Bedelibry Prolog: An introduction

Nathan Bedell

October 24, 2019

Preface

> Concepts lead us to make investigations. They are the expression of our interest and direct our interest.
>
> Wittgenstein

Ineviatably, the question that one has to answer when implementing or designing a new programming language is: *Why?*. We already have Haskell, Scala, Prolog, Scheme, Clojure, Java, Kotlin, C#, F#, Nim, Idris, Agda, Isabelle/Hol, Elm, ReasonML, Javascript – not to mention some of the more obscure offerings out there like Hume, Curry, Mercury, Ceylon, Caledon, Twelf, Flora-2 – and thousands of others. Why do we need a new programming language?

Initially, Bedelibry Prolog was developed to be used as a query and schema definiton language for *Bedelibry* – am open-source library and knowledge representation framework oriented towards helping working researchers organize their notes, references, and thoughts on the topics that they are interested in by allowing the user to make *ontologies* for different subjects, and making *semantic annotations* in their notes in a formal language to aid the future retrieval of facts relevant to their research via Bedelibry's query language. And thus, the language was designed due to the bad fit of existing solutions.

The first choice for the query language of Bedelibry was Prolog – as one of the requirements of the query language for the project was that it must support *deductive rules* that allow making inferences from different basic facts. SQL alone would not do. I wanted to be able to make queries that involve complicated logical inferences, not simply queries that involve retrieving data from a store. Prolog also had the advantage that it's syntax was clean, elegant, and familiar – being

similar enough to the language of propositional logic.

However, one of the issues with Prolog is its lack of a type system. One of the design requirements of Bedelibry is that it allows the user to declare *ontologies* – and it should be easy to check that a given prolog file typechecks against an ontology. Of course, the idea of a "Prolog with types" is not new. This requirement led me to consider existing solutions such as Twelf, Lambda Prolog, HiLog, HOPES, Issabelle/HOL, Mercury, Curry, Caledon, and Flora-2.

However, these projects either did not meet my needs by being too focused on *proof search* and *formal verification* rather than on knowledge representation, being unmaintained, and/or difficult to install and distribute, or by moving too far away from the simple Prolog-like syntax that I wanted to support.

In addition, in doing my own research into logic programming, and playing around with Prolog itself, I discovered that Prolog had some features that I did not want to support. The default depth first search and the ubiquitous "cut" feature made it difficult to work with Prolog as a query engine for a general-purpose query engine for a knowledge base with facts and rules (in my opinion, where logic programming really shines!). In short, I did not find Prolog's approach to combining logical and imperative features very satisfying. This led me to consider the excellent "pure Prolog" implementation by Ken Friis Larsen, which supported a breadth first search strategy, and being "pure", did not support any of Prolog's imperative features that I was not fond of. In addition, Ken Friis Larsen's implementation was in *Haskell* – a language I am very familiar with and fond of.

And so, I started hacking away! I added a proper REPL, some new syntatic features of my own creation, a type system, modules, a compiler – and this all came together to create the new programming language that I call Bedelibry Prolog.

Bedelibry Prolog was created with it's application to Bedelibry in mind – the interpreter itself can be run in server mode to provide a REST API that the rest of the Bedelibry ecosystem can interact with. Relations can be labeled "stored" to indicate that queries involving these relations involve facts that might be stored in the central bedelibry server (which uses a SQL backend for persistance). Thus, the language itself was concieved in a similar vein to Emacs Lisp or Guile Scheme – general purpose programming languages, but tied to specific

application(s), and with features which makes integration into those applications smooth and simple.

However, after it's conception as a typed Prolog with application-specific features tied to Bedelibry, Bedelibry Prolog eventually became a more general experiment in programming language design. I wanted to take the features of pure logic programming I liked, and combine them with what I saw as the good parts of other programming paradigms. I wanted good support for algebraic data types like in Haskell, but with more flexibility – similar to the OOP paradigm, but without what I see as the mistakes of enterprise OOP languages such as C# and Java. I wanted "pure-by-default" functional programming like in Haskell, with "escape hatches" like using *StateT* and *IO*, but with more flexibility, and rather than the implementation of imperative features in Prolog.

And so, Bedelibry Prolog was born. Somewhere in-between statically typed functional programming a la Haskell, logic programming, and object-oriented programming a la Scala – what, to me, seems like a hole in the design space of multi-paradigm languages that needed to be filled. A new paradigm inbetween the cracks of the old, with *types*, *relations*, *procedures*, *entities*, and *rules* as the main building blocks.

So, why bother? For one thing, I think this project came to the point that it did out of pure curiosity on my part. Since I first learned how to code in Python, and I was exposed to the idea of *functional programming*, I have always been interested in how different programming paradigms allow the user to express ideas more succinctly, and at the appropriate level of abstraction. New concepts, and new paradigms can be powerful in this way. And thus, I hope that in this book, I can convince the reader that the concepts in Bedelibry Prolog are worthwhile, and can make them into better programmers.

Contents

Chapter 1

Introduction

1.1 Organization of this book

This book is intended to be both an introduction, and documentation for the Bedelibry Prