Embedding Programming Languages: PROLOG in HASKELL

by

Mehul Chandrakant Solanki

Bachelor of Engineering in Computer Science and Engineering Mumbai University 2012

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

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

November 2015

Abstract

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

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

0.1 Thesis Statement

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

We explore embedding domain specific languages in HASKELL

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is this chapter about

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

1.2 Beginnings

Programming has become an integral part of working and interacting with computers and day by day more and more complex problems are being tackled using the power of programming technologies. It is possibly the only way to talk to computers and hence the need for a robust and multi purpose programming language has never been more urgent. The desirability of a programming language depends on a lot of factors such as the ease of use, the features and functionalities that it provides, adaptability and what sort of problems can it solve. One is spoilt for choice with a number of options for a wide variety of programming paradigms, for example Object Oriented Languages.

Over the last decade the declarative style of programming has gained popularity. The methodologies that have stood out are the Functional and Logical Approaches. The former is based on Functions and Lambda Calculus, while the latter is based on Horn Clause

Logic. Each of them has its own advantages and aws. How does one choose which approach to adopt? Perhaps one does not need to choose! This document looks at the attempts, improvements and future possibilities of uniting Haskell, a Purely Functional Programming Language and Prolog, a Logical Programming Language so that one is not forced to choose.

1.3 Thesis Statement

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

1.4 Problem Statement

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

Prolog is a language that has a hard time being adopted. Born in an era where procedural languages were receiving a lot of attention, it suered from competing against another new kid on the block: C. Some of the problems were of its own making. Basic features like modules were not provided by all compilers. Practical features for real world problems were added in an ad hoc way resulting in the loss of its purely declarative charm. Some say that PROLOG is fading away, [89, 140, 139]. It is apparently not used for building large programs [154, 117, 67]. However there are a lot of good things about Prolog: it is ideal for search problems; it has a simple syntax, and a strong underlying theory. It is a language that should not die away.

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

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

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

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

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

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

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

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

This can be achieved in two ways,

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

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

1.5 Thesis Organization

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

1.6 Chapter Recap

Chapter 2

Background

2.1 What is this chapter about

Programming Languages fall into different categories also known as ”paradigms”. They exhibit different characteristics according to the paradigm they fall into. It has been argued [72] that rather than classifying a language into a particular paradigm, it is more accurate that a language exhibits a set of characteristics from a number of paradigms. Either way, the broader the scope of a language the more the expressibility or use it has.

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

Functional Programming, [59] gets its name as the fundamental concept is to apply mathematical functions to arguments to get results. A program itself consists of functions and functions only which when applied to arguments produce results without changing the state that is values on variables and so on. Higher order functions allow functions to be passed as arguments to other functions. The roots lie in λ-calculus [166], a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application using variable binding and substitution. It can be thought as the smallest programming language [107], a single rule and a single function definition scheme. In particular there are typed and untyped λ calculi. In the untyped λ calculus functions have no predetermined type whereas typed lambda calculus puts restriction on what sort(type) of data can a function work with. SCHEME is based on the untyped variant while ML and HASKELL are based on typed λ calculus. Most typed λ calculus langauges are based on Hindley-Milner or Damas-Milner or Damas- Hindley-Milner [164] type system. The ability of the type system to give the most general type of a program without any help (annotation). The algorithm [22] 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.

Logical Programming, [119] on the other hand is based on formal logic. A program is a set of rules and formulæ in symbolic logic that are used to derive new formulas from the old ones. This is done until the one which gives the solution is not derived.

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

PROLOG [154] 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 [77] functional languages around and is the first language to incorporate Monads [142] for safe *IO*. Monads can be described as composable computation descriptions [152] . Each monad consists of a description of what has action has to be executed, how the action has to be run and how to combine such computations. An action can describe an impure or side-effecting computation, for example, *IO* can be performed outside the language but can be brought together with pure functions inside in a program resulting in a separation and maintaining safety with practicality. HASKELL computes results lazily and is strongly typed.

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

The selection of languages is not uncommon and this not only the case with HASKELL, PROLOG seems to be the all time favourite for ”let's implement PROLOG in the language X for proving it's power and expressibility”. The PROLOG language has been partially implemented [34] in other languages like SCHEME [116], LISP [70, 105, 106], JAVA [154, 62], JAVASCRIPT [63] and the list [99] goes on and on.

The technique of embedding is a shallow one, it is as if the embedded language floats over the host. Over time there has been an approach that branches out, which is Paradigm Integration. A lot of work has been done on Unifying the Theories of Programming [36, 14, 100, 177, 56, 46]. All sorts of hybrid languages which have characteristics from more than one paradigm are coming into the mainstream.

Before moving on, let us take a look at some terms related to the content above. To begin with Foreign Function Interfaces (FFI) [165], a mechanism by which a program written in one programming language can make use of services written in another. For example, a function written in C can be called within a program written in HASKELL and vice versa through the FFI mechanism. Currently the HASKELL foreign function interface works only for one language. Another notable example is the Common Foreign Function Interface (CFFI) [13] for LISP which provides fairly complete support for C functions and data. JAVA provides the Java Native Interface(JNI) for the working with other languages. Moreover there are services that provide a common platform for multiple languages to work with each other and run their programs. They can be termed as multi lingual run times which lay down a common layer for languages to use each others functions. An example for this is the Microsoft Common Language Runtime (CLR) [161] which is an implementation of the Common Language Infrastructure (CLI) standard [160].

Another important concept is meta programming [168], which involves writing computer programs that write or manipulate other programs. The language used to write meta programs is known as the meta language while the the language in which the program to be modified is written is the object language. If both of them are the same then the language is said to be reflective. HASKELL programs can be modified using Template HASKELL [52] an extension to the language which provides services to jump between the two types of programs. The abstract syntax trees in the form of HASKELL data types can be modified at compile time which playing with the code and going back and forth.

A specific tool used in meta programming is quasi quotation [80, 145, 159], permits HASKELL expressions and patterns to be constructed using domain specific, programmer-defined concrete syntax. For example, consider a particular application that requires a complex data type. To accommodate the same it has to represented using HASKELL syntax and preforming pattern matching may turn into a tedious task. So having the option of using specific syntax reduces the programmer from this burden and this is where a quasi-quoter comes into the picture. Template HASKELL provides the facilities mentioned above. For example, consider the following code in PROLOG to append two lists, going through the code, the first rule says that and empty list appended with any list results in the list itself. The second predicate matches the head of the first and the resulting lists and then recurs on the tails. The same in HASKELL,

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

Consider the Object Functional Programming Language, SCALA [180], it is purely functional but with objects and classes. With the above in mind, coming back to the problem of implementing PROLOG in HASKELL. There have been quite a few attempts to "merge" the two programming languages from different programming paradigms. The attempts fall into two categories as follows,

1. Embedding, where PROLOG is merely translated to the host language HASKELL or a Foreign Function Interface.
2. Paradigm Integration, developing a hybrid programming language that is a Functional Logic Programming Language with a set of characteristics derived from both the participating languages.

The approaches listed above are next in line for discussions.

2.2 Chapter Recap

Chapter 3

Accomplished Work

3.1 What is this chapter about

3.2 Current Work

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

1. Very few embedded implementations exist which offer a perspective into the job at hand. One of the earliest implementations [65] is for an older specification of HASKELL called HASKELL 98 `hugs`. It is more of a proof of concept providing a mechanism to include variable search strategies in order to produce a result. Another implementation [178] based of it simplifies the notation to a list format. Nonetheless, both implementations lack simplicity and support for basic PROLOG features such as *cuts, fails, assert* among others.

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

complete and most lack many practical parts of PROLOG.

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

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

And as far as the idea of merging paradigms goes, it is not the main focus of this thesis and can be more of an ”add-on”. A handful of crossover hybrid languages based on HASKELL exist, CURRY [138] being the prominent one. Moving away from HASKELL and exploring other languages from different paradigms, a respectable number of crossover implementations exist but again most of them have faded out.

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

3.3 Contributions

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

3.4 Improved Contributions

1. Most languages have a recursive abstract syntax which restricts the eDSL in terms of its capability to *open up* the language i.e. to include meta syntactic variables, adding 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 a polymorphically typed embedded language within HASKELL

   To test it out we adopt the closed PROLOG like language defined in [109] and open it up. And for the unification part we use [130], which provides a generic unification algorithm implementation encapsulated into a monad.

2. (Prototype 2) does the what a PROLOG query resolver would do given a query and a knowledge base. The mechanism for the same is adopted from [109]. 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.

3. (Prototype 3) demonstrates the modularity of the unification process of the query

resolver with multiple search strategies.

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 only produced only when the actions are executed.

3.5 Thesis Contributions

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

   You can make an Flatterm int

   but you cannot make term int

   adding quantifiers

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

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

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

3.6 What work was done in terms of points

1. Literature review on eDSL's.

2. Short survey on multi paradigm declarative languages.
3. Accumulated and evaluated PROLOG in HASKELL.
4. Defined a procedure to open up a language starting from a generic recursive abstract syntax.
5. Made a few libraries to work together.
6. Some stuff for monadic unification.
7. Something to show it was modular and independent of the original grammar.
8. Something to show that the unification part is independent of the search strategy and hence multiple ones can be used, possibly simultaneously to find a solution.
9. Creating a micro language to represent and encapsulate IO operation in an eDSL so that the it remains pure even after interpretation and only produces side effects when the action is actually executed and hence in some way it can be controlled.

3.7 Chapter Recap

3.8 What is this chapter about

Chapter 4

Embedding a Programming Language into another Programming Language

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

4.1 The Informal Content from Blogs, Articles and Internet Discussions

Before moving on to the formal content such as publications, modules and libraries let's take a look at some of the unofficially published content. This subsection takes a look at the information, thoughts and discussions that are currently taking place from time to time on the internet. A lot of interesting content is generated which has often led to some formal 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 [73], which lists a number of PROLOG implementations in a variety of languages like LISP, SCHEME, SCALA, JAVA, JAVASCRIPT, RACKET [116] and so on. Moreover the discussion focusses on a lot of critical points that should be considered in a translation of PROLOG to the host language regarding types and modules among others.

One of the implementations discussed redirects us to one of the most earliest implementations of PROLOG in HASKELL for Hugs 98, called Mini PROLOG [65]. Although this implementation takes as reference the working of the PROLOG Engine and other details, it still is an unofficial implementation with almost no documentation, support or ongoing development. Moreover, it comes with an option of three engines to play with but still lacks complete list support and a lot of practical features that PROLOG has and this seems to be a common problem with the only other implementation that exists, [178].

