLOG



```
?-  k(s(g),Y)  =  k(X,t(k)).
X  =  s(g)
Y  =  t(k)
yes
```

whose operation is illustrated in Figure 8.1.

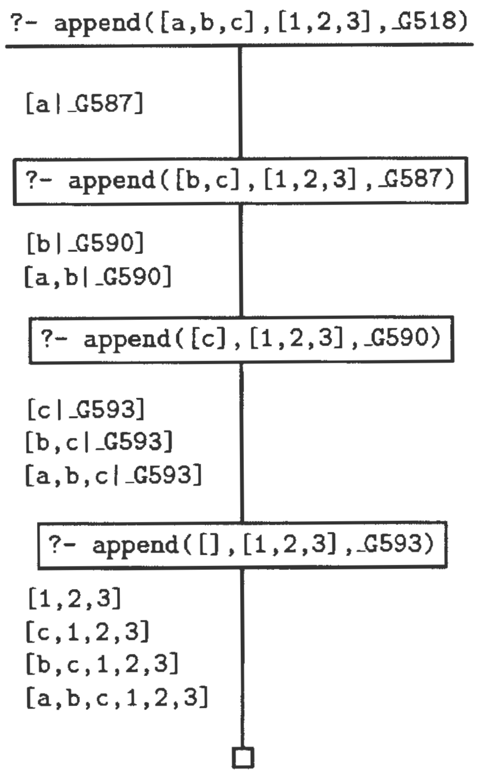

?- append([a,b,c],[1,2,3],_G518)
[a|_G587]
?- append([b,c],[1,2,3],_G587)
[b|_G590]
[a,b|_G590]
?- append([c],[1,2,3],_G590)
[c|_G593]
[b,c|_G593]
[a,b,c|_G593]
?- append([],[1,2,3],_G593)
[1,2,3]
[c,1,2,3]
[b,c,1,2,3]
[a,b,c,1,2,3]

Figure 8.1: Trace for append [131]

8.5 The execution models of PROLOG

The description of how PROLOG relates to logic programming is paraphrased from Chapter 6 of [130]. Logic programming languages are adapted from abstract interpreters for logic programs. To implement a logic programming language such as PROLOG two major decisions about the resolver must be taken:

1. Scheduling policy: A scheduling policy defines how the additions and deletions of goals from the resolvent is performed by the interpreter. For instance, PROLOG adopts a stack scheduling policy. The resolvent is maintained as a stack which involves popping the topmost goal for reduction and pushing derived goals back.

2. Search strategy: Most problem solving systems are built by state-space search to

obtain desired solutions and involve problems in selecting possible alternatives while searching through a solution space. [97]. For instance, PROLOG simulates the non-deterministic choice of reducing clause by sequential search and backtracking. The first goal whose head unifies with the goal is chosen. If no match is found then the computation is unwound to the last choice point and the next unifiable clause is chosen.

For terminating queries PROLOG generates all possible solutions of the goal with respect to the PROLOG program. It performs a complete depth first traversal of a particular search tree for the goal by always choosing the leftmost goal. Listing 8.8 shows a sample trace for a query using the knowledge base from Listing 8.7.

A popular implementation of PROLOG is 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.

Listing 8.7 Tracing a simple PROLOG computation [130] Clauses

```
father(abraham,isaac). male(isaac).
father(haran,lot). male(lot).
father(haran,milcah). female(yiscah).
father(haran,yiscah). female(milcah).
son(X,Y) :- father(Y,X), male(X).
daughter(X,Y) :- father(Y,X), female(X).
```

Listing 8.8 Tracing a simple PROLOG computation [130] : Output

```
son(X,haran)?
        father(haran,X)                                     X=lot
        male(lot)
                        true
                Output: X=lot
                        ;
father(haran,x)                                             X=milcah
male(milcah)         f
father(haran,X)                                             X=yiscah
male(yiscah)         f
                        no (more) solutions
```

8.6 cuts in PROLOG

Consider the example in Listing 8.9. A sample query p(X) will result in 8.10.

Listing 8.9 A cut-free PROLOG computation [99]

```
p(X) :- a(X).
p(X) :- b(X),c(X),d(X),e(X).
p(X) :- f(X).
a(1).
b(1).
c(1).
b(2).
c(2).
d(2).
e(2).
f(3).
```

Listing 8.10 cut-free PROLOG computation output[99]

```
X = 1 ;
X = 2 ;
X = 3 ;
no
```

A `cut` represented as `!` operator [179]. If PROLOG finds a `cut` in a rule, it will not backtrack on the choices it has made. Consider the example below:

```
p(X) :- b(X),c(X),!,d(X),e(X).
```

The result for a sample query `p(X)`:

```
X = 1 ; no
```

Only a single solution is obtained because the `cut` prevents backtracking. The Figure 8.2 shows the trace for the query with the `cut` operator.

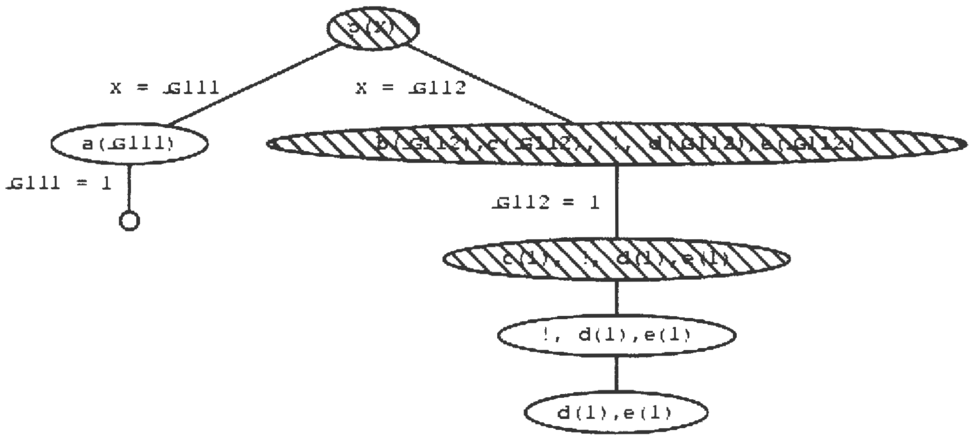


Figure 8.2: Trace with `cut` (taken from [99])

Recapitulating, this chapter provided information on PROLOG as a logic programming language.

Chapter 9

Related Concepts

This chapter discusses concepts which are related to the work presented in this thesis which may not bear a direct point of contact but contribute to understanding.

9.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [26].

In HASKELL it is implemented using the `map` and `fold` functions. `map` takes as arguments: an operation and a list of values, and applies the operation to each element in the list. The `map` operation is as follows:

```
map (+3) [1,5,3,1,6]
```

and results in

```
[4,8,6,4,9]
```

`fold` takes as arguments: an operation, an accumulator and a list, and applies the operation on the accumulator and each element one at a time till the end of the list and returns the final accumulator.

The `fold` operation is as follows:

```
foldr (+) 0 [4,8,6,4,9]
```

and results in,

```
31
```

9.2 Type systems

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

The advantages of static typing [82] are :

- earlier detection of errors,
- better documentation in terms of type signatures,
- more opportunities for compiler optimizations,
- increased run-time efficiency, and
- better developer tools;

whereas for dynamic typing the advantages are:

- less rigidity,
- suitability, and
- re-usability.

9.3 Residuation and narrowing

Lastly some details on the working of functional logic programming languages, residuation and narrowing [50, 163]. Residuation involves delaying of function calls until they are deterministic, that is, deterministic reduction of functions with partial data. This principle is used in languages like ESCHER [78], LIFE [141], NUE-PROLOG [77] and OZ [21]. Narrowing on the other hand is a mixture of reduction in functional languages and unification in logic languages. In narrowing, a variable is bound a value within the specified constraints and try to find a solution, values are generated while searching rather than just for testing. The languages based on this approach are ALF [139], BABEL [54], LPG [11] and CURRY [143].

Chapter 10

Prototype 1

This chapter demonstrates a fairly generic procedure of creating an open eDSL in HASKELL along with *monadic unification*. As a proof of concept, the implementation consists of creating a PROLOG-like open language whose unification procedure is carried out in a monad.

10.1 Ingredients

In this chapter we work with four pieces of software to develop a working implementation of embedded PROLOG. These are:

1. PROLOG

   The language itself has a number of sub components. The ones relevant to this thesis are:

   (a) Language, the syntax, semantics.

   (b) Database, or the knowledge base where the rules are stored.

   (c) Unification

   (d) PROLOG has to satisfy a list of goals while maintaining variable bindings and choice points. For a non empty list of list of goals, it looks through the knowledge base for matching rules and attempts at unifying the terms and repeats until all goals have been satisfied. If more than one option is available, they are recorded as choice points which are later used for backtracking and finding other possible solutions.

   (e) Lastly, the query resolver which combines the unification and search strategy to return a result.

2. `prolog-0.2.0.1` [114]

   One of the existing implementation of PROLOG in HASKELL, though partial, provides a starting point for the implementation as it provides certain components to test our approach. The main components of this library are adopted from PROLOG and modified. These are:

   (a) the language, adopted from PROLOG but trimmed down;

   (b) the database;

   (c) the unifier;

   (d) the REPL;

   (e) the interpreter which consists of a parsing mechanism and resembles the query resolver.

3. `unification-fd` [137]

   This library provides tools for first-order structural unification over general structure types along with mechanisms for a modifiable generic unification algorithm implementation.

   The relevant components are:

   (a) the `Unifiable` class;

   (b) the `UTerm` data type;

   (c) variable implementations `STVar` and `IntVar`;

   (d) the `Binding` monad; and

   (e) Unification (`unify` and `unifyOccurs`).

4. Prototype 1

   This implementation applies to practice the procedure to create an open language to accommodate types, custom variables, quantifiers and logic and recovering primitives while preserving the structure of a language commonly defined by a recursive abstract syntax tree. The resulting language is then adapted to apply a PROLOG-like unification.

   The implementation consists of the following components:

   (a) an open language,

   (b) compatibility with the unification library [137],

   (c) variable bindings, and

   (d) monadic unification.

Each of the components are discussed in the following sections.

10.2 Prototype architecture

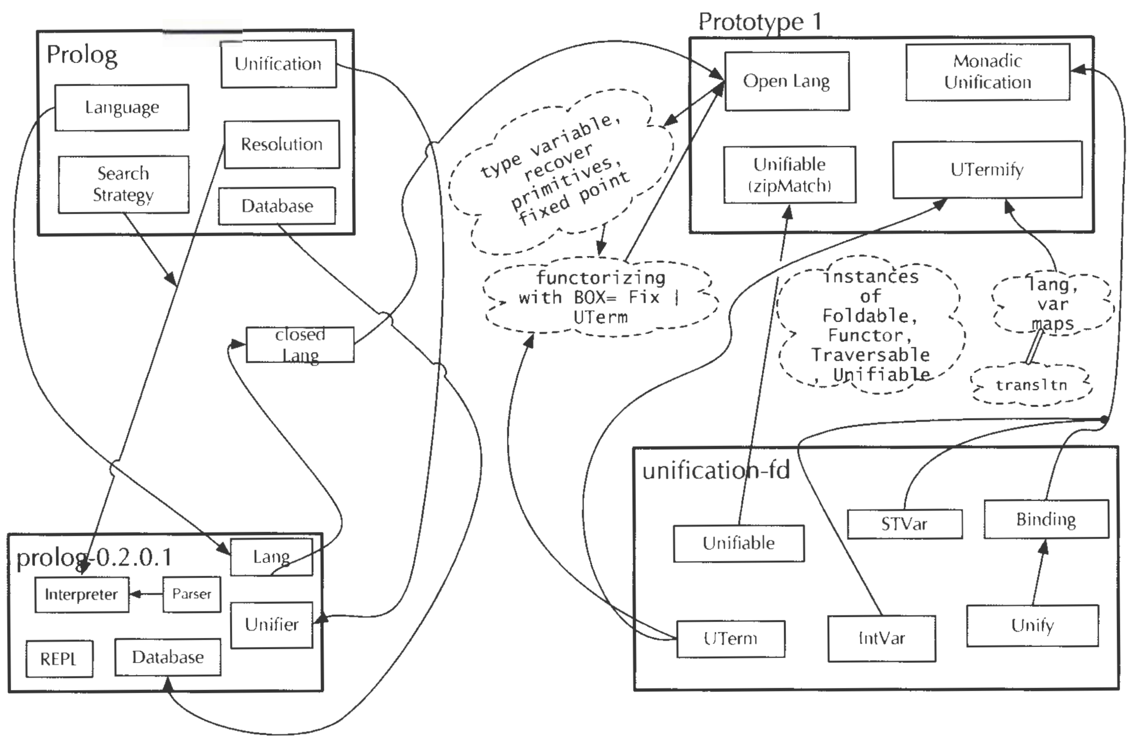


Figure 10.1: Architecture of Prototype 1

Chapter 8 provides a general description of the working of PROLOG. In this prototype we explore the unification aspect only.

This prototype demonstrates the process of creating an isomorphic data type to replicate the target language type system while conforming to the host language.

We create a PROLOG-like language using a recursive abstract grammar in HASKELL using HASKELL's `data` statement. We then convert it to a non-recursive version whose fixed point is isomorphically equivalent to the original. This gives us a more open implementation which will be discussed in the sections to come.

The rest of the procedure includes managing library compatibility for the language and, more importantly, monadic unification.

10.3 Creating a data type

To start we need to define an abstract syntax for a PROLOG-like language. Consider the language in Listing 10.1, which has been adopted from [114].

Listing 10.1 A classic recursive grammar

```
data VariableName = VariableName Int String
          deriving (Eq, Data, Typeable, Ord)

data Atom = Atom !String | Operator !String
          deriving (Eq, Ord, Data, Typeable)

data Term = Struct Atom [Term]
            | Var VariableName
            | Wildcard
            | Cut Int
          deriving (Eq, Data, Typeable)
```

Even though *Term* has a number of constructors the resulting construct has a single type. Hence, a binary function would have type

```
foo :: Term -> Term -> Term
```

Listing 10.1 is a classic example of using a recursive data type to define the abstract syntax of a language. One of the issues with the datatype in Listing 10.1 is that it is not possible to distinguish the structure of the data from the data type itself [120]. Moreover, the primitives of the language (see [8]) are not accessible, as the language can have expressions of only one type, "`Term`".

Consider the code in Listing 10.2.

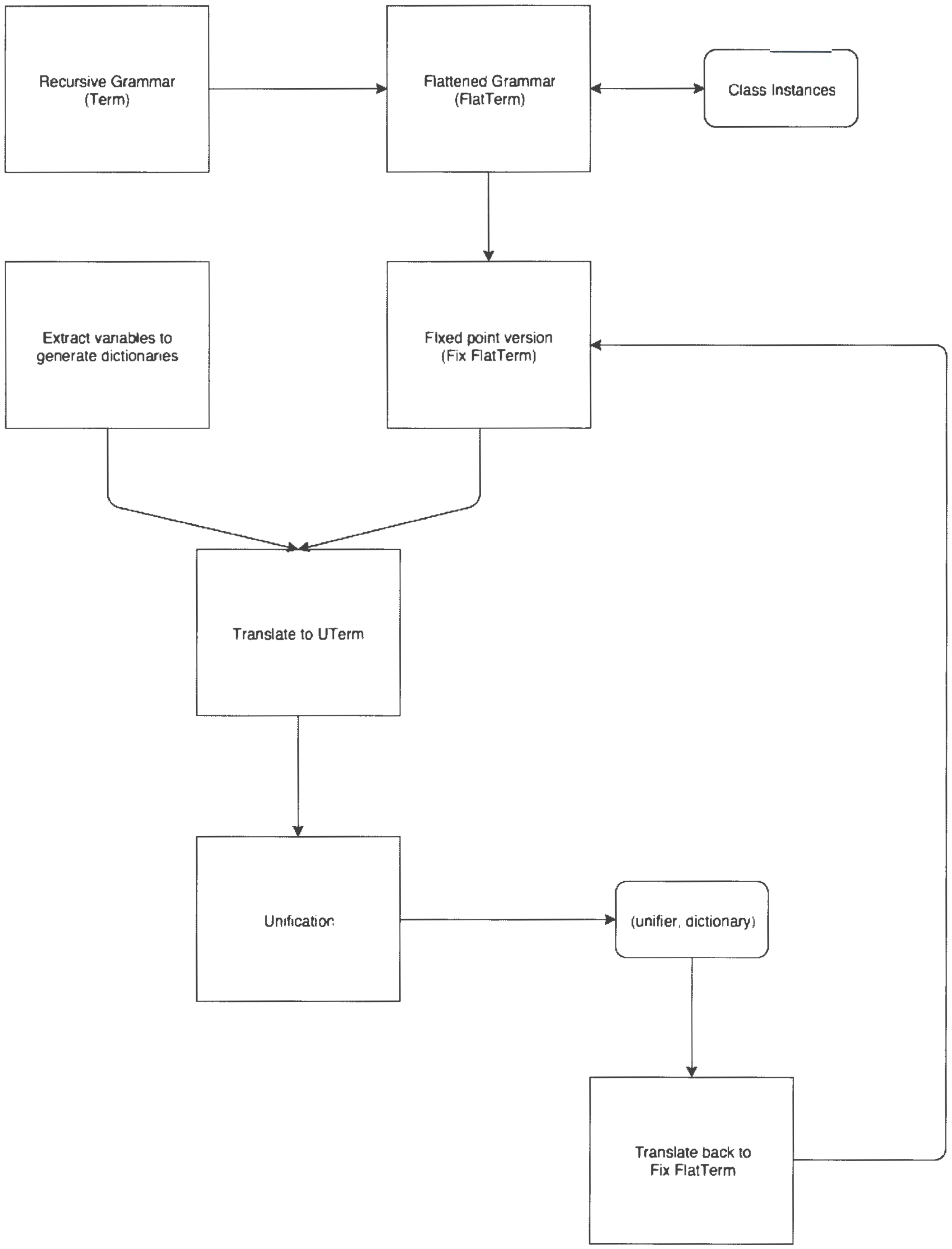

Recursive Grammar (Term)
Flattened Grammar (FlatTerm)
Class Instances
Extract variables to generate dictionaries
Fixed point version (Fix FlatTerm)
Translate to UTerm
Unification
(unifier, dictionary)
Translate back to Fix FlatTerm

Figure 10.2: Prototype 1 Implementation architecture

Listing 10.2 A flattened (non-recursive) grammar

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

One result of coding as in Listing 10.2 rather than as in Listing 10.1 (lines 6–9) is that the non-recursive type `FlatTerm` is modular and generic as the structure `FlatTerm` is separate from the type of its subterms which is a. The above language can be of any type `FlatTerm` a where a can be of any type at all. A more accurate way of saying it would be that a can be any *kind* in HASKELL.

In type theory, a kind is the type of a type constructor or, less commonly, the type of a higher-order type operator. A kind system is essentially a simply-typed λ-calculus 'one level up', endowed with a primitive type, denoted $*$ and called 'type', which is the kind of any (monomorphic) data type (see [153]). Listing 10.3 describes kinds in HASKELL.

Listing 10.3 `kinds` in HASKELL

```
Int :: *
Maybe :: * -> *
Maybe Bool :: *
a -> a :: *
[] :: * -> *
(->) :: * -> * -> *
```

Simply speaking we can have something like

```
FlatTerm Bool
```

and a generic function like,

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

One problem remains: how does one represent deep expressions of the above language, for example something of the form,

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

and how to represent it generically to perform operations on it, since

```
(FlatTerm a) != (FlatTerm (FlatTerm a))
```

because with our original grammar all the expression that could be defined would be represented by a single entity `Term`, no matter how deep they were.

The approach to tackling this problem is to find the "fixed-point" of `FlatTerm`. After infinitely many iterations we should get to a fixed point where further iterations make no difference. It means that applying one more `FlatTerm` would not change anything—HASKELL provides fixed-points in two forms, one for data and one for types.

In type constructor form,

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

which we apply to our abstract syntax.

The resulting language is of the form,

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

simply speaking all the expressions resulting from `FlatTerm` can be represented by the type signature `Fix FlatTerm`.

A sample function working with such expressions would be of the form,

```
func :: Fix FlatTerm -> Fix FlatTerm
```

Generically speaking, the language can be expanded for additional functionality without changing or modifying the base structure. Consider the scenario where the language needs to accommodate additional type of terms. There are two approaches:

1. Manually modify the structure of the language, as shown in Listing 10.4.

Listing 10.4 A manually enhanced recursive grammar

```
type Atom              = String

data VariableName      = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)

data Term = Struct Atom [Term]
            | Var VariableName
            | Wildcard
            | Cut Int
            | New_Constructor_1 .........
            | New_Constructor_2 .........
      deriving (Eq, Data, Typeable)
```

   This would then trigger a ripple effect throughout the architecture because accommodations need to be made for the new functionality.

2. The other option would be to *functorize* language like we did by adding a type variable which can be used to plug something that provides the functionality into the language. Since we needed the fixed point of the language we used `Fix` but generically one could add custom functionality. For instance, using `Extended` from Listing 10.5 we have `Extended FlatTerm` that is isomorphic to the type defined in Listing 10.4.

Listing 10.5 The `Extended` type constructor

```
data Extended f = New_Constructor_1 ...
                | New_Constructor_2 ....
                | Base (f (Extended f))
```

Figure 10.3 and Figure 10.4 show the extension of the example in Listing 10.1 using the manual and functorized approach respectively.

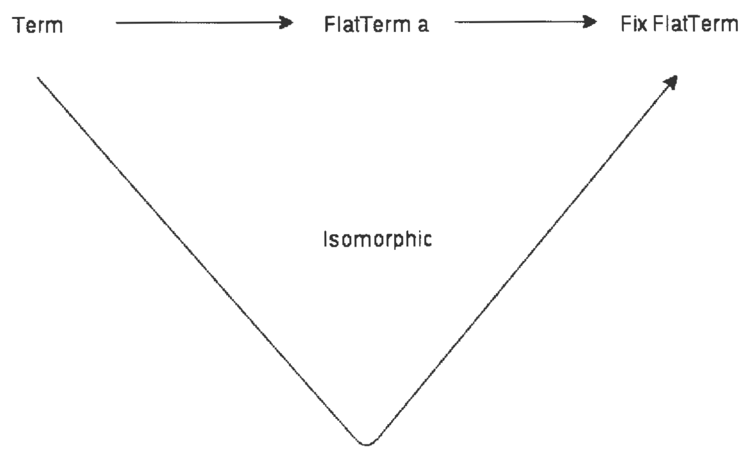


Figure 10.3: Manually Extension of data type

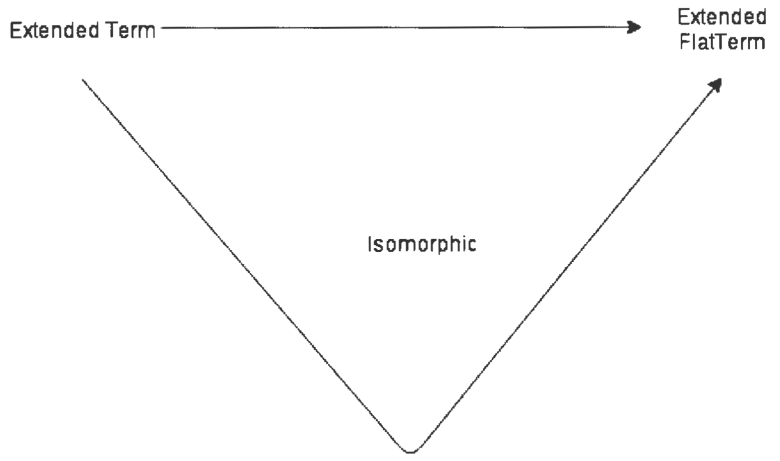


Figure 10.4: Automatic Extension of data type

10.4 Making the language compatible with `unification-fd`

Our language is now opened up and ready for expansion, but it still needs to conform to the requirements of the `unifiication-fd` library ([137]) for the unification algorithm to work.

The library provides functionality for first-order structural unification over general structure types along with mutable variable bindings.

In this section we discuss

1. Functor hierarchy.
2. Required instances the language must have.
3. Mutable variables.
4. Variable bindings.
5. Monadic unification.
6. Replicating PROLOG unification in HASKELL

Classes in HASKELL are like containers with certain properties which can be thought of as functions. When a data type creates an instance of a class the function(s) can be applied to each element / primitive in the data type.

The data here is the PROLOG abstract syntax and the containers are `Functor`, `Foldable`, and `Traversable`. Figure 10.5 shows the relation between the different classes.

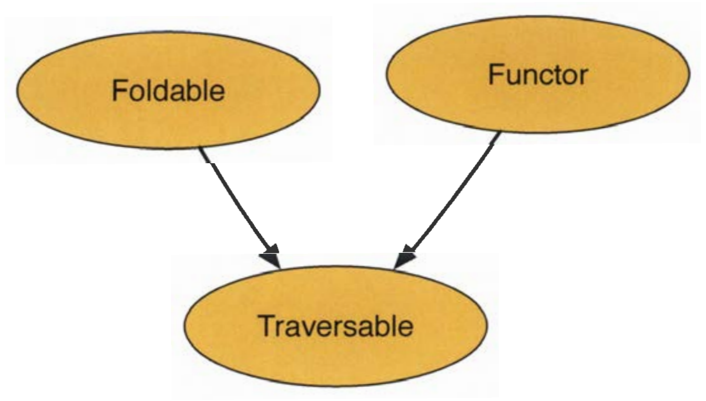


Figure 10.5: Functor Hierarchy (simplified from [155])

The `Functor` and `Foldable` instances provide functions for applying map-reduce to the data structure as described in Chapter 9. The primary issue is that at the end of the operation the structure of the data type is lost which would not help our cause since the result of a query must be a list of substitutions which are essentially pairs of language variables with language values (language constructs).

