

Embedding Programming Languages: PROLOG in HASKELL

by

Mehul Chandrakant Solanki

Bachelor of Engineering in Computer Science and Engineering Mumbai University 2012

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

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

November 2015

© Mehul Chandrakant Solanki, 2015

Abstract

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

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

0.1 Thesis Statement

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

We explore embedding domain specific languages in HASKELL

TABLE OF CONTENTS

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

5.4	Functional Logic Programming Languages	27
5.5	From chap 9	28
5.6	Theory	29
5.7	Implementations	29
5.8	Miscellaneous / Possibly Related Content	30
6	Related Concepts	31
7	Prolog in ---- other languages	33
7.1	Theory	33
7.2	Implementations	34
7.3	Important People	34
7.4	Miscellaneous / Possibly Related Content	34
8	Prolog in Haskell	35
8.1	Theory	35
8.2	Implementations	36
8.3	Important People	37
8.4	Miscellaneous / Possibly Related Content	37
9	Quasiquotation	39
9.1	Theory	39
9.2	Implementations	39
9.3	Miscellaneous / Possibly Related Content	40
9.4	What is Quasiquotation ?	40
9.5	Quasiquotaion in HASKELL	40
10	Meta Syntactic Variables	42
11	Haskell or Why Haskell ?	44
12	Prolog or Why Prolog ?	47
13	Prototype 1	52
13.1	About this chapter	52
13.2	How Prolog works ?	52
13.3	What we do in this Prototype	54
13.4	Creating a data type	55
13.5	Working with the language	57
13.6	Black box	58
13.7	Something about unification-fd and Monadic Unification	58
14	Prototype 2.1	65
14.1	About this chapter	65
14.2	How prolog-0.2.0.1 works	65
14.3	What we do in this prototype?	67

14.4	Current implementation (prolog-0.2.0.1)	68
14.5	Modifications	69
14.6	Results	74
15	Prototype 3	75
15.1	Unification	75
15.2	Resolution	75
15.3	Search strategies	76
15.4	Stack Engine	76
15.5	Pure Engine	77
15.6	Andorra Engine	80
15.7	Current Unification	82
15.8	Syntax Modification	85
15.9	Monadic Unification	94
16	Prototype 4	96
16.1	I/O is pure	96
16.2	Dr. Casperson Pure IO	105
16.3	Mehul Pure IO	105
17	Work Completed	108
17.1	What we are doing	108
17.2	Unifiable Data Structures	108
17.3	Why Fix is necessary?	108
17.4	Dr. Casperson's Explanation	110
17.5	The other fix	110
17.6	The Fix we use	111
17.7	Opening up or Extending language Explanation using Box Analogy	113
18	Results	116
18.1	Types	116
18.2	Lazy Evaluation	117
18.3	Opening up the Language	117
18.4	Quasi Quotation	117
18.5	Template Haskell	117
18.6	Higher Order Functions	117
18.7	I/O	118
18.8	Mutability	118
18.9	Unification	118
18.10	Monads	118
19	Future Scope	119
20	Conclusion / Expected Outcomes	121

21 Editing to do	122
21.1 Editing suggestions from David	124
Bibliography	128

List of Tables

List of Figures

13.1	Trace for append [124]	54
13.2	A sample Minted figure	64
14.1	A language-processing system [1]	66
14.2	Phases of Compiler [1]	67
17.1	Functor Hierarchy [148]	109
21.1	A sample Minted figure	125

Chapter 1

Introduction

1.1 Beginnings

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

Over the last decade the declarative style of programming has gained popularity. The methodologies that have stood out are the Functional and Logical Approaches. The former is based on Functions and Lambda Calculus, while the latter is based on Horn Clause Logic. Each of them has its own advantages and awns. How does one choose which approach to adopt? Perhaps one does not need to choose! This document looks at the attempts, improvements and future possibilities of uniting HASKELL, a Purely Functional Programming Language and PROLOG, a Logical Programming Language so that one is not

19 forced to choose.

20 **1.2 Thesis Statement**

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

25 **1.3 Problem Statement**

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

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

43 that should not die away.

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

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

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

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

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

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

70 problem according to a paradigm.

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

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

75 This can be achieved in two ways,

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

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

89 1.4 Thesis Organization

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

⁹⁵ Logic Languages). Some of the indirectly related content [Chapter 6](#) and finishing off with
⁹⁶ the [Chapter 7](#), the expected outcomes.

97 Chapter 2

98 Background

99 Programming Languages fall into different categories also known as "paradigms". They
100 exhibit different characteristics according to the paradigm they fall into. It has been argued
101 [72] that rather than classifying a language into a particular paradigm, it is more accurate
102 that a language exhibits a set of characteristics from a number of paradigms. Either way,
103 the broader the scope of a language the more the expressibility or use it has.

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

108 Functional Programming, [59] gets its name as the fundamental concept is to apply
109 mathematical functions to arguments to get results. A program itself consists of functions
110 and functions only which when applied to arguments produce results without changing the
111 state that is values on variables and so on. Higher order functions allow functions to be
112 passed as arguments to other functions. The roots lie in λ -calculus [166], a formal system
113 in mathematical logic and computer science for expressing computation based on function
114 abstraction and application using variable binding and substitution. It can be thought as the
115 smallest programming language [107], a single rule and a single function definition scheme.

116 In particular there are typed and untyped λ calculi. In the untyped λ calculus functions have
117 no predetermined type whereas typed lambda calculus puts restriction on what sort(type)
118 of data can a function work with. SCHEME is based on the untyped variant while ML
119 and HASKELL are based on typed λ calculus. Most typed λ calculus languages are based
120 on Hindley-Milner or Damas-Milner or Damas- Hindley-Milner [164] type system. The
121 ability of the type system to give the most general type of a program without any help
122 (annotation). The algorithm [22] works by initially assigning undefined types to all inputs,
123 next check the body of the function for operations that impose type constraints and go on
124 mapping the types of each of the variables, lastly unifying all of the constraints giving the
125 type of the result.

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

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

133 PROLOG [154] being one of the most dominant Logic Programming Languages has
134 spawned a number of distributions and is present from academia to industry.

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

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

146 The selection of languages is not uncommon and this not only the case with HASKELL,
147 PROLOG seems to be the all time favourite for "let's implement PROLOG in the language
148 X for proving it's power and expressibility". The PROLOG language has been partially
149 implemented [34] in other languages like SCHEME [116], LISP [70, 105, 106], JAVA [154,
150 62], JAVASCRIPT [63] and the list [99] goes on and on.

151 The technique of embedding is a shallow one, it is as if the embedded language floats
152 over the host. Over time there has been an approach that branches out, which is Paradigm
153 Integration. A lot of work has been done on Unifying the Theories of Programming [36,
154 14, 100, 177, 56, 46]. All sorts of hybrid languages which have characteristics from more
155 than one paradigm are coming into the mainstream.

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

169 Another important concept is meta programming [168], which involves writing com-

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

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

```

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

```

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

- 193 1. Embedding, where PROLOG is merely translated to the host language HASKELL or
194 a Foreign Function Interface.
- 195 2. Paradigm Integration, developing a hybrid programming language that is a Func-
196 tional Logic Programming Language with a set of characteristics derived from both
197 the participating languages.
- 198 The approaches listed above are next in line for discussions.

199 Chapter 3

200 Accomplished Work

201 3.1 Current Work

202 There have been several attempts at embedding PROLOG into HASKELL which are dis-
203 cussed below along with the shortcomings.

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

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

214 3. In the case of libraries, a few exist, most are old and are not currently maintained or
215 updated. Many provide only a shell through which one has to do all the work, which

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

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

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

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

231 3.2 Contributions

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

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

244 3.3 Improved Contributions

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

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

254 2. ([Prototype 2](#)) does the what a PROLOG query resolver would do given a query and a
255 knowledge base. The mechanism for the same is adopted from [109]. The embedded
256 language is modified as per the procedure in ([Prototype 1](#)) and the monadic unifica-
257 tion part is plugged into the existing architecture to demonstrate that it is independent
258 of the other components. Lastly the result is converted into the original language via
259 a translate function.

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

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

265 produced only when the actions are executed.

266 **3.4 Thesis Contributions**

267 1. Prototype 1 does flattening language opening up the language (binding monad) adding
268 custom variables monadic unification (stuff happens in a bubble) $\text{rec type} \rightarrow \text{non rec}$
269 $\text{type} \rightarrow \text{fix non rec type isomorphically} == \text{rec type}$

270 You can make an Flatterm int

271 but you cannot make term int

272 adding quantifiers

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

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

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

280 **3.5 What work was done in terms of points**

281 1. Literature review on eDSL's.

282 2. Short survey on multi paradigm declarative languages.

283 3. Accumulated and evaluated PROLOG in HASKELL.

284 4. Defined a procedure to open up a language starting from a generic recursive abstract
285 syntax.

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

Chapter 4

Embedding a Programming Language into another Programming Language

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

4.1 The Informal Content from Blogs, Articles and Internet Discussions

Before moving on to the formal content such as publications, modules and libraries let's take a look at some of the unofficially published content. This subsection takes a look at the information, thoughts and discussions that are currently taking place from time to time on the internet. A lot of interesting content is generated which has often led to some formal

310 content.

311 A lot has been talked about embedding languages and also the techniques and methods
312 to do so. It might not seem such a hot topic as such but it has always been a part of any pro-
313 gramming language to work and integrate their code with other programming languages.
314 One of the top discussions are in, Lambda the Ultimate, The Programming Languages
315 Weblog [73], which lists a number of PROLOG implementations in a variety of languages
316 like LISP, SCHEME, SCALA, JAVA, JAVASCRIPT, RACKET [116] and so on. Moreover the
317 discussion focusses on a lot of critical points that should be considered in a translation of
318 PROLOG to the host language regarding types and modules among others.

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

326 Adding fuel to fire, is the question on PROLOG's existence and survival [139, 89, 140,
327 117] since its use in industry is far scarce than the leading languages of other paradigms.
328 The purely declarative nature lacks basic requirements such as support for modules. And
329 then there is the ongoing comparison between the siblings [179] of the same family, the
330 family of Declarative Languages. Not to forget HASKELL also has some tricks [143] up its
331 sleeve which enables encoding of search problems.

332 4.2 Related Books

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

335 few constructs to the language to bring together both styles of Declarative Programming
336 and capture the essence of PROLOG. Moreover, HASKELL also claims for it to be suitable
337 for basic Logic Programming naturally using the List Monad [144]. A general out look
338 towards implementing PROLOG has also been discussed by [71] to push the ideas forward.

339 4.3 Related Papers

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

- 346 • Papers from Mike Spivey and Silvija Seres

347 The work presented in the series [121, 113, 114, 120, 111] attempts to encapsulate
348 various aspects of an embedding of PROLOG in HASKELL. Being the very first doc-
349 umented formal attempt, the work is influenced by similar embeddings of PROLOG
350 in other languages like SCHEME and LISP. Although the host language has distinct
351 characteristics such as lazy evaluation and strong type system the proposed scheme
352 tends to be general as the aim here is to achieve PROLOG like working not a multi
353 paradigm declarative language. PROLOG predicates are translated to HASKELL func-
354 tions which produce a stream of results lazily depicting depth first search with sup-
355 port for different strategies and practical operators such as *cut* and *fail* with higher
356 order functions. The papers provide a minimalistic extension to HASKELL with only
357 four new constructs. Though no implementation exists, the synthesis and transforma-
358 tion techniques for functional programs have been *logicalised* and applied to PRO-
359 LOG programs. Another related work [122] looks through conventional data types so

360 as to adapt to the problems at hand so as to accommodate and jump between search
361 strategies.