Adding fuel to fire, is the question on PROLOG's existence and survival [139, 89, 140, 117] since its use in industry is far scarce than the leading languages of other paradigms. The purely declarative nature lacks basic requirements such as support for modules. And then there is the ongoing comparison between the siblings [179] of the same family, the family of Declarative Languages. Not to forget HASKELL also has some tricks [143] up its sleeve which enables encoding of search problems.

4.2 Related Books

As HASKELL is relatively new in terms of being popular, its predecessors like SCHEME have explored the territory of embedding quite profoundly [27], which aims at adding a few constructs to the language to bring together both styles of Declarative Programming and capture the essence of PROLOG. Moreover, HASKELL also claims for it to be suitable for basic Logic Programming naturally using the List Monad [144]. A general out look towards implementing PROLOG has also been discussed by [71] to push the ideas forward.

4.3 Related Papers

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

- Papers from Mike Spivey and Silvija Seres

  The work presented in the series [121, 113, 114, 120, 111] attempts to encapsulate various aspects of an embedding of PROLOG in HASKELL. Being the very first documented formal attempt, the work is influenced by similar embeddings of PROLOG in other languages like SCHEME and LISP. Although the host language has distinct characteristics such as lazy evaluation and strong type system the proposed scheme tends to be general as the aim here is to achieve PROLOG like working not a multi paradigm declarative language. PROLOG predicates are translated to HASKELL functions which produce a stream of results lazily depicting depth first search with support for different strategies and practical operators such as *cut* and *fail* with higher order functions. The papers provide a minimalistic extension to HASKELL with only four new constructs. Though no implementation exists, the synthesis and transformation techniques for functional programs have been *logicalised* and applied to PROLOG programs. Another related work [122] looks through conventional data types so as to adapt to the problems at hand so as to accommodate and jump between search strategies.

- Other works related or based on the above

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

4.4 Related Libraries in Haskell

- Prolog Libraries

  To replicate Prolog like capabilities Haskell seems to be already in the race with a host of related libraries. First we begin with the libraries about Prolog itself, a few exist [126] being a preliminary or "mini Prolog" as such with not much in it to be able to be uselul, [127] is all powerful but is an Foreign Function Interface so it is "Prolog in Haskell" but we need Prolog for it, [109] which is the only implementation that comes the closest to something like an actual practical Prolog. But all they give is a small interpreter, none or a few practical features, incomplete support for lists, minor or no monadic support and an REPL without the ability to ”write a Prolog Program File”.

- Logic Libraries

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

- Unification Libraries

  The more specific the feature the lesser the support in Haskell. Moving on to the other distinct feature of Prolog is Unification, two libraries exist [130], [101] that unify two Prolog Terms and return the resulting substitution.

- Backtracking

  Another important aspect of PROLOG is backtracking. To simulate it in HASKELL, the libraries [41, 118] use monads. Moreover, there is a package for the EGISON programming language [57] which supports non-linear pattern-matching with backtracking.

4.5 From chap 7

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

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

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

A lot of the above can be skipped or taken from the base language if an embedding approach is chosen. For an embedded domain specific language the functionality is translated and written as an add on. The result can be thought of as a library. But the difference between an ordinary library and an eDSl is the feature set provided and the degree of embedding [150]. For example, reading a file and parsing its contents to perform certain operations to return *string* results is a shallow form of embedding as the generation of code, results is not native nor are the functions processing them dealing with embedded data types as such. On the other hand, building data structures in the base language which represent the target language expression would be called a deep embedding approach.

The snippet of Haskell code below describes Prolog entities,

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

```
            | PString   !String
            | PInteger  !Integer
            | PFloat    !Double
            | Flat [FlatItem]
            | Cut Int
      deriving (Eq, Data, Typeable)
```

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

As discussed in the

4.6 Theory

1. Papers

    (a) Embedding an interpreted language using higher-order functions, [102]

    (b) Building domain-specific embedded languages, [58]

    (c) Embedded interpreters, [11]

    (d) Cayenne – a Language With Dependent Types, [6]

    (e) Foreign interface for PLT Scheme, [10]

    (f) Dot-Scheme: A PLT Scheme FFI for the .NET framework, [96]

    (g) Application-specific foreign-interface generation, [103]

    (h) Embedding S in other languages and environments, [76]

2. Books

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

3. Articles / Blogs / Discussions

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

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

   (c) Linguistic Abstraction, [92]

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

4. Websites

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

4.7 Implementations

1. Lots of them I guess

4.8 Important People

1. ????

4.9 Miscellaneous / Possibly Related Content

1. ????

4.10 Chapter Recap

Chapter 5

Multi Paradigm Languages (Functional Logic Languages)

5.1 What is this chapter about

Over the years another approach has branched off from embedding languages, to merge and/or integrate programming languages from different paradigms. Let us take an example of the SCALA Programming Language [180], a hybrid Object-Functional Programming Language which takes a leaf from each of the two books. In this thesis, the languages in question are HASKELL and PROLOG. This section takes a look at the literature on Multi Paradigm Languages, mainly Functional Logic Programming Languages that combine two of the most widespread Declarative Programming Styles.

A peak into language classification reveals that it is not always a straight forward task to segregate languages according to their features and/or characteristics. Turns out that there are a number of notions which play a role in deciding where the language belongs. Many a times a language ends up being a part of almost all paradigms due extensive libraries. Simply speaking, a multi-paradigm programming language is a programming language that supports more than one programming paradigm [72], more over as Timothy Budd puts it [170] ”The idea of a multi paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms.”

5.2 The Informal Content from Blogs, Articles and Internet Discussions

- Multi Paradigm Languages

  A lot has been talked and discussed on coming to clear grounds about the classification of programming languages. If the conventional ideology is considered then the scope of each language is pretty much infinite as small extension modules replicate different feature sets which are not naturally native to the language itself. The definitions of multi paradigm languages across the web [170, 90, 15] converge to roughly the same thing that of providing a framework to work with different styles with a list of languages [167, 33] that ticks the boxes. Generally speaking, it does not feel all that hot or popular in programming circles; one reason could be that it is a very broad topic and specifying details can clear the fog.

- Functional Logic Programming Languages

  Continuing from the previous section, narrowing down the search by considering only multi paradigm declarative languages namely, Functional Logical programming languages. By doing so a large amount of information pops up, from articles that give brief description and mentions [158, 155] to the implementing techniques [3] which give a brief overview of the aim and also the backdrop of publications.

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

5.3 Literature and Publications

- Multi Paradigm Languages

  Possibly one of the most important works towards bringing programming styles together is the book by C.A.R. Hoare [56] which points out that among the large number of programming paradigms and/or theories the unification theory serves as a complementary rather than a replacement to relate the universe. As as always since we are talking about HASKELL we have to include monads and unifying theories using monads [46].

- Functional Logic Programming Languages

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

  One of the most prominent multi paradigm languages in HASKELL is CURRY [5]. Th syntax is borrowed from the parent language and so are a lot of the features. Taking a recap, a functional programming language works on the notion of mathematical functions while a logic programming language is based on predicate logic. The strong points of CURRY are that the features or basis of the language are general and are visible in a number of languages like [29]. The language can play with problems from both worlds. In a problem where there are no unknowns and/or variables the language behaves like a functional language which is pattern matching the rules and execute the respective bodies. In the case of missing information, it behaves like PROLOG; a sub-expression e is evaluated on the conditions that it should satisfy which constraint the possible values of e. This brings us to the first important feature of functional logic languages *narrowing*. The expressions contain *free variables*; simply speaking incomplete information that needs to be *unified* to a value depending on the constraints of the problem. The language introduces only a few new constructs to support non determinism and choice. Firstly, *narrowing* (=:=), which deals with the expressions and unknown values and binds them with appropriate values. The next one is the *choice* operator (?) for non-deterministic operations. Lastly, for unifying variables and values under some conditions, (&) operator has been provided to add constraints to the equation. Putting it all together, it gives us the feel of a logic language for something that looks very much like HASKELL. Unification is like two way pattern matching and with a similar analogy CURRY is a HASKELL that works both ways and hence variables can be on either sides. Although the language can do a lot but gaps do exist such as the improvement of narrowing techniques.

5.4 Some Multi Paradigm Languages

The list of multi paradigm languages is huge, but in this thesis we will mostly stick to Functional Logical programming languages. Beginning with functional hybrids, a small project language called VIRGIL [137], combining objects to work with functions and procedures. On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [156]. This can be justified as object oriented programming has been one of the most dominant styles of programming and hence even HASKELL has one called O'HASKELL [91] though it last saw a release back in 2001. Another prominent implementation is OCAML [169, 95] which adds object oriented capabilities with a powerful type system and module support. This is the case with most of the languages in this section hardly a few have survived as the new ones incorporated the positives of the old. As mentioned before one of the most poplar [77] and widely usage both in academia and industry is the SCALA [180] programming language stands out.

5.5 Functional Logic Programming Languages

Knowing that there is quite some amount of literature out there on these type of languages, it is fairly easy to say that there have been numerous attempts at specifications and/or implementations. Sadly though not many have survived leave alone being successful as a result of the competition. Only the ones that are easily available or have an implementation or have been cited or referred by other attempts have been included as the list is long and does not reflect the main intention of the document. Beginning with the ones from Australia, which seems to be a popular destination for fiddling with PROLOG and merging paradigms. As of now there have been three popular ones, beginning with NEU PROLOG, [78], OZ (MOZART PROGRAMMING SYSTEM) [23] and MERCURY [30]. Delving deeper the languages feel more like extensions of PROLOG rather than hybrids. Starting with MERCURY which a boundary between deterministic and non-deterministic programs, similarly NUE PROLOG has special support for functions while OZ gives concurrent constraint programming plus distributed support, with different function types for goal solving and expression rewriting. ESCHER [79] comes very close to HASKELL with monads, higher order functions and lazy evaluation. Taking a look at PROLOG variants, CIAO [20]; a preprocessor to PROLOG for functional syntax support, λ PROLOG [88] aims at modular higher order programming with abstract data types in a logical setting, BABEL [53, 85, 84] combines pure PROLOG with a first order functional notation, LIFE [136] is for Logic, Inheritance, Functions and Equations in PROLOG syntax with currying and other features like functional languages and others [12, 81].

The functional language SCHEME is a very popular choice for this sort of a thing. With a book [27] and an implementation to accompany [28, 129] which seems to have translated into HASKELL, [61, 42, 141].

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 [134, 29].

5.6 From chap 9

Unifying / Marrying / Merging / Combining Programming Paradigms / Theories

5.7 Theory

- Papers
    1. Unifying Theories of Programming with Monads, [46]
    2. Symposium on Unifying Theories of Programming, 2006, [36].
    3. Symposium on Unifying Theories of Programming, 2008, [14].
    4. Symposium on Unifying Theories of Programming, 2010, [100].
    5. Symposium on Unifying Theories of Programming, 2012, [177].
- Books
    1. Unifying Theories of Programming, [56]
- Articles / Blogs / Discussions
    1. ???
- Websites
    1. ???

5.8 Implementations

1. Scala
2. Virgil
3. CLOS, Common Lisp Object System

4. Visual Prolog

5. ????

5.9 Miscellaneous / Possibly Related Content

1. ???

5.10 Chapter Recap

Chapter 6

Related Concepts

6.1 What is this chapter about

There are some technicalities which are indirectly related to the problem but do not bare a point of contact. The underpinnings of the languages throw some more light on the how different languages work to solve a problem. Different programming paradigms incorporate different operational mechanisms. For example, PROLOG programs execute on the Warren Abstract Machine [2] which has three different storage usages; a global stack for compound terms, for environment frames and choice points and lastly the trail to record which variables bindings ought to be undone on backtracking.

Constraint programming [163] is closely related to the declarative programming paradigm in the sense that the relations between variables is specified in the form of constraints. For example, consider a program to solve a simultaneous equation, now adding on to that restricting the range of the values that the variables can possible take, thus adding constraints to the possible solutions. Related to the same are Constraint Handling Rules [162], which are extensions to a language, simply speaking adding constraints to a language like PROLOG.

Lastly some details on the working of functional logic programming languages, residuation and narrowing [51, 157]. Residuation involves delaying of functions calls until they are deterministic, that is, deterministic reduction of functions with partial data. This principle is used in languages like ESCHER [79], LIFE [136], NUE-PROLOG [78] and OZ [23]. 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 [134], BABEL [53], LPG [12] and CURRY [138].

F-Algebras

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

An F-algebra consists of:

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

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

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

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

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

6.2 Chapter Recap

Chapter 7

Prolog in ____ other languages

Prolog in ______

7.1 Theory

- Papers
    1. QLog, [70]
    2. LogLisp Motivation, design, and implementation, [105]
- Books
    1. Warrens Abstract Machine A TUTORIAL RECONSTRUCTION, [2]
    2. LOGLISP: an alternative to PROLOG, [106]
- Articles / Blogs / Discussions
    1. Hello
- Websites
    1. Hello

7.2 Implementations

1. Castor : Logic paradigm for C++, [87]
2. GNU Prolog for Java, [47]
3. JLog - Prolog in Java, [62]
4. JScriptLog - Prolog in Java, [63]
5. Quintus Prolog, [97]
6. Yield Prolog, [99]
7. Racklog, [116]

7.3 Important People

1. ???

7.4 Miscellaneous / Possibly Related Content

1. ???

Chapter 8

Prolog in Haskell

Prolog in Haskell

8.1 Theory

- Papers

    1. Embedding Prolog in Haskell / Functional Reading of Logic Programs, [121]
    2. Algebra of Logic Programming, [113]
    3. The Algebra of Logic Programming, [111]
    4. Optimisation Problems in Logic Programming : An Algebraic Approach, [112]
    5. Higher Order Transformation of Logic Programs, [114]
    6. The Algebra of Searching, [120]
    7. FUNCTIONAL PEARL Combinators for breadth-first search, [122]
    8. Type Logic Variables, K Classen, [21]
    9. A Type-Safe Embedding of Constraint Handling Rules into Haskell Wei-Ngan Chin, Mar-tin Sulzmann and Meng Wang, [19]

10. Prological Features in a Functional Setting Axioms and Implementation, R Hinze, [55]
11. Escape from Zurg: An Exercise in Logic Programming, [39]

- Books

  1. The Reasoned Schemer, Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, [27]
  2. Programming Languages: Application and Interpretation, Shriram Krishnamurthi, Chapters 33-34 of PLAI discuss Prolog and implementing Prolog, [71]

- Articles / Blogs / Discussions

  1. Lambda the Ultimate, Programming Languages, [73]
  2. Takashi's Workplace (Implementation), [178]
  3. Haskell vs. Prolog Comparison, [123]

- Websites

  1. Logic Programming in Haskell, [143]

8.2 Implementations

1. A Prolog in Haskell, Takashi's Workplace, [178]
2. Mini Prolog for Hugs 98, [65]
3. Nano Prolog, [126]
4. Prolog, [109]
5. cspm-To-Prolog, [43]

6. prolog-graph, [9]
7. prolog-graph-lib, [108]
8. hswip, [127]

8.3 Important People

1. Mike Spivey
2. Silvija Seres

8.4 Miscellaneous / Possibly Related Content

1. Unification Libraries
   - (a) unification-fd, [130]
   - (b) cmu, [101]
2. Logic Libraries
   - (a) logicct, [25], [26]
   - (b) logic-classes, [**?**]
   - (c) proplogic, [44]
   - (d) cflp, [40]
   - (e) logic-grows-on-trees, [24]
3. Concatenative Programming
   - (a) peg, [31]
4. Constraint Programming and Constraint Handling Rules

(a) monadiccp, [104]

(b) monadicccp-gecode, [133]

(c) csp, [7]

(d) liquid fix point, [110]

Chapter 9

Quasiquotation

9.1 Theory

1. Papers
   (a)
2. Books
   (a)
3. Articles / Blogs / Discussions
   (a)
4. Websites
   (a) Quasiquotation Wikipedia, [159]
   (b) Quasiquotation in Haskell, [145]

9.2 Implementations

1.

9.3 Miscellaneous / Possibly Related Content

1.

9.4 What is Quasiquotation ?

1. [54]

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

2. [32]

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

3. Quasiquotation Wikipedia, [159]

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

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

9.5 Quasiquotaion in HASKELL

[145, 80]

Quasiquoting allows programmers to use custom, domain-specific syntax to construct fragments of their program. Along with HASKELL's existing support for domain specific languages, 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.

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

9.6 Chapter Recap

9.7 What is this chapter about

Chapter 10

Meta Syntactic Variables

Some sources for the topic

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

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

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

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

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

[16]

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

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

10.1 Chapter Recap

Chapter 11

Haskell or Why Haskell ?

11.1 What is this chapter about

In this chapter we discuss the properties of HASKELL

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

1. Why a Functional Language?

2. HASKELL as a functional programming language Haskell is an advanced purely-functional programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language [149]. It is one of the popular functional programming languages [77]. HASKELL is widely used in the industry [153].

   Shifting a bit to Embedded Domain Specific Languages (EDSLs) such as Emacs LISP. Opting for embedding provides a ”shortcut” to create a language which may be designed to provide specific functionality. Designing a language from scratch would require writing a parser, code generator / interpreter and possibly a debugger, not to mention all the routine stuff that every language needs like variables, control structures and arithmetic types. All of the aforementioned are provided by the host language; in this case HASKELL. Examples for the same can be found here [66, 83] which talk about introducing combinator libraries for custom functionality.

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

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

   (a) Monads

   Control flow defines the order/ manner of execution of statements in a program[172]. 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 [135]. 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[173].

   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[146]. A monad gives a bubble within the language to allow modification of control flow without affecting the rest of the universe. This is especially useful while handling side effects.

   A related topic would be of persistence languages, architectures and data structures. Persistent programming is concerned with creating and manipulating data in a manner that is independent of its lifetime [86]. A persistent data structure supports access to multiple versions which may arise after modifications [35, 68]. 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. And HASKELL allows creation of custom monads relieving the burden of dealing with a fixed model of the host language.

(b) 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 [151].

11.2 Chapter Recap

Chapter 12

Prolog or Why Prolog ?

12.1 What is this chapter about

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

1. Why a Logic Programming Language ?

2. PROLOG as a logic programming language.

   PROLOG is a general purpose logic programming language mainly used in artificial intelligence and computational linguistics. It is a Declarative language i.e. a program is a set of facts an rules running a query on which will return a result. The relation between them is defined by clauses using *Horn Clauses*[154]. PROLOG is very popular and has a number of implementations [171] for different purposes.

3. Why embed PROLOG ?

   (a) Existing Implementations

      As a starting point a few publications and implementations helped in exploring the topic. The shortcomings were clearly visible to work and improve upon giving a starting point.

(b) Simple Syntax [154]

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

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.

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses, a Turing-complete subset of first-order predicate logic. There are two types of clauses: Facts and rules.

[94] In Prolog all data objects are called terms Atomic terms
Come in two forms, atoms and integers. Atoms (this is a misnomer as in logic predicates are called atoms and atoms are called constants. However, we’ll stick to the Prolog convention.) Strings of alphanumerics and _, starting with a lower case alphabetic. Strings enclosed in ’single quotes’ Integers are numeric Example

```
geoff
'the cat and the rat'
'ABCD'
123
```

Function terms

Functions have the form ¡functor¿(¡term¿,¡term¿) Functor starts with a lower case alphabetic. Example

```
prerequisite_to(adv_ai)
grade_attained_in(prerequisite_to(adv_ai),pass)
```

The number of arguments is the arity of the function. When referring to a functor, it is written with its arity in the format ¡functor¿/¡arity¿. This is also true for atoms, whose arity is 0. Note that this is a recursive definition. The view of functions as trees Operators Some functors are used in infix notation, e.g. 5+4 Operators do not cause the associated function to be carried out. Variables Uppercase or _ for start of variables Example

```
Who
What
_special
_
```

Variables in Prolog are rather different to those in most other languages. Further discussion and use is deferred until later.

(c) Simple Semantics

Under a declarative reading, the order of rules, and of goals within rules, is irrelevant since logical disjunction and conjunction are commutative. Procedurally, however, it is often important to take into account Prolog's execution strategy, either for efficiency reasons, or due to the semantics of impure built-in predicates for which the order of evaluation matters. Also, as Prolog interpreters try to unify clauses in the order they're provided, failing to give a correct ordering can lead to infinite recursion.

In this subsection the operational semantics of CHR in Prolog are presented informally. They do not differ essentially from other CHR systems. When a constraint is called, it is considered an active constraint and the system will try to apply the rules to it. Rules are tried and executed sequentially in the order they are written.

[98]

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head, they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraints can be found, or the matching or the guard fails, then the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule there are multiple constraints in the head, the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint.