Enter `Traversable`. It allows `reduce` whilst preserving the shape of the structure. We create the necessary instances as shown in Listing 10.6.

Listing 10.6 `FlatTerm` class instances

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

instance Foldable (FlatTerm) where
  foldMap = T.foldMapDefault

instance Traversable (FlatTerm) where
   traverse f (Struct atom x)    =   Struct atom <$>
      sequenceA (Prelude.map f x)
   traverse _ (Var v)    =   pure (Var v)
   traverse _ Wildcard   =   pure (Wildcard)
   traverse _ (Cut i)    =    pure (Cut i)
```

The above lay the foundation to work with the library. Coming back to the library, the language must have the `Unifiable` instance. This works in tandem with the `UTerm` data type. The `UTerm` data type captures the recursive structure of logic terms, i.e., given some functor *t* which describes the constructors of our logic terms, and some type *v* which describes our logic variables, the type `UTerm` *t v* is the type of logic terms: trees with multiple layers of *t* structure and leaves of type *v*. The `Unifiable` class gives one step of the unification process. Just as we only need to specify one level of the ADT (i.e., `T`) and then we can use the library's `UTerm` to generate the recursive ADT, so we only need to specify one level of the unification (i.e., `zipMatch`) and then we can use the library's operators to perform the recursive unification. This is shown in Figure 10.7.

Listing 10.7 `FlatTerm` instance of `zipMatch`

```
instance Unifiable (FlatTerm) where
   zipMatch (Struct al ls) (Struct ar rs) =
     if (al == ar) && (length ls == length rs)
     then Struct al <$> pairWith (\l r -> Right (l,r)) ls rs
     else Nothing
   zipMatch Wildcard _ = Just Wildcard
   zipMatch _ Wildcard = Just Wildcard
   zipMatch (Cut i1) (Cut i2) =
     if (i1 == i2) then Just (Cut i1) else Nothing
```

Unification involves side effects of binding logic variables to terms. To allow and keep track of these effects we use the binding monad which provides facilities to generate fresh logic variables and perform look ups on dictionaries. By default two logic variable implementations exist:

1. The `IntVar` implementation uses `Int` as the names of variables, and uses an `IntMap` to keep track of the environment.

2. The `STVar` implementation uses `STRef`s, so we can use actual mutation for binding logic variables, rather than keeping an explicit environment around.

The `ST monad` is similar to the `IO monad` but is escapable. An `ST` action is of the form:

```
ST s a
```

a is the return type of the result of the computation in thread s. The s keeps references to objects inside the `ST monad` from leaking to the outside of the `ST monad` meaning the actions can only affect their own thread. To escape we require:

```
runST :: (forall s. ST s a) -> a
```

The action a must be universal in s, meaning the threading association is not known prohibiting to other threads and thus `runST` is pure [154, 129]. This is the idea behind

STVars which are implemented using STRefs. Hence to use "true mutability" we carry out the computation in the ST monad.

This implementation uses STVars and STBindings to implement the BindingMonad but a custom implementation could also be used inside the BindingMonad. For our language expressions to be unifiable we must deal with the variables in the expressions being compared. For that we extract the variables and then convert them into a dictionary consisting of a free variable for each language variable as shown in Figure 10.8.

Listing 10.8 Creating a variable dictionary

```haskell
variableExtractor :: Fix FlatTerm -> [Fix FlatTerm]
variableExtractor (Fix x) = case x of
   (Struct _ _) ->  Foldable.foldMap variableExtractor x
   (Var v)      ->  [Fix $ Var v]
   _            ->  []

variableNameExtractor :: Fix FlatTerm -> [VariableName]
variableNameExtractor (Fix x) = case x of
   (Struct _ _)     -> Foldable.foldMap variableNameExtractor x
   (Var v)          -> [v]
   _                -> []

variableSet :: [Fix FlatTerm] -> S.Set (Fix FlatTerm)
variableSet a = S.fromList a

variableNameSet :: [VariableName] -> S.Set (VariableName)
variableNameSet a = S.fromList a

varsToDictM :: (Ord a, Unifiable t) =>
    S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
varsToDictM set = foldrM addElt Map.empty set where
  addElt sv dict = do
    iv <- freeVar
    return $! Map.insert sv iv dict
```

A language to STVar dictionary is only one part of the unification procedure, the terms themselves should be made compatible for the in built unify procedure to perform look ups for the variables in them. The dictionary along with the fixed point version flattened of the term, as shown in Figure 10.9.

Listing 10.9 Conversion to UTerm

```
uTermify
  :: Map VariableName (ST.STVar s (FlatTerm))
  -> UTerm FlatTerm (ST.STVar s (FlatTerm))
  -> UTerm FlatTerm (ST.STVar s (FlatTerm))
uTermify varMap ux = case ux of
  UT.UVar _                  -> ux
  UT.UTerm (Var v)          -> maybe (error "bad map") UT.UVar $
                                   Map.lookup v varMap
  -- UT.UTerm t               -> UT.UTerm £! fmap (uTermify varMap)
  UT.UTerm (Struct a xs)     -> UT.UTerm $ Struct a $!
                                          fmap (uTermify varMap) xs
  UT.UTerm (Wildcard)        -> UT.UTerm Wildcard
  UT.UTerm (Cut i)           -> UT.UTerm (Cut i)

translateToUTerm ::
    Fix FlatTerm -> ST.STBinding s
            (UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)),
             Map VariableName (ST.STVar s (FlatTerm)))
translateToUTerm e1Term = do
  let vs = variableNameSet $ variableNameExtractor e1Term
  varMap <- varsToDictM vs
  let t2 = uTermify varMap . unfreeze $ e1Term
  return (t2,varMap)
```

and for later use to convert them back, as shown in Figure 10.10.

Listing 10.10 Conversion from UTerm

```
vTermify :: Map Int VariableName ->
            UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)) ->
            UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm))
vTermify dict t1 = case t1 of
  UT.UVar x  -> maybe (error "logic") (UT.UTerm . Var) $
                         Map.lookup (UT.getVarID x) dict
  UT.UTerm r ->
    case r of
      Var iv   -> t1
      _         -> UT.UTerm . fmap (vTermify dict) $ r

translateFromUTerm ::
    Map VariableName (ST.STVar s (FlatTerm)) ->
    UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)) -> Prolog
translateFromUTerm dict uTerm =
  P .  maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm where
    rot3 f a k v = f k v a
    inserter k v = Map.insert (UT.getVarID v) k
    forKV dict initial fn = Map.foldlWithKey' (rot3 fn) initial dict
    varIdDict = forKV dict Map.empty inserter
```

The variable dictionaries and UTermified language expressions are unified in the binding monad as shown in Figure 10.11 and Figure 10.12.

Listing 10.11 Unification code

```
monadicUnification ::
  (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s)) =>
  (forall s. (Fix FlatTerm) -> (Fix FlatTerm) ->
   ErrorT (UT.UFailure (FlatTerm) (ST.STVar s (FlatTerm)))
   (ST.STBinding s) (UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)),
                     Map VariableName (ST.STVar s (FlatTerm))))
monadicUnification t1 t2 = do
  (x1,d1) <- lift . translateToUTerm $ t1
  (x2,d2) <- lift . translateToUTerm $ t2
  x3 <- U.unify x1 x2
  return $! (x3, d1 `Map.union` d2)
```

Listing 10.12 Driver code

```
runUnify ::
  (forall s. (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  =>
      (ErrorT
          (UT.UFailure FlatTerm (ST.STVar s FlatTerm))
          (ST.STBinding s)
          (UT.UTerm FlatTerm (ST.STVar s FlatTerm),
             Map VariableName (ST.STVar s FlatTerm)))
     )
  -> [(VariableName, Prolog)]
runUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> extractUnifier dict
```

The final reconversion to return a list of substitutions, called extractUnifier in Figure 10.12, is shown in Figure 10.13.

Listing 10.13 Variable substitution list extraction

```
extractUnifier ::
  (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  => (forall s. Map VariableName (STVar s FlatTerm)
      -> (ST.STBinding s [(VariableName, Prolog)])
     )
extractUnifier dict = do
  let ld1 = Map.toList dict
  ld2 <- Control.Monad.Error.sequence
         [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v]
  let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 `zip` ld2]
      ld4 = [ (k,v) | (k,v2) <- ld3,
                    let v = translateFromUTerm dict v2 ]
  return ld4
```

Listing 10.14 shows fixed point versions of sample terms.

Listing 10.14 Sample terms

```haskell
fix1 = Fix $ Struct "a" [(Fix $ Var $ VariableName 0 "x"), (Fix $ Cut 0),
                (Fix $ Wildcard)]

fix2 = (Fix $ Var $ VariableName 1 "x")
```

Listing 10.15 shows the result of the execution for `runUnify`.

Listing 10.15 Output for `runUnify`

```
runUnify $ monadicUnification fix1 fix2
[(VariableName 1 "x",
 P (Struct "a" [Var (VariableName 0 "x"),Cut 0,Wildcard]))]
```

Recapitulating, this chapter provides the ideas of language modification and monadic unification along with their respective implementations.

Chapter 11

Prototype 2

This chapter attempts to infuse the generic methodology from Chapter 10 in a current PROLOG implementation [114] and make the unification "monadic".

This chapter discusses the idea of implementing a basic working PROLOG query resolver while using `prolog-0.2.0.1` [114] as the base implementation. The language modifications and unification mechanism have been taken from Chapter 10 and adapted to fit in with the other components such as the search strategy of the library.

11.1 How `prolog-0.2.0.1` works

The `prolog-0.2.0.1` library ([114]) was written by Matthias Bartsch and consists of 14 HASKELL files. It implements data base assertions and cuts but lacks any IO facilities. Moreover, the abstract syntax used to implement the language is rigid and leaves very little to no scope for extension. Figure 11.1 describes its architecture.

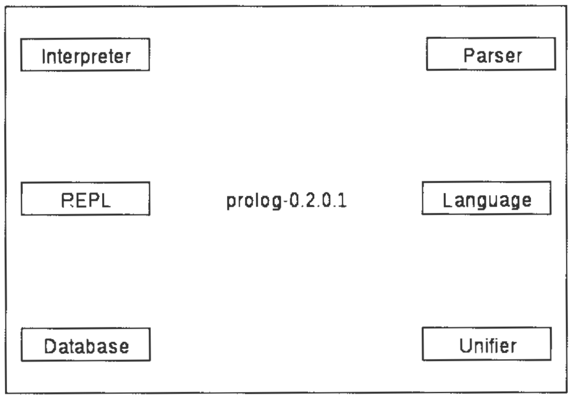


Figure 11.1: `prolog-0.2.0.1` [114] architecture

From the Listing 11.1 we will focus on `Terms`, since the others just add wrappers around expressions which can be created by it. This language suffers from most of the problems discussed in the previous chapter. The above is used to construct PROLOG "terms" which are of a "single type".

The implementation consists of components that one would find in a language processing system (see Figure 11.2).

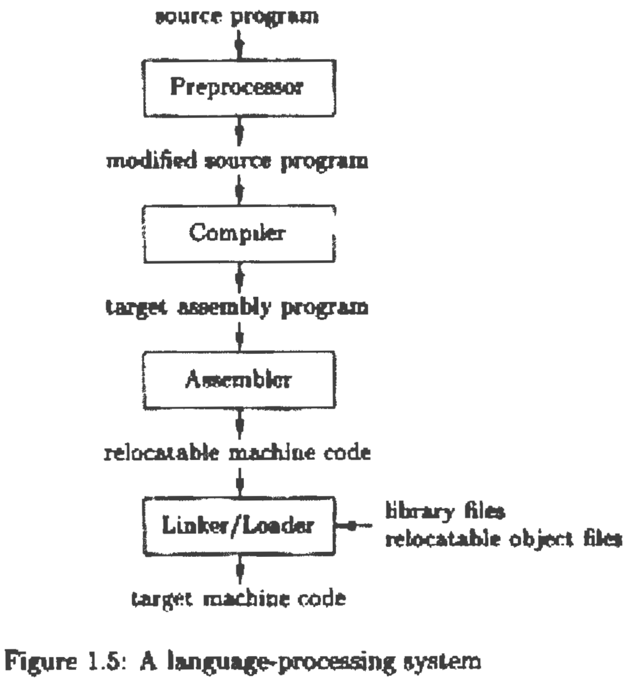


Figure 11.2: A language-processing system (taken from [1])

They specifically contain parts of a compiler (see Figure 11.3).

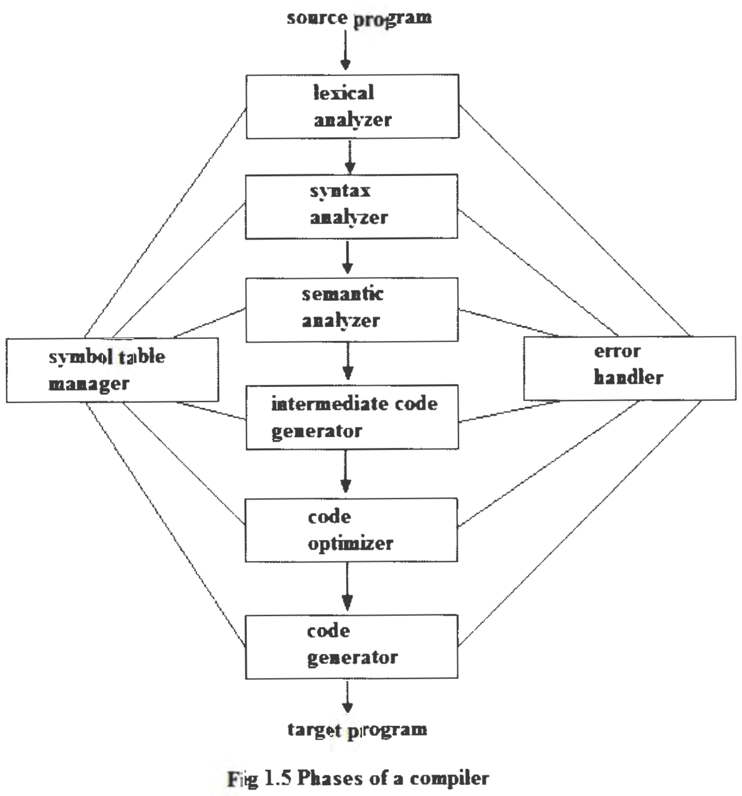


Figure 11.3: Phases of Compiler [1]

The architecture for a compiler as described in Figure 11.3 is not needed since HASKELL provides most of them. Nonetheless, the library has the following major components as shown in Figure 11.1:

- the syntax which defines the language,
- the database which stores the expressions and language constructs,
- the parser,
- the interpreter,
- the unifier,
- the Read-Eval-Print Loop (REPL).

To prove the modularity of the approach used in Chapter 10 for language modification and monadic unification only the abstract syntax and unifier will be customized.

11.2 What we do in this prototype

Figure 11.4 describes the components of this implementation and the relation between them.

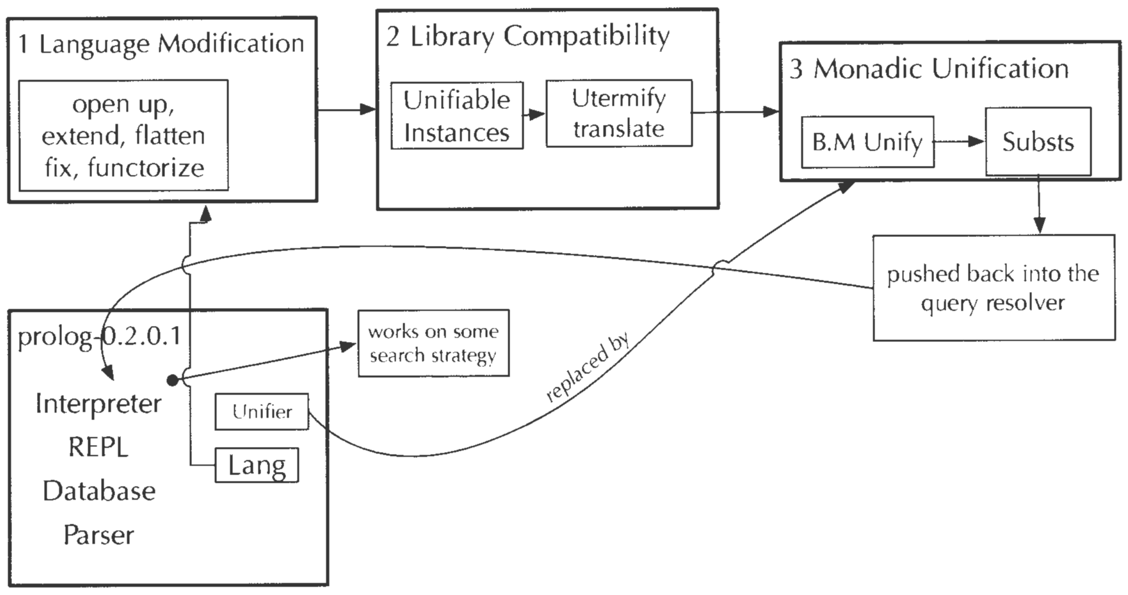


Figure 11.4: Architecture of Prototype 2

Unification is a part of how PROLOG works to produce a solution to the query. The `unification` procedure tells us whether or not two terms can be made equal. Before reaching that state, two terms must be gathered depending upon matching rules in the knowledge base. This is where a search strategy comes into play along with a backtracking mechanism.

Putting everything together forms the PROLOG query resolver. Given a query and a knowledge base, the query resolver matches the input query with the rules in the knowledge base to create a list of goals to satisfy to generate an unifier along with saving choice points on the way for backtracking.

This chapter discusses how the abstract syntax and query resolver from [114] can be adapted to the concepts and implementation from Chapter 10. This not only proves how generic and modular the approach from the previous chapter is, but also the working PROLOG system as a whole.

11.3 Procedure

11.3.1 Flatten the language by introducing a type variable

The first component describes the process of creating a modified version of the current abstract syntax used by the library as shown in Listing 11.1.

Listing 11.1 Original Recursive Grammar

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

var = Var . VariableName 0
cut = Cut 0

data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
            | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
      deriving (Data, Typeable)

rhs :: Clause -> [Term] -> [Goal]
rhs (Clause   _ rhs) = const rhs
rhs (ClauseFn _ fn ) = fn

data VariableName = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)

type Atom         = String
type Goal         = Term
type Program      = [Clause]
```

The grammar suffers from most of the drawbacks mentioned in Chapter 10. This implementation focuses on creating a working PROLOG query resolver with monadic unification. The base implementation adopted from prolog-0.2.0.1 [114] already has a working implementation, but lacks benefits achieved from our approach mention in the previous chapter.

The non-recursive implementation described in Listing 11.2 introduces a type variable which separates the structure from the data itself.

Listing 11.2 Flattened (non-recursive) grammar

```
data FTS a = FS Atom [a] | FV VariableName | FW | FC Int
                          deriving (Show, Eq, Typeable, Ord)
```

We implement the necessary instances to make the language unifiable as shown in Listing 11.3.

Listing 11.3 Instances for flattened grammar

```
instance Functor (FTS) where
  fmap                        = T.fmapDefault
instance Foldable (FTS) where
  foldMap                     = T.foldMapDefault
instance Traversable (FTS) where
    traverse f (FS atom xs) =   FS atom <$> sequenceA (Prelude.map f xs)
    traverse _ (FV v)       =   pure (FV v)
    traverse _ FW           =   pure (FW)
    traverse _ (FC i)       =   pure (FC i)
instance Unifiable (FTS) where
  zipMatch (FS al ls) (FS ar rs) =
        if (al == ar) && (length ls == length rs)
        then FS al <$> pairWith (\l r -> Right (l,r)) ls rs else Nothing
  zipMatch FW _                  = Just FW
  zipMatch _ FW                  = Just FW
  zipMatch (FC i1) (FC i2)       =
         if (i1 == i2) then Just (FC i1) else Nothing
```

Lastly, the fixed point version is created using the Fix constructor

Listing 11.4 Fixed point of flattened grammar

```
newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)

unP :: Prolog -> Fix FTS
unP (P x) = x
```

The above approach allows us to jump between the environments (grammars) easily provided the back and forth conversion capabilities from Listing 11.6 and Listing 11.5.

Listing 11.5 `prolog-0.2.0.1` Monadic Unification Conversion Functions

```haskell
termFlattener :: Term -> Fix FTS
termFlattener = DFF.ana oneLevel where
  oneLevel :: Term -> FTS Term
  oneLevel x = case x of
    { Var v ->   FV v ; Wildcard -> FW ; Cut i -> FC i ;
      Struct a xs ->  FS a xs }

unFlatten :: Fix FTS -> Term
unFlatten = DFF.cata levelOne where
  levelOne :: FTS Term -> Term
  levelOne x = case x of
    { FV v -> Var v ; FW -> Wildcard ; FC i -> Cut i ;
      FS a xs -> Struct a xs }


variableExtractor :: Fix FTS -> [Fix FTS]
variableExtractor (Fix x) = case x of
  (FS _ xs)   ->  Prelude.concat $ Prelude.map variableExtractor xs
  (FV v)     ->  [Fix $ FV v]
  _        ->  []

variableNameExtractor :: Fix FTS -> [VariableName]
variableNameExtractor (Fix x) = case x of
  (FS _ xs) -> Prelude.concat $ Prelude.map variableNameExtractor xs
  (FV v)     -> [v]
  _          -> []

variableSet :: [Fix FTS] -> S.Set (Fix FTS)
variableSet a = S.fromList a

variableNameSet :: [VariableName] -> S.Set (VariableName)
variableNameSet a = S.fromList a

varsToDictM :: (Ord a, Unifiable t) =>
    S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
varsToDictM set = foldrM addElt Map.empty set where
  addElt sv dict = do
    iv <- freeVar
    return $! Map.insert sv iv dict
```

Listing 11.6 `prolog-0.2.0.1` Monadic Unification Translation Functions

```haskell
type USTerm t s = UTerm t (ST.STVar s t)

uTermify :: Map VariableName (ST.STVar s FTS) -> USTerm FTS s -> UTerm FTS s
uTermify varMap ux = case ux of
  UT.UVar _         -> ux
  UT.UTerm (FV v)   -> maybe (error "bad map") UT.UVar $ Map.lookup v varMap
  UT.UTerm t        -> UT.UTerm $! fmap (uTermify varMap) t

translateToUTerm ::
    Fix FTS -> ST.STBinding s
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
translateToUTerm e1Term = do
  let vs = variableNameSet $ variableNameExtractor e1Term
  varMap <- varsToDictM vs
  let t2 = uTermify varMap . unfreeze $ e1Term
  return (t2,varMap)


-- | vTermify recursively converts @UVar x@ into @UTerm (VarA x).
-- This is a subroutine of @ translateFromUTerm @.  The resulting
-- term has no (UVar x) subterms.

helper :: Map Int VariableName -> ST.STVar s FTS -> USTerm FTS s
helper dict v  = maybe (error "logic") (UT.UTerm . FV) $
                  Map.lookup (UT.getVarID v) dict

vTermify :: Map Int VariableName -> USTerm FTS s -> USTerm FTS s
vTermify dict t1 = vTermify2 (helper dict) t1 where
  vTermify2 f t1 = case t1 of
    UT.UVar x  -> f x
    UT.UTerm r -> UT.UTerm . fmap (vTermify2 f) $ r

reverseDict :: Map VariableName (SVar s) -> Map Int VariableName
reverseDict dict = varIdDict where
  forKV dict initial fn = Map.foldlWithKey' (\a k v -> fn k v a) initial dict
  varIdDict = forKV dict Map.empty $ \ k v -> Map.insert (UT.getVarID v) k

translateFromUTerm ::
    Map VariableName (ST.STVar s (FTS)) -> USTerm FTS s -> Prolog
translateFromUTerm dict =
  P .  maybe (error "Logic") id . freeze . vTermify (reverseDict dict)
```

After the language is opened up, the next step is to replace the current unification procedure from `prolog-0.2.0.1` [114] with one similar to Chapter 10.

The current unification uses basic pattern matching to unify terms. This approach tightly couples the unification to the embedded language meaning that every language change requires rewriting the unification procedure. Secondly, the unification procedure is correct but not very efficient. Given that `unification-fd` [137] provides an efficient implementation of imperative unification algorithms, we have another reason to replace the unification mechanism from `prolog-0.2.0.1` [114].

The results produced by the query resolver are shown in Listing 11.7.

Listing 11.7 `prolog-0.2.0.1` `Unifier`

```
type Unifier      = [Substitution]
type Substitution = (VariableName, Term)
```

A `Unifier` is a list of `Substitutions` which binds a variable to a value.