362 • Other works related or based on the above

363 Continuing from above, [21] taps into the advantages of the host language to em-
364 bed a typed functional logic programming language. This results in typed logical
365 predicates and a backtracking monad with support for various data types and search
366 strategies. Though not very efficient nor practical the method aims at a more ele-
367 gant translation of programs from one language to the other. While other papers [39]
368 attempt at exercising `HASKELL` features without adding anything new rather doing
369 something new with what is available. Specifically speaking, using `HASKELL` type
370 classes to express general structure of a problem while the solutions are instances.
371 [55] replicates `PROLOG`'s control operations in `HASKELL` suggesting the use of the
372 `HASKELL State Monad` to capture and maintain a global state. The main contribu-
373 tions are a Backtracking Monad Transformer that can enrich any monad with back-
374 tracking abilities and a monadic encapsulation to turn a `PROLOG` predicate into a
375 `HASKELL` function.

376 4.4 Related Libraries in Haskell

377 • Prolog Libraries

378 To replicate Prolog like capabilities Haskell seems to be already in the race with a
379 host of related libraries. First we begin with the libraries about Prolog itself, a few
380 exist [126] being a preliminary or "mini Prolog" as such with not much in it to be able
381 to be useful, [127] is all powerful but is an Foreign Function Interface so it is "Prolog
382 in Haskell" but we need Prolog for it, [109] which is the only implementation that
383 comes the closest to something like an actual practical Prolog. But all they give is a
384 small interpreter, none or a few practical features, incomplete support for lists, minor

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

387 • Logic Libraries

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

395 • Unification Libraries

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

399 • Backtracking

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

404 **4.5 From chap 7**

405 Embedding a language into another language has been explored with a variety of languages.
406 Attempts have been made to build Domain Specific Languages from the host languages
407 [58], Foreign Function Interfaces [10]

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

- 411 1. Syntax
- 412 2. Semantics
- 413 3. Standard Library
- 414 4. Runtime System
- 415 5. Parsers
- 416 6. Code Generators
- 417 7. Interpreters
- 418 8. Debuggers

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

428 The snippet of HASKELL code below describes PROLOG entities,

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

```

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

```

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

431 As discussed in the

432 **4.6 Theory**

433 1. Papers

- 434 (a) Embedding an interpreted language using higher-order functions, [102]
- 435 (b) Building domain-specific embedded languages, [58]
- 436 (c) Embedded interpreters, [11]
- 437 (d) Cayenne – a Language With Dependent Types, [6]
- 438 (e) Foreign interface for PLT Scheme, [10]
- 439 (f) Dot-Scheme: A PLT Scheme FFI for the .NET framework, [96]
- 440 (g) Application-specific foreign-interface generation, [103]
- 441 (h) Embedding S in other languages and environments, [76]

442 2. Books

- 443 (a) ?????????

444 3. Articles / Blogs / Discussions

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

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

447 (c) Linguistic Abstraction, [92]

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

449 4. Websites

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

451 **4.7 Implementations**

452 1. Lots of them I guess

453 **4.8 Important People**

454 1. ????

455 **4.9 Miscellaneous / Possibly Related Content**

456 1. ????

457 Chapter 5

458 Multi Paradigm Languages (Functional 459 Logic Languages)

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

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

475 paradigms.”

476 **5.1 The Informal Content from Blogs, Articles and Inter-** 477 **net Discussions**

478 • Multi Paradigm Languages

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

488 • Functional Logic Programming Languages

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

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

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 [56] which points out that among the large number of programming paradigms and/or theories the unification theory serves as a complementary rather than a replacement to relate the universe. As as always since we are talking about HASKELL we have to include monads and unifying theories using monads [46].

• Functional Logic Programming Languages

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

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

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

532 **5.3 Some Multi Paradigm Languages**

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

545 **5.4 Functional Logic Programming Languages**

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

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

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

Finally talking about CURRY, one of the most popular HASKELL based multi paradigm languages with support for deterministic and non-deterministic computations. Contributing to the same there have been some predecessors [134, 29].

5.5 From chap 9

Unifying / Marrying / Merging / Combining Programming Paradigms / Theories

574 **5.6 Theory**

575 • Papers

- 576 1. Unifying Theories of Programming with Monads, [46]
- 577 2. Symposium on Unifying Theories of Programming, 2006, [36].
- 578 3. Symposium on Unifying Theories of Programming, 2008, [14].
- 579 4. Symposium on Unifying Theories of Programming, 2010, [100].
- 580 5. Symposium on Unifying Theories of Programming, 2012, [177].

581 • Books

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

583 • Articles / Blogs / Discussions

- 584 1. ???

585 • Websites

- 586 1. ???

587 **5.7 Implementations**

- 588 1. Scala
- 589 2. Virgil
- 590 3. CLOS, Common Lisp Object System
- 591 4. Visual Prolog
- 592 5. ????

593 **5.8 Miscellaneous / Possibly Related Content**

594 1. ???

Chapter 6

Related Concepts

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

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

Lastly some details on the working of functional logic programming languages, residuation and narrowing [51, 157]. Residuation involves delaying of functions calls until they are deterministic, that is, deterministic reduction of functions with partial data. This princi-

614 ple is used in languages like ESCHER [79], LIFE [136], NUE-PROLOG [78] and Oz [23].
615 Narrowing on the other hand is a mixture of reduction in functional languages and unifi-
616 cation in logic languages. In narrowing, a variable is bound a value within the specified
617 constraints and try to find a solution, values are generated while searching rather than just
618 for testing. The languages based on this approach are ALF [134], BABEL [53], LPG [12]
619 and CURRY [138].

620 F-Algebras

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

623 An F-algebra consists of:

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

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

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

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

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

633 **Chapter 7**

634 **Prolog in _____ other languages**

635 Prolog in _____

636 **7.1 Theory**

637 • Papers

638 1. QLog, [70]

639 2. LogLisp Motivation, design, and implementation, [105]

640 • Books

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

642 2. LOGLISP: an alternative to PROLOG, [106]

643 • Articles / Blogs / Discussions

644 1. Hello

645 • Websites

646 1. Hello

647 **7.2 Implementations**

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

655 **7.3 Important People**

- 656 1. ???

657 **7.4 Miscellaneous / Possibly Related Content**

- 658 1. ???

659 **Chapter 8**

660 **Prolog in Haskell**

661 Prolog in Haskell

662 **8.1 Theory**

663 • Papers

664 1. Embedding Prolog in Haskell / Functional Reading of Logic Programs, [121]

665 2. Algebra of Logic Programming, [113]

666 3. The Algebra of Logic Programming, [111]

667 4. Optimisation Problems in Logic Programming : An Algebraic Approach, [112]

668 5. Higher Order Transformation of Logic Programs, [114]

669 6. The Algebra of Searching, [120]

670 7. FUNCTIONAL PEARL Combinators for breadth-first search, [122]

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

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

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

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

677 • Books

678 1. The Reasoned Schemer, Daniel P. Friedman, William E. Byrd, Oleg Kiselyov,
679 [27]

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

682 • Articles / Blogs / Discussions

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

684 2. Takashi's Workplace (Implementation), [178]

685 3. Haskell vs. Prolog Comparison, [123]

686 • Websites

687 1. Logic Programming in Haskell, [143]

688 **8.2 Implementations**

689 1. A Prolog in Haskell, Takashi's Workplace, [178]

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

691 3. Nano Prolog, [126]

692 4. Prolog, [109]

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

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

697 **8.3 Important People**

- 698 1. Mike Spivey
- 699 2. Silvija Seres

700 **8.4 Miscellaneous / Possibly Related Content**

- 701 1. Unification Libraries
 - 702 (a) unification-fd, [130]
 - 703 (b) cmu, [101]
- 704 2. Logic Libraries
 - 705 (a) logicct, [25], [26]
 - 706 (b) logic-classes, [?]
 - 707 (c) proplogic, [44]
 - 708 (d) cflp, [40]
 - 709 (e) logic-grows-on-trees, [24]
- 710 3. Concatenative Programming
 - 711 (a) peg, [31]
- 712 4. Constraint Programming and Constraint Handling Rules

- 713 (a) monadiccp, [104]
- 714 (b) monadicccp-gecode, [133]
- 715 (c) csp, [7]
- 716 (d) liquid fix point, [110]

717 **Chapter 9**

718 **Quasiquotation**

719 **9.1 Theory**

720 1. Papers

721 (a)

722 2. Books

723 (a)

724 3. Articles / Blogs / Discussions

725 (a)

726 4. Websites

727 (a) Quasiquotation Wikipedia, [159]

728 (b) Quasiquotation in Haskell, [145]

729 **9.2 Implementations**

730 1.

731 **9.3 Miscellaneous / Possibly Related Content**

732 1.

733 **9.4 What is Quasiquotation ?**

734 1. [54]

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

738 2. [32]

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

744 3. Quasiquotation Wikipedia, [159]

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

748 [176] The usemention distinction is a foundational concept of analytic philosophy,[1]
749 according to which it is necessary to make a distinction between using a word (or
750 phrase) and mentioning it

751 **9.5 Quasiquotaion in HASKELL**

752 [145, 80]

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

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

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

761 Chapter 10

762 Meta Syntactic Variables

763 Some sources for the topic

764 [174] A metasyntactic variable is a placeholder name used in computer science, a word
765 without meaning intended to be substituted by some objects pertaining to the context where
766 it is used. The word foo as used in IETF Requests for Comments is a good example. By
767 mathematical analogy, a metasyntactic variable is a word that is a variable for other words,
768 just as in algebra letters are used as variables for numbers. Any symbol or word which does
769 not violate the syntactic rules of the language can be used as a metasyntactic variable.

770 [17] A name used in examples and understood to stand for whatever thing is under
771 discussion, or any random member of a class of things under discussion. The word foo is
772 the canonical example. To avoid confusion, hackers never (well, hardly ever) use foo or
773 other words like it as permanent names for anything. In filenames, a common convention
774 is that any filename beginning with a metasyntactic-variable name is a scratch file that may
775 be deleted at any time.

776 Metasyntactic variables are so called because they are variables in the metalanguage
777 used to talk about programs etc; they are variables whose values are often variables (as in
778 usages like the value of $f(\text{foo}, \text{bar})$ is the sum of foo and bar). However, it has been plausibly
779 suggested that the real reason for the term metasyntactic variable is that it sounds good. To

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

783 [60] In programming, a metasyntactic (which derives from meta and syntax) variable is
784 a variable (a changeable value) that is used to temporarily represent a function . Examples
785 of metasyntactic variables include (but are by no means limited to) ack, bar , baz, blarg,
786 wibble, foo , fum, and qux. Metasyntactic variables are sometimes used in developing a
787 conceptual version of a program or examples of programming code written for illustrative
788 purposes.

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

791 [16]

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

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

Chapter 11

Haskell or Why Haskell ?

In this chapter we discuss the properties of HASKELL

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

1. Why a Functional Language?

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

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

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

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

(a) Monads

Control flow defines the order/ manner of execution of statements in a program[172]. The specification is set by the programming language. Generally, in the case of imperative languages the control flow is sequential while for a functional language is recursion [135]. For example, JAVA has a top down sequential execution approach. The declarative style consists of defining components of programs i.e. computations not a control flow[173].

This is where HASKELL shines by providing something called a *monad*. Functional Programming Languages define computations which then need to be ordered in some way to form a combination[146]. A monad gives a bubble within the language to allow modification of control flow without affecting the rest of the universe. This is especially useful while handling side effects.

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

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

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

847 (b) Lazy Evaluation

848 Another property of HASKELL is laziness or lazy evaluation which means that
849 nothing is evaluated until it is necessary. This results in the ability to define
850 infinite data structures because at execution only a fragment is used [151].

Chapter 12

Prolog or Why Prolog ?

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

1. Why a Logic Programming Language ?

2. PROLOG as a logic programming language.

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

3. Why embed PROLOG ?

(a) Existing Implementations

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

(b) Simple Syntax [154]

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

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

872 Numbers can be floats or integers. Many Prolog implementations also provide
873 unbounded integers and rational numbers.

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

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

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

887 [94] In Prolog all data objects are called terms Atomic terms
888 Come in two forms, atoms and integers. Atoms (this is a misnomer as in logic
889 predicates are called atoms and atoms are called constants. However, we'll
890 stick to the Prolog convention.) Strings of alphanumerics and `_`, starting with a
891 lower case alphabetic. Strings enclosed in 'single quotes' Integers are numeric
892 Example

```
1 geoff
2 'the cat and the rat'
3 'ABCD'
4 123
```


893
894
895

Function terms

Functions have the form $f(\text{term}_1, \text{term}_2)$ Functor starts with a lower case alphabetic. Example

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

896
897
898
899
900
901

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

```
1 Who
2 What
3 _special
4 _
```

902
903

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

904

(c) Simple Semantics

905
906
907
908
909
910
911

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

912
913
914
915

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

they are written.

[98]

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head, they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraints can be found, or the matching or the guard fails, then the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule there are multiple constraints in the head, the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint. This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

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

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

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

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

943 C. simpagation The simpagation rule removes the constraints in its head
 944 after the and then calls its body. It is an optimization of simplification
 945 rules of the form: $[constraints_1, constraints_2 \Leftarrow constraints_1,$
 946 $body]$ Namely, in the simpagation form: $[constraints_1 \setminus constraints_2$
 947 $\Leftarrow body]$ The constraints_1 constraints are not called in the body.

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

950 iii. Pragmas The semantics of the pragmas are:

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

953 (d) Universal Horn Clauses

954 (e) Unification

955 (f) Definite Clause Grammar

Chapter 13

Prototype 1

13.1 About this chapter

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

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

13.2 How Prolog works ?

Looking at how PROLOG works [125].

Most PROLOG distributions have three types of terms:

1. Constants.

2. Variables.

3. Complex terms.

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

972 The possibilities could be,

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

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

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

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

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

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

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

982 (b) all their corresponding arguments unify, and

983 (c) the variable instantiations are compatible.

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

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

986 For example, consider the append function

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

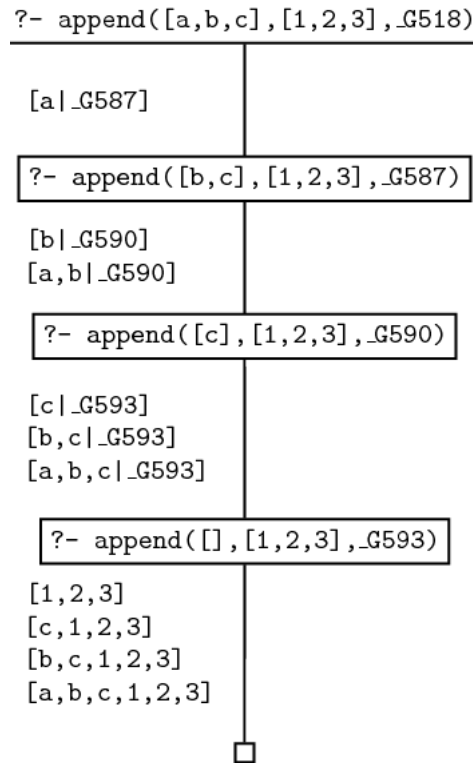


Figure 13.1: Trace for append [124]

987 13.3 What we do in this Prototype

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

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

992 The language defined is recursive in nature.

993 We convert it into a non recursive data type.

994 Basically we do Unification monadically.

995 **13.4 Creating a data type**

996 A type system consists of a set of rules to define a "type" to different constructs in a pro-
997 gramming language such as variables, functions and so on. A static type system requires
998 types to be attached to the programming constructs before hand which results in finding
999 errors at compile time and thus increase the reliability of the program. The other end is the
1000 dynamic type system which passes through code which would not have worked in former
1001 environment, it comes of as less rigid.

1002 The advantages of static typing [82]

- 1003 1. Earlier detection of errors
- 1004 2. Better documentation in terms of type signatures
- 1005 3. More opportunities for compiler optimizations
- 1006 4. Increased run-time efficiency
- 1007 5. Better developer tools

1008 For dynamic typing

- 1009 1. Less rigid
- 1010 2. Ideal for prototyping / unknown / changing requirements or unpredictable behaviour
- 1011 3. Re-usability

1012 **Transitional paragraph** An ideal case would would be something that is dont
1013 know what to write

1014 To start with, replicating the single type "term" in PROLOG one must consider the dis-
 1015 tinct constructs it can be associated to such as complex structures (for example predicates,
 1016 clauses etc.), don't cares, cuts, variables and so on.

1017 Consider the language below,

```

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

```

1018 Even though *Term* has a number of constructors the resulting construct has a single
 1019 type. Hence, a function would still be untyped / singly typed,

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

1020 The above data type is recursive as seen in the constructor,

```
Struct Atom [Term]
```

1021 One of the issues with the above is that it is not possible to distinguish the structure of
 1022 the data from the data type itself [115]. Consider the following, a reduced version of the
 1023 above data type,


```

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

```

1024 Also one cannot create Quantifiers plus logic

1025 To split a data type into two levels, a single recursive data type is replaced by two related
 1026 data types. Consider the following,

```

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

```

1027 One result of the approach is that the non-recursive type *FlatTerm* is modular and
 1028 generic as the structure "FlatTerm" is separate from it's type which is "a". Simply speaking
 1029 we can have something like

```
FlatTerm Bool
```

1030 and a generic fuinction like,

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

1031 13.5 Working with the language

1032 Creating instances,

```

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

```

```

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

```

1033 After flattening do fixing,

1034 Opening up the language somehow so as to accommodate your own variables.

1035 13.6 Black box

1036 13.7 Something about unification-fd and Monadic Unifi- 1037 cation

1038 Library [130]

1039 Tutorial 1 [131]

1040 Tutorial 2 [132]

1041 1. What library provides ?

1042 This module provides first-order structural unification over general structure types.

1043 It also provides the standard suite of functions accompanying unification (applying
1044 bindings, getting free variables, etc.).

1045 The implementation makes use of numerous optimization techniques. First, we use
1046 path compression everywhere (for weighted path compression see `Control.Unification.Ranked`).
1047 Second, we replace the occurs-check with visited-sets. Third, we use a technique for
1048 aggressive opportunistic observable sharing; that is, we track as much sharing as
1049 possible in the bindings (without introducing new variables), so that we can compare
1050 bound variables directly and therefore eliminate redundant unifications.

1051 2. Unifiable stuff

1052 The basic class for generating, reading, and writing to bindings stored in a monad.
1053 These three functionalities could be split apart, but are combined in order to simplify
1054 contexts. Also, because most functions reading bindings will also perform path com-
1055 pression, there's no way to distinguish "true" mutation from mere path compression.
1056 The superclass constraints are there to simplify contexts, since we make the same
1057 assumptions everywhere we use `BindingMonad`.

1058 In order to use our `T` data type with the rest of the API, we'll need to give a Unifi-
1059 able instance for it. Before we do that we'll have to give `Functor`, `Foldable`, and
1060 `Traversable` instances. These are straightforward and can be automatically derived
1061 with the appropriate language pragmas.

1062 The `Unifiable` class gives one step of the unification process. Just as we only need
1063 to specify one level of the ADT (i.e., `T`) and then we can use the library's `UTerm` to
1064 generate the recursive ADT, so too we only need to specify one level of the unification
1065 (i.e., `zipMatch`) and then we can use the library's operators to perform the recursive
1066 unification, subsumption, etc.

1067 The `zipMatch` function takes two arguments of type `t a`. The abstract `t` will be our
1068 concrete `T` type. The abstract `a` is polymorphic, which ensures that we can't mess

1069 around with more than one level of the term at once. If we abandon that guarantee,
1070 then you can think of it as if a is $\text{UTerm } T \ v$. Thus, $t \ a$ means $T \ (\text{UTerm } T \ v)$; and
1071 $T \ (\text{UTerm } T \ v)$ is essentially the type $\text{UTerm } T \ v$ with the added guarantee that the
1072 values aren't in fact variables. Thus, the arguments to `zipMatch` are non-variable
1073 terms.

1074 The `zipMatch` method has the rather complicated return type: `Maybe (t (Either a`
1075 `(a,a)))`. Let's unpack this a bit by thinking about how unification works. When we
1076 try to unify two terms, first we look at their head constructors. If the constructors
1077 are different, then the terms aren't unifiable, so we return `Nothing` to indicate that
1078 unification has failed. Otherwise, the constructors match, so we have to recursively
1079 unify their subterms. Since the T structures of the two terms match, we can return
1080 `Just t0` where $t0$ has the same T structure as both input terms. Where we still have to
1081 recursively unify subterms, we fill $t0$ with `Right(l,r)` values where l is a subterm of the
1082 left argument to `zipMatch` and r is the corresponding subterm of the right argument.
1083 Thus, `zipMatch` is a generalized zipping function for combining the shared structure
1084 and pairing up substructures. And now, the implementation:

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

1085 Where `list-extras>Data.List.Extras.Pair.pairWith` is a version of `zip` which returns
1086 `Nothing` if the lists have different lengths. So, if the names m and n match, and
1087 if the two arguments have the same number of subterms, then we pair those subterms
1088 off in order; otherwise, either the names or the lengths don't match, so we return
1089 `Nothing`.

1090 3. UTerm stuff

1091 The type of terms generated by structures `t` over variables `v`. The structure type should
1092 implement `Unifiable` and the variable type should implement `Variable`.

1093 The `Show` instance doesn't show the constructors, in order to improve legibility for
1094 large terms.

1095 All the category theoretic instances (`Functor`, `Foldable`, `Traversable`,...) are provided
1096 because they are often useful; however, beware that since the implementations must
1097 be pure, they cannot read variables bound in the current context and therefore can
1098 create incoherent results. Therefore, you should apply the current bindings before
1099 using any of the functions provided by those classes.

1100 4. `STVar` stuff

1101 This module defines an implementation of unification variables using the `ST` monad.

1102 5. `IntVar` stuff

1103 This module defines a state monad for functional pointers represented by integers as
1104 keys into an `IntMap`. This technique was independently discovered by Dijkstra et al.
1105 This module extends the approach by using a state monad transformer, which can
1106 be made into a backtracking state monad by setting the underlying monad to some
1107 `MonadLogic` (part of the `logict` library, described by Kiselyov et al.).

1108 Atze Dijkstra, Arie Middelkoop, S. Doaitse Swierstra (2008) Efficient Functional
1109 Unification and Substitution, Technical Report UU-CS-2008-027, Utrecht Univer-
1110 sity.

1111 Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry (2005) Back-
1112 tracking, Interleaving, and Terminating Monad Transformers, ICFP

1113 A "mutable" unification variable implemented by an integer. This provides an en-
1114 tirely pure alternative to truly mutable alternatives (like `STVar`), which can make
1115 backtracking easier.