This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

A suspended constraint is eligible as a passive constraint for an active constraint. The other way it may interact again with the rules is when a variable appearing in the constraint becomes bound to either a non-variable or another variable involved in one or more constraints. In that case the constraint is triggered, i.e. it becomes an active constraint and all the rules are tried.

i. Rule Types There are three different kinds of rules, each with its specific semantics:

   A. simplification The simplification rule removes the constraints in its head and calls its body.

B. propagation The propagation rule calls its body exactly once for the constraints in its head.

C. simpagation The simpagation rule removes the constraints in its head after the and then calls its body. It is an optimization of simplification rules of the form: [constraints_1, constraints_2 <=> constraints_1, body] Namely, in the simpagation form: [constraints_1 \constraints_2 <=> body] The constraints_1 constraints are not called in the body.

ii. Rule Names Naming a rule is optional and has no semantic meaning. It only functions as documentation for the programmer.

iii. Pragmas The semantics of the pragmas are:

iv. passive(Identifier) The constraint in the head of a rule Identifier can only match a passive constraint in that rule.

(d) Universal Horn Clauses

(e) Unification

(f) Definite Clause Grammar

12.2 Chapter Recap

Chapter 13

Prototype 1

13.1 About this chapter

This chapter demonstrates a ”fairly generic” procedure of creating an open embedded domain specific language 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.

13.2 Components

There are four main components that we work with to develop a working implementation of embedded PROLOG using the concepts mentioned above.

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) The search strategy which is used to list and accomplish goals.

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

2. prolog-0.2.0.1 [109]

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

(a) Language, adopted from PROLOG but trimmed down.

(b) Database

(c) Unifier

(d) REPL

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

3. unification-fd [130]

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) Unifiable Class

(b) UTerm data type

(c) Variables, STVar, IntVar

(d) Binding Monad

(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 [130]

   (c) Variable Bindings

   (d) Monadic Unification

Each of the components are discussed in the following sections.

13.3 How Prolog works ?

To replicate PROLOG we look into how it works [125].

Most PROLOG distributions have three types of terms:

1. Constants.

2. Variables.

3. Complex terms.

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

The possibilities could be,

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

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

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

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

```
?- 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.

```
?- 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 were the language attempts to find a solution for the given query using the rules provided in the knowledge base. The other part is actually reaching a point where two terms need to be unified i.e searching. Together they form the query resolver in PROLOG.

For example, consider the append function

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

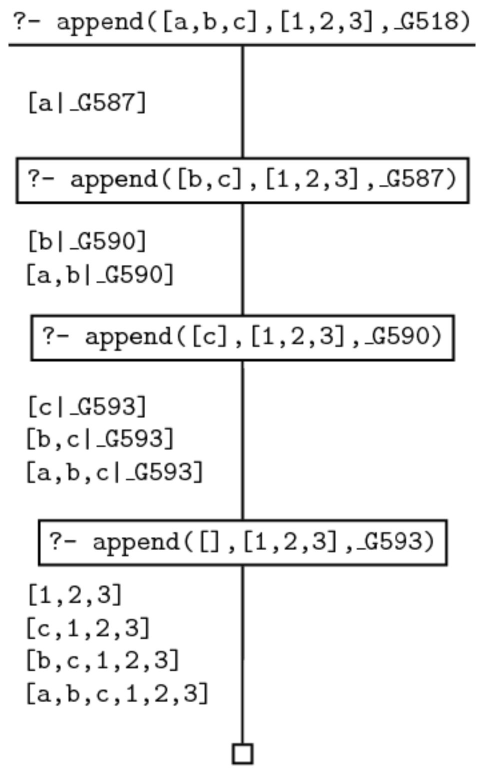

Figure 13.1: Trace for append [124]

In this prototype we explore the unification aspect only.

13.4 What we do in this Prototype

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

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

The language defined is recursive in nature.

We convert it into a non recursive data type.

Basically we do Unification monadically.

13.5 Creating a data type

To start we need to define a abstract syntax for the PROLOG like language. But there is a conflict between the type systems as we shall discuss.

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

The advantages of static typing [82]

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

For dynamic typing

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

Since HASKELL is statically type we would need to define a ”typed” language which would have a number of constructs representing different terms in PROLOG such as complex structures (for example predicates, clauses etc.), don’t cares, cuts, variables and so on.

Consider the language below which has been adopted from [109],

```
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)
data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
            | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
      deriving (Data, Typeable)
type Program = [Sentence]
type Body    = [Goal]
data Sentence = Query   Body
              | Command Body
              | C Clause
      deriving (Data, Typeable)
```

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

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

The above is a classic example of a recursive grammar to define the abstract syntax of a language. One of the issues with the above is that it is not possible to distinguish the structure of the data from the data type itself [115]. Moreover, the primitives of the language are not accessible as the language can have expressions of only one type i.e. ”Term”. The solution to would be to add a type constructor

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

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

One result of the approach is that the non-recursive type *FlatTerm* is modular and generic as the structure ”FlatTerm” is separate from it’s type which is ”a”. The above language can be of any type *a*. A more accurate way of saying it would be that *a* can be a *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 lambda calculus ’one level up,’ endowed with a primitive type, denoted * and called ’type,’ which is the kind of any (monomorphic) data type for example [147],

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

Simply speaking we can have something like

```haskell
FlatTerm Bool
```

and a generic fuinction like,

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

Although one problem remains, how does one represent infinitely nested / deep expressions of the above language, for example something of the form,

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

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

```haskell
(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 infinitely deep they were.

The approach to tackling this problem is to find the ”fixed-point”. After infinitely many iterations we should get to a fix point where further iterations make no difference. It means that applying one more ExprF would not change anything a fix point does not move under FlatTerm.

HASKELL provides it in two forms,

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

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

   The above function results in an infinite application stream,

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

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

2. And in type constructor form,

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

   which we apply to our abstract syntax.

The resulting language is of the form,

```haskell
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,

1. Manually modifying the structure of the language,

```
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 thoroughout the architecture because accomodations 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. Consider the following example,

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

   then something like,

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

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

13.6 Working with the language

Our language now opened up and ready for expansion, still needs to conform to the requirements of the [130] so that the generic

Creating 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 <$>
                                        sequenceA (Prelude.map f x)
          traverse _ (Var v)        =        pure (Var v)
          traverse _ Wildcard        =        pure (Wildcard)
          traverse _ (Cut i)        =        pure (Cut i)
instance Unifiable (FlatTerm) where
        zipMatch (Struct al ls) (Struct ar rs) =
                if (al == ar) && (length ls == length rs)
                        then Struct al <$>
                                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
instance Applicative (FlatTerm) where
        pure x = Struct "" [x]
        _ <*> Wildcard        =        Wildcard
        _ <*> (Cut i)        =        Cut i
        _ <*> (Var v)        =        (Var v)
        (Struct a fs) <*> (Struct b xs) = Struct (a ++ b) [f x | f <- fs, x <- xs]
```

After flattening do fixing,

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

13.7 Black box

13.8 Something about unification-fd and Monadic Unification

Library [130]

Tutorial 1 [131]

Tutorial 2 [132]

1. What library provides ?

   This module provides first-order structural unification over general structure types. It also provides the standard suite of functions accompanying unification (applying bindings, getting free variables, etc.).

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

2. Unifiable stuff

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

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

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

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 too 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, subsumption, etc.

The zipMatch function takes two arguments of type t a. The abstract t will be our concrete T type. The abstract a is polymorphic, which ensures that we can't mess around with more than one level of the term at once. If we abandon that guarantee, then you can think of it as if a is UTerm T v. Thus,t a means T (UTerm T v); and T (UTerm T v) is essentially the type UTerm T v with the added guarantee that the values aren't in fact variables. Thus, the arguments to zipMatch are non-variable terms.

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

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

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

3. UTerm stuff

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

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

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

4. STVar stuff

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

5. IntVar stuff

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

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

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

A ”mutable” unification variable implemented by an integer. This provides an entirely pure alternative to truly mutable alternatives (like STVar), which can make backtracking easier.

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

6. Binding Monad Stuff

A monad for handling STVar bindings.

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

7. U.unify stuff

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

8. U.unifyOccurs

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

9. Translation stuff

13.9 Chapter Recap

```
monadicUnification :: (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)))))
monadicUnification t1 t2 = do
--  let
--     t1f = termFlattener t1
--     t2f = termFlattener t2
  (x1,d1) <- lift . translateToUTerm $ t1
  (x2,d2) <- lift . translateToUTerm $ t2
  x3 <- U.unify x1 x2
  --get state from somehwere, state -> dict
  return $! (x3, d1 `Map.union` d2)


goUnify ::
  (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)]
goUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test --ERROR
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> f1 dict


f1 ::
  (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  => (forall s. Map VariableName (STVar s FlatTerm)
      -> (ST.STBinding s [(VariableName, Prolog)])
     )
f1 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
```

Figure 13.2: A sample Minted figure

Chapter 14

Prototype 2.1

14.1 About this chapter

This chapter attempts to infuse the generic methodology from 13 in a current PROLOG implementation [109] and make the unification ”monadic”.

14.2 How prolog-0.2.0.1 works

As described in the previous chapter about extending languages to incorporate functionality, this prototype applies the procedure to the eDSL in [109].

The original abstract syntax used by the library,

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

type Atom         = String

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

data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
```

```
                | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
        deriving (Data, Typeable)

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

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

The implementation consists of components that one would find in a Language Processing System 14.1,

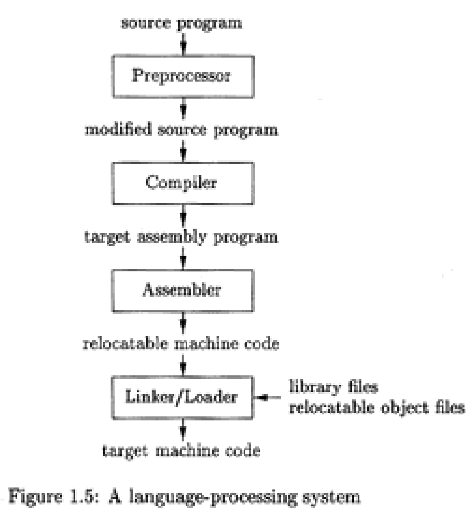

Figure 14.1: A language-processing system [1]

specifically speaking, parts of a compiler 14.2,

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

1. Syntax, defining the language.

2. Database, to create a storage for the expressions.

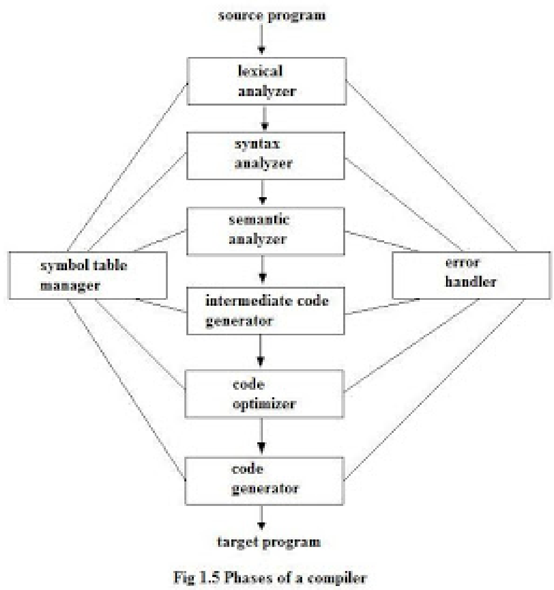

Figure 14.2: Phases of Compiler [1]

3. Parser.

4. Interpreter.

5. Unifier.

6. REPL.

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

14.3 What we do in this prototype?

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

We do complete PROLOG query resolution like stuff.

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

14.4 Current implementation (prolog-0.2.0.1)

The current unification uses basic pattern matching to unify the terms

```
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')
```

14.5 Modifications

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

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

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

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)
    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

instance Applicative (FTS) where
  pure x                  =   FS "" [x]
  _          <*>   FW     =   FW
  _        <*>   (FC i)      =   FC i
  _        <*>   (FV v)      = (FV v)
  (FS a fs) <*> (FS b xs)    = FS (a ++ b) [f x | f <- fs, x <- xs]
```

Aditionally helper functions for converting expressions between the two domains and translation to *UTerm*.

```
termFlattener :: Term -> Fix FTS
termFlattener (Var v)            =   DFF.Fix $ FV v
termFlattener (Wildcard)         =   DFF.Fix FW
termFlattener (Cut i)            =   DFF.Fix $ FC i
termFlattener (Struct a xs)      =   DFF.Fix $ FS a (Prelude.map termFlattener xs)

unFlatten :: Fix FTS -> Term
unFlatten (DFF.Fix (FV v))       =   Var v
unFlatten (DFF.Fix FW)           =   Wildcard
unFlatten (DFF.Fix (FC i))       =   Cut i
unFlatten (DFF.Fix (FS a xs))    =   Struct a (Prelude.map unFlatten 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) =>
```

```haskell
    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


uTermify
  :: Map VariableName (ST.STVar s (FTS))
  -> UTerm FTS (ST.STVar s (FTS))
  -> UTerm FTS (ST.STVar s (FTS))
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
  UT.UTerm (FS a xs)     -> UT.UTerm $ FS a $! fmap (uTermify varMap) xs
  UT.UTerm (FW)          -> UT.UTerm FW
  UT.UTerm (FC i)        -> UT.UTerm (FC i)

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.

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

translateFromUTerm ::
```

```
    Map VariableName (ST.STVar s (FTS)) ->
    UT.UTerm (FTS) (ST.STVar s (FTS)) -> Prolog
translateFromUTerm dict uTerm =
  P .  maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm 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


-- | 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 is likely a mistake that needs fixing

unifyTerms :: Fix FTS -> Fix FTS -> Maybe (Map VariableName (Prolog))
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) ->
    ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
        (ST.STBinding s)
        (Map VariableName (Prolog))
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 (ST.STVar s (FTS)) -> ST.STBinding s (Map VariableNam
makeDict sVarDict =
    flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do
```

```haskell
        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 (ST.STVar s (FTS)), Map VariableName (ST.STVar s (FTS))) ->
    ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
    (ST.STBinding s) (Map VariableName (Prolog))
makeDicts (svDict1, svDict2) = do
  let svDict3 = (svDict1 `Map.union` svDict2)
  let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
  applyBindingsAll ivs
  -- the interface below is dangerous because Map.union is left-biased.
  -- variables that are duplicated across terms may have different
  -- bindings because `translateToUTerm` is run separately on each
  -- term.
  lift . makeDict $ svDict3
```

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

```haskell
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
--  let
--     t1f = termFlattener t1
--     t2f = termFlattener t2
  (x1,d1) <- lift . translateToUTerm $ t1
  (x2,d2) <- lift . translateToUTerm $ t2
  x3 <- U.unify x1 x2
  --get state from somehwere, state -> dict
  return $! (x3, d1 `Map.union` d2)


goUnify ::
  (forall s. (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  =>
```

```haskell
      (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)]
goUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test --ERROR
  case answer of
    (Left _)           -> return []
    (Right (_, dict))  -> f1 dict


f1 ::
  (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (forall s. Map VariableName (STVar s FTS)
      -> (ST.STBinding s [(VariableName, Prolog)])
      )
f1 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
  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) (te
```

14.6 Results

It works,

14.7 Chapter Recap

Chapter 15

Prototype 3

15.1 What is this chapter about

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

term1 and term2 are matched and an assignment is the result

now this may be a part of a query resolution procedure

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

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

15.2 Unification

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

15.3 Resolution

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

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

[37]

[175]

15.4 Search strategies

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

15.5 Stack Engine

```
-- Stack based Prolog inference engine
-- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
-- and for Hugs 1.3 June 1996.
--
-- Suitable for use with Hugs 98.
--

module StackEngine( version, prove ) where

import Prolog
import st
import Interact

version = "stack based"

--- Calculation of solutions:

-- the stack based engine maintains a stack of triples (s,goal,alts)
-- corresponding to backtrack points, where s is the stitution at that
-- point, goal is the outstanding goal and 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 tp is a new goal that must be proved and u is
```

```haskell
-- a unifying stitution that must be combined with the stitution 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.

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 ]

-- The use of a stack enables backtracking to be described explicitly,
-- in the following ‘state-based’ definition of prove:

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


--- Special definitions for the cut predicate:

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

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

--- End of Engine.hs
```

15.6 Pure Engine

```
-- The Pure Prolog inference engine (using explicit prooftrees)
-- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
-- and for Hugs 1.3 June 1996.
--
-- Suitable for use with Hugs 98.
--

module PureEngine( version, prove ) where

import Prolog
import st
import Interact
import Data.List(nub)

version = "tree based"

--- Calculation of solutions:

-- Each node in a prooftree corresponds to:
-- either: a solution to the current goal, represented by Done s, where s
--         is the required stitution
-- or:     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 []

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 ]
{--
pt 1 nullst [] = Done (nullst)

pt n s (g:gs)

renClauses :- Rename variables in a clause, the parameters are the database, an
                         (head of list) resulting in a clause.
```

```
unify :- take the head of the list and and match with head of clause from renCla

app :- function for applying (st) to (Terms)
the new list is formed by replacing the cluase head with its body and applying t

so the new parameters for pt are

(n+1) (the old stitution + the new one from unify) (the list formed after applyi


Working of a small example

The database,
(foldl addClause emptyDb [((:-) (Struct "hello" []) []), ((:-) (Struct "hello" [
hello.
hello(world).
hello:-world.
hello(X_1).

The other parameters are 1 nullst(as mentioned in the prove function).

For the list of goals, [(Struct "hello" []), (Struct "hello" [(Struct "world" []

1. [Struct "hello" []] :: [Term]

* Rule 1 does not apply

* Rule 2 does apply,

(tm:- tp) <- renClauses db 1 (Struct "hello" [])

tm ==> "hello , hello(world) , hello , hello(X_1) , "
tp ==> "[] , [] , [world] , [] , "









--}

```

```haskell


-- 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

--- End of PureEngine.hs
```

15.7 Andorra Engine

```haskell
{-
By Donald A. Smith, December 22, 1994, based on Mark Jones' PureEngine.

This inference engine implements a variation of the Andorra Principle for
logic programming. (See references at the end of this file.) The basic
idea is that instead of always selecting the first goal in the current
list of goals, select a relatively deterministic goal.

For each goal g in the list of goals, calculate the resolvents that would
result from selecting g.  Then choose a g which results in the lowest
number of resolvents.  If some g results in 0 resolvents then fail.
(This would occur for a goal like:  ?- 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
(i.e., only a single clause matches) 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=[].
-}

module AndorraEngine( version, prove ) where

import Prolog
import st
import Interact
```

```haskell

version = "Andorra Principle Interpreter (select deterministic goals first)"

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])])]
--  For each pair in the list that we return, the first element of the
--  pair is the list of unresolved goals; the second element is the list
--  of resolvents of the selected goal, where a resolvent is a pair
--  consisting of a stitution and a list of new goals.
resolve_selecting_each_goal goals db n = [(gs, gResolvents) |
      (g,gs) <- delete goals, let gResolvents = resolve db g n]

-- The unselected goals from above are not passed in.
resolve :: Database -> Term -> Int -> [(st,[Term])]
resolve db g n = [(u,tp) | (tm:-tp)<-renClauses db n g, u<-unify g tm]
-- u is not yet applied to tp, since it is possible that g won't be selected.
-- Note that unify could be nondeterministic.

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)
-- It seems to me that there is an opportunity for a clever compiler to
-- optimize this code a lot. In particular, there should be no need to
-- determine the total length of a goal list if it is known that
-- there is a shorter goal list in allResolvents ... ?

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

{- An optimized, incremental version of the above interpreter would use
  a data representation in which for each goal in "goals" we carry around
  the list of resolvents.  After each resolution step we update the lists.
-}

{- References

   Seif Haridi & Per Brand, "Andorra Prolog, an integration of Prolog
   and committed choice languages" in Proceedings of FGCS 1988, ICOT,
   Tokyo, 1988.

   Vitor Santos Costa, David H. D. Warren, and Rong Yang, "Two papers on
   the Andorra-I engine and preprocessor", in Proceedings of the 8th
   ICLP. MIT Press, 1991.

   Steve Gregory and Rong Yang, "Parallel Constraint Solving in
   Andorra-I", in Proceedings of FGCS'92. ICOT, Tokyo, 1992.

   Sverker Janson and Seif Haridi, "Programming Paradigms of the Andorra
   Kernel Language", in Proceedings of ILPS'91. MIT Press, 1991.

   Torkel Franzen, Seif Haridi, and Sverker Janson, "An Overview of the
   Andorra Kernel Language", In LNAI (LNCS) 596, Springer-Verlag, 1992.
-}
```

15.8 Current Unification

```
{-# LANGUAGE  DeriveDataTypeable,
              ViewPatterns,
              ScopedTypeVariables,
              DefaultSignatures,
              TypeOperators,
              TypeFamilies,
              DataKinds,
              DataKinds,
              PolyKinds,
```

```
                 OverlappingInstances,
                 TypeOperators,
                 LiberalTypeSynonyms,
                 TemplateHaskell,
                 AllowAmbiguousTypes,
                 ConstraintKinds,
                 Rank2Types,
                 MultiParamTypeClasses,
                 FunctionalDependencies,
                 FlexibleContexts,
                 FlexibleInstances,
                 UndecidableInstances
                 #-}

-- stitutions and Unification of Prolog Terms
-- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
-- and for Hugs 1.3 June 1996.
--
-- Suitable for use with Hugs 98.
--

module st where

import Prolog
import CustomSyntax
import Data.Map as Map
import Data.Maybe
import Data.Either

--Unification
import Control.Unification.IntVar
import Control.Unification.STVar as ST

import Control.Unification.Ranked.IntVar
import Control.Unification.Ranked.STVar

import Control.Unification.Types as UT

import Control.Monad.State.UnificationExtras
import Control.Unification as U

-- Monads
import Control.Monad.Error
import Control.Monad.Trans.Except

```

```haskell
import Data.Functor.Fixedpoint as DFF

--State
import Control.Monad.State.Lazy
import Control.Monad.ST
import Control.Monad.Trans.State as Trans

infixr 3 @@
infix  4 ->-

--- stitutions:

type st = Id -> Term

newtype stP = stP { unstP :: st }

-- instance Show stP where
--  show (i) = show £ Var i
-- stitutions are represented by functions mapping identifiers to terms.
--
-- app s     extends the stitution s to a function mapping terms to terms
{--
Looks like an apply function that applies a stitution function tho the variables
--}


-- nullst is the empty stitution which maps every identifier to the same identif



-- i ->- t   is the stitution which maps the identifier i to the term t, but oth


-- s1@@ s2  is the composition of stitutions s1 and s2
--           N.B.  app is a monoid homomorphism from (st,nullst,(@@))
--           to (Term -> Term, id, (.)) in the sense that:
--                  app (s1 @@ s2) = app s1 . app s2
--                 s @@ nullst = s = nullst @@ s

app                      :: st -> Term -> Term
app s (Var i)             = s i
app s (Struct a ts)       = Struct a (Prelude.map (app s) ts)
{--
app (stFunction) (Struct "hello" [Var (0, "Var")])
hello(Var_2) :: Term
```

```

--}


nullst                :: st
nullst i               = Var i
{--
nullst (0, "Var")
Var :: Term
--}


--
(->-)                     :: Id -> Term -> st
(i ->- t) j | j==i         = t
            | otherwise    = Var j
{--
:t (->-) (1,"X") (Struct "hello" [])
(1,"X") ->- Struct "hello" [] :: (Int,[Char]) -> Term
--}


-- Function composition for applying two stitution functions.
(@@)                      :: st -> st -> st
s1 @@ s2                   = app s1 . s2
```

15.9 Syntax Modification

```
{-# LANGUAGE  DeriveDataTypeable,
              ViewPatterns,
              ScopedTypeVariables,
              FlexibleInstances,
              DefaultSignatures,
              TypeOperators,
              FlexibleContexts,
              TypeFamilies,
              DataKinds,
              OverlappingInstances,
              DataKinds,
              PolyKinds,
              TypeOperators,
              LiberalTypeSynonyms,
```

```haskell
             TemplateHaskell,
             RankNTypes,
             AllowAmbiguousTypes,
             MultiParamTypeClasses,
             FunctionalDependencies,
             ConstraintKinds,
             ExistentialQuantification
             #-}

module CustomSyntax where

import Data.Generics (Data(..), Typeable(..))
import Data.List (intercalate)
import Data.Char (isLetter)

import Control.Monad.State.UnificationExtras
import Control.Unification as U


import Data.Functor.Fixedpoint as DFF


import Control.Unification.IntVar
import Control.Unification.STVar as ST

import Control.Unification.Ranked.IntVar
import Control.Unification.Ranked.STVar

import Control.Unification.Types as UT



import Data.Traversable as T
import Data.Functor
import Data.Foldable
import Control.Applicative


import Data.List.Extras.Pair
import Data.Map as Map
import Data.Set as S


import Control.Monad.Error
import Control.Monad.Trans.Except
```

```haskell
import Prolog

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)
                                     else Nothing
        zipMatch (FV v1) (FV v2) = if (v1 == v2) then Just (FV v1)
                else Nothing
        zipMatch _ _ = Nothing

instance Applicative FTS where
        pure x = FS "" [x]
        (FS a fs) <*> (FS b xs)   = FS (a ++ b) [f x | f <- fs, x <- xs]
        --other cases
{--
instance Monad FTS where
        func =
instance Variable FTS where
        func =

instance BindingMonad FTS where
        func =
--}

data VariableName = VariableName Int String
```

```
idToVariableName :: Id -> VariableName
idToVariableName (i, s) = VariableName i s

variablenameToId :: VariableName -> Id
variablenameToId (VariableName i s) = (i,s)

termFlattener :: Term -> Fix FTS
termFlattener (Var v)          =   DFF.Fix $ FV v
termFlattener (Struct a xs)    =   DFF.Fix $ FS a (Prelude.map termFlattener xs)

unFlatten :: Fix FTS -> Term
unFlatten (DFF.Fix (FV v))      =   Var v
unFlatten (DFF.Fix (FS a xs))   =   Struct a (Prelude.map unFlatten 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]
--  _        ->  []

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


uTermify
  :: Map Id (ST.STVar s (FTS))
  -> UTerm FTS (ST.STVar s (FTS))
  -> UTerm FTS (ST.STVar s (FTS))
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
  UT.UTerm (FS a xs)    -> UT.UTerm $ FS a $! fmap (uTermify varMap) xs


translateToUTerm ::
    Fix FTS -> ST.STBinding s
            (UT.UTerm (FTS) (ST.STVar s (FTS)),
             Map Id (ST.STVar s (FTS)))
translateToUTerm e1Term = do
  let vs = variableNameSet $ variableIdExtractor 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 routine of @ translateFromUTerm @.  The resulting
-- term has no (UVar x) terms.

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

translateFromUTerm ::
    Map Id (ST.STVar s (FTS)) ->
    UT.UTerm (FTS) (ST.STVar s (FTS)) -> Prolog
translateFromUTerm dict uTerm =
  P .  maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm 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


-- | 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 is likely a mistake that needs fixing

unifyTerms :: Fix FTS -> Fix FTS -> Maybe (Map Id (Prolog))
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 ->
    ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
        (ST.STBinding s)
        (Map Id (Prolog))
unifyTermsX t1 t2 = do
    (x1,d1) <- lift . translateToUTerm $ t1
    (x2,d2) <- lift . translateToUTerm $ t2
    _ <- 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 Id (ST.STVar s (FTS)) -> ST.STBinding s (Map Id (Prolog))
makeDict sVarDict =
    flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do
        Just xx <- UT.lookupVar $ iKey
        return $! (translateFromUTerm sVarDict) xx
```

```haskell


-- | recover the bindings for the variables of the two terms
-- unified from the monad.

makeDicts ::
    (Map Id (ST.STVar s (FTS)), Map Id (ST.STVar s (FTS))) ->
    ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
    (ST.STBinding s) (Map Id (Prolog))
makeDicts (svDict1, svDict2) = do
  let svDict3 = (svDict1 `Map.union` svDict2)
  let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
  applyBindingsAll ivs
  -- the interface below is dangerous because Map.union is left-biased.
  -- variables that are duplicated across terms may have different
  -- bindings because `translateToUTerm` is run separately on each
  -- term.
  lift . makeDict $ svDict3

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 Id (ST.STVar s (FTS)))
test1 = do
    let
        t1a = (Fix $ FV $ (0, "x"))
        t2a = (Fix $ FV $ (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 Id (ST.STVar s (FTS)))
test2 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
        t2a = (Fix $ FV $ (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 Id (ST.STVar s (FTS)))
test3 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
        t2a = (Fix $ FV $ (0, "x"))
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unify x1 x2
    return (x3, d1 `Map.union` d2)
{--
goTest test3
"ok:    STVar -9223372036854775807
[(VariableName 0 \"x\",STVar -9223372036854775808)]"
--}

test4 ::
  ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
         (ST.STBinding s)
          (UT.UTerm (FTS) (ST.STVar s (FTS)),
           Map Id (ST.STVar s (FTS)))
test4 = do
    let
        t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
        t2a = (Fix $ FV $ (0, "x"))
    (x1,d1) <- lift . translateToUTerm $ t1a --error
    (x2,d2) <- lift . translateToUTerm $ t2a
    x3 <- U.unifyOccurs x1 x2
    return (x3, d1 `Map.union` d2)
{--
goTest test4
"ok:    STVar -9223372036854775807
[(VariableName 0 \"x\",STVar -9223372036854775808)]"
--}

test5 ::
```

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

goTest :: (Show b) => (forall 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))))) -> String
goTest test = ST.runSTBinding $ do
  answer <- runErrorT $ test
  return $! case answer of
    (Left x)  -> "error: " ++ show x
    (Right y) -> "ok:    " ++ show y


----------------------------------------------------------------
----------------------------------------------------------------
---------------GLUE-CODE----------------------------------------
{--
monadicUnify :: 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)))
monadicUnify 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)

--}

-- type st = Id -> Term
```

```

-- Convert result from monadicUnify to [st]
{--
goMonadicTest :: (Show b) => (forall 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))))) -> [st]
goMonadicTest test = ST.runSTBinding £ do
  answer <- runErrorT £ test
  return £! case answer of
    (Left x)  -> [nullst]
    (Right y) -> convertTost y
--}

--(Id, STVar s FTS)
--convertTost :: Map Id (ST.STVar s FTS) -> [(Id, ST.STVar s FTS)]
{--
convertTost m = convertTost1 Map.toAscList m

convertTost1 (id, ST.STVar _ fts):xs = (id, (unFlatten fts)) : convertTost1 xs
--}
```

15.10 Monadic Unification

```
monadicUnification :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s)) => (forall
            (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
  --get state from somehwere, state -> dict
  return $! (x3, d1 `Map.union` d2)


goUnify ::
  (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)]
goUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test --ERROR
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> f1 dict


f1 ::
  (BindingMonad FTS (STVar s FTS) (ST.STBinding s))
  => (forall s. Map Id (STVar s FTS)
      -> (ST.STBinding s [(Id, Prolog)])
      )
f1 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


--unify :: Term -> Term -> [st]
unify t1 t2 = stConvertor (goUnify (monadicUnification t1 t2))


varX :: Term
varX = Var (0,"x")

varY :: Term
varY = Var (1,"y")


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

15.11 Chapter Recap

Chapter 16

Prototype 4

16.1 What is this chapter about

Our aim to embedd IO into the DSL

So something like a ”data” declaration for IO operations

16.2 I/O is pure

[128]

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

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

How can this possibly be pure?

In this post I want to explain exactly why I/O in Haskell is pure. Ill do it by building up data structures that represent blocks of code. These data structures can later be executed, and they cause effects to occur - but until that point well always work with pure functions, never with effects.

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

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

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.

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

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)
```

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

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

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

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.

Thats a bit of a mouthful, but writing this combinator isnt too hard. When we reach the final Return, we apply the function f to get a new action. For the other constructors, we keep the form of the action the same, and just thread seqio through the seqio constructor.

Using seqio we can define the action that gets input from stdin and immediately prints it to the screen:

or even more complicated actions:

```
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 (admittedly with pretty unpleasant syntax), its really a value of type IOAction (). In Haskell, code can be data and data can be code.

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

```
*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 ()
          )
        )
      )
    )
  )
)
```

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

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

The main benefit of doing this is that we can now sequence actions using Haskells do notation, which desugars into calls to (¿¿=), and hence to seqio. Our earlier hello example can now be written as:

```haskell
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!")
```

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

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. Its easy to write such a function just by turning it into the approprate calls to putStrLn and getLine.

```haskell
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)
```

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

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

Is there any practical use to this?

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

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

```haskell
-- http://chris-taylor.github.io/blog/2013/02/09/io-is-not-a-side-effect/

data IOAction a =
-- A container for a value of type a.
                  Return a
-- A container holding a String to be printed to stdout, followed by another IOA
                | Put String (IOAction a)
-- A container holding a function from String -> IOAction a, which can be applie
                | Get (String -> IOAction a)
{--

Return 1

Put "hello" (Return ())
Put "hello" (
  Return ()
)

Put "hello" (Return 1)
Put "hello" (
  Return 1
)

Put "hello" (get)
Put "hello" (
  Get (£0 ->
    Return "£0"
  )
)

Get put
```

```
Get (£0 ->
  Put "£0" (
    Return ()
  )
)

--}

-- Read and return
get :: IOAction String
get   = Get Return
{--

Get (£0 ->
  Return "£0"
)

--}

-- Print and return.
put :: String -> IOAction ()
put s = Put s (Return ())
{--

put "hello"
Put "hello" (
  Return ()
)

--}

-- (>>=) Action sequencer and combiner :- read -> write -> read -> write -> ....
seqio :: IOAction a -> (a -> IOAction b) -> IOAction b
--        (First action   (Take and perform
--        which generates  next action)
--        value a)
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)

--Take input and print.
echo :: IOAction ()
echo = get `seqio` put
{--

```

```
Get (£0 ->
  Put "£0" (
    Return ()
  )
)

--}

hello :: IOAction ()
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")
{--

Put "What is your name?" (
  Get (£0 ->
    Put "What is your age?" (
      Get (£1 ->
        Put "Hello £0!" (
          Put "You are £1 years old" (
            Return ()
          )
        )
      )
    )
  )
)

run hello
What is your name?
Mehul
What is your age?
25
Hello Mehul!
You are 25 years old

--}

-- hello in "do" block since IOAction is a Monad
hello2 :: IOAction ()
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!")
{--

Put "What is your name?" (
  Get ($0 ->
    Put "What is your age?" (
      Get ($1 ->
        Put "Hello, $0!" (
          Put "You are $1 years old!" (
            Return ()
          )
        )
      )
    )
  )
)

run hello2
What is your name?
Mehul
What is your age?
25
Hello, Mehul!
You are 25 years old!

--}

-- where the effects happen.
-- "Real" IO functions like return, putStrLn, getLine.
run :: IOAction a -> IO a
run (Return a) = return a
run (Put s io) = putStrLn s >> run io
run (Get f)    = getLine >>= run . f
{--

run (Return 1)
1

run (Put "hello" get)
hello
1
"1"
```

```

run (Get put)
1
1

--}


-- Glue code that makes everything play nice --

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

instance Show a => Show (IOAction a) where
  show io = go 0 0 io
    where
      go m n (Return a) = ind m "Return " ++ show a
      go m n (Put s io) = ind m "Put " ++ show s ++ " (\n" ++ go (m+2) n io ++ "\n
      go m n (Get g)    = let i = "$" ++ show n
                          in ind m "Get (" ++ i ++ " -> \n" ++ go (m+2) (n+1) (g i

      ind m s = replicate m ' ' ++ s

-- IOAction is also a Functor --

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

mapio (+1) (Return 1)
Return 2

mapio (id) (Put "hello" get)
Put "hello" (
  Get (£0 ->
    Return "£0"
  )
)

mapio (id) (Get put)
Get (£0 ->
  Put "£0" (
```

```
    Return ()
  )
)

--}

instance Functor IOAction where
    fmap = mapio
```

16.3 Dr. Casperson Pure IO

```
-- Prolog IO

{--
FREE MONADS
In general, a structure is called free when it is left-adjoint to a forgetful fu
In this specific instance, the Term data type is a higher-order functor that map
a functor f to the monad Term f ; this is illustrated by the above two instance
definitions. This Term functor is left-adjoint to the forgetful functor from mon
to their underlying functors.
--}

data Term f a = Pure a
                          | Impure (f (Term f a))

main                                                          = undefined

instance Functor f => Functor (Term f) where
        fmap f (Pure x )                          = Pure (f x )
        fmap f (Impure t)                          = Impure (fmap (fmap f ) t)

instance Functor f => Monad (Term f) where
        return x                                        = Pure x
        (Pure x )         >>=         f                 = f x
        (Impure t)         >>=         f                 = Impure (fmap (>>= f ) t
```

16.4 Mehul Pure IO

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

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

```haskell
import Data.Traversable
import Control.Monad
import Data.Functor
import Control.Applicative
import System.IO

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



data OneBinding = Pair VariableName VariableName


--data MiniLang a = MyData a | Empty | Input

--runInIO :: PrologResult -> IO [OneBinding]


data PrologIO a = Input (IO a) | Output (a -> IO ()) | PrologData a | Empty
--                                deriving (Show, Eq, Ord)
{--
instance Functor (PrologIO) where
        fmap f Empty                                            = Empty
        fmap f (Input (IO a))                            = Input (IO (f a))
--        fmap f (Output (a -> IO ()))          = Output (a -> IO ())
--        fmap f (PrologData a)                            = PrologData (f a)
--}

instance Monad PrologIO where
                 return a = PrologData a
--                  (Input i) >>= (Output o) = i >>= (\a -> (o a))
```

```haskell

instance (Show a) => Show (PrologIO a) where
        show (Empty)                 = show "No result"
        show (PrologData a) = show a
--        show (Input f)                 = show (f ++ "")
--        show (Output )


-- (>>=) Action sequencer and combiner :- read -> write -> read -> write -> ....
seqio :: PrologIO a -> (a -> PrologIO b) -> PrologIO b
--       (First action    (Take and perform
--       which generates  next action)
--       value a)
seqio (PrologData a)         f          = f a
--seqio (Output o)                 f          = \a -> o a
--seqio (Input i)                   f          = \s -> (seqio (i s) f) --



{--
instance Applicative PrologIO where
        func =

instance Traversable PrologIO where
        traverse f Empty                                        = Empty
        traverse f (Input (IO a))                         = Input (IO (f a))
        traverse f (Output (a -> IO ()))        = Output ((a) -> IO ())
        traverse f (PrologData a)                         = PrologData (f a)
--}


concate :: PrologIO t -> PrologIO t -> IO ()
concate (Input f1) (Output f2) = do
        x <- f1
        f2 x
{--
concate (Input getLine) (Output putStrLn)
Loading package list-extras-0.4.1.4 ... linking ... done.
Loading package syb-0.5.1 ... linking ... done.
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package containers-0.5.5.1 ... linking ... done.
Loading package transformers-0.4.3.0 ... linking ... done.
Loading package mtl-2.2.1 ... linking ... done.
Loading package logict-0.6.0.2 ... linking ... done.
```

```
Loading package unification-fd-0.10.0.1 ... linking ... done.
1
1
--}
```

16.5 Chapter Recap

Chapter 17

Work Completed

17.1 What is this chapter about

17.2 What we are doing

A partial implementation of the logic programming language PROLOG is provided by the library `prolog-0.2.0.1`. One of the objectives is to implement monadic unification using the library [130].

17.3 Unifiable Data Structures

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

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

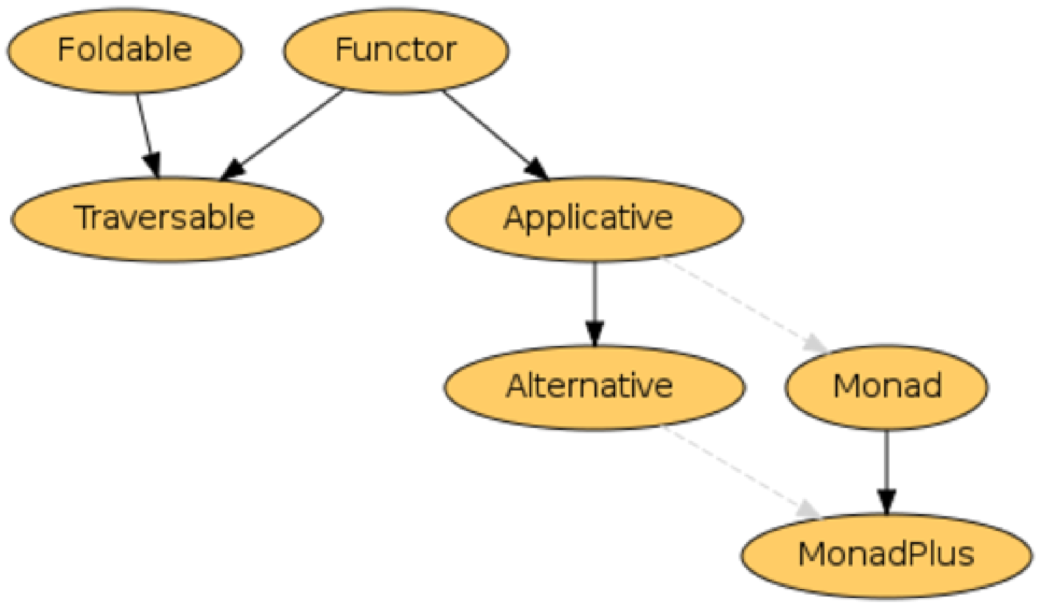

Figure 17.1: Functor Hierarchy [148]

17.4 Why `Fix` is necessary?

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

In our case consider the following example,

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

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

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

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

```
Fix FlatTerm
```

resulting in the PROLOG Data Type

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

17.5 Dr. Casperson's Explanation

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

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

A sample Tree would be,

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

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

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

A sample FlatTerm would be similar to Tree.

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

17.6 The other fix

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

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

The above function results in an infinite application stream,

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

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

17.7 The Fix we use

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

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

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

How to generate expressions Using grammars, for example

```
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr
```

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

```
ExprF a = Const Int
        | Add a a
        | Mul a a
```

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

```haskell
(ExprF (ExprF (ExprF a)))
```

How to generate really deep expressions Keep on applying

```haskell
ExprF
```

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

How do that in HASKELL In HASKELL, we can express the fix point of a type constructor f as a type:

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

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

```haskell
type Expr = Fix ExprF
```

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

```haskell
Fix ExprF
```

So rather than writing separate functions for,

```haskell
(ExprF a)

(ExprF (ExprF a))

(ExprF (ExprF (ExprF a)))

(ExprF (ExprF (ExprF ....)))
```

We write a function from,

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

17.8 Opening up or Extending language Explanation using Box Analogy

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

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

```
type Atom                  = String

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

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

   The above language represent a stripped down version of PROLOG from [109]. The pool of the expressions that can be generated from *Term* are restricted to the constructors,

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

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

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

```
type Atom = String

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

data FlatTerm a =
                Struct Atom [a]
        | Var VariableName
```

```
         | Wildcard
         | Cut Int deriving (Show, Eq, Ord)
```

The above language can be of any type *a*. A more accurate way of saying it would be that *a* can be a *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 lambda calculus 'one level up,' endowed with a primitive type, denoted * and called 'type,' which is the kind of any (monomorphic) data type for example [147],

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

Simply speaking the *a* can be changed.

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

   (a) Manually modifying the structure of the language,

```
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 thorughout the architecture because accomodations need to be made for the new functionality.

(b) 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. Consider the following example,

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

then something like,

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

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

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

$x \cdot y + x \cdot z$

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

$$\begin{aligned} &\forall x \forall y \forall z \qquad x \cdot y + x \cdot z \\ &= x \cdot (y + z) \end{aligned}$$

5. Multiple modifications

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

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

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

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

17.9 Chapter Recap

Chapter 18

Results

18.1 What is this chapter about

18.2 Types

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

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

Our preliminary implementation is as follows,

```
type Atom = String

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

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

{--
Output :-

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

--}
```

which in PROLOG would look like, 1

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

18.3 Lazy Evaluation 2

18.4 Opening up the Language 3

Flattening 4

Fixing 5

MetaSyntactic Variables 6

18.5 Quasi Quotation 1

18.6 Template Haskell 2

18.7 Higher Order Functions 3

```
% Mehul Solanki.

% Higher Order Functions.

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

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

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

18.8 I/O 4

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

18.9 Mutability

18.10 Unification

18.11 Monads

18.12 Chapter Recap

Chapter 19

Future Scope

19.1 What is this chapter about

---

1. Quasi quoter to get something like,

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

   where X is a PROLOG variable and y is a HASKELL variable injected into the expression

2. We already have variable search strategies, what if the query resolver could be instructed to use a particular search strategy to get the result.

```
queryResolver searcStrategy query knowledgeBase
```

3. Add database operations

4. Multi type variable Language

5. Pure + IO Combined Language

```
data ResultWithIO typevariableforpureexpressions typevariableforioexpressions
        = PureConstructor_1 ....
        | PureConstructor_2 ....
```

```
        | IOContrcutor_1 .....
        | IOContructor_2 ...
        | ContructorWithBoth_1 .....
        | ContructorWithBoth_2 .....
        deriving(........)
```

6.

19.2 Chapter Recap

Chapter 20

Conclusion / Expected Outcomes

20.1 What is this chapter about

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

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

20.2 Chapter Recap

Chapter 21

Editing to do

This Chapter needs to be removed from the final work.

Meeting on 5th Novemeber 2015

1. Write about this chapter and chapter conclusion for all chapters
2. Till haskell why haskell chapter 11 wait for feedback
3. In the remaining chapters write according to flow == move around stuff or add new content.

2015-10-29

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

Either

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

Mehul

5. Chapter 13.5 needs fleshing out.

6. Rewrite (Section) Chapter 3.2 . You are now in a position to state what your contributions are. In some sense everything else flows around this.

7. Fix the reference at the bottom of page 2:

   `citewikipro- log,somogyi1995logic,website:prolog1000db`. **SOLVED**

8. Write enough of Chapters 13–16 that we can decide what material is needed in Chapters 9, 10, and **??**.

9. [mainly done] If you don't like the shape of the paragraphs that you get without `paragraph`, use something like

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

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

10. Rewrite (Section) Chapter 3 in formal English.

11. "re-curses" means to swear again (*p* 9). **Changed to recurs**

12. I am not sure that I agree with the use of "reflective" on *p* 8 (*l* 25). Reflection often means run-time introspection (for instance the Java `.getClass()` method). In computer science, reflection is the ability of a computer program to examine (see type introspection) and modify its own structure and behavior (specifically the values, meta-data, properties and functions) at runtime.

13. Supply your credentials in the front material (what degrees do you have?). (Search for `%% Supply your credentials` in `proposal1.tex`.)

14. The abstract is too long. UNBC guidelines limit Masters' theses abstracts to 150 words.

15. Citation `logic-classes` is not defined (in `./prologinhaskell.tex`).

David

16. Clean up the non-exclusive license page in unbcthesis.cls
17. Incorporate unbcthesis.cls into Mehul's work.
18. Review Chapter 2
19. Review Chapter 3
20. Review Chapter 4
21. Review Chapter 5
22. Review Chapter 6
23. Review Chapter 7
24. Review Chapter 8
25. Review Chapter 18

21.1 Editing suggestions from David

Thoughts on Chapter 14

- Do not use naked `\refs`: "*the generic methodology from 13*" should be "*the generic methodology from* **Chapter** *13*".
- You should say more about [109], either here or in an earlier section and reference that discussion here. For instance, it isn't clear that `prolog-0.2.0.1` comes from [109].
- See my comments below. I suspect that longer listings should be separate figures.

```haskell
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
          | PString   !String
          | PInteger  !Integer
          | PFloat    !Double
          | Flat [FlatItem]
          | Cut Int
      deriving (Eq, Data, Typeable)
data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
            | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
      deriving (Data, Typeable)
type Program = [Sentence]
type Body    = [Goal]
data Sentence = Query   Body
              | Command Body
              | C Clause
      deriving (Data, Typeable)
```

Figure 21.1: A sample Minted figure

- Line 7 on p 55 is not a complete sentence.
- I suspect that § 14.2 should start with a sentence like

> The `prolog-0.2.0.1` ([109]) was written by Indira Ghandi and consists of 718 HASKELL files. It implements data base assertions and cuts but lacks any IO facilities…

and then go on to discuss the syntax.

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

1. the text should be loose—as you have it, or floated to a figure, as shown in Figure 21.1.

2. I am also not sure whether I like the inlined code, or whether I would prefer to have it `\inputminted` from a HASKELL file. I suppose that this depends on your work-flow. Thoughts?

I am not sure what conventions you are following with respect to code in text. At some point you have `FlatTerm` in italics (á la *FlatTerm*); at other points you have it typeset in straight double quotes (”FlatTerm”) and I don’t know what the different typesetting implies.

Just above Section 13.5 you mention a generic function `map`, which for STANDARD ML and HASKELL readers likely means the function with signature `(a -> b) -> ([a] -> [b])`. Why not `fmap`?

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

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

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

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

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

Thoughts on 1.4 Rename to “Thesis Organization”.

Thoughts on Chapter 2 Here are some potential keywords from Chapter 2: • Hindley-Milner type systems • Horn clauses • λ-calculi • HASKELL • SCALA • declarative programming languages • foreign function interfaces • functional programming • implementing Prolog in other languages • language embedding • language families • language paradigms • logic programming • meta-programming • monads • paradigm integration • quasi-quotation • the typed λ-calculus • the untyped λ-calculus .

What is the overall message?

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 and Forêt Des Flambertins, *Warrens abstract machine a tutorial reconstruction*, (1999).

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

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

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

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

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

[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] c2 wiki, *Metasyntactic variables*, September 2011.

[17] Catb, *Metasynatactic variables*.

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

[19] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] 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.

[36] Steve Dunne and Bill Stoddart (eds.), *Unifying theories of programming, first international symposium, utp 2006, walworth castle, county durham, uk, february 5-7, 2006, revised selected papers*, Lecture Notes in Computer Science, vol. 4010, Springer, 2006.

[37] Joshua Eckroth, *Prolog resolution*, April 2014.

[38] ———, *Prolog unification*, April 2014.

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

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

[41] ———, *stream-monad*, September 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

[55] 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.

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

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

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

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

[60] What is Tech Target, *Metasynatactic variables*, September 2005.

[61] JaimieMurdock, *Haskell kanren*, March 2012.

[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] Paul Johnson, *Why haskell is good for embedded domain specific languages*, January 2008.

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

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

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

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

[69] Anton Kholomiov, *data-fix*, February 2013.

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

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

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

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

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

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

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

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

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

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

[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: The end of the cold war between programming languages*, Citeseer, 2004.

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

[84] 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.

[85] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[98] SWI Prolog, *swi prolog syntax and semantics*.

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

[100] 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.

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

[102] 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.

[103] 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.

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

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

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

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

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

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

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

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

[112] Silvija Seres and Shin-Cheng Mu, *Optimisation problems in logic programming: an algebraic approach*, (2000).

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

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

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

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

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

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

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

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

[121] ———, *Embedding prolog in haskell*, Proceedings of Haskell, vol. 99, Citeseer, 1999, pp. 1999–28.

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

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

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

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

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

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

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

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

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

[131] ———, *Unification tutorial 1*, October 2015.

[132] ———, *Unification tutorial 2*, October 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[150] ______, *Embedded domain specific languages*, September 2014.

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

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

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

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

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

[156] ______, *Common lisp object system*, December 2013.

[157] ______, *Curry programming language*, December 2013.

[158] ______, *Functional logic programming*, May 2013.

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

[160] ______, *Common language infrastructure*, February 2014.

[161] ______, *Common language runtime*, March 2014.

[162] ______, *Constraint handling rules*, March 2014.

[163] ______, *Constraint programming*, March 2014.

[164] ______, *Damas-hindley-milner type system*, February 2014.

[165] ______, *Foreign function interface*, January 2014.

[166] ______, *Lambda calculus*, March 2014.

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

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

[169] ———, *Ocaml*, March 2014.

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

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

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

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

[174] ———, *Metasyntactic variable*, October 2015.

[175] ———, *Resolution*, October 2015.

[176] ———, *Usemention distinction*, October 2015.

[177] 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.

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

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

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