The unification procedure is shown in Figure 11.8. Each language expression is matched based on its structure and returns a `Unifier` .

Listing 11.8 prolog-0.2.0.1 Unification

```
unify, unify_with_occurs_check :: MonadPlus m => Term -> Term
-> m Unifier

unify = fix unify'

unify_with_occurs_check =
   fix $ \self t1 t2 -> if (t1 `occursIn` t2 || t2 `occursIn` t1)
                           then fail "occurs check"
                           else unify' self t1 t2
 where
   occursIn t = everything (||) (mkQ False (==t))

unify' :: MonadPlus m => (Term -> Term -> m Unifier) -> Term ->
Term -> m [(VariableName, Term)]

-- If either of the terms are don't cares then no unifiers exist
unify' _ Wildcard _ = return []
unify' _ _ Wildcard = return []

-- If one is a variable then equate the term to its value which
-- forms the unifier
unify' _ (Var v) t  = return [(v,t)]
unify' _ t (Var v)  = return [(v,t)]

-- Match the names and the length of their parameter list and
-- then match the elements of list one by one.
unify' self (Struct a1 ts1) (Struct a2 ts2)
            | a1 == a2 && same length ts1 ts2 =
            unifyList self (zip ts1 ts2)

unify' _ _ _ = mzero

same :: Eq b => (a -> b) -> a -> a -> Bool
same f x y = f x == f y

-- Match the elements of each of the tuples in the list.
unifyList :: Monad m => (Term -> Term -> m Unifier) ->-
[(Term, Term)] -> m Unifier
unifyList _ [] = return []
unifyList unify ((x,y):xys) = do
   u  <- unify x y
   u' <- unifyList unify (Prelude.map (both (apply u)) xys)
   return (u++u')
```

The modification to the unification procedure is described in Listing 11.9, Listing 11.10, Listing 11.11, Listing 11.13, Listing 11.15 and Listing 11.14.

Listing 11.9 prolog-0.2.0.1 Monadic Unification Functions

```haskell
-- | Unify two (E1 a) terms resulting in maybe a dictionary
-- of variable bindings (to terms).
--
-- NB !!!!  The current interface assumes that the variables
-- in t1 and t2 are disjoint.  This likely needs fixing.
type SVar s = ST.STVar s (FTS)
type PrologMap = Map VariableName Prolog
type UTExcept t v m r = ExceptT (UT.UFailure t v) m r

unifyTerms :: Fix FTS -> Fix FTS -> Maybe PrologMap
unifyTerms t1 t2 = ST.runSTBinding $ do
  answer <- runExceptT $ unifyTermsX t1 t2
  return $! either (const Nothing) Just answer

-- | Unify two (E1 a) terms resulting in maybe a dictionary
-- of variable bindings (to terms).
-- This routine works in the unification monad
unifyTermsX :: (Fix FTS) -> (Fix FTS)
               -> UTExcept FTS (SVar s) (ST.STBinding s) PrologMap
unifyTermsX t1 t2 = do
    (x1,d1) <- lift . translateToUTerm $ t1
    (x2,d2) <- lift . translateToUTerm $ t2
    _ <- U.unify x1 x2
    makeDicts $ (d1,d2)

mapWithKeyM :: (Ord k,Applicative m,Monad m)
               => (k -> a -> m b) -> Map k a -> m (Map k b)
mapWithKeyM = Map.traverseWithKey
makeDict :: Map VariableName (SVar s) -> ST.STBinding s PrologMap
makeDict sVarDict =
    flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do
        Just xx <- UT.lookupVar $ iKey
        return $! (translateFromUTerm sVarDict) xx

-- | recover the bindings for the variables of the two terms
-- unified from the monad.
makeDicts :: (Map VariableName (SVar s), Map VariableName (SVar s))
             -> UTExcept FTS (SVar s) (ST.STBinding s) PrologMap
makeDicts (svDict1, svDict2) = do
  let svDict3 = (svDict1 `Map.union` svDict2)
  let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
  applyBindingsAll ivs
  lift . makeDict $ svDict3
```

Listing 11.10 `prolog-0.2.0.1` Monadic Unification Tests and Extraction 1

```haskell
instance (UT.Variable v, Functor t) => Error (UT.UFailure t v) where {}

test1 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
           (ST.STBinding s)
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
test1 = do
    let
        t1a = (Fix $ FV $ VariableName 0 "x")
        t2a = (Fix $ FV $ VariableName 1 "y")
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unify x1 x2
    return (x3, d1 `Map.union` d2)

test2 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
           (ST.STBinding s)
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
test2 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ VariableName 0 "x"])
        t2a = (Fix $ FV $ VariableName 1 "y")
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unify x1 x2
    return (x3, d1 `Map.union` d2)

test3 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
           (ST.STBinding s)
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
test3 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ VariableName 0 "x"])
        t2a = (Fix $ FV $ VariableName 0 "x")
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unify x1 x2
    return (x3, d1 `Map.union` d2)
```

Listing 11.11 `prolog-0.2.0.1` Monadic Unification Tests and Extraction 2

```haskell
test4 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
           (ST.STBinding s)
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
test4 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ VariableName 0 "x"])
        t2a = (Fix $ FV $ VariableName 0 "x")
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unifyOccurs x1 x2
    return (x3, d1 `Map.union` d2)

test5 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
           (ST.STBinding s)
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))
test5 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ VariableName 0 "x"])
        t2a = (Fix $ FS "b" [Fix $ FV $ VariableName 0 "y"])
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unify x1 x2
    return (x3, d1 `Map.union` d2)
```

In this implementation unification is a three part procedure as follows:

1. `monadicUnification`

   Firstly, we take a couple of flattened terms in fixed point and convert them into `UTerm`-format so that we can apply the `unify` function provided by `unification-fd` and results in a `unifier`. The type signature of `unify` is shown in Listing 11.12.

Listing 11.12 `unification-fd` `unify` type signature

```
unify ::
(BindingMonad t v m, Fallible t v e, MonadTrans em, Functor (em m),
        MonadError e (em m))
        => UTerm t v
    -> UTerm t v
    -> em m (UTerm t v)
```

We also return a union of the variable dictionaries for each of the terms required to convert the result into the eDSL. Listing 11.13 the procedure for `monadicUnification`.

Listing 11.13 `monadicUnification` function

```
monadicUnification :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
 => (forall s. ((Fix FTS) -> (Fix FTS) ->
      ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
            (ST.STBinding s) (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map VariableName (ST.STVar s (FTS)))))
monadicUnification t1 t2 = do
  (x1,d1) <- lift . translateToUTerm $ t1
  (x2,d2) <- lift . translateToUTerm $ t2
  x3 <- U.unify x1 x2
  return $! (x3, d1 `Map.union` d2)
```

2. `extractUnifier`

Retranslate the terms back to fixed flat terms and return results. Listing 11.14 returns list of `VariableName`, `Prolog` pairs.

Listing 11.14 `extractUnifier` function

```
extractUnifier ::
  (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (forall s. Map VariableName (STVar s FTS)
      -> (ST.STBinding s [(VariableName, Prolog)]))
extractUnifier dict = do
  let ld1 = Map.toList dict
  ld2 <- Control.Monad.Error.sequence
          [v1 | (k,v) <- ld1, let v1 = UT.lookupVar v]
  let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 `zip` ld2]
      ld4 = [ (k,v) | (k,v2) <- ld3, let v = translateFromUTerm dict v2 ]
  return ld4
```

3. `runUnify`

   This function executes the above operations in the `BindingMonad` and return the necessary results. Listing 11.15 shows the `runUnify` function.

Listing 11.15 `runUnify` function

```
runUnify ::
  (forall s. (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (ErrorT
        (UT.UFailure FTS (ST.STVar s FTS))
        (ST.STBinding s)
        (UT.UTerm FTS (ST.STVar s FTS),
            Map VariableName (ST.STVar s FTS))))
  -> [(VariableName, Prolog)]
runUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> extractUnifier dict
```

4. `unify`

   Putting it all together we perform the unification on the two terms and return the `Unifier` as originally intended. Listing 11.16

Listing 11.16 `unify`

```haskell
unifierConvertor :: [(VariableName, Prolog)] -> Unifier
unifierConvertor xs =
  Prelude.map (\(v, p) -> (v, (unFlatten $ unP $ p))) xs

unify :: MonadPlus m => Term -> Term -> m Unifier
unify t1 t2 = unifierConvertor
                (goUnify
                  (monadicUnification
                    (termFlattener t1) (termFlattener t2)
                  )
                )
```

Recapitulating, this chapter provided an implementation of a `Prolog`-like language adopted from [114] and incorporated a monadic unifier built from [137] to create a PROLOG query resolver.

Chapter 12

Prototype 3

This chapter discusses the procedure to infuse multiple search strategies into a PROLOG query resolver with monadic unification. The base implementation for this prototype is `Mini Prolog` [64].

12.1 `Mini Prolog` [64] architecture

`Mini Prolog` is based on an older implementation of HASKELL called `Hugs 98`. The architecture of the library is described in the Figure 12.1. The main components are as follows:

1. the language itself,
2. multiple search strategies used by the query resolver,
3. a parser,
4. a unification mechanism,
5. an interpreter,
6. a knowledge base, and

7. a REPL.

The main highlight of this implementation is the fact that the query resolver can work with multiple search strategies decided at compile time. A query request will consist of the query itself i.e., the terms to be unified, a knowledge base storing the clauses, the unification procedure and finally a user provided search strategy.

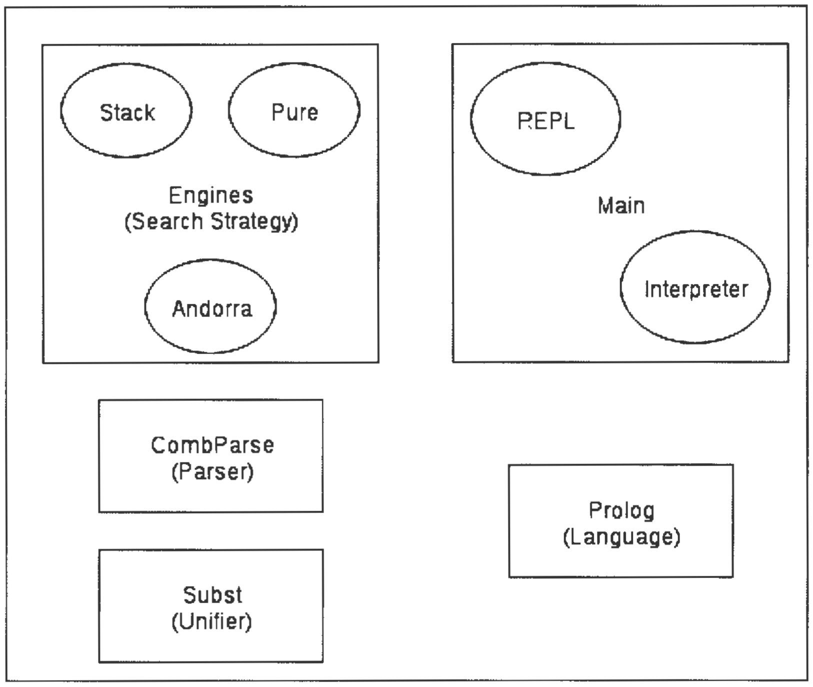


Figure 12.1: Mini PROLOG architecture

12.2 Prototype architecture

The focus of this prototype is to embed the language modification procedure and monadic unification into [64] so to further prove the generality and modularity of the approach from the previous prototypes. The architecture for this prototype is beautifully illustrated by Figure 12.2.

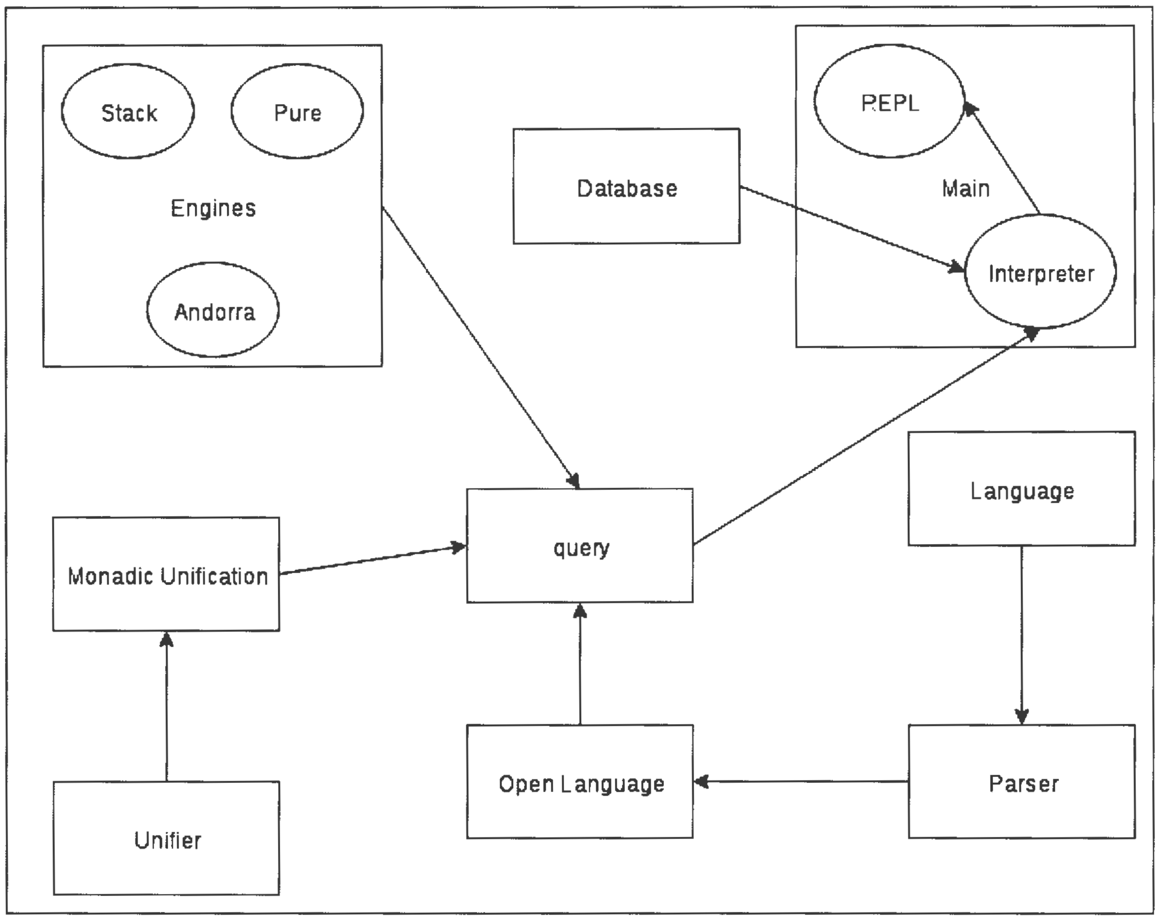


Figure 12.2: Architecture of Prototype 3

Since we are aiming for modularity, most components in the Figure 12.1 are untouched. The abstract syntax is modified to conform to the `unification-fd` library [137]. Looking at the center of the figure you will find *query*. This component takes as input the terms to be unified in modified language form, a search strategy, the knowledge base and the monadic unifier, and returns a list of substitutions as required by `Mini Prolog` library [64]. Each of the components in Figure 12.2 will be discussed in the sections to come.

12.3 Engines (search strategies)

This section corresponds to the top left component of Figure 12.2. Below are the description of the various engines (these are called search strategies in 8.5).

12.3.1 The Stack engine

The stack based engine works on a stack of triples `(s,goal,alts)` corresponding to backtrack choice points, where:

1. `s` is the substitution at that point,
2. `goal` is the outstanding goal and
3. `alts` is a list of possible ways of extending the current proof to find a solution.

Each member of `alts` is a pair `(tp,u)` where:

1. `tp` is a new goal that must be proved and
2. `u` is a unifying substitution that must be combined with the substitution `s`.

The list of relevant clauses at each step in the execution is produced by attempting to unify the head of the current goal with a suitably renamed clause from the database.

Listing 12.1 represents the Stack engine.

Listing 12.1 Stack engine from `Mini Prolog` [64]

```
type Stack = [ (st, [Term], [Alt]) ]
type Alt   = ([Term], st)

alts       :: Database -> Int -> Term -> [Alt]
alts db n g = [ (tp,u) | (tm:-tp) <- renClauses db n g, u <- unify g tm ]

prove      :: Database -> [Term] -> [st]
prove db gl = solve 1 nullst gl []
 where
   solve :: Int -> st -> [Term] -> Stack -> [st]
   solve n s []     ow          = s : backtrack n ow
   solve n s (g:gs) ow
                    | g==theCut = solve n s gs (cut ow)
                    | otherwise = choose n s gs (alts db n (app s g)) ow

   choose :: Int -> st -> [Term] -> [Alt] -> Stack -> [st]
   choose n s gs []          ow = backtrack n ow
   choose n s gs ((tp,u):rs) ow = solve (n+1) (u@@s) (tp++gs) ((s,gs,rs):ow)

   backtrack                    :: Int -> Stack -> [st]
   backtrack n []                = []
   backtrack n ((s,gs,rs):ow)    = choose (n-1) s gs rs ow

theCut    :: Term
theCut     = Struct "!" []

cut                       :: Stack -> Stack
cut ss                     = []
```

12.3.2 The Pure engine

The pure engine works on `Prooftree`s. Each node in a `Prooftree` corresponds to:

1. Either, a solution to the current goal, represented by `Done s`, where `s` is the required substitution, or,

2. a choice between a number of trees `ts`, each corresponding to a proof of a `goal` of the current goal, represented by `Choice ts`. The proof tree corresponding to an

unsolvable goal is `Choice []`.

Listing 12.2 represents the Pure engine.

Listing 12.2 Pure engine from `Mini Prolog` [64]

```
data Prooftree = Done st  |  Choice [Prooftree]

-- prooftree uses the rules of Prolog to construct a suitable proof tree for
--            a specified goal
prooftree   :: Database -> Int -> st -> [Term] -> Prooftree
prooftree db = pt
 where pt               :: Int -> st -> [Term] -> Prooftree
       pt n s []      = Done s
       pt n s (g:gs) = Choice [ pt (n+1) (u@@s) (map (app u) (tp++gs))
                             | (tm:-tp)<-renClauses db n g, u<-unify g tm ]

-- DFS Function
-- search performs a depth-first search of a proof tree, producing the list
-- of solution stitutions as they are encountered.
search              :: Prooftree -> [st]
search (Done s)      = [s]
search (Choice pts)  = [ s | pt <- pts, s <- search pt ]


prove    :: Database -> [Term] -> [st]
prove db  = search . prooftree db 1 nullst
```

12.3.3 The Andorra engine

This inference engine implements a variation of the Andorra Principle for logic programming adapted from [51]. The main difference here is to select a relatively deterministic goal and not the first one. Upon selecting a goal:

1. for each goal g in the list of goals, calculate the resolvents that would result from selecting g, and

2. then choose a g which results in the lowest number of resolvents.

If some g results in no resolvents then it is regarded as a failure. For instance,

```
?- append(A,B,[1,2,3]),equals(1,2).)
```

PROLOG would not perform this optimization and would instead search and backtrack wastefully. If some g results in a single resolvent then that g will get selected; by selecting and resolving g, bindings are propagated sooner, and useless search can be avoided, since these bindings may prune away choices for other clauses. For example:

```
?- append(A,B,[1,2,3]),B=[].
```

Listing 12.3 represents the Andorra engine.

Listing 12.3 Andorra engine from `Mini Prolog` [64]

```
solve   :: Database -> Int -> st -> [Term] -> [st]
solve db = slv where
slv           :: Int -> st -> [Term] -> [st]
slv n s [] = [s]
slv n s goals =
let allResolvents = resolve_selecting_each_goal goals db n in
let (gs,gres) =  findMostDeterministic allResolvents in
concat [slv (n+1) (u@@s) (map (app u) (tp++gs)) | (u,tp) <- gres]

resolve_selecting_each_goal::
[Term] -> Database -> Int -> [([Term],[(st,[Term])])]
resolve_selecting_each_goal goals db n = [(gs, gResolvents) |
  (g,gs) <- delete goals, let gResolvents = resolve db g n]

resolve :: Database -> Term -> Int -> [(st,[Term])]
resolve db g n = [(u,tp) | (tm:-tp)<-renClauses db n g, u<-unify g tm]

findMostDeterministic:: [([Term],[(st,[Term])])] -> ([Term],[(st,[Term])])
findMostDeterministic  allResolvents = minF comp allResolvents where
comp:: (a,[b]) -> (a,[b]) -> Bool
comp (_,gs1) (_,gs2) = (length gs1) < (length gs2)

delete ::  [a] -> [(a,[a])]
delete l = d l [] where
d :: [a] -> [a] ->  [(a,[a])]
d [g] sofar = [ (g,sofar) ]
d (g:gs) sofar = (g,sofar++gs) : (d gs (g:sofar))

minF                :: (a -> a -> Bool) -> [a] -> a
minF f (h:t) = m h t where
--    m :: a -> [a] -> a
m sofar [] = sofar
m sofar (h:t) = if (f h sofar) then m h t else m sofar t

prove    :: Database -> [Term] -> [st]
prove db  = solve db 1 nullst
```

12.4 Language

12.4.1 Current language

Listing 12.4 shows the abstract syntax of `Mini Prolog` [64]. A term can either be a variable or a complex term with an atom as a head. A `Clause` consists of a head of type `Term` and a body of type `[Term]`.

Listing 12.4 Current abstract syntax grammar in `Mini Prolog` [64]

```

type Atom      = String

data Term      = Var Id | Struct Atom [Term]

data Clause    = Term :- [Term]

data Database = Db [(Atom,[Clause])]

instance Eq Term where
     Var v        == Var w        =  v==w
     Struct a ts == Struct b ss =  a==b && ts==ss
     _            == _            =  False

```

12.4.2 Language modification

Listing 12.5 describes the necessary modifications required to adapt the language for monadic unification. This procedure consists of opening the language and adding the necessary instances for `unification-fd` [137] compatibility.

Listing 12.5 Language modification

```haskell
data FTS a = forall a . FV Id | FS Atom [a]
           deriving (Eq, Show, Ord, Typeable)
newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)

unP :: Prolog -> Fix FTS
unP (P x) = x

instance Functor FTS where
  fmap = T.fmapDefault

instance Foldable FTS where
  foldMap = T.foldMapDefault

instance Traversable FTS where
  traverse f (FS atom xs) = FS atom <$> sequenceA (Prelude.map f xs)
  traverse _ (FV v) =     pure (FV v)

instance Unifiable FTS where
  zipMatch (FS al ls) (FS ar rs) =
    if (al == ar) && (length ls == length rs)
    then FS al <$> pairWith (\l r -> Right (l,r)) ls rs
    else Nothing
  zipMatch (FV v1) (FV v2) = if (v1 == v2) then Just (FV v1)
                             else Nothing
  zipMatch _ _ = Nothing
```

12.5 Unification from `Mini Prolog` [64]

12.5.1 Current unification

Listing 12.6 describes the current unification mechanism which works on substitutions. The `unify` function compares the two terms and returns a list of substitutions as with the base implementations from the previous prototypes.

Listing 12.6 Current unification procedure in `Mini Prolog` [64]

```
newtype SubstP = SubstP { unSubstP :: Subst }

app                         :: Subst -> Term -> Term
app s (Var i)                = s i
app s (Struct a ts)          = Struct a (Prelude.map (app s) ts)

nullSubst                   :: Subst
nullSubst i                  = Var i

(->-)                       :: Id -> Term -> Subst
(i ->- t) j | j==i           = t
            | otherwise      = Var j

(@@)                        :: Subst -> Subst -> Subst
s1 @@ s2                     = app s1 . s2

unify :: Term -> Term -> [Subst]
unify (Var x)       (Var y)       = if x==y then [nullSubst] else [x->-Var y]
unify (Var x)       t2            = [ x ->- t2 | x `notElem` varsIn t2 ]
unify t1            (Var y)       = [ y ->- t1 | y `notElem` varsIn t1 ]
unify (Struct a ts) (Struct b ss) = [ u | a==b, u<-listUnify ts ss ]

listUnify :: [Term] -> [Term] -> [Subst]
listUnify []     []     = [nullSubst]
listUnify []     (r:rs) = []
listUnify (t:ts) []     = []
listUnify (t:ts) (r:rs) = [ u2 @@ u1 | u1<-unify t r,
                                       u2<-listUnify (map (app u1) ts)
                                                     (map (app u1) rs) ]
```

The shortcomings are translated from the language to the unification as it is based on basic pattern matching.

12.5.2 Monadic unification

Listing 12.7 shows the procedure for monadic unification. Most components of the procedure remain similar to the ones in previous prototypes.