1116 N.B., because this implementation is pure, we can use it for both ranked and unranked
1117 monads.

1118 6. Binding Monad Stuff

1119 A monad for handling STVar bindings.

1120 Run the ST ranked binding monad. N.B., because STVar are rank-2 quantified, this
1121 guarantees that the return value has no such references. However, in order to remove
1122 the references from terms, you'll need to explicitly apply the bindings and ground
1123 the term.

1124 7. U.unify stuff

1125 Unify two terms, or throw an error with an explanation of why unification failed.
1126 Since bindings are stored in the monad, the two input terms and the output term
1127 are all equivalent if unification succeeds. However, the returned value makes use of
1128 aggressive opportunistic observable sharing, so it will be more efficient to use it in
1129 future calculations than either argument.

1130 8. U.unifyOccurs

1131 A variant of unify which uses occursIn instead of visited-sets. This should only
1132 be used when eager throwing of occursFailure errors is absolutely essential (or for
1133 testing the correctness of unify). Performing the occurs-check is expensive. Not only
1134 is it slow, it's asymptotically slow since it can cause the same subterm to be traversed
1135 multiple times.


```

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

```

Figure 13.2: A sample Minted figure

Chapter 14

Prototype 2.1

14.1 About this chapter

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

14.2 How prolog-0.2.0.1 works

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

The original abstract syntax used by the library,

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

```

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

```

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

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

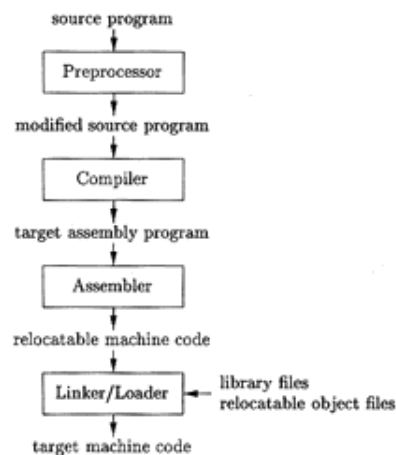


Figure 1.5: A language-processing system

Figure 14.1: A language-processing system [1]

specifically speaking, parts of a compiler 14.2,

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

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

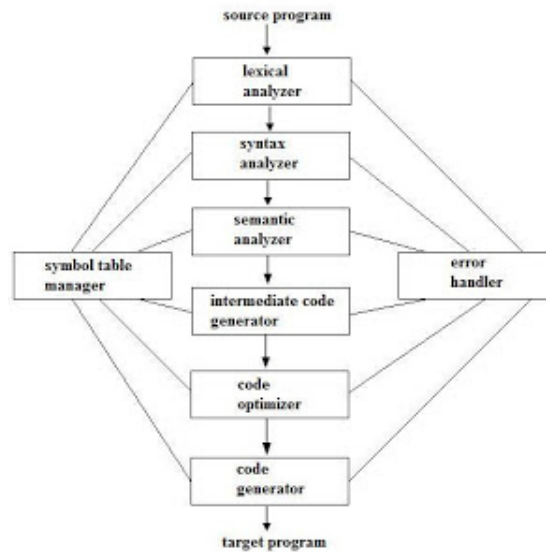


Fig 1.5 Phases of a compiler

Figure 14.2: Phases of Compiler [1]

1157 3. Parser.

1158 4. Interpreter.

1159 5. Unifier.

1160 6. REPL.

1161 To prove the modularity of the approach for language modification and monadic unifi-
 1162 cation only the abstract syntax and unifier will be customized.

1163 14.3 What we do in this prototype?

1164 In the first prototype we just did unification of two terms not query resolution.

1165 We do complete PROLOG query resolution like stuff.

1166 13 provides a generic procedure / methodology to convert a language into monadic
 1167 unifiable form

1168 14.4 Current implementation (prolog-0.2.0.1)

1169 The current unification uses basic pattern matching to unify the terms

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

```

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

```

1170 14.5 Modifications

1171 The resulting language is not far from what we did in 13 apart from the fact that the *Term*
1172 expressions are encapsulated to form *Clauses* which in turn form a *Program*.

1173 Moreover, the required instances make the language compatible with the unification
1174 procedure.

```

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

```

```

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

```

1175 Additionally helper functions for converting expressions between the two domains and
1176 translation to *UTerm*.

```

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

```

```

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

```

```

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

```



```

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

```

1177 Take original expressions flatten fix convert unify run it STBinding monad to extract
1178 substitutions.

```

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

```

```

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

```

1179 14.6 Results

1180 It works,

Chapter 15

Prototype 3

When two terms are to be unified we can use 13 ,
term1 and term2 are matched and an assignment is the result
now this may be a part of a query resolution procedure
to reach the point where two terms need to unified will happen through some sort of
search strategy
and our approach is independent of that, and this prototype is a proof of concept to
implementing query resolution using unification with variable search strategy

15.1 Unification

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

15.2 Resolution

this where the complete procedure takes place after the query is passed along with the
knowledge
the resolver searches to create and a list of goals and then tries to achieve each one.

1197 [37]

1198 [175]

1199 15.3 Search strategies

1200 The base implementation used for this prototype is [65] and below are the search strategies

1201 15.4 Stack Engine

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

```

31
32 alts      :: Database -> Int -> Term -> [Alt]
33 alts db n g = [ (tp,u) | (tm:-tp) <- renClauses db n g, u <- unify g tm ]
34
35 -- The use of a stack enables backtracking to be described explicitly,
36 -- in the following 'state-based' definition of prove:
37
38 prove     :: Database -> [Term] -> [st]
39 prove db gl = solve 1 nullst gl []
40 where
41   solve :: Int -> st -> [Term] -> Stack -> [st]
42   solve n s []      ow      = s : backtrack n ow
43   solve n s (g:gs) ow
44       | g==theCut = solve n s gs (cut ow)
45       | otherwise = choose n s gs (alts db n (app s g)) ow
46
47   choose :: Int -> st -> [Term] -> [Alt] -> Stack -> [st]
48   choose n s gs []      ow = backtrack n ow
49   choose n s gs ((tp,u):rs) ow = solve (n+1) (u@@s) (tp++gs) ((s,gs,rs):ow)
50
51   backtrack      :: Int -> Stack -> [st]
52   backtrack n []      = []
53   backtrack n ((s,gs,rs):ow) = choose (n-1) s gs rs ow
54
55
56 --- Special definitions for the cut predicate:
57
58 theCut      :: Term
59 theCut      = Struct "!" []
60
61 cut         :: Stack -> Stack
62 cut ss      = []
63
64 --- End of Engine.hs

```

1202 15.5 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 st
12 import Interact
13 import Data.List(nub)
14
15 version = "tree based"
16
17 --- Calculation of solutions:
18
19 -- Each node in a proof tree corresponds to:
20 -- either: a solution to the current goal, represented by Done s, where s
21 --          is the required substitution
22 -- or:      a choice between a number of trees ts, each corresponding to a
23 --          proof of a goal of the current goal, represented by Choice ts.
24 --          The proof tree corresponding to an unsolvable goal is Choice []
25
26 data Prooftree = Done st | Choice [Prooftree]
27
28 -- prooftree uses the rules of Prolog to construct a suitable proof tree for
29 --          a specified goal
30 prooftree :: Database -> Int -> st -> [Term] -> Prooftree
31 prooftree db = pt
32   where pt :: Int -> st -> [Term] -> Prooftree
33         pt n s [] = Done s
34         pt n s (g:gs) = Choice [ pt (n+1) (u@@s) (map (app u) (tp++gs))
35                                | (tm:-tp)<-renClauses db n g, u<-unify g tm ]
36   {--
37   pt 1 nullst [] = Done (nullst)
38
39   pt n s (g:gs)
40
41   renClauses :- Rename variables in a clause, the parameters are the database, an
42                 (head of list) resulting in a clause.
43
44   unify :- take the head of the list and and match with head of clause from renCla
45
46   app :- function for applying (st) to (Terms)
47   the new list is formed by replacing the cluase head with its body and applying t
48
49   so the new parameters for pt are
50

```

```

51  (n+1) (the old stitution + the new one from unify) (the list formed after applyi
52
53
54  Working of a small example
55
56  The database,
57  (foldl addClause emptyDb [((:-) (Struct "hello" []) []), ((:-) (Struct "hello" [
58  hello.
59  hello(world).
60  hello:-world.
61  hello(X_1).
62
63  The other parameters are 1 nullst(as mentioned in the prove function).
64
65  For the list of goals, [(Struct "hello" []), (Struct "hello" [(Struct "world" [
66
67  1. [Struct "hello" []] :: [Term]
68
69  * Rule 1 does not apply
70
71  * Rule 2 does apply,
72
73  (tm:- tp) <- renClauses db 1 (Struct "hello" [])
74
75  tm ==> "hello , hello(world) , hello , hello(X_1) , "
76  tp ==> "[] , [] , [world] , [] , "
77
78
79
80
81
82
83
84
85
86  --}
87
88
89
90  -- DFS Function
91  -- search performs a depth-first search of a proof tree, producing the list
92  -- of solution stitutions as they are encountered.
93  search :: ProofTree -> [st]
94  search (Done s)      = [s]
95  search (Choice pts) = [ s | pt <- pts, s <- search pt ]

```

```

96
97
98 prove    :: Database -> [Term] -> [st]
99 prove db = search . prooftree db 1 nullst
100
101 --- End of PureEngine.hs

```

1203 15.6 Andorra Engine

```

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

```



```

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

```

```

79   the list of resolvents. After each resolution step we update the lists.
80 -}
81
82 {- References
83
84   Seif Haridi & Per Brand, "Andorra Prolog, an integration of Prolog
85   and committed choice languages" in Proceedings of FGCS 1988, ICOT,
86   Tokyo, 1988.
87
88   Vitor Santos Costa, David H. D. Warren, and Rong Yang, "Two papers on
89   the Andorra-I engine and preprocessor", in Proceedings of the 8th
90   ICLP. MIT Press, 1991.
91
92   Steve Gregory and Rong Yang, "Parallel Constraint Solving in
93   Andorra-I", in Proceedings of FGCS'92. ICOT, Tokyo, 1992.
94
95   Sverker Janson and Seif Haridi, "Programming Paradigms of the Andorra
96   Kernel Language", in Proceedings of ILPS'91. MIT Press, 1991.
97
98   Torkel Franzen, Seif Haridi, and Sverker Janson, "An Overview of the
99   Andorra Kernel Language", In LNAI (LNCS) 596, Springer-Verlag, 1992.
100 -}

```

1204 15.7 Current Unification

```

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

```

```

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

```

```

63 infix 4 ->-
64
65 --- stitutions:
66
67 type st = Id -> Term
68
69 newtype stP = stP { unstP :: st }
70
71 -- instance Show stP where
72 --   show (i) = show £ Var i
73 -- stitutions are represented by functions mapping identifiers to terms.
74 --
75 -- app s      extends the stitution s to a function mapping terms to terms
76 {--
77 Looks like an apply function that applies a stitution function tho the variables
78 --}
79
80
81 -- nullst is the empty stitution which maps every identifier to the same identif
82
83
84
85 -- i ->- t    is the stitution which maps the identifier i to the term t, but oth
86
87
88 -- s1@@ s2    is the composition of stitutions s1 and s2
89 --           N.B. app is a monoid homomorphism from (st,nullst,(@@))
90 --           to (Term -> Term, id, (..)) in the sense that:
91 --           app (s1 @@ s2) = app s1 . app s2
92 --           s @@ nullst = s = nullst @@ s
93
94 app                :: st -> Term -> Term
95 app s (Var i)      = s i
96 app s (Struct a ts) = Struct a (Prelude.map (app s) ts)
97 {--
98 app (stFunction) (Struct "hello" [Var (0, "Var")])
99 hello(Var_2) :: Term
100
101 --}
102
103
104 nullst              :: st
105 nullst i            = Var i
106 {--
107 nullst (0, "Var")

```

