1 Embedding a Programming Language into another Programming Language

The art of embedding a programming language into another one has been explored and played around with 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/or 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 along with some other implementations and embedding languages in general.

1.1 The Informal Content from Blogs, Articles and Internet Discussions

Before moving on to the formal content like publications and modules and/or libraries we shall *throw light upon the streets*. This subsection takes a look at the information, thoughts and discussions that take 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/or 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 would be 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, modules among others. One of the implementations discussed redirects us to one of the most earliest implementations of PROLOG in HASKELL for Hugs 98, called Mini PROLOG [**?**]. 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 and 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, would be 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.

1.2 Related Books

As HASKELL is relatively new in terms of being popular, its predecessors like SCHEME have explored this territory 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.

1.3 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

A series of papers [?, ?, ?, ?, ?] covers the topic in a sufficiently thorough manner. The attempt throws light on the subject of Embedding Prolog in Haskell from all aspects. Moreover, it is one of the first formal attempts at Embedding Prolog in Haskell. It takes quite some leads from implementations of Prolog in other languages like Scheme and Lisp and also some Multi Paradigm Declarative Languages. But the difference here being that Lisp is strict while Haskell is lazy which leads to a natural backtracking behaviour. The basic idea being that each Prolog Predicate is translated to a Haskell Function which will work on lists and produce a Stream of results lazily. The aim was never to develop a Hybrid Functional Logic Programming Language but to put forth a set of general rules for embedding. Moreover the initial model is for a more Pure Declarative Prolog rather than the Practical one. The extension is very minimalistic with only four new constructs and/or functions, Conjunction, Disjunction, Unification and the Existential Quantifier. A general technique has also been described for converting a Prolog Predicate to a Haskell Function. The Prolog terms are still untyped and the search is carried out in a Breadth First manner but there is no support for Higher Order Functions in the first revision and a implementation does not exist.

The follow up papers then provide support for Depth First Search and include practical features like *not* and *cut* operators. Pushing it further, data types to work with Depth First Search and also a more General Strategy to interchange mechanisms. Covering up some downfalls, a more compositional approach like the one in functional programming languages is proposed to incorporate higher order functions (higher order predicates). Synthesis and Transformation techniques for Functional Programs have been *logicalized* and applied to Prolog Programs.

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 .

- Other works relate and/or based on the above

  This section takes a look at the improvements at the attempts mostly based on the work from the previous section. Some work is done by the one of the authors above while the other prominent others in the same field.

  Taking the idea further, an attempt [?] brings the HASKELL type system to PROLOG resulting in something called typed logic variables. The aim that has been stated is to embed a simple typed functional logic programming language giving HASKELL logic programming features. Another important aspect that has been touched upon in the paper is the backtracking ability of PROLOG which has been replicated using a monad. The implementation does not support practical predicates like cuts or assert or retract and so on leave alone the details, for example if a predicate is half way through and then a new fact is asserted into the database, the question arises is this change taken into account or the old database is used. The main issues tackled here are how to translate PROLOG predicates elegantly into HASKELL functions, addressed with ground terms and a *Class* which can be used for unification; overloaded according to the terms being unified. The second issue is how to make the implementation more suited to the problem, for example PROLOG by default works on depth first search which may not be a fair strategy if the requirement is for visiting all possible options to a certain degree of depth, in which case breadth first search is more suitable. The idea is to make the language modifiable with respect to search strategies. Though issues with efficiency in terms of syntax, the unification algorithm and also the lack of practical features exist.

  While some attempt at doing problems suitable for Prolog in Haskell [?]. The exercise subjects a set of students to solve a puzzle in PROLOG and then in HASKELL and comparing the results and experiences. It turns out that, after trying things out in PROLOG, HASKELL seems easier as the general structure of a problem is expressed using a type class and then the procedure to find a solution is an instance. The result being that the HASKELL type system helps.

Another attempt [?] looks at playing with data types used to implement Breadth First Search in HASKELL. PROLOG does a depth first search implicitly, so if it is to be replicated in HASKELL it will be forced to do it and hence it is an explicit behaviour. As a lazy language, the host is capable of working with possibly infinite data structures and this where the problem is as depth first search will end up going down one and only one branch. So the other branches will never be explored so as to contribute to the solutions and hence better to do a depth first search to make the procedure more fair by visiting several nodes at the same depth. Rather than using *lists* of *streams*, a better solution is to use *bags* implemented as finite lists which preserve the properties such as the associativity of join. The idea being that the operators behave like $\vee$ and $\wedge$ of logic programming in terms of properties. The category is of operators, while the initial object is the search tree and the morphisms being the different search strategies.

A lot of the investigations above have been based of [?] which tries to replicate PROLOG's control operations in HASKELL. But as there are implications of the host language resulting in the loss of data base operations. But having a global state using the *State Monad* does not seem too difficult. *Monads* have usages which are internal; that is structuring programs and external; usages that is extending the language. The monads in this paper add PROLOG like capabilities but do not extend the capabilities of HASKELL as such. 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. The aim here is to deal with the axiomatic details of the embedding. There are monads for encapsulation, backtracking, exception handling and input and output along with their respective transformers. Although a set of axioms are provided which show a consistent behaviour but nothing in the lines of completeness exists.

1.4 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.