Listing 12.7 Monadic unification

```haskell
monadicUnification :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s)) =>
  (forall s. (Term -> Term -> ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
      (ST.STBinding s) (UT.UTerm (FTS) (ST.STVar s (FTS)),
          Map Id (ST.STVar s (FTS)))))
monadicUnification t1 t2 = do
  let
    t1f = termFlattener t1
    t2f = termFlattener t2
  (x1,d1) <- lift . translateToUTerm $ t1f
  (x2,d2) <- lift . translateToUTerm $ t2f
  x3 <- U.unify x1 x2
  return $! (x3, d1 `Map.union` d2)

runUnify ::
  (forall s. (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (ErrorT
        (UT.UFailure FTS (ST.STVar s FTS))
        (ST.STBinding s)
        (UT.UTerm FTS (ST.STVar s FTS),
            Map Id (ST.STVar s FTS))))
  -> [(Id, Prolog)]
runUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> extractUnifier dict

extractUnifier :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (forall s. Map Id (STVar s FTS)
      -> (ST.STBinding s [(Id, Prolog)]))
extractUnifier dict = do
  let ld1 = Map.toList dict
  ld2 <- sequence [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v]
  let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 `zip` ld2]
      ld4 = [ (k,v) | (k,v2) <- ld3, let v = translateFromUTerm dict v2 ]
  return ld4

substConvertor :: [(Id, Prolog)] -> [Subst]
substConvertor xs = Prelude.map (\(varId, p) -> (->-) varId
                                    (unFlatten $ unP $ p)) xs

unify t1 t2 = substConvertor (runUnify (monadicUnification t1 t2))
```

The major changes occur in `substConvertor` used to convert the result of monadic unification into the original `Subst` form and `unify`.

12.6 Summary

Recapitulating, this chapter provided us with a working implementation of a PROLOG-like interpreter with the option to change the search strategy further proving the modularity and genericity of the language modification and monadic unification procedure.

Chapter 13

Prototype 4

The aim of this prototype is to embed `IO` operations within the definition of an eDSL so as to allow the chaining and control of operations of the language irrespective of them being pure or impure.

13.1 HASKELL `IO` is pure

The discussion in this section is mainly paraphrased from [135].

HASKELL calls itself a pure functional programming language. Every function in HASKELL is a function in the mathematical sense (i.e., "pure"). Even side-effecting `IO` operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers [52]. Consider the example in Listing 13.1 describing the `getLine` function in HASKELL.

Listing 13.1 HASKELL `getLine`

```
Prelude> x <- getLine
Hello
Prelude> x
"Hello"
```

IO actions can be embedded by building up data structures which can then be executed to cause side-effects, but until that point they are pure. Consider the Listing 13.2 describing an example for the same.

Listing 13.2 IO action data type taken from [135]

```
data IOAction a = Return a
                | Put String (IOAction a)
                | Get (String -> IOAction a)
```

IOAction is one of the following three types:

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

The Return constructor is the terminal operation for any program written in IOAction. Some simple actions include the one that prints to stdout before returning ():

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

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

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

A program is a sequence of actions. Operators for chaining actions and then performing them in a particular order would be required to execute a program. We could have the second IOAction depend on the return value of the first one. Consider the seqio operator described in Listing 13.3.

Listing 13.3 seqio operation

```
seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
seqio (Return a) f = f a
seqio (Put s io) f = Put s (seqio io f)
seqio (Get g)    f = Get (\s -> seqio (g s) f)
```

We want to take the IOAction a supplied in the first argument, get its return value (which is of type a) and feed that to the function in the second argument, getting an IOAction b out, which can be sequenced with the first IOAction a. Listing 13.4 describes an example of chaining IOActions and Listing 13.5 shows the output.

Listing 13.4 Example operation with IOActions

```
hello = put "What is your name?"      `seqio` \_    ->
        get                           `seqio` \name ->
        put "What is your age?"       `seqio` \_    ->
        get                           `seqio` \age  ->
        put ("Hello " ++ name ++ "!") `seqio` \_    ->
        put ("You are " ++ age ++ " years old")
```

Although this looks like imperative code, it's really a value of type IOAction (). In HASKELL, code can be data and data can be code.

Listing 13.5 Output of example operation

```
*Main> print hello
Put "What is your name?" (
  Get ($0 ->
    Put "What is your age?" (
      Get ($1 ->
        Put "Hello $0!" (
          Put "You are $1 years old" (
            Return ()
          )
        )
      )
    )
  )
)
```

IOAction is a monad. Listing 13.6 shows the instance for the same.

Listing 13.6 IOAction Monad

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

The main benefit of doing this is that we can now sequence actions using HASKELL's do notation. Listing 13.7 describes the example from Listing 13.4:

Listing 13.7 Example operation using do notation

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

Since no code is executed, till this the above example is pure and side-effect free.

To see the effects, we need to define a function that takes an `IOAction a` and converts it into a value of type `IO a`, which can then be executed by the interpreter or the runtime system. Listing 13.8 shows the `run` function for `IOAction`.

Listing 13.8 `run` function for `IOAction`

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

Listing 13.9 shows the output for the `run` function.

Listing 13.9 Output for `run` function

```
*Main> run hello
What is your name?
Chris
What is your age?
29
Hello Chris!
You are 29 years old
```

`IOAction` is a mini-language for doing impure, side-effecting code. It restricts the language constructs to only reading from `stdin` and writing to `stdout` in effect creating a safe eDSL.

13.2 Prototype architecture

Figure 13.1 shows architecture for prototype 4.

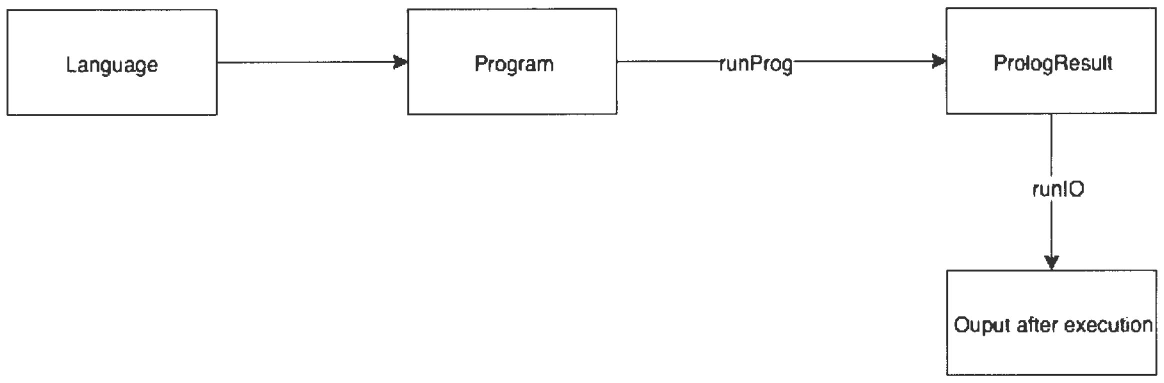


Figure 13.1: Prototype 4 architecture

This architecture adopts a two stage interpretation procedure with `runProg` and `runIO`. Consider the PROLOG-like language described in Listing 13.10.

Listing 13.10 Language with pure and impure constructors

```
data Prolog = Pure_Constructor_1 ...
            | Pure_Constructor_2 ...
            | Impure_Constructor_1 ...
            | Impure_Constructor_2 ...
        deriving (Show, Eq, Ord)
```

This abstract grammar encapsulates constructors which represent pure and impure actions. The first stage of the interpretation, i.e., `runProg` takes a program and returns a `PrologResult` as shown in Listing 13.11.

Listing 13.11 PROLOG-like language with `IO` constructors

```
data PrologResult
   = NoResult
   | Cons Unifier PrologResult
   | IOIn (IO String) (String -> PrologResult)
   | IOOut (IO ()) PrologResult
```

The `NoResult` is used for termination. If a most general unifier is reached then a `Cons` construct is returned. The last two constructors are for read and write operations respectively.

Listing 13.12 describes the type signature of `runProg`.

Listing 13.12 `runProg` type signature

```
runProg :: Prolog -> PrologResult
```

Till this point in the interpretation of the program no `IO` operations have been executed. As described in the previous section we construct functions which upon execution will produce results and hence the partially interpreted program is still pure.

The second stage of interpretation, i.e., `runIO` involves executing the impure actions within the `IO Monad` to produce the desired side effects. In this prototype, the impure actions as seen in Listing 13.11 are:

1. `IOIn (IO String) (String -> PrologResult)`, and
2. `IOIn (IO String) (String -> PrologResult)`

Listing 13.13 describes the type signature of `runIO`.

Listing 13.13 `runIO` type signature

```
runIO :: PrologResult -> IO [Unifier]
```

`runIO` accepts as input the result from `runProg`, i.e., `PrologResult` and executes each action in the `IO Monad`.

13.3 Summary

Recapitulating, this prototype gives an architecture for a two stage interpretation strategy for an eDSL. In this process the first stage produces a pure interpreted program while the latter executes each action to produce output. This approach provides modularity and control over the side effecting actions for execution.

Chapter 14

Future Scope

This chapter discusses additions that could be made to the contributions of this thesis.

14.1 PROLOG quasiquoter with antiquotation

As discussed in Section 2.4 and Section 7.6, a quasiquoter provides an economy of expression to describe a program in the eDSL. Listing 14.1 shows an example.

Listing 14.1 A sample quasi quoted expression for PROLOG in HASKELL

```
Clause [pr| $(X) is $(Y)|] [[pr| $(X) = $(eval(varY))|]]
```

This not only provides a simpler interface but also allows interchangeable usage of programming constructs of different languages in the same expression. For the example above, `X` is a PROLOG variable and `varY` is a HASKELL variable injected into the expression

14.2 Runtime search strategy selection

The PROLOG interpreter in Chapter 12 can work with multiple search strategies. The search must be provided at compile time and hence one must make an estimated guess to the nature of the problem to select the most appropriate choice. A possible solution to this problem could be the addition of providing and/or switching the search strategy at runtime. Listing 14.2 depicts an example query request.

Listing 14.2 Query resolver with variable search strategy

```
queryResolver searcStrategy query knowledgeBase
```

A potential improvement would be add a construct to the abstract grammar itself which can allow to change the search strategy dynamically.

14.3 Database operations for PROLOG

Operations for manipulating the clause database containing facts and clauses are provided with many PROLOG distributions such as SWI PROLOG [103]. Operations such `assertz` [102] would provide the ability to modify the knowledge base at runtime. This is only partially implemented by `prolog-0.2.0.1` [114].

Moreover, SWI PROLOG provides multiple mechanisms for storage and modification as described in [103].

14.4 Multi type variable language

The eDSL `FlatTerm` has a single type variable i.e.,

```
FlatTerm a
```

This restricts the constructors (generated terms) to all be of the same type or no type. An eDSL with multiple type variables will provide its constructors with multiple options resulting in multi-type terms generated by the grammar. Listing 14.3 shows an example.

Listing 14.3 Multi type variable eDSL

```
data FlatTerm a b c
        = Constructor_1 a
        | Constrcutor_2 b
        | ...
```

In the above example, the `c` type variable is not used by any constructor in `FlatTerm` and the fixed point will be calculated around it.

Continuing the example from the previous section, with multi typed constructors one can have constructor specific unification. Generally speaking, quantifiers and logic can be programmed per constructor adding to the extensibility of the language.

14.5 Additions and extensions to prototype 4

Chapter 13 defines a grammar for encapsulating and ultimately controlling impure actions in a language. The constructors are split into two types representing pure and impure actions. A program would be a chaining/ sequence of these actions. The possibility of a constructor having a pure and an impure component would be possible if the language has multiple type variables. Listing 14.4 provides an example.

Listing 14.4 Grammar with hybrid constructors

```
data ResultWithIO a b
        = PureConstructor_1 ....
        | PureConstructor_2 ....
        | IOContrcutor_1 .....
        | IOContructor_2 ...
        | ContructorWithBoth_1 .....
        | ContructorWithBoth_2 .....
        deriving(........)
```

Moreover, Chapter 13 lacks an accompanying implementation.

Recapitulating, this chapter provided us with points where this thesis could be extended in the future along with suitable samples for the respective implementations.

Chapter 15

Conclusion

This thesis set out to explore the various approaches used for bringing features of multiple languages into a single programming environment, and has identified embedding and paradigm integration as being the major ones. This study has also sought to improve the existing work on implementing PROLOG in HASKELL. One contribution of this thesis is a PROLOG-like language in HASKELL which not only is closer to a "real" PROLOG distribution but also provides logic programming functionality as natively as possible in the host language.

During this process we have also thrown light on the subject of eDSLs in HASKELL and the support for the same. Moreover, we have provided a methodology for replicating results achieved in this thesis.

HASKELL has been shown to be an effective tool for embedding DSLs. If used correctly HASKELL's lazy nature can be utilized to calculate results lazily; as in returning only a single result from all the possible ones. Opening up the language enables us to take advantage of the existing classes such as `Functor, Applicative, Monad, Foldable, Traversable`. Adding a type variable not only allows us to inject custom functionality into the language but permits jumping between the different grammars since they are isomorphic. It is fairly straight forward to embed a recursive untyped grammar into a typed language such as HASKELL by defining a data type with multiple constructors. Monads provide the functionality to define custom control flow mechanisms along with encapsulation of impure side effecting computations. Moreover, due to the nature of monads, a approach similar to Prototype 4 can not only be possible but also generalized, i.e., the separation of interpreting different types of code such as pure or impure.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Hassan Aït-Kaci, *Warren's abstract machine: A tutorial reconstruction*, MIT Press, 1991.