```

108  Var :: Term
109  --}
110
111
112  --
113  (->-) :: Id -> Term -> st
114  (i ->- t) j | j==i      = t
115              | otherwise = Var j
116  {--
117  :t (->-) (1,"X") (Struct "hello" [])
118  (1,"X") ->- Struct "hello" [] :: (Int,[Char]) -> Term
119  --}
120
121
122  -- Function composition for applying two stitution functions.
123  (@@) :: st -> st -> st
124  s1 @@ s2      = app s1 . s2

```

1205 15.8 Syntax Modification

```

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

```

```

23
24 module CustomSyntax where
25
26 import Data.Generics (Data(..), Typeable(..))
27 import Data.List (intercalate)
28 import Data.Char (isLetter)
29
30 import Control.Monad.State.UnificationExtras
31 import Control.Unification as U
32
33
34 import Data.Functor.Fixedpoint as DFF
35
36
37 import Control.Unification.IntVar
38 import Control.Unification.STVar as ST
39
40 import Control.Unification.Ranked.IntVar
41 import Control.Unification.Ranked.STVar
42
43 import Control.Unification.Types as UT
44
45
46
47 import Data.Traversable as T
48 import Data.Functor
49 import Data.Foldable
50 import Control.Applicative
51
52
53 import Data.List.Extras.Pair
54 import Data.Map as Map
55 import Data.Set as S
56
57
58 import Control.Monad.Error
59 import Control.Monad.Trans.Except
60
61
62 import Prolog
63
64 data FTS a = forall a . FV Id | FS Atom [a] deriving (Eq, Show, Ord, Typeable)
65
66 newtype Prolog = P (Fix FTS) deriving (Eq, Show, Ord, Typeable)
67

```

```

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

```

```

113 termFlattener (Struct a xs)      =   DFF.Fix $ FS a (Prelude.map termFlattener xs)
114
115 unFlatten :: Fix FTS -> Term
116 unFlatten (DFF.Fix (FV v))      =   Var v
117 unFlatten (DFF.Fix (FS a xs))   =   Struct a (Prelude.map unFlatten xs)
118
119
120 variableExtractor :: Fix FTS -> [Fix FTS]
121 variableExtractor (Fix x) = case x of
122   (FS _ xs)   -> Prelude.concat $ Prelude.map variableExtractor xs
123   (FV v)      -> [Fix $ FV v]
124 -- _         -> []
125
126 variableIdExtractor :: Fix FTS -> [Id]
127 variableIdExtractor (Fix x) = case x of
128   (FS _ xs) -> Prelude.concat $ Prelude.map variableIdExtractor xs
129   (FV v)   -> [v]
130
131 {--
132  variableNameExtractor :: Fix FTS -> [VariableName]
133  variableNameExtractor (Fix x) = case x of
134    (FS _ xs) -> Prelude.concat $ Prelude.map variableNameExtractor xs
135    (FV v)    -> [v]
136    _         -> []
137  --}
138
139 variableSet :: [Fix FTS] -> S.Set (Fix FTS)
140 variableSet a = S.fromList a
141
142 variableNameSet :: [Id] -> S.Set (Id)
143 variableNameSet a = S.fromList a
144
145
146 varsToDictM :: (Ord a, Unifiable t) =>
147   S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
148 varsToDictM set = foldrM addElt Map.empty set where
149   addElt sv dict = do
150     iv <- freeVar
151     return $! Map.insert sv iv dict
152
153
154 uTermify
155   :: Map Id (ST.STVar s (FTS))
156   -> UTerm FTS (ST.STVar s (FTS))
157   -> UTerm FTS (ST.STVar s (FTS))

```



```

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

```

```

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

```

```

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

```

```

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

```

```

338     (x1,d1) <- lift . translateToUTerm $ t1a --error
339     (x2,d2) <- lift . translateToUTerm $ t2a
340     x3 <- U.unify x1 x2
341     return (x3, d1 'Map.union' d2)
342
343 goTest :: (Show b) => (forall s .
344   (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
345     (ST.STBinding s)
346     (UT.UTerm (FTS) (ST.STVar s (FTS)),
347       Map Id (ST.STVar s (FTS)))) -> String
348 goTest test = ST.runSTBinding $ do
349   answer <- runErrorT $ test
350   return $! case answer of
351     (Left x)  -> "error: " ++ show x
352     (Right y) -> "ok:    " ++ show y
353
354
355 -----
356 -----
357 -----GLUE-CODE-----
358 {--
359 monadicUnify :: Term -> Term -> ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
360   (ST.STBinding s)
361   (UT.UTerm (FTS) (ST.STVar s (FTS)),
362     Map Id (ST.STVar s (FTS)))
363 monadicUnify t1 t2 = do
364   let
365       t1f = termFlattener t1
366       t2f = termFlattener t2
367       (x1,d1) <- lift . translateToUTerm £ t1f
368       (x2,d2) <- lift . translateToUTerm £ t2f
369       x3 <- U.unify x1 x2
370       return (x3, d1 'Map.union' d2)
371
372 --}
373
374 -- type st = Id -> Term
375
376 -- Convert result from monadicUnify to [st]
377 {--
378 goMonadicTest :: (Show b) => (forall s .
379   (ErrorT (UT.UFailure (FTS) (ST.STVar s (FTS)))
380     (ST.STBinding s)
381     (UT.UTerm (FTS) (ST.STVar s (FTS)),
382       Map Id (ST.STVar s (FTS)))) -> [st]

```

```

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

```

1206 15.9 Monadic Unification

```

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

```

```

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

```

Chapter 16

Prototype 4

Our aim to embedd IO into the DSL

So something like a "data" declaration for IO operations

16.1 I/O is pure

[128]

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

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

How can this possibly be pure?

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

1221 never with effects.

1222 Lets look at a simplified form of I/O, where we only care about reading from stdin,
1223 writing to stdout and returning a value. We can model this with the IOAction data type.

1224 That is, an IOAction is one of the following three things:

- 1225 1. A container for a value of type a,
- 1226 2. A container holding a String to be printed to stdout, followed by another IOAction a,
1227 or
- 1228 3. A container holding a function from String -> IOAction a, which can be applied to
1229 whatever String is read from stdin.

1230 Notice that the only terminal constructor is Return that means that any IOAction must
1231 be a combination of Get and Put constructors, finally ending in a Return.

1232 Some simple actions include the one that prints to stdout before returning ():

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

1233 and the action that reads from stdin and returns the string unchanged:

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

1234 To build up a language for doing I/O we need to be able to combine and sequence
1235 actions. We want the ability to perform an IOAction a followed by an IOAction b, and
1236 return some result.

1237 In fact, we could have the second IOAction depend on the return value of the first one -
1238 that is, we need a sequencing combinator of the following type:

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

1239 We want to take the IOAction a supplied in the first argument, get its return value (which
1240 is of type a) and feed that to the function in the second argument, getting an IOAction b
1241 out, which can be sequenced with the first IOAction a.

1242 Thats a bit of a mouthful, but writing this combinator isnt too hard. When we reach the
1243 final Return, we apply the function f to get a new action. For the other constructors, we
1244 keep the form of the action the same, and just thread seqio through the seqio constructor.

1245 Using seqio we can define the action that gets input from stdin and immediately prints
1246 it to the screen:
1247 or even more complicated actions:

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

1248 Although this looks like imperative code (admittedly with pretty unpleasant syntax), its
1249 really a value of type IOAction (). In Haskell, code can be data and data can be code.

1250 In the gist Ive defined a function to convert an IOAction to a String, which allows them
1251 to be printed, so you can load the file into GHCi and verify that hello is in fact just data:

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

1252 It will surprise no one to learn that IOAction is a monad. In fact weve already defined the
1253 necessary bind operation in seqio, so getting the Monad instance is trivial:

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

1254 The main benefit of doing this is that we can now sequence actions using Haskell's `do`
1255 notation, which desugars into calls to `(\l_<=)`, and hence to `seqio`. Our earlier hello example
1256 can now be written as:

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

1257 Remember though, that this is still just defining a value of type `IOAction ()` - no code is
1258 executed, and no effects occur! So far, this post is 100 % pure.

1259 To see the effects, we need to define a function that takes an `IOAction a` and converts
1260 it into a value of type `IO a`, which can then be executed by the interpreter or the runtime
1261 system. It's easy to write such a function just by turning it into the appropriate calls to
1262 `putStrLn` and `getLine`.

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

1263 You can now load up GHCi and apply `run` to any action - a value of type `IO a` will be
1264 returned, and then immediately executed by the interpreter:

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

1265 Is there any practical use to this?

1266 Yes - an `IOAction` is a mini-language for doing I/O. In this mini language you are
1267 restricted to only reading from `stdin` and writing to `stdout` - there is no accessing files,

1268 spawning threads or network I/O.

1269 In effect we have a safe domain-specific language. If a user of your program or library
1270 supplies a value of type `IOAction a`, you know that you are free to convert it to an `IO a` using
1271 `run` and execute it, and it will never do anything except reading from `stdin` and writing to
1272 `stdout` (not that those things aren't potentially dangerous in themselves, but)

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

```

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

```

```

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

```

```

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

```

```

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

```



```

217
218 instance Functor IOAction where
219     fmap = mapio

```

1273 16.2 Dr. Casperson Pure IO

```

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

```

1274 16.3 Mehul Pure IO

1275 So when the program is getting interpreted the interpreter encounters an IO operation which
1276 then gets "interpreted" to the above and it continues normally.

1277 The interpreted program is still pure since the IO actions have not been executed

1278 if the running is done inside a monad then the IO still is pure.

```

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

```

```

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

```

Chapter 17

Work Completed

17.1 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 [130].

17.2 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 17.1.

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

17.3 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.

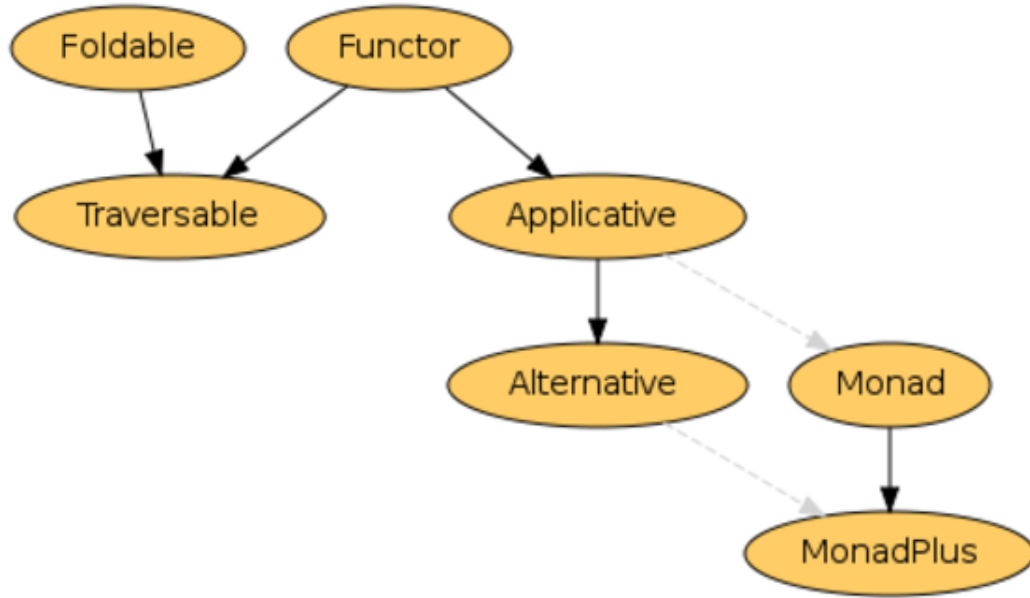


Figure 17.1: Functor Hierarchy [148]

1294 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)

```

1295 A FlatTerm can be of infinite depth which due to the reason stated above cannot be
 1296 accounted for during application function. The resulting type signature would be of the
 1297 form,

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

1298 Enter the Fix same as the function as a data type. The above would be simply reduced
 1299 to,

```
Fix FlatTerm
```

1300 resulting in the PROLOG Data Type

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

1301 17.4 Dr. Casperson's Explanation

1302 A recursive data type in HASKELL is where one value of some type contains values of that
1303 type, which in turn contain more values of the same type and so on. Consider the following
1304 example.

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

1305 A sample Tree would be,

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

1306 The above structure can be infinitely deep since HASKELL is a *lazy* programming lan-
1307 guage. But working with an infinitely deep / nested structure is not possible and will result
1308 in a *occurs check* error. This is because writing a type signature for a function to deal
1309 with such a parameter is not possible. One option would be to *flatten* the data type by the
1310 introduction of a type variable. Consider the following,

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

1311 A sample FlatTerm would be similar to Tree.

1312 The FlatTree is recursive but does not reference itself. But it too can be infinitely deep
1313 and hence writing a function to work on the structure is not possible.

1314 17.5 The other fix

1315 The `fix` function in the `Control.Monad.Fix` module allows for the definition of recursive
1316 functions in HASKELL. Consider the following scenario,

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

1317 The above function results in an infinite application stream,

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

1318 A fixed point of a function f is a value a such that $f\ a == a$. This is where the name of
1319 `fix` comes from: it finds the least-defined fixed point of a function.

1320 17.6 The Fix we use

1321 Fix-point type allows to define generic recursion schemes [69]. [8]

1322 **What is Algebra** Naively speaking algebra gives us the ability to perform calculations
1323 with numbers and symbols.

1324 **What can algebra do** The ability to form and evaluate expressions.

1325 **How to generate expressions** Using grammars, for example

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

1326 **How to uncover primitives from a recursive type** Make it non-recursive by defining a
1327 type function, otherwise known as type constructor,

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

1328 **How to create a nested structure from the above** The fractally recursive structure of `Expr`
1329 can be generated by repeatedly applying `ExprF` to itself.

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

1330 **How to generate really deep expressions** Keep on applying

```
ExprF
```

1331 **Is there a better way** After infinitely many iterations we should get to a fix point where
1332 further iterations make no difference. It means that applying one more ExprF would
1333 not change anything – a fix point does not move under ExprF. It's like adding one to
1334 infinity: you get back infinity.

1335 **How do that in HASKELL** In HASKELL, we can express the fix point of a type construc-
1336 tor f as a type:

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

1337 With that, we can redefine Expr as a fixed point of ExprF:

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

1338 **Any other benefits** Writing functions is simpler. You can have the terms of all depths
1339 encapsulated under the same type, i.e.

```
Fix ExprF
```

1340 So rather than writing separate functions for,

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

1341 We write a function from,

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


17.7 Opening up or Extending language Explanation using Box Analogy

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

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

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

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

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

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

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

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

```

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

```

1352 The above language can be of any type a . A more accurate way of saying it would
 1353 be that a can be a *kind* in HASKELL.

1354 In type theory, a kind is the type of a type constructor or, less commonly, the type
 1355 of a higher-order type operator. A kind system is essentially a simply typed lambda
 1356 calculus 'one level up,' endowed with a primitive type, denoted $*$ and called 'type,'
 1357 which is the kind of any (monomorphic) data type for example [147],

```

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

```

1358 Simply speaking the a can be changed.

1359 3. It gives the language the capability to be expanded. Adding some functionality to the
 1360 original language could be done in a no. of ways

1361 (a) Manually modifying the structure of the language,

```

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

```

1362 This would then trigger a ripple effect throughout the architecture because ac-
 1363 comodations need to be made for the new functionality.

1364 (b) The other option would be to *functorize* language like we did by adding a type
 1365 variable which can be used to plug something that provides the functionality
 1366 into the language. Consider the following example,

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

1367 then something like,

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

1368 is possible. Since we needed the fixed point of the language we used *Fix* but
 1369 generically one could add multiple custom functionality.

1370 4. If we have a grammar that support an expressions like,

1371 $x \cdot y + x \cdot z$

1372 Once the language is 'functorized' one can add quantifiers and logic to the language
 1373 to do something like,

1374 $\forall x \forall y \forall z \quad x \cdot y + x \cdot z$
 1375 $= x \cdot (y + z)$

1376 5. Multiple modifications

1377 6. As is with the original language it can be wrapped with multiple other data structures,

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

1378 and so on. But the core expression can only be of type *Term*.

1379 Whereas a *FlatTerm* expression can not only have an outer wrapper but also its type
 1380 is 'open'.

Chapter 18

Results

18.1 Types

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

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

Our preliminary implementation is as follows,

```
type Atom = String

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

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

```

{--
Output :-

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

--}

```

1394 which in PROLOG would look like,

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

1395 **18.2 Lazy Evaluation**

1396 **18.3 Opening up the Language**

1397 **Flattening**

1398 **Fixing**

1399 **MetaSyntactic Variables**

1400 **18.4 Quasi Quotation**

1401 **18.5 Template Haskell**

1402 **18.6 Higher Order Functions**

```
% Mehul Solanki.
```

```
% Higher Order Functions.
```

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

```

:- use_module(library(apply)).

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

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

```

1403 18.7 I/O

```

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

```

1404 18.8 Mutability

1405 18.9 Unification

1406 18.10 Monads

Chapter 19

Future Scope

1. Quasi quoter to get something like,

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

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

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

```
1 queryResolver searchStrategy query knowledgeBase
```

3. Add database operations

4. Multi type variable Language

5. Pure + IO Combined Language

```
1 data LangWithIO typevariableforpureexpressions typevariableforioexpressions
2     = PureConstructor_1 ....
3     | PureConstructor_2 ....
4     | IOContrcutor_1 .....
5     | IOConstructor_2 ...
6     | ConstructorWithBoth_1 .....
7     | ConstructorWithBoth_2 .....
8     deriving(.....)
```


1418 **Chapter 20**

1419 **Conclusion / Expected Outcomes**

1420 The aim of this study is to experiment with two different languages working together and/or
1421 contributing in providing a solution. Mixing and matching conflicting characteristics may
1422 lead to a behaviour similar to that of a multi paradigm language. The points to be looked at
1423 are efficiency of the emulation, semantics of the resulting embedding.

1424 Moreover, this will be an attempt to answer the question how practical PROLOG fits
1425 into HASKELL.

Chapter 21

Editing to do

This Chapter needs to be removed from the final work.

Meeting on 5th Novemeber 2015

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

2015-10-29

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

Either

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

Mehul

5. Chapter 13.5 needs fleshing out.
6. **Rewrite (Section) Chapter 3.2**. You are now in a position to state what your contributions are. In some sense everything else flows around this.
7. Fix the reference at the bottom of page 2:
`citewikipro- log,somogyi1995logic,website:prolog1000db`. **SOLVED**
8. Write enough of Chapters 13–16 that we can decide what material is needed in Chapters 9, 10, and ??.
9. [mainly done] If you don't like the shape of the paragraphs that you get without paragraph, use something like
`\setlength{\parindent}{3em}`
`\setlength{\parskip}{2\baselineskip}`
 to adjust either the initial paragraph indent, or the inter-paragraph space.
10. Rewrite (Section) Chapter 3 in formal English.
11. “re-curses” means to swear again (*p* 9). **Changed to recurs**
12. I am not sure that I agree with the use of “reflective” on *p* 8 (*l* 25). Reflection often means run-time introspection (for instance the Java `.getClass()` method). In computer science, reflection is the ability of a computer program to examine (see type introspection) and modify its own structure and behavior (specifically the values, meta-data, properties and functions) at runtime.
13. Supply your credentials in the front material (what degrees do you have?). (Search for `%% Supply your credentials in proposal1.tex`.)
14. The abstract is too long. UNBC guidelines limit Masters' theses abstracts to 150 words.

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

David

16. Clean up the non-exclusive license page in `unbcthesis.cls`

17. Incorporate `unbcthesis.cls` into Mehul’s work.

18. Review Chapter 2

19. Review Chapter 3

20. Review Chapter 4

21. Review Chapter 5

22. Review Chapter 6

23. Review Chapter 7

24. Review Chapter 8

25. Review Chapter 18

21.1 Editing suggestions from David

Thoughts on Chapter 14

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

```

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

```

Figure 21.1: A sample Minted figure

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

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

and then go on to discuss the syntax.

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

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

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

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

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

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

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

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

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

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

Thoughts on 1.4 Rename to "Thesis Organization".

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

What is the overall message?

1426 Bibliography

- 1427 [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, tech-*
1428 *niques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA,
1429 USA, 1986.
- 1430 [2] Hassan Aït-Kaci and Forêt Des Flambertins, *Warrens abstract machine a tutorial*
1431 *reconstruction*, (1999).
- 1432 [3] Sergio Antoy, *Implementing functional logic programming languages*.
- 1433 [4] ———, *Sergio antoy home page*.
- 1434 [5] Sergio Antoy and Michael Hanus, *Functional logic programming*, Communications
1435 of the ACM **53** (2010), no. 4, 74–85.
- 1436 [6] Lennart Augustsson, *Cayenne – a language with dependent types*, IN INTER-
1437 NATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, ACM Press,
1438 1998, pp. 239–250.
- 1439 [7] Andrei Barbu, *The csp package*, August 2013, [http://hackage.haskell.org/](http://hackage.haskell.org/package/csp)
1440 [package/csp](http://hackage.haskell.org/package/csp).
- 1441 [8] FP Complete Bartosz Milewski, *Understanding algebras*, October 2013.
- 1442 [9] Matthias Bartsch, *The prolog-graph package*, September 2011, [http://hackage.](http://hackage.haskell.org/package/prolog-graph)
1443 [haskell.org/package/prolog-graph](http://hackage.haskell.org/package/prolog-graph).
- 1444 [10] Eli Barzilay and Dmitry Orlovsky, *Foreign interface for plt scheme*, on Scheme and
1445 Functional Programming (2004), 63.
- 1446 [11] Nick Benton, *Embedded interpreters*, Journal of Functional Programming **15** (2005),
1447 no. 4, 503–542.
- 1448 [12] Didier Bert, Pascal Drabik, and Rachid Echahed, *Lpg: A generic, logic and func-*
1449 *tional programming language*, STACS 87, Springer, 1987, pp. 468–469.
- 1450 [13] James Bielman and Lus Oliveira, *Common lisp foreign function interface*, March
1451 2014.

- 1452 [14] Andrew Butterfield (ed.), *Unifying theories of programming, second international*
 1453 *symposium, utp 2008, dublin, ireland, september 8-10, 2008, revised selected pa-*
 1454 *pers*, Lecture Notes in Computer Science, vol. 5713, Springer, 2010.
- 1455 [15] C2, *Multi paradigm programming language*, September 2012.
- 1456 [16] c2 wiki, *Metasyntactic variables*, September 2011.
- 1457 [17] Catb, *Metasynatactic variables*.
- 1458 [18] Prolog Development Center, *Visual prolog*, June 2013.
- 1459 [19] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang, *A type-safe embedding of con-*
 1460 *straint handling rules into haskell*, Technical report School of Computing, National
 1461 University of Singapore, Boston, MA, USA (2003).
- 1462 [20] Ciao, *Ciao programming language*, August 2011.
- 1463 [21] Koen Claessen and Peter Ljunglöf, *Typed logical variables in haskell.*, Electr. Notes
 1464 Theor. Comput. Sci. **41** (2000), no. 1, 37.
- 1465 [22] Code Commit, *Hindley milner type system*, December 2008.
- 1466 [23] Mozart Consortium, *Oz / mozart*, March 2013.
- 1467 [24] Gregory Crosswhite, *The logicgrowsontrees package*, September 2013, [http://](http://hackage.haskell.org/package/LogicGrowsOnTrees)
 1468 hackage.haskell.org/package/LogicGrowsOnTrees.
- 1469 [25] DanDoel, *The logict package*, August 2013, [http://hackage.haskell.org/](http://hackage.haskell.org/package/logict)
 1470 [package/logict](http://hackage.haskell.org/package/logict).
- 1471 [26] ———, *The logict package example*, August 2013, [http://okmij.org/ftp/](http://okmij.org/ftp/Computation/monads.html)
 1472 [Computation/monads.html](http://okmij.org/ftp/Computation/monads.html).
- 1473 [27] Oleg Kiselyov Daniel P. Friedman, William E. Byrd, *The reasoned schemer*, The
 1474 MIT Press, Cambridge Massachusetts, London England, 2005.
- 1475 [28] William E. Byrd Daniel P. Friedman and Oleg Kiselyov, *Kanren*, March 2009.
- 1476 [29] Universidad Complutense de Madrid, *Toy*, Decmeber 2006.
- 1477 [30] University Of Melbourne Computer Science department, *Mercury programming lan-*
 1478 *guage*, February 2014.
- 1479 [31] Dustin DeWeese, *The peg package*, April 2012, [http://hackage.haskell.org/](http://hackage.haskell.org/package/peg)
 1480 [package/peg](http://hackage.haskell.org/package/peg).
- 1481 [32] Free Dictionary, *Quasi-quotation*.
- 1482 [33] Open Directory Project dmoz, *Multi paradigm*, November 2013.

- 1483 [34] SWI Prolog Documentation, *Embedding swi-prolog in other applications*, June
1484 2013, <http://www.swi-prolog.org/pldoc/man?section=embedded>.
- 1485 [35] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan,
1486 *Making data structures persistent*, Proceedings of the eighteenth annual ACM sym-
1487 posium on Theory of computing, ACM, 1986, pp. 109–121.
- 1488 [36] Steve Dunne and Bill Stoddart (eds.), *Unifying theories of programming, first in-*
1489 *ternational symposium, utp 2006, walworth castle, county durham, uk, february 5-*
1490 *7, 2006, revised selected papers*, Lecture Notes in Computer Science, vol. 4010,
1491 Springer, 2006.
- 1492 [37] Joshua Eckroth, *Prolog resolution*, April 2014.
- 1493 [38] ———, *Prolog unification*, April 2014.
- 1494 [39] Martin Erwig, *Escape from zurg: an exercise in logic programming*, Journal of Func-
1495 tional Programming **14** (2004), no. 03, 253–261.
- 1496 [40] Sebastian Fischer, *The cflp package*, June 2009, [http://hackage.haskell.org/](http://hackage.haskell.org/package/cflp)
1497 [package/cflp](http://hackage.haskell.org/package/cflp).
- 1498 [41] ———, *stream-monad*, September 2012.
- 1499 [42] Adam C. Foltzer, *Molog*, March 2013.
- 1500 [43] Marc Fontaine, *The cspm-toprolog package*, August 2013, [http://hackage.](http://hackage.haskell.org/package/CSPM-ToProlog)
1501 [haskell.org/package/CSPM-ToProlog](http://hackage.haskell.org/package/CSPM-ToProlog).
- 1502 [44] David Fox, *The proplogic package*, April 2012, [http://hackage.haskell.org/](http://hackage.haskell.org/package/PropLogic)
1503 [package/PropLogic](http://hackage.haskell.org/package/PropLogic).
- 1504 [45] ———, *The logic-classes package*, October 2013, [http://hackage.haskell.](http://hackage.haskell.org/package/logic-classes)
1505 [org/package/logic-classes](http://hackage.haskell.org/package/logic-classes).
- 1506 [46] Jeremy Gibbons, *Unifying theories of programming with monads*, Unifying Theories
1507 of Programming, Springer, 2013, pp. 23–67.
- 1508 [47] GNU, *Gnu prolog for java*, August 2010, [http://www.gnu.org/software/](http://www.gnu.org/software/gnuprologjava/)
1509 [gnuprologjava/](http://www.gnu.org/software/gnuprologjava/).
- 1510 [48] Michael Hanus, *Michael hanus home page*.
- 1511 [49] Michael Hanus, *Multi-paradigm declarative languages*, Logic Programming,
1512 Springer, 2007, pp. 45–75.
- 1513 [50] Michael Hanus, *Functional logic programming*, February 2009.
- 1514 [51] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro, *Curry: A truly*
1515 *functional logic language*, Proc. ILPS, vol. 95, 1995, pp. 95–107.

- 1516 [52] Haskellwiki, *Template haskell*, October 2013.
- 1517 [53] Juan Jose Moreno Navarro Herbert Kuchen, *Babel programming language*, January
1518 1988.
- 1519 [54] Ernest Lepore Herman Cappelen, *Quotation*, January 2012.
- 1520 [55] Ralf Hinze et al., *Prological features in a functional setting axioms and implemen-*
1521 *tation.*, Fuji International Symposium on Functional and Logic Programming, Cite-
1522 seer, 1998, pp. 98–122.
- 1523 [56] Charles Anthony Richard Hoare and Jifeng He, *Unifying theories of programming*,
1524 vol. 14, Prentice Hall Englewood Cliffs, 1998.
- 1525 [57] Satoshi Egi Ryo Tanaka Takahisa Watanabe Kentaro Honda, *Egison package*, March
1526 2014.
- 1527 [58] Paul Hudak, *Building domain-specific embedded languages*, ACM Comput. Surv.
1528 **28** (1996), no. 4es, 196.
- 1529 [59] John Hughes, *Why functional programming matters*, The computer journal **32**
1530 (1989), no. 2, 98–107.
- 1531 [60] What is Tech Target, *Metasynatactic variables*, September 2005.
- 1532 [61] JaimieMurdock, *Haskell kanren*, March 2012.
- 1533 [62] JLogic, *Jlog - prolog in java*, September 2012, <http://jlogic.sourceforge.net/index.html>.
- 1534 [63] ———, *Jscriptlog - prolog in javascript*, September 2012, <http://jlogic.sourceforge.net/index.html>.
- 1535 [64] Paul Johnson, *Why haskell is good for embedded domain specific languages*, January
1536 2008.
- 1537 [65] Mark P Jones, *Mini-prolog for hugs 98*, June 1996, <http://darcs.haskell.org/hugs98/demos/prolog/>.
- 1538 [66] Simon L Peyton Jones, Jean-Marc Eber, and Julian Seward, *Composing contracts:*
1539 *An adventure in financial engineering*, FME, vol. 2021, 2001, p. 435.
- 1540 [67] Mark Kantrowitz, *The prolog 1000 database*, August 2012.
- 1541 [68] David Karger, *Persistent data structures*, September 2005.
- 1542 [69] Anton Kholomiov, *data-fix*, February 2013.
- 1543 [70] H Jan Komorowski, *Qlog: The programming environment for prolog in lisp*, Logic
1544 Programming (1982), 315–324.

- 1548 [71] Shriram Krishnamurthi, *Programming languages: Application and interpretation*,
1549 ch. 33-34, pp. 295–305, 307–311, Brown Univ., 2007.
- 1550 [72] ———, *Teaching programming languages in a post-linnaean age*, SIGPLAN Not.
1551 **43** (2008), no. 11, 81–83.
- 1552 [73] The Programming Languages Weblog Lambda The Ultimate, *Embedding prolog in*
1553 *haskell*, July 2004, <http://lambda-the-ultimate.org/node/112>.
- 1554 [74] ———, *Embedding one language into another*, March 2005, [http://](http://lambda-the-ultimate.org/node/578)
1555 lambda-the-ultimate.org/node/578.
- 1556 [75] ———, *Application-specific foreign-interface generation*, October 2006, [http://](http://lambda-the-ultimate.org/node/2304)
1557 lambda-the-ultimate.org/node/2304.
- 1558 [76] Duncan Temple Lang, *Embedding s in other languages and environments*, Proceed-
1559 ings of DSC, vol. 2, 2001, p. 1.
- 1560 [77] LangPop.com, *Programming language popularity*, October 2013.
- 1561 [78] University of Melbourne Lee Naish, *Neu prolog*, February 1991.
- 1562 [79] John W Lloyd, *Programming in an integrated functional and logic language*, Journal
1563 of Functional and Logic Programming **3** (1999), no. 1-49, 68–69.
- 1564 [80] Geoffrey Mainland, *Why it's nice to be quoted: quasiquoting for haskell*, Proceed-
1565 ings of the ACM SIGPLAN workshop on Haskell workshop, ACM, 2007, pp. 73–82.
- 1566 [81] Yonathan Malachi, Zohar Manna, and Richard Waldinger, *Tablog: The deductive-*
1567 *tableau programming language*, Proceedings of the 1984 ACM Symposium on LISP
1568 and functional programming, ACM, 1984, pp. 323–330.
- 1569 [82] Erik Meijer and Peter Drayton, *Static typing where possible, dynamic typing when*
1570 *needed: The end of the cold war between programming languages*, Citeseer, 2004.
- 1571 [83] Bertrand Meyer, *Eiffel as a framework for verification*, Verified Software: Theories,
