Embedding Programming Languages: Prolog in Haskell

A Master's Thesis by

Mehul Chandrakant Solanki

230108015 solanki@unbc.ca

29 10 2015

Submitted to the graduate faculty of the

MCPS

in partial fulfillment of the requirements

for the Master's Thesis and

subsequent MSc. in Computer Science

Committee Members:

Dr. David Casperson, Committee Chair

Dr. Alex Aravind

Dr. Mark Shegelski

Outline

List of Figures

List of Tables

Abstract

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

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

1 Introduction

1.1 Beginnings

Computers have become a part of everyone's life. From the ones in our pockets to the ones on desks or in our school bags, working or in fact living without them is difficult if not impossible. All the more reason to know how to use one. Simply speaking just using a computer these days is not enough. To be able to utilise their true potential, one must go deeper and communicate with them. This is where the art of programming steps in.

Programming has become an integral part of working and interacting with computers and day by day more and more complex problems are being tackled using the power of programming technologies. It is possibly the only way to talk to computers and hence the need for a robust and multi purpose programming language has never been more urgent. The desirability of a programming language depends on a lot of factors such as the ease of use, the features and functionalities that it provides, adaptability and what sort of problems can it solve. One is spoilt for choice with a number of options for a wide variety of programming paradigms, for example Object Oriented Languages. Over the last decade the declarative style of programming has gained popularity. The methodologies that have stood out are the Functional and Logical Approaches. The former is based on Functions and Lambda Calculus, while the latter is based on Horn Clause Logic. Each of them has its own advantages and aws. How does one choose which approach to adopt? Perhaps one does not need to choose! This document looks at the attempts, improvements and future possibilities of uniting Haskell, a Purely Functional Programming Language and Prolog, a Logical Programming Language so that one is not forced to choose.

1.2 Thesis Statement

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

1.3 Problem Statement

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

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

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

using PROLOG?

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

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

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

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

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

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

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

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

This can be achieved in two ways,

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

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

1.4 Proposal Organization

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

Chapter 6 and finishing off with the Chapter 7, the expected outcomes.

2 Background

Programming Languages fall into different categories also known as "paradigms". They exhibit different characteristics according to the paradigm they fall into. It has been argued [?] 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, [?] 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 [?], 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 [?], 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 [?] type system. The ability of the type system to give the most general type of a program without any help (annotation). The algorithm [?] 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, [?] 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 [?] 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 [?] functional languages around and is the first language to incorporate Monads [?] for safe *IO*. Monads can be described as composable computation descriptions [?] . 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 [?] in other languages like SCHEME [?], LISP [?, ?, ?],

JAVA [?, ?], JAVASCRIPT [?] and the list [?] 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 [?, ?, ?, ?, ?, ?]. 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) [?], 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) [?] 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) [?] which is an implementation of the Common Language Infrastructure (CLI) standard [?].

Another important concept is meta programming [?], 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 [?] 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 [?, ?, ?], 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

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

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

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

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

The approaches listed above are next in line for discussions.

3 Proposed Work

3.1 Current Work

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

1. Very few embedded implementations exist which offer a perspective into the job at hand. One of the earliest implementations [**?**] 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 [**?**] based of it simplifies the notation to a list format. Nonetheless, both implementations lack simplicity and support for basic PROLOG features such as *cuts, fails, assert* among others.

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

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

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

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

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

3.2 Contributions

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

3.3 Thesis Contributions

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

type → non rec type → fix non rec type isomorphically == rec type

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

4 Embedding a Programming Language into another Programming Language

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

4.1 The Informal Content from Blogs, Articles and Internet Discussions

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

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

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

mentations of PROLOG in HASKELL for Hugs 98, called Mini PROLOG [?]. 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, [?].

Adding fuel to fire, is the question on PROLOG's existence and survival [?, ?, ?, ?] 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 [?] of the same family, the family of Declarative Languages. Not to forget HASKELL also has some tricks [?] 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 [?], 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 [?]. A general out look towards implementing PROLOG has also been discussed by [?] 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 [**?**, **?**, **?**, **?**, **?**] 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 [**?**] 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, [**?**] 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 [**?**] 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. [?] 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 [?] being a preliminary or "mini Prolog" as such with not much in it to be able to be uselul, [?] is all powerful but is an Foreign Function Interface so it is "Prolog in Haskell" but we need Prolog for it, [?] 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. [?] is a continuation-based, backtracking, logic programming monad which sort of depicts Prolog's backtracking behaviour. Prolog is heavily based on formal logic, [?] provides a powerful system for Propositional Logic. Others include small hybrid languages [?] and Parallelising Logic

Programming and Tree Exploration [**?**].

- 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 [**?**], [**?**] 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 [**?**, **?**] use monads. Moreover, there is a package for the EGISON programming language [**?**] which supports non-linear pattern-matching with backtracking.

5 Multi Paradigm Languages (Functional Logic Languages)

Over the years another approach has branched off from embedding languages, to merge and/or integrate programming languages from different paradigms. Let us take an example of the Scala Programming Language [**?**], 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 [**?**], more over as Timothy Budd puts it [**?**] "The idea of a multi paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms."

5.1 The Informal Content from Blogs, Articles and Internet Discussions

- Multi Paradigm Languages

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

modules replicate different feature sets which are not naturally native to the language itself. The definitions of multi paradigm languages across the web [?, ?, ?] converge to roughly the same thing that of providing a framework to work with different styles with a list of languages [?, ?] 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 [?, ?] to the implementing techniques [?] 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 [?] 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 [?] and Sergio Antoy [?].

5.2 Literature and Publications

- Multi Paradigm Languages

  Possibly one of the most important works towards bringing programming styles together is the book by C.A.R. Hoare [?] 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 [**?**].

- Functional Logic Programming Languages

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

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

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

5.3 Some Multi Paradigm Languages

The list of multi paradigm languages is huge, but in this thesis we will mostly stick to Functional Logical programming languages. Beginning with functional hybrids, a small project language called VIRGIL [?], combining objects to work with functions and procedures. On similar lines is COMMON OBJECT LISP SYSTEM (CLOS) [?]. 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 [?] though it last saw a release back in 2001. Another prominent implementation is OCAML [?, ?] 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 [?] and widely usage both in academia and industry is the SCALA [?] programming language stands out.

5.4 Functional Logic Programming Languages

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

popular ones, beginning with Neu Prolog, [?], Oz (Mozart Programming System) [?] and Mercury [?]. 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 [?] comes very close to Haskell with monads, higher order functions and lazy evaluation. Taking a look at Prolog variants, Ciao [?]; a preprocessor to Prolog for functional syntax support, λ Prolog [?] aims at modular higher order programming with abstract data types in a logical setting, Babel [?, ?, ?] combines pure Prolog with a first order functional notation, LIFE [?] is for Logic, Inheritance, Functions and Equations in Prolog syntax with currying and other features like functional languages and others [?, ?].

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

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

6 Related Work

There are some technicalities which are indirectly related to the problem but do not bare a point of contact. The underpinnings of the languages throw some more light on the how different languages work to solve a problem. Different programming paradigms incorporate different operational mechanisms. For example, PROLOG programs execute on the Warren Abstract Machine [?] 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 [?] 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 [?], 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 [?, ?]. 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 [?], LIFE [?], NUE-PROLOG [?] and OZ [?]. 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 [?], BABEL [?], LPG [?] and CURRY [?].

7 Embedding a Programming Language into another Programming Language

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

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 [**?**]. 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

7.1 Theory

1. Papers

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

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

   (c) Embedded interpreters, [?]

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

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

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

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

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

2. Books

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

3. Articles / Blogs / Discussions

   (a) Embedding one language into another, [**?**]

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

   (c) Linguistic Abstraction, [**?**]

   (d) LISP, Unification and Embedded Languages, [**?**]

4. Websites

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

7.2 Implementations

1. Lots of them I guess

7.3 Important People

1. ????

7.4 Miscellaneous / Possibly Related Content

1. ????

8 Prolog in ____

Prolog in _____

8.1 Theory

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

8.2 Implementations

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

6. Yield Prolog, [**?**]

7. Racklog, [**?**]

8.3 Important People

1. ???

8.4 Miscellaneous / Possibly Related Content

1. ???

9 Prolog in Haskell

Prolog in Haskell

9.1 Theory

- Papers

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

- Books

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

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

- Articles / Blogs / Discussions

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

- Websites

1. Logic Programming in Haskell, [?]

9.2 Implementations

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

9.3 Important People

1. Mike Spivey
2. Silvija Seres

9.4 Miscellaneous / Possibly Related Content

1. Unification Libraries
   (a) unification-fd, [**?**]
   (b) cmu, [**?**]
2. Logic Libraries
   (a) logicct, [**?**], [**?**]
   (b) logic-classes, [**?**]
   (c) proplogic, [**?**]
   (d) cflp, [**?**]
   (e) logic-grows-on-trees, [**?**]
3. Concatenative Programming
   (a) peg, [**?**]
4. Constraint Programming and Constraint Handling Rules
   (a) monadiccp, [**?**]
   (b) monadicccp-gecode, [**?**]
   (c) csp, [**?**]
   (d) liquid fix point, [**?**]

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

Unifying / Marrying / Merging / Combining Programming Paradigms / Theories

10.1 Theory

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

10.2 Implementations

1. Scala
2. Virgil

3. CLOS, Common Lisp Object System

4. Visual Prolog

5. ????

10.3 Miscellaneous / Possibly Related Content

1. ???

11 Functional Logic Programming Languages

Functional Logic Programming Languages

11.1 Theory

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

11.2 Implementations

1. Hello

11.3 Miscellaneous / Possibly Related Content

1. Hello

12 Quasiquotation

12.1 Theory

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

12.2 Implementations

1.

12.3 Miscellaneous / Possibly Related Content

1.

13 Meta Syntactic Variables

Some sources for the topic

[?] 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.

[?] 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:

[?] 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.

[?]

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, ..."

14 Related Terms or Keywords

Related Terms / Keywords

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

15 Haskell or Why Haskell ?

In this chapter we discuss the properties of Haskell

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

1. Haskell as a functional programming language

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

   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 [**?**, **?**] which talk about introducing combinator libraries for custom functionality.

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

2. Looking at Haskell as a tool for embedding domain specific languages[**?**]

(a) Monads

Control flow defines the order/ manner of execution of statements in a program[**?**]. 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 [**?**]. 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[**?**].

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[**?**]. 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 [**?**]. A persistent data structure supports access to multiple versions which may arise after modifications [**?**, **?**]. 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 [**?**].

16 Prolog or Why Prolog ?

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

1. PROLOG as a logic programming language.

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

2. Why embed PROLOG ?

17 Miscellaneous or Possibly Related Content

Miscellaneous / Possibly Related Content

1. ???

18 Prototype 1

18.1 About this chapter

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

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

18.2 How Prolog works ?

Looking at how PROLOG works [**?**].

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.

For example, consider the append function

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

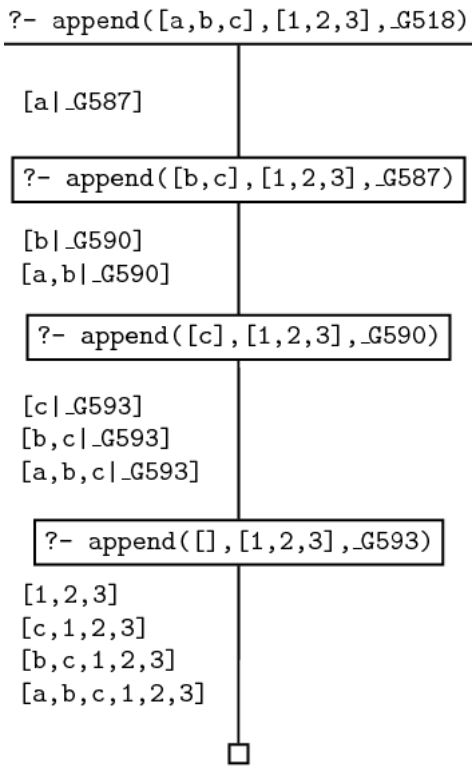

Figure 1: Trace for append [?]

18.3 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.

18.4 Creating a data type

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

The advantages of static typing [**?**]

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

For dynamic typing

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

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

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

Consider the language below,

```
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)
```

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

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

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

```
Struct Atom [Term]
```

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

```
type Atom         = String
data VariableName = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)
data Term = Struct Atom [Term]
```

```
          | Var VariableName
          | Wildcard -- Don't cares
          | Cut Int
      deriving (Eq, Data, Typeable)
```

Also one cannot create Quantifiers plus logic

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

```
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". Simply speaking we can have something like

```
FlatTerm Bool
```

and a generic fuinction like,

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

18.5 Working with the language

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 <- f
```

After flattening do fixing,

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

18.6 Black box

19 Prototype 2.1

19.1 About this chapter

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

19.2 How prolog-0.2.0.1 works

The original syntax used by the library,

```haskell
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)

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

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

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

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

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

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

An interpreter to solve a query and lastly the unifier,

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

19.3 What we do in this prototype?

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

We do complete PROLOG query resolution like stuff.

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

19.4 Current implementation (prolog-0.2.0.1)

The current unification uses basic pattern matching to unfiy 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')
```

19.5 Modifications

The first modification is to the language is to make it compatible with the library whcih provides this nice generic mechanism a perform unification in a mondic manner.

Fixing, flattening, creating necessary instances

```
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]
```

some translation and helper functions .....

and finally the unification

```
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))
  =>
      (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
```

19.6 Results

It works,

20 Prototype 2.2

nothing to do here

21 Prototype 3

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

term1 and term2 are matched and an assignment is the result

now this may be a part of a query resolution procedure

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

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

21.1 Unification

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

21.2 Resolution

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

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

[**?**]

21.3 Search strategies

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

21.4 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 Subst
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 substitution at t
-- point, goal is the outstanding goal and alts is a list of possible
-- of extending the current proof to find a solution.  Each member of
-- is a pair (tp,u) where tp is a new subgoal that must be proved and
-- a unifying substitution that must be combined with the substitution
--
-- the list of relevant clauses at each step in the execution is produ
-- by attempting to unify the head of the current goal with a suitably
-- renamed clause from the database.

type Stack = [ (Subst, [Term], [Alt]) ]
type Alt   = ([Term], Subst)

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] -> [Subst]
prove db gl = solve 1 nullSubst gl []
 where
   solve :: Int -> Subst -> [Term] -> Stack -> [Subst]
   solve n s []     ow          = s : backtrack n ow
```

```
43    solve n s (g:gs) ow
44                    | g==theCut = solve n s gs (cut ow)
45                    | otherwise = choose n s gs (alts db n (app s g)) o
46
47    choose :: Int -> Subst -> [Term] -> [Alt] -> Stack -> [Subst]
48    choose n s gs []          ow = backtrack n ow
49    choose n s gs ((tp,u):rs) ow = solve (n+1) (u@@s) (tp++gs) ((s,gs,rs
50
51    backtrack                   :: Int -> Stack -> [Subst]
52    backtrack n []               = []
53    backtrack n ((s,gs,rs):ow)   = choose (n-1) s gs rs ow
54
55
56  --- Special definitions for the cut predicate:
57
58  theCut    :: Term
59  theCut     = Struct "!" []
60
61  cut                  :: Stack -> Stack
62  cut ss                = []
63
64  --- End of Engine.hs
```

21.4.1 Pure Engine

```
1   -- The Pure Prolog inference engine (using explicit prooftrees)
2   -- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
3   -- and for Hugs 1.3 June 1996.
4   --
5   -- Suitable for use with Hugs 98.
6   --
7
8   module PureEngine( version, prove ) where
9
10  import Prolog
11  import Subst
12  import Interact
13  import Data.List(nub)
14
15  version = "tree based"
16
17  --- Calculation of solutions:
18
```

```
19 -- Each node in a prooftree corresponds to:
20 -- either: a solution to the current goal, represented by Done s, where
21 --         is the required substitution
22 -- or:     a choice between a number of subtrees ts, each corresponding
23 --         proof of a subgoal of the current goal, represented by Choice
24 --         The proof tree corresponding to an unsolvable goal is Choice
25
26 data Prooftree = Done Subst  |  Choice [Prooftree]
27
28 -- prooftree uses the rules of Prolog to construct a suitable proof tree
29 --           a specified goal
30 prooftree   :: Database -> Int -> Subst -> [Term] -> Prooftree
31 prooftree db = pt
32  where pt           :: Int -> Subst -> [Term] -> Prooftree
33        pt n s []     = Done s
34        pt n s (g:gs) = Choice [ pt (n+1) (u@@s) (map (app u) (tp++gs))
35                               | (tm:-tp)<-renClauses db n g, u<-unify g
36 {--
37 pt 1 nullSubst [] = Done (nullSubst)
38
39 pt n s (g:gs)
40
41 renClauses :- Rename variables in a clause, the parameters are the data
42                         (head of list) resulting in a clause.
43
44 unify :- take the head of the list and and match with head of clause fi
45
46 app :- function for applying (Subst) to (Terms)
47 the new list is formed by replacing the cluase head with its body and a
48
49 so the new parameters for pt are
50
51 (n+1) (the old substitution + the new one from unify) (the list formed
52
53
54 Working of a small example
55
56 The database,
57 (foldl addClause emptyDb [((:-) (Struct "hello" []) []), ((:-) (Struct
58 hello.
59 hello(world).
60 hello:-world.
61 hello(X_1).
62
63 The other parameters are 1 nullSubst(as mentioned in the prove functio
```

```
64
65  For the list of goals, [(Struct "hello" []), (Struct "hello" [(Struct
66
67  1. [Struct "hello" []] :: [Term]
68
69  * Rule 1 does not apply
70
71  * Rule 2 does apply,
72
73  (tm:- tp) <- renClauses db 1 (Struct "hello" [])
74
75  tm ==> "hello , hello(world) , hello , hello(X_1) , "
76  tp ==> "[] , [] , [world] , [] , "
77
78
79
80
81
82
83
84
85
86  --}
87
88
89
90  -- DFS Function
91  -- search performs a depth-first search of a proof tree, producing the
92  -- of solution substitutions as they are encountered.
93  search               :: Prooftree -> [Subst]
94  search (Done s)       = [s]
95  search (Choice pts)   = [ s | pt <- pts, s <- search pt ]
96
97
98  prove    :: Database -> [Term] -> [Subst]
99  prove db  = search . prooftree db 1 nullSubst
100
101 --- End of PureEngine.hs
```

21.4.2 Andorra Engine

```
1  {-
2  By Donald A. Smith, December 22, 1994, based on Mark Jones' PureEngine
```

```
3
4  This inference engine implements a variation of the Andorra Principle
5  logic programming. (See references at the end of this file.) The basic
6  idea is that instead of always selecting the first goal in the current
7  list of goals, select a relatively deterministic goal.
8
9  For each goal g in the list of goals, calculate the resolvents that wo
10 result from selecting g.  Then choose a g which results in the lowest
11 number of resolvents.  If some g results in 0 resolvents then fail.
12 (This would occur for a goal like:  ?- append(A,B,[1,2,3]),equals(1,2)
13 Prolog would not perform this optimization and would instead search
14 and backtrack wastefully.  If some g results in a single resolvent
15 (i.e., only a single clause matches) then that g will get selected;
16 by selecting and resolving g, bindings are propagated sooner, and usel
17 search can be avoided, since these bindings may prune away choices for
18 other clauses.  For example: ?- append(A,B,[1,2,3]),B=[].
19 -}
20
21 module AndorraEngine( version, prove ) where
22
23 import Prolog
24 import Subst
25 import Interact
26
27 version = "Andorra Principle Interpreter (select deterministic goals fi
28
29 solve   :: Database -> Int -> Subst -> [Term] -> [Subst]
30 solve db = slv where
31    slv           :: Int -> Subst -> [Term] -> [Subst]
32    slv n s [] = [s]
33    slv n s goals =
34     let allResolvents = resolve_selecting_each_goal goals db n in
35       let (gs,gres) =  findMostDeterministic allResolvents in
36           concat [slv (n+1) (u@@s) (map (app u) (tp++gs)) | (u,tp) <- g
37
38 resolve_selecting_each_goal::
39     [Term] -> Database -> Int -> [([Term],[(Subst,[Term])])]
40 --  For each pair in the list that we return, the first element of the
41 --  pair is the list of unresolved goals; the second element is the li
42 --  of resolvents of the selected goal, where a resolvent is a pair
43 --  consisting of a substitution and a list of new goals.
44 resolve_selecting_each_goal goals db n = [(gs, gResolvents) |
45       (g,gs) <- delete goals, let gResolvents = resolve db g n]
46
47 -- The unselected goals from above are not passed in.
```

```
48 resolve :: Database -> Term -> Int -> [(Subst,[Term])]
49 resolve db g n = [(u,tp) | (tm:-tp)<-renClauses db n g, u<-unify g tm]
50 -- u is not yet applied to tp, since it is possible that g won't be se
51 -- Note that unify could be nondeterministic.
52
53 findMostDeterministic:: [([Term],[(Subst,[Term])])] -> ([Term],[(Subst
54 findMostDeterministic  allResolvents = minF comp allResolvents where
55    comp:: (a,[b]) -> (a,[b]) -> Bool
56    comp (_,gs1) (_,gs2) = (length gs1) < (length gs2)
57 -- It seems to me that there is an opportunity for a clever compiler t
58 -- optimize this code a lot. In particular, there should be no need to
59 -- determine the total length of a goal list if it is known that
60 -- there is a shorter goal list in allResolvents ... ?
61
62 delete ::  [a] -> [(a,[a])]
63 delete l = d l [] where
64    d :: [a] -> [a] ->  [(a,[a])]
65    d [g] sofar = [ (g,sofar) ]
66    d (g:gs) sofar = (g,sofar++gs) : (d gs (g:sofar))
67
68 minF                :: (a -> a -> Bool) -> [a] -> a
69 minF f (h:t) = m h t where
70 --   m :: a -> [a] -> a
71      m sofar [] = sofar
72      m sofar (h:t) = if (f h sofar) then m h t else m sofar t
73
74 prove    :: Database -> [Term] -> [Subst]
75 prove db  = solve db 1 nullSubst
76
77 {- An optimized, incremental version of the above interpreter would us
78   a data representation in which for each goal in "goals" we carry aro
79   the list of resolvents.  After each resolution step we update the li
80 -}
81
82 {- References
83
84    Seif Haridi & Per Brand, "Andorra Prolog, an integration of Prolog
85    and committed choice languages" in Proceedings of FGCS 1988, ICOT,
86    Tokyo, 1988.
87
88    Vitor Santos Costa, David H. D. Warren, and Rong Yang, "Two papers
89    the Andorra-I engine and preprocessor", in Proceedings of the 8th
90    ICLP. MIT Press, 1991.
91
92    Steve Gregory and Rong Yang, "Parallel Constraint Solving in
```

```
93     Andorra-I", in Proceedings of FGCS'92. ICOT, Tokyo, 1992.
94
95     Sverker Janson and Seif Haridi, "Programming Paradigms of the Andor
96     Kernel Language", in Proceedings of ILPS'91. MIT Press, 1991.
97
98     Torkel Franzen, Seif Haridi, and Sverker Janson, "An Overview of th
99     Andorra Kernel Language", In LNAI (LNCS) 596, Springer-Verlag, 1992
100 -}
```

21.5 Current Unification

```
1  {-# LANGUAGE  DeriveDataTypeable,
2                ViewPatterns,
3                ScopedTypeVariables,
4                DefaultSignatures,
5                TypeOperators,
6                TypeFamilies,
7                DataKinds,
8                DataKinds,
9                PolyKinds,
10               OverlappingInstances,
11               TypeOperators,
12               LiberalTypeSynonyms,
13               TemplateHaskell,
14               AllowAmbiguousTypes,
15               ConstraintKinds,
16               Rank2Types,
17               MultiParamTypeClasses,
18               FunctionalDependencies,
19               FlexibleContexts,
20               FlexibleInstances,
21               UndecidableInstances
22               #-}
23
24 -- Substitutions and Unification of Prolog Terms
25 -- Mark P. Jones November 1990, modified for Gofer 20th July 1991,
26 -- and for Hugs 1.3 June 1996.
27 --
28 -- Suitable for use with Hugs 98.
29 --
30
31 module Subst where
32
```

```
33 import Prolog
34 import CustomSyntax
35 import Data.Map as Map
36 import Data.Maybe
37 import Data.Either
38
39 --Unification
40 import Control.Unification.IntVar
41 import Control.Unification.STVar as ST
42
43 import Control.Unification.Ranked.IntVar
44 import Control.Unification.Ranked.STVar
45
46 import Control.Unification.Types as UT
47
48 import Control.Monad.State.UnificationExtras
49 import Control.Unification as U
50
51 -- Monads
52 import Control.Monad.Error
53 import Control.Monad.Trans.Except
54
55 import Data.Functor.Fixedpoint as DFF
56
57 --State
58 import Control.Monad.State.Lazy
59 import Control.Monad.ST
60 import Control.Monad.Trans.State as Trans
61
62 infixr 3 @@
63 infix  4 ->-
64
65 --- Substitutions:
66
67 type Subst = Id -> Term
68
69 newtype SubstP = SubstP { unSubstP :: Subst }
70
71 -- instance Show SubstP where
72 --  show (i) = show $ Var i
73 -- substitutions are represented by functions mapping identifiers to t
74 --
75 -- app s     extends the substitution s to a function mapping terms to
76 {--
77 Looks like an apply function that applies a substitution function tho
```

```
78  --}
79
80
81  -- nullSubst is the empty substitution which maps every identifier to
82
83
84
85  -- i ->- t   is the substitution which maps the identifier i to the te
86
87
88  -- s1@@ s2  is the composition of substitutions s1 and s2
89  --           N.B.  app is a monoid homomorphism from (Subst,nullSubst,
90  --           to (Term -> Term, id, (.)) in the sense that:
91  --                  app (s1 @@ s2) = app s1 . app s2
92  --                s @@ nullSubst = s = nullSubst @@ s
93
94  app                      :: Subst -> Term -> Term
95  app s (Var i)             = s i
96  app s (Struct a ts)       = Struct a (Prelude.map (app s) ts)
97  {--
98  app (substFunction) (Struct "hello" [Var (0, "Var")])
99  hello(Var_2) :: Term
100
101 --}
102
103
104 nullSubst                :: Subst
105 nullSubst i               = Var i
106 {--
107 nullSubst (0, "Var")
108 Var :: Term
109 --}
110
111
112 --
113 (->-)                    :: Id -> Term -> Subst
114 (i ->- t) j | j==i        = t
115             | otherwise   = Var j
116 {--
117 :t (->-) (1,"X") (Struct "hello" [])
118 (1,"X") ->- Struct "hello" [] :: (Int,[Char]) -> Term
119 --}
120
121
122 -- Function composition for applying two substitution functions.
```

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

21.6 Syntax Modification

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

```
39
40 import Control.Unification.Ranked.IntVar
41 import Control.Unification.Ranked.STVar
42
43 import Control.Unification.Types as UT
44
45
46
47 import Data.Traversable as T
48 import Data.Functor
49 import Data.Foldable
50 import Control.Applicative
51
52
53 import Data.List.Extras.Pair
54 import Data.Map as Map
55 import Data.Set as S
56
57
58 import Control.Monad.Error
59 import Control.Monad.Trans.Except
60
61
62 import Prolog
63
64 data FTS a = forall a . FV Id | FS Atom [a] deriving (Eq, Show, Ord, Ty
65
66 newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)
67
68 unP :: Prolog -> Fix FTS
69 unP (P x) = x
70
71 instance Functor FTS where
72         fmap = T.fmapDefault
73
74 instance Foldable FTS where
75          foldMap = T.foldMapDefault
76
77 instance Traversable FTS where
78         traverse f (FS atom xs) = FS atom <$> sequenceA (Prelude.map f
79         traverse _ (FV v) =         pure (FV v)
80
81 instance Unifiable FTS where
82         zipMatch (FS al ls) (FS ar rs) = if (al == ar) && (length ls ==
83                                          then FS al <$> pairWith (\l r ->
```

```
84                                          else Nothing
85          zipMatch (FV v1) (FV v2) = if (v1 == v2) then Just (FV v1)
86                  else Nothing
87          zipMatch _ _ = Nothing
88
89  instance Applicative FTS where
90          pure x = FS "" [x]
91          (FS a fs) <*> (FS b xs)    = FS (a ++ b) [f x | f <- fs, x <- xs
92          --other cases
93  {--
94  instance Monad FTS where
95          func =
96  instance Variable FTS where
97          func =
98
99  instance BindingMonad FTS where
100         func =
101 --}
102
103 data VariableName = VariableName Int String
104
105 idToVariableName :: Id -> VariableName
106 idToVariableName (i, s) = VariableName i s
107
108 variablenameToId :: VariableName -> Id
109 variablenameToId (VariableName i s) = (i,s)
110
111 termFlattener :: Term -> Fix FTS
112 termFlattener (Var v)           =   DFF.Fix $ FV v
113 termFlattener (Struct a xs)     =   DFF.Fix $ FS a (Prelude.map termFla
114
115 unFlatten :: Fix FTS -> Term
116 unFlatten (DFF.Fix (FV v))      =   Var v
117 unFlatten (DFF.Fix (FS a xs))   =   Struct a (Prelude.map unFlatten xs)
118
119
120 variableExtractor :: Fix FTS -> [Fix FTS]
121 variableExtractor (Fix x) = case x of
122   (FS _ xs)    ->  Prelude.concat $ Prelude.map variableExtractor xs
123   (FV v)       ->  [Fix $ FV v]
124 --  _          ->  []
125
126 variableIdExtractor :: Fix FTS -> [Id]
127 variableIdExtractor (Fix x) = case x of
128         (FS _ xs) -> Prelude.concat $ Prelude.map variableIdExtractor x
```

```
129           (FV v) -> [v]
130
131 {--
132 variableNameExtractor :: Fix FTS -> [VariableName]
133 variableNameExtractor (Fix x) = case x of
134   (FS _ xs) -> Prelude.concat $ Prelude.map variableNameExtractor xs
135   (FV v)     -> [v]
136   _          -> []
137 --}
138
139 variableSet :: [Fix FTS] -> S.Set (Fix FTS)
140 variableSet a = S.fromList a
141
142 variableNameSet :: [Id] -> S.Set (Id)
143 variableNameSet a = S.fromList a
144
145
146 varsToDictM :: (Ord a, Unifiable t) =>
147     S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
148 varsToDictM set = foldrM addElt Map.empty set where
149   addElt sv dict = do
150     iv <- freeVar
151     return $! Map.insert sv iv dict
152
153
154 uTermify
155   :: Map Id (ST.STVar s (FTS))
156   -> UTerm FTS (ST.STVar s (FTS))
157   -> UTerm FTS (ST.STVar s (FTS))
158 uTermify varMap ux = case ux of
159   UT.UVar _              -> ux
160   UT.UTerm (FV v)        -> maybe (error "bad map") UT.UVar $ Map.looku
161  -- UT.UTerm t             -> UT.UTerm $! fmap (uTermify varMap) t
162   UT.UTerm (FS a xs)     -> UT.UTerm $ FS a $! fmap (uTermify varMap) x
163
164
165 translateToUTerm ::
166     Fix FTS -> ST.STBinding s
167               (UT.UTerm (FTS) (ST.STVar s (FTS)),
168                Map Id (ST.STVar s (FTS)))
169 translateToUTerm e1Term = do
170   let vs = variableNameSet $ variableIdExtractor e1Term
171   varMap <- varsToDictM vs
172   let t2 = uTermify varMap . unfreeze $ e1Term
173   return (t2,varMap)
```

```
174
175
176 -- | vTermify recursively converts @UVar x@ into @UTerm (VarA x).
177 -- This is a subroutine of @ translateFromUTerm @.  The resulting
178 -- term has no (UVar x) subterms.
179
180 vTermify :: Map Int Id ->
181             UT.UTerm (FTS) (ST.STVar s (FTS)) ->
182             UT.UTerm (FTS) (ST.STVar s (FTS))
183 vTermify dict t1 = case t1 of
184   UT.UVar x  -> maybe (error "logic") (UT.UTerm . FV) $ Map.lookup (UT
185   UT.UTerm r ->
186     case r of
187       FV iv   -> t1
188       _       -> UT.UTerm . fmap (vTermify dict) $ r
189
190 translateFromUTerm ::
191     Map Id (ST.STVar s (FTS)) ->
192     UT.UTerm (FTS) (ST.STVar s (FTS)) -> Prolog
193 translateFromUTerm dict uTerm =
194   P .  maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm
195     forKV dict initial fn = Map.foldlWithKey' (\a k v -> fn k v a) init
196     varIdDict = forKV dict Map.empty $ \ k v -> Map.insert (UT.getVarI
197
198
199 -- | Unify two (E1 a) terms resulting in maybe a dictionary
200 -- of variable bindings (to terms).
201 --
202 -- NB !!!!
203 -- The current interface assumes that the variables in t1 and t2 are
204 -- disjoint.  This is likely a mistake that needs fixing
205
206 unifyTerms :: Fix FTS -> Fix FTS -> Maybe (Map Id (Prolog))
207 unifyTerms t1 t2 = ST.runSTBinding $ do
208   answer <- runExceptT $ unifyTermsX t1 t2
209   return $! either (const Nothing) Just answer
210
211 -- | Unify two (E1 a) terms resulting in maybe a dictionary
212 -- of variable bindings (to terms).
213 --
214 -- This routine works in the unification monad
215
216 unifyTermsX ::
217     Fix FTS -> Fix FTS ->
218     ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
```

```
219           (ST.STBinding s)
220           (Map Id (Prolog))
221 unifyTermsX t1 t2 = do
222     (x1,d1) <- lift . translateToUTerm $ t1
223     (x2,d2) <- lift . translateToUTerm $ t2
224     _ <- unify x1 x2
225     makeDicts $ (d1,d2)
226
227
228
229 mapWithKeyM :: (Ord k,Applicative m,Monad m)
230                => (k -> a -> m b) -> Map k a -> m (Map k b)
231 mapWithKeyM = Map.traverseWithKey
232
233
234 makeDict ::
235             Map Id (ST.STVar s (FTS)) -> ST.STBinding s (Map Id (Prolo
236 makeDict sVarDict =
237     flip mapWithKeyM sVarDict $ \ _ -> \ iKey -> do
238         Just xx <- UT.lookupVar $ iKey
239         return $! (translateFromUTerm sVarDict) xx
240
241
242 -- | recover the bindings for the variables of the two terms
243 -- unified from the monad.
244
245 makeDicts ::
246     (Map Id (ST.STVar s (FTS)), Map Id (ST.STVar s (FTS))) ->
247     ExceptT  (UT.UFailure (FTS) (ST.STVar s (FTS)))
248     (ST.STBinding s) (Map Id (Prolog))
249 makeDicts (svDict1, svDict2) = do
250   let svDict3 = (svDict1 `Map.union` svDict2)
251   let ivs = Prelude.map UT.UVar . Map.elems $ svDict3
252   applyBindingsAll ivs
253   -- the interface below is dangerous because Map.union is left-biased
254   -- variables that are duplicated across terms may have different
255   -- bindings because `translateToUTerm` is run separately on each
256   -- term.
257   lift . makeDict $ svDict3
258
259 instance (UT.Variable v, Functor t) => Error (UT.UFailure t v) where {
260
261 test1 ::
262   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
263             (ST.STBinding s)
```

```
264             (UT.UTerm (FTS) (ST.STVar s (FTS)),
265              Map Id (ST.STVar s (FTS)))
266 test1 = do
267     let
268         t1a = (Fix $ FV $ (0, "x"))
269         t2a = (Fix $ FV $ (1, "y"))
270     (x1,d1) <- lift . translateToUTerm $ t1a --error
271     (x2,d2) <- lift . translateToUTerm $ t2a
272     x3 <- U.unify x1 x2
273     return (x3, d1 `Map.union` d2)
274
275
276 test2 ::
277   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
278             (ST.STBinding s)
279              (UT.UTerm (FTS) (ST.STVar s (FTS)),
280               Map Id (ST.STVar s (FTS)))
281 test2 = do
282     let
283         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
284         t2a = (Fix $ FV $ (1, "y"))
285     (x1,d1) <- lift . translateToUTerm $ t1a --error
286     (x2,d2) <- lift . translateToUTerm $ t2a
287     x3 <- U.unify x1 x2
288     return (x3, d1 `Map.union` d2)
289
290
291 test3 ::
292   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
293             (ST.STBinding s)
294              (UT.UTerm (FTS) (ST.STVar s (FTS)),
295               Map Id (ST.STVar s (FTS)))
296 test3 = do
297     let
298         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
299         t2a = (Fix $ FV $ (0, "x"))
300     (x1,d1) <- lift . translateToUTerm $ t1a --error
301     (x2,d2) <- lift . translateToUTerm $ t2a
302     x3 <- U.unify x1 x2
303     return (x3, d1 `Map.union` d2)
304 {--
305 goTest test3
306 "ok:    STVar -9223372036854775807
307 [(VariableName 0 \"x\",STVar -9223372036854775808)]"
308 --}
```

```
309
310 test4 ::
311   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
312             (ST.STBinding s)
313              (UT.UTerm (FTS) (ST.STVar s (FTS)),
314               Map Id (ST.STVar s (FTS)))
315 test4 = do
316     let
317         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
318         t2a = (Fix $ FV $ (0, "x"))
319     (x1,d1) <- lift . translateToUTerm $ t1a --error
320     (x2,d2) <- lift . translateToUTerm $ t2a
321     x3 <- U.unifyOccurs x1 x2
322     return (x3, d1 `Map.union` d2)
323 {--
324 goTest test4
325 "ok:    STVar -9223372036854775807
326 [(VariableName 0 \"x\",STVar -9223372036854775808)]"
327 --}
328
329 test5 ::
330   ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
331             (ST.STBinding s)
332              (UT.UTerm (FTS) (ST.STVar s (FTS)),
333               Map Id (ST.STVar s (FTS)))
334 test5 = do
335     let
336         t1a = (Fix $ FS "a" [Fix $ FV $ (0, "x")])
337         t2a = (Fix $ FS "b" [Fix $ FV $ (0, "y")])
338     (x1,d1) <- lift . translateToUTerm $ t1a --error
339     (x2,d2) <- lift . translateToUTerm $ t2a
340     x3 <- U.unify x1 x2
341     return (x3, d1 `Map.union` d2)
342
343 goTest :: (Show b) => (forall s .
344   (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
345             (ST.STBinding s)
346              (UT.UTerm (FTS) (ST.STVar s (FTS)),
347               Map Id (ST.STVar s (FTS))))) -> String
348 goTest test = ST.runSTBinding $ do
349   answer <- runErrorT $ test
350   return $! case answer of
351     (Left x)  -> "error: " ++ show x
352     (Right y) -> "ok:    " ++ show y
353
```

```
354
355 ------------------------------------------------------------------
356 ------------------------------------------------------------------
357 --------------GLUE-CODE-------------------------------------------
358 {--
359 monadicUnify :: Term -> Term -> ErrorT (UT.UFailure (FTS) (ST.STVar s
360             (ST.STBinding s)
361              (UT.UTerm (FTS) (ST.STVar s (FTS)),
362               Map Id (ST.STVar s (FTS)))
363 monadicUnify t1 t2 = do
364         let
365                 t1f = termFlattener t1
366                 t2f = termFlattener t2
367         (x1,d1) <- lift . translateToUTerm $ t1f
368         (x2,d2) <- lift . translateToUTerm $ t2f
369         x3 <- U.unify x1 x2
370         return (x3, d1 `Map.union` d2)
371
372 --}
373
374 -- type Subst = Id -> Term
375
376 -- Convert result from monadicUnify to [Subst]
377 {--
378 goMonadicTest :: (Show b) => (forall s .
379   (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
380             (ST.STBinding s)
381              (UT.UTerm (FTS) (ST.STVar s (FTS)),
382               Map Id (ST.STVar s (FTS))))) -> [Subst]
383 goMonadicTest test = ST.runSTBinding $ do
384   answer <- runErrorT $ test
385   return $! case answer of
386     (Left x)  -> [nullSubst]
387     (Right y) -> convertToSubst y
388 --}
389
390 --(Id, STVar s FTS)
391 --convertToSubst :: Map Id (ST.STVar s FTS) -> [(Id, ST.STVar s FTS)]
392 {--
393 convertToSubst m = convertToSubst1 Map.toAscList m
394
395 convertToSubst1 (id, ST.STVar _ fts):xs = (id, (unFlatten fts)) : conv
396 --}
```

21.7 Monadic Unification

```
monadicUnification :: (BindingMonad FTS (STVar s FTS) (ST.STBinding s)
            (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
```

```
43
44
45 --unify :: Term -> Term -> [Subst]
46 unify t1 t2 = substConvertor (goUnify (monadicUnification t1 t2))
47
48
49 varX :: Term
50 varX = Var (0,"x")
51
52 varY :: Term
53 varY = Var (1,"y")
54
55
56 substConvertor :: [(Id, Prolog)] -> [Subst]
57 substConvertor xs = Prelude.map (\(varId, p) -> (->-) varId (unFlatten
```

22 Prototype 4

Our aim to embedd IO into the DSL

So something like a "data" declaration for IO operations

```haskell
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 a
                | Put String (IOAction a)
-- A container holding a function from String -> IOAction a, which can
                | Get (String -> IOAction a)
```

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

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

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

23 Work Completed

23.1 What we are doing

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

23.2 Unifiable Data Structures

For a data type to be Unifiable, it must have instances of Functor, Foldable and Traversable. The interaction between different classes is depicted in figure 2.

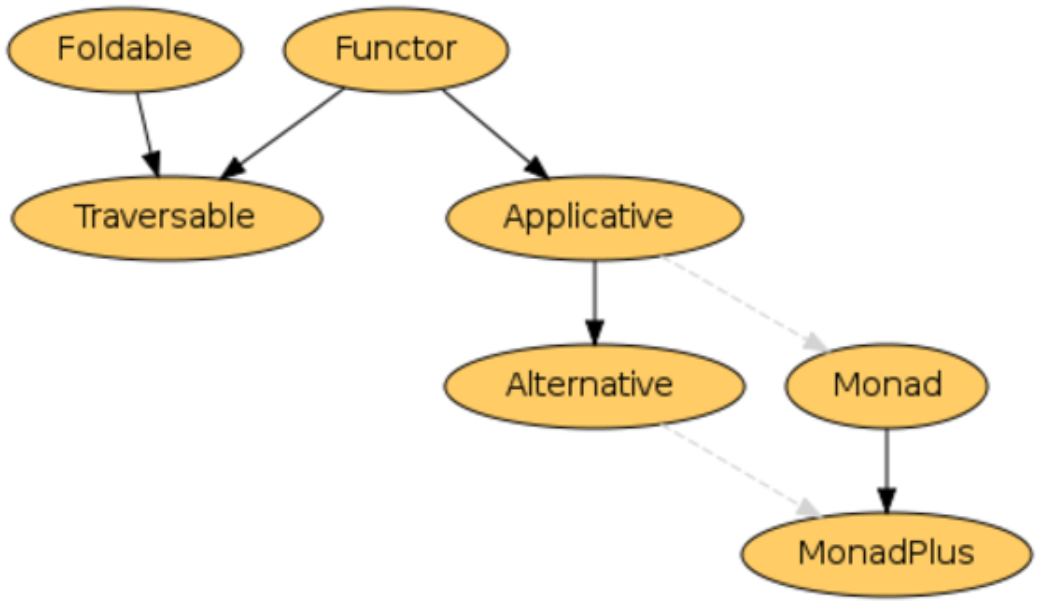

Figure 2: Functor Hierarchy [**?**]

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

23.3 Why `Fix` is necessary?

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

In our case consider the following example,

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

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

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

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

```
Fix FlatTerm
```

resulting in the Prolog Data Type

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

23.4 Dr. Casperson's Explanation

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

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

A sample `Tree` would be,

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

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

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

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

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

23.5 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.

23.6 The Fix we use

Fix-point type allows to define generic recursion schemes [**?**].

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

24 Results

24.1 Types

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

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

Our preliminary implementation is as follows,

```
type Atom = String

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

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

{--
Output :-

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

--}
```

which in PROLOG would look like,

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

24.2 Lazy Evaluation

24.3 Opening up the Language

Flattening

Fixing

MetaSyntactic Variables

24.4 Quasi Quotation

24.5 Template Haskell

24.6 Higher Order Functions

```
% Mehul Solanki.

% Higher Order Functions.

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

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

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

24.7 I/O

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

24.8 Mutability

24.9 Unification

24.10 Monads

25 Conclusion / Expected Outcomes

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

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

26 Editing to do

This Chapter needs to be removed from the final work.

2015-10-29

1. Abstract is too long and incorrect.

2. Remove first ¶ paragraph from intro.

Either

3. Rename "`proposal.*`" to "`thesis-solanki.*`".

4. Switch the thesis style to UNBC thesis style. (Not urgent, if this breaks other tools, we can do this last, but it would be nice to have a sense of what the thesis is going to look like.)

5. Check the rules for spacing in the bibliography to ensure that we have them right.

Mehul

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 18–22 that we can decide what material is needed in Chapters 12, 13, and 14.

9. [TEXnical] Remove the `\paragraph{}`s from the running text. LaTeX ends a paragraph every time that it encounters two end-of-lines with only whitespace between them. `\par` does the same thing.

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

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

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

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

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

11. Bump the sectioning levels up by one. That is, what is currently a section should become a chapter, what is currently a subsection should become a section, and so on. It may not make sense to do this until you have switch to thesis.sty.

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

13. 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.

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

David

15. Review Chapter 1

16. Review Chapter 2

17. Review Chapter 3

18. Review Chapter 4

19. Review Chapter 5

20. Review Chapter 6

21. Review Chapter 7

22. Review Chapter 8

23. Review Chapter 18

26.1 Editing suggestions from David

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

1. Undergraduate Physics students

2. Undergraduate Computer Science students

3. Future graduate students of Casperson who have just begun their thesis work.

4. Simon Peyton-Jones.

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

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

Thoughts on 1.4 Rename to "Thesis Organization".

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

What is the overall message?