[3] Sergio Antoy, *Implementing functional logic programming languages*.

[4] ______, *Sergio antoy home page*.

[5] Sergio Antoy and Michael Hanus, *Functional logic programming*, Communications of the ACM **53** (2010), no. 4, 74–85.

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

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

[8] FP Complete Bartosz Milewski, *Understanding algebras*, October 2013.

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

[10] Nick Benton, *Embedded interpreters*, Journal of Functional Programming **15** (2005), no. 4, 503–542.

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

[12] James Bielman and Luís Oliveira, *Common lisp foreign function interface*, March 2014.

[13] Nikolaj Bjorner, Ken McMillan, and Andrey Rybalchenko, *On solving universally quantified horn clauses*, Static Analysis, Springer, 2013, pp. 105–125.

[14] Andrew Butterfield (ed.), *Unifying theories of programming, second international symposium, utp 2008, dublin, ireland, september 8-10, 2008, revised selected papers*, Lecture Notes in Computer Science, vol. 5713, Springer, 2010.

[15] C2, *Multi paradigm programming language*, September 2012.

[16] Prolog Development Center, *Visual prolog*, June 2013.

[17] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang, *A type-safe embedding of constraint handling rules into haskell*, Technical reportSchool of Computing, National University of Singapore, Boston, MA, USA (2003), 30.

[18] Ciao, *Ciao programming language*, August 2011.

[19] Koen Claessen and Peter Ljunglöf, *Typed logical variables in haskell.*, Electr. Notes Theor. Comput. Sci. **41** (2000), no. 1, 37.

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

[21] Mozart Consortium, *Oz / mozart*, March 2013.

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

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

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

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

[26] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: simplified data processing on large clusters*, Communications of the ACM **51** (2008), no. 1, 107–113.

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

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

[29] Free Dictionary, *Quasi-quotation*.

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

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

[32] Dan Doel, *The logict package*, August 2013, `http://hackage.haskell.org/package/logict`.

[33] ______, *The logict package example*, August 2013, `http://okmij.org/ftp/Computation/monads.html`.

[34] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan, *Making data structures persistent*, Proceedings of the eighteenth annual ACM symposium on Theory of computing, ACM, 1986, pp. 109–121.

[35] Steve Dunne and Bill Stoddart (eds.), *Unifying theories of programming, first international symposium, utp 2006, february 5-7, 2006, revised selected papers*, Lecture Notes in Computer Science, vol. 4010, Walworth Castle, County Durham, UK, Springer, February 2006.

[36] Switzerland École Polytechnique Fédérale de Lausanne (EPFL) Lausanne, *Scala programming language*, 2002-2014.

[37] Martin Erwig, *Escape from zurg: an exercise in logic programming*, Journal of Functional Programming **14** (2004), no. 03, 253–261.

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

[39] ______, *stream-monad*, September 2012.

[40] Adam C. Foltzer, *Molog*, March 2013.

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

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

[43] ______, *The logic-classes package*, October 2013, `http://hackage.haskell.org/package/logic-classes`.

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

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

[46] Software Guild, *Programming languages through the years*, July 2015.

[47] Michael Hanus, *Michael hanus home page*.

[48] Michael Hanus, *Multi-paradigm declarative languages*, Logic Programming, Springer, 2007, pp. 45–75.

[49] Michael Hanus, *Functional logic programming*, February 2009.

[50] Michael Hanus, Herbert Kuchen, and Juan José Moreno-Navarro, *Curry: A truly functional logic language*, Proc. ILPS (Leon Sterling, ed.), MIT Press, 1995, pp. 95–107.

[51] Seif Haridi and Sverker Janson, *Kernel andorra prolog and its computational model*, SICS Research Report (1990), 15.

[52] Haskell, *Haskell website*, January 2015.

[53] Haskellwiki, *Template haskell*, October 2013.

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

[55] Ernest Lepore Herman Cappelen, *Quotation*, January 2012.

[56] Roger Hindley, *The principal type-scheme of an object in combinatory logic*, Transactions of the american mathematical society (1969), 29–60.

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

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

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

[60] Paul Hudak, *Building domain-specific embedded languages*, ACM Comput. Surv. **28** (1996), no. 4es, 196.

[61] John Hughes, *Why functional programming matters*, The computer journal **32** (1989), no. 2, 98–107.

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

[63] ———, *Jscriptlog - prolog in javascript*, September 2012, `http://jlogic.sourceforge.net/index.html`.

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

[65] Simon L Peyton Jones, Jean-Marc Eber, and Julian Seward, *Composing contracts: An adventure in financial engineering*, FME, vol. 2021, 2001, p. 435.

[66] Mark Kantrowitz, *The prolog 1000 datbase*, August 2012.

[67] David Karger, *Persistent data structures*, September 2005.

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

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

[70] ______, *Teaching programming languages in a post-linnaean age*, SIGPLAN Not. **43** (2008), no. 11, 81–83.

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

[72] ______, *Embedding one language into another*, March 2005, `http://lambda-the-ultimate.org/node/578`.

[73] ______, *Application-specific foreign-interface generation*, October 2006, `http://lambda-the-ultimate.org/node/2304`.

[74] Duncan Temple Lang, *Embedding s in other languages and environments*, Proceedings of DSC, vol. 2, 2001, p. 1.

[75] LangPop.com, *Programming language popularity*, October 2013.

[76] learn Prolog Now, *Horn clauses.*

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

[78] John W Lloyd, *Programming in an integrated functional and logic language*, Journal of Functional and Logic Programming **3** (1999), no. 1-49, 68–69.

[79] Ricardo J. Machado, Rita Suzana Maciel, Julia Rubin, and Goetz Botterweck, *Model-based methodologies for pervasive and embedded software: 8th international workshop, mompes 2012, essen, germany, september 4, 2012, revised ... / programming and software engineering)*, Springer Publishing Company, Incorporated, 2013.

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

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

[82] Erik Meijer and Peter Drayton, *Static Typing Where Possible, Dynamic Typing When Needed*, Workshop on Revival of Dynamic Languages, 2005.

[83] Bertrand Meyer, *Eiffel as a framework for verification*, Verified Software: Theories, Tools, Experiments, Springer, 2008, pp. 301–307.

[84] Robin Milner, *A theory of type polymorphism in programming*, Journal of computer and system sciences **17** (1978), no. 3, 348–375.

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

[86] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo, *Logic programming with functions and predicates: The language babel*, The Journal of Logic Programming **12** (1992), no. 3, 191–223.

[87] R Morrison and MP Atkinson, *Persistent languages and architectures*, Security and Persistence, Springer, 1990, pp. 9–28.

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

[89] Jaimie Murdock, *Haskell kanren*, March 2012.

[90] Gopalan Nadathur, *λ prolog*, September 2013.

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

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

[93] Johan Nordlander, *O'haskell*, January 2001.

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

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

[96] University of Miami, *Prolog introduction*, March 2012.

[97] Hayato Ohwada and Fumio Mizoguchi, *Managing search in parallel logic programming*, Logic Programming'87, Springer, 1987, pp. 148–177.

[98] Ocaml Org, *Ocaml programming language*, March 2014.

[99] Johan Bos Patrick Blackburn and Kristina Striegnitz, *The cut*, February 2003.

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

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

[102] SWI Prolog, *assertz in swi prolog*.

[103] ———, *Database operations in swi prolog*.

[104] Yield Prolog, *Yield prolog*, October 2011, `http://yieldprolog.sourceforge.net/`.

[105] Shengchao Qin (ed.), *Unifying theories of programming - third international symposium, utp 2010, shanghai, china, november 15-16, 2010. proceedings*, Lecture Notes in Computer Science, vol. 6445, Springer, 2010.

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

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

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

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

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

[111] John Alan Robinson and EE Silbert, *Loglisp: an alternative to prolog*, School of Computer and Information SCience, Syracuse University, 1980.

[112] Raúl Rojas, *A tutorial introduction to the lambda calculus*, DOI= http://www. utdallas. edu/˜ gupta/courses/apl/lambda. pdf (2004), 9.

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

[114] ———, *The prolog package*, June 2012, `http://hackage.haskell.org/package/prolog`.

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

[116] Silvija Seres, *The algebra of logic programming*, Ph.D. thesis, Oxford University, 2001.

[117] Silvija Seres and Shin-Cheng Mu, *Optimisation problems in logic programming: an algebraic approach*, Logic Programming and Software Engineering (2000), 19.

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

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

[120] Tim Sheard and Emir Pasalic, *Two-level types and parameterized modules*, Journal of Functional Programming **14** (2004), no. 05, 547–587.

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

[122] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard O'Keefe, *Logic programming for the real world*, Proceedings of the ILPS, vol. 95, 1995, pp. 83–94.

[123] Andy Sonnenburg, *logicst*, April 2013.

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

[125] JM Spivey and Silvija Seres, *Embedding prolog in haskell*, Proceedings of Haskell, vol. 99, Citeseer, 1999, pp. 1999–28.

[126] Michael Spivey, *Functional pearls combinators for breadth-first search*, Journal of Functional Programming **10** (2000), no. 4, 397–408.

[127] Mike Spivey and Silvija Seres, *The algebra of searching*, PROCEEDINGS OF A SYMPOSIUM IN CELEBRATION OF THE WORK OF, MacMillan, 1999.

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

[129] ———, *How does the st monad work?*, September 2012.

[130] Leon Sterling and Ehud Shapiro, *The art of prolog (2nd ed.): Advanced programming techniques*, MIT Press, Cambridge, MA, USA, 1994.

[131] Patrick Blackburn Johan Bos Kristina Striegnitz, *Learn prolog now*, January 2012.

[132] ———, *Learn prolog now*, January 2012.

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

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

[135] Chris Taylor, *Io is pure*, February 2013.

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

[137] Wren Thornton, *The unification-fd package*, July 2012, `http://hackage.haskell.org/package/unification-fd`.

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

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

[140] Dalhousie University, *Control flow*, January 2012.

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

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

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

[144] Arie Van Deursen, Paul Klint, and Joost Visser, *Domain-specific languages: An annotated bibliography.*, Sigplan Notices **35** (2000), no. 6, 26–36.

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

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

[147] Job Vranish, *minikanrent*, March 2013.

[148] Philip Wadler, *Comprehending monads*, Mathematical Structures in Computer Science **2** (1992), no. 04, 461–493.

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

[150] ______, *Logic programming example in haskell*, February 2010.

[151] ______, *Quasiquotation in haskell*, January 2014, `http://www.haskell.org/haskellwiki/Quasiquotation`.

[152] Haskell Wiki, *Monads as computation*, December 2011.

[153] ______, *Kind*, August 2012.

[154] ______, *St monad*, July 2012.

[155] ______, *Foldable and traversable*, January 2013.

[156] ______, *The haskell programming language*, October 2013.

[157] ______, *Haskell/laziness*, November 2014.

[158] ______, *Monads in haskell*, January 2014.

[159] ______, *Haskell in industry*, June 2015.

[160] Wikipedia, *Prolog wikipedia*, March 2004.

[161] ______, *Functional logic programming languages*, February 2005.

[162] ———, *Common lisp object system*, December 2013.

[163] ———, *Curry programming language*, December 2013.

[164] ———, *Functional logic programming*, May 2013.

[165] ———, *Quasiquotation*, Novemeber 2013, `http://en.wikipedia.org/wiki/Quasi-quotation`.

[166] ———, *Common language infrastructure*, February 2014.

[167] ———, *Common language runtime*, March 2014.

[168] ———, *Damas-hindley-milner type system*, February 2014.

[169] ———, *Foreign function interface*, January 2014.

[170] ———, *Lambda calculus*, March 2014.

[171] ———, *List of multi paradigm languages*, March 2014.

[172] ———, *Meta programming*, March 2014.

[173] ———, *Ocaml*, March 2014.

[174] ———, *Programming paradigm*, March 2014.

[175] ———, *Comparison of prolog implementations*, August 2015.

[176] ———, *Control flow*, August 2015.

[177] ———, *Declarative programming*, September 2015.

[178] ———, *Timeline of programming languages*, November 2015.

[179] ———, *cut in prolog*, January 2016.

[180] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi (eds.), *Unifying theories of programming, 4th international symposium, utp 2012, paris, france, august 27-28, 2012, revised selected papers*, Lecture Notes in Computer Science, vol. 7681, Springer, 2013.

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

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