1572 Tools, Experiments, Springer, 2008, pp. 301–307.
- 1573 [84] Juan José Moreno-Navarro and Mario Rodriguez-Artalejo, *Babel: A functional and*
1574 *logic programming language based on constructor discipline and narrowing*, Alge-
1575 braic and Logic Programming, Springer, 1988, pp. 223–232.
- 1576 [85] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo, *Logic programming with*
1577 *functions and predicates: The language babel*, The Journal of Logic Programming
1578 **12** (1992), no. 3, 191–223.
- 1579 [86] R Morrison and MP Atkinson, *Persistent languages and architectures*, Security and
1580 Persistence, Springer, 1990, pp. 9–28.

- 1581 [87] MPprogramming.com, *Castor : Logic paradigm for c++*, August 2010, [http://](http://www.mpprogramming.com/cpp/)
1582 www.mpprogramming.com/cpp/.
- 1583 [88] Gopalan Nadathur, *λ prolog*, September 2013.
- 1584 [89] Mark J Nelson, *Why did prolog lose steam?*, August 2010, [http://www.kmjn.org/](http://www.kmjn.org/notes/prolog_lost_steam.html)
1585 [notes/prolog_lost_steam.html](http://www.kmjn.org/notes/prolog_lost_steam.html).
- 1586 [90] Mozilla Developer Network, *Multi paradigm language*, February 2014.
- 1587 [91] Johan Nordlander, *O'haskell*, January 2001.
- 1588 [92] Kurt Nrmak Department of Computer Science Aalborg University Denmark,
1589 *Linguistic abstraction*, July 2013, [http://people.cs.aau.dk/~normark/](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_)
1590 [prog3-03/html/notes/languages_themes-intro-sec.html#languages_](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_)
1591 [intro-sec_section-title_1](http://people.cs.aau.dk/~normark/prog3-03/html/notes/languages_themes-intro-sec.html#languages_).
- 1592 [93] University of Maryland Medical Center, *Lisp, unification and embedded languages*,
1593 October 2012, <http://www.cs.unm.edu/~luger/ai-final2/LISP/>.
- 1594 [94] University of Miami, *Prolog introduction*, March 2012.
- 1595 [95] Ocaml Org, *Ocaml programming language*, March 2014.
- 1596 [96] Pedro Pinto, *Dot-scheme: A plt scheme ffi for the .net framework*, Workshop on
1597 Scheme and Functional Programming, Citeseer, 2003.
- 1598 [97] Quintus Prolog, *Embeddability*, December 2003, [http://quintus.sics.se/isl/](http://quintus.sics.se/isl/quintuswww/site/embed.html)
1599 [quintuswww/site/embed.html](http://quintus.sics.se/isl/quintuswww/site/embed.html).
- 1600 [98] SWI Prolog, *swi prolog syntax and semantics*.
- 1601 [99] Yield Prolog, *Yield prolog*, October 2011, [http://yieldprolog.sourceforge.](http://yieldprolog.sourceforge.net/)
1602 [net/](http://yieldprolog.sourceforge.net/).
- 1603 [100] Shengchao Qin (ed.), *Unifying theories of programming - third international symposium, utp 2010, shanghai, china, november 15-16, 2010. proceedings*, Lecture Notes
1604 in Computer Science, vol. 6445, Springer, 2010.
- 1605
- 1606 [101] John Ramsdell, *The cmu package*, February 2013, [http://hackage.haskell.](http://hackage.haskell.org/package/cmu)
1607 [org/package/cmu](http://hackage.haskell.org/package/cmu).
- 1608 [102] Norman Ramsey, *Embedding an interpreted language using higher-order functions*
1609 *and types*, Proceedings of the 2003 workshop on Interpreters, virtual machines and
1610 emulators, ACM, 2003, pp. 6–14.
- 1611 [103] John Reppy and Chunyan Song, *Application-specific foreign-interface generation*,
1612 Proceedings of the 5th international conference on Generative programming and
1613 component engineering, ACM, 2006, pp. 49–58.

- 1614 [104] Maik Riechert, *The monadiccp package*, July 2013, <http://hackage.haskell.org/package/monadiccp>.
1615
- 1616 [105] J Alan Robinson and Ernest E Sibert, *Loglisp: Motivation, design, and implementa-*
1617 *tion*, 1982.
- 1618 [106] John Alan Robinson and EE Silbert, *Loglisp: an alternative to prolog*, School of
1619 Computer and Information Science, Syracuse University, 1980.
- 1620 [107] Raúl Rojas, *A tutorial introduction to the lambda calculus*, DOI= <http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf> (2004).
1621
- 1622 [108] Daniel Seidel, *The prolog-graph-lib package*, June 2012, <http://hackage.haskell.org/package/prolog-graph-lib>.
1623
- 1624 [109] ———, *The prolog package*, June 2012, <http://hackage.haskell.org/package/prolog>.
1625
- 1626 [110] Eric Seidel, *The liquid-fixpoint package*, September 2013, <http://hackage.haskell.org/package/liquid-fixpoint>.
1627
- 1628 [111] Silvija Seres, *The algebra of logic programming*, Ph.D. thesis, 2001.
- 1629 [112] Silvija Seres and Shin-Cheng Mu, *Optimisation problems in logic programming: an*
1630 *algebraic approach*, (2000).
- 1631 [113] Silvija Seres, J Michael Spivey, and CAR Hoare, *Algebra of logic programming.*,
1632 ICLP, 1999, pp. 184–199.
- 1633 [114] Silvija Seres and Michael Spivey, *Higher-order transformation of logic programs*,
1634 Logic Based Program Synthesis and Transformation, Springer, 2001, pp. 57–68.
- 1635 [115] Tim Sheard and Emir Pasalic, *Two-level types and parameterized modules*, Journal
1636 of Functional Programming **14** (2004), no. 05, 547–587.
- 1637 [116] Dorai Sitaram, *Racklog: Prolog-style logic programming*, January 2014.
- 1638 [117] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard OKeefe, *Logic*
1639 *programming for the real world*, Proceedings of the ILPS, vol. 95, 1995, pp. 83–94.
- 1640 [118] Andy Sonnenburg, *logicst*, April 2013.
- 1641 [119] JM Spivey, *An introduction to logic programming through prolog*, 1995.
- 1642 [120] JM Spivey and Silvija Seres, *The algebra of searching*, Festschrift in honour of
1643 CAR Hoare (1999).
- 1644 [121] ———, *Embedding prolog in haskell*, Proceedings of Haskell, vol. 99, Citeseer,
1645 1999, pp. 1999–28.

- 1646 [122] Michael Spivey, *Functional pearls combinators for breadth-first search*, Journal of
1647 Functional Programming **10** (2000), no. 4, 397–408.
- 1648 [123] Stackoverflow, *Haskell vs. prolog comparison*, December 2009, [http://](http://stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison)
1649 stackoverflow.com/questions/1932770/haskell-vs-prolog-comparison.
- 1650 [124] Patrick Blackburn Johan Bos Kristina Striegnitz, *Learn prolog now*, January 2012.
- 1651 [125] ———, *Learn prolog now*, January 2012.
- 1652 [126] Jurrien Stutterheim, *The nanoprolog package*, December 2011, [http://hackage.](http://hackage.haskell.org/package/NanoProlog)
1653 [haskell.org/package/NanoProlog](http://hackage.haskell.org/package/NanoProlog).
- 1654 [127] Evgeny Tarasov, *The hswip package*, August 2010, [http://hackage.haskell.](http://hackage.haskell.org/package/hswip)
1655 [org/package/hswip](http://hackage.haskell.org/package/hswip).
- 1656 [128] Chris Taylor, *Io is pure*, February 2013.
- 1657 [129] William E. Byrd The Reasoned Schemer’ (MIT Press, 2005) by Daniel P. Friedman
1658 and Oleg Kiselyov, *minikanren*.
- 1659 [130] Wren Thornton, *The unification-fd package*, July 2012, [http://hackage.](http://hackage.haskell.org/package/unification-fd)
1660 [haskellhttp://yieldprolog.sourceforge.net/.org/package/](http://hackage.haskell.org/package/unification-fd)
1661 [unification-fd](http://hackage.haskell.org/package/unification-fd).
- 1662 [131] ———, *Unification tutorial 1*, October 2015.
- 1663 [132] ———, *Unification tutorial 2*, October 2015.
- 1664 [133] Jan Tikovsky, *The monadiccp-gecode package*, January 2014, [http://hackage.](http://hackage.haskell.org/package/monadiccp-gecode)
1665 [haskell.org/package/monadiccp-gecode](http://hackage.haskell.org/package/monadiccp-gecode).
- 1666 [134] Carnegie Mellon University, *Algebraic logic functional programming language*,
1667 February 1995.
- 1668 [135] Dalhousie University, *Control flow*, January 2012.
- 1669 [136] Simon Fraiser University, *Life programming language*, March 1998.
- 1670 [137] Los Angeles University of California, *Virgil programming language*, March 2012.
- 1671 [138] Germany University of Kiel, *Curry programming language*, September 2013.
- 1672 [139] Maarten van Emden, *Who killed prolog?*, August 2010, [http://vanemden.](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/)
1673 [wordpress.com/2010/08/21/who-killed-prolog/](http://vanemden.wordpress.com/2010/08/21/who-killed-prolog/).
- 1674 [140] Andre Vellino, *Prolog’s death*, August 2010, [http://synthese.wordpress.com/](http://synthese.wordpress.com/2010/08/21/prologs-death/)
1675 [2010/08/21/prologs-death/](http://synthese.wordpress.com/2010/08/21/prologs-death/).
- 1676 [141] Job Vranish, *minikanrent*, March 2013.

- 1677 [142] Philip Wadler, *Comprehending monads*, Mathematical Structures in Computer Sci-
1678 ence **2** (1992), no. 04, 461–493.
- 1679 [143] Haskell Website, *Logic programming example*, February 2010, http://www.haskell.org/haskellwiki/Logic_programming_example.
1680
- 1681 [144] ———, *Logic programming example in haskell*, February 2010.
- 1682 [145] ———, *Quasiquote in haskell*, January 2014, <http://www.haskell.org/haskellwiki/Quasiquote>.
1683
- 1684 [146] Haskell Wiki, *Monads as computation*, December 2011.
- 1685 [147] ———, *Kind*, August 2012.
- 1686 [148] ———, *Foldable and traversable*, January 2013.
- 1687 [149] ———, *The haskell programming language*, October 2013.
- 1688 [150] ———, *Embedded domain specific languages*, September 2014.
- 1689 [151] ———, *Haskell/laziness*, November 2014.
- 1690 [152] ———, *Monads in haskell*, January 2014.
- 1691 [153] ———, *Haskell in industry*, June 2015.
- 1692 [154] Wikipedia, *Prolog wikipedia*, March 2004.
- 1693 [155] ———, *Functional logic programming languages*, February 2005.
- 1694 [156] ———, *Common lisp object system*, December 2013.
- 1695 [157] ———, *Curry programming language*, December 2013.
- 1696 [158] ———, *Functional logic programming*, May 2013.
- 1697 [159] ———, *Quasiquote*, November 2013, <http://en.wikipedia.org/wiki/Quasi-quote>.
1698
- 1699 [160] ———, *Common language infrastructure*, February 2014.
- 1700 [161] ———, *Common language runtime*, March 2014.
- 1701 [162] ———, *Constraint handling rules*, March 2014.
- 1702 [163] ———, *Constraint programming*, March 2014.
- 1703 [164] ———, *Damas-hindley-milner type system*, February 2014.
- 1704 [165] ———, *Foreign function interface*, January 2014.
- 1705 [166] ———, *Lambda calculus*, March 2014.

- 1706 [167] ———, *List of multi paradigm languages*, March 2014.
- 1707 [168] ———, *Meta programming*, March 2014.
- 1708 [169] ———, *Ocaml*, March 2014.
- 1709 [170] ———, *Programming paradigm*, March 2014.
- 1710 [171] ———, *Comparison of prolog implementations*, August 2015.
- 1711 [172] ———, *Control flow*, August 2015.
- 1712 [173] ———, *Declarative programming*, September 2015.
- 1713 [174] ———, *Metasyntactic variable*, October 2015.
- 1714 [175] ———, *Resolution*, October 2015.
- 1715 [176] ———, *Usemention distinction*, October 2015.
- 1716 [177] Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi (eds.), *Unifying*
 1717 *theories of programming, 4th international symposium, utp 2012, paris, france, au-*
 1718 *gust 27-28, 2012, revised selected papers*, Lecture Notes in Computer Science, vol.
 1719 7681, Springer, 2013.
- 1720 [178] Takashi's Workplace, *A prolog in haskell*, April 2009, [http://propella.](http://propella.blogspot.in/2009/04/prolog-in-haskell.html)
 1721 [blogspot.in/2009/04/prolog-in-haskell.html](http://propella.blogspot.in/2009/04/prolog-in-haskell.html).
- 1722 [179] xkcd, *Haskell vs prolog, or giving haskell a choice*, February 2009, [http://](http://echochamber.me/viewtopic.php?f=11&t=35369)
 1723 echochamber.me/viewtopic.php?f=11&t=35369.
- 1724 [180] Switzerland cole Polytechnique Fdrale de Lausanne (EPFL) Lausanne, *Scala pro-*
 1725 *gramming language*, 2002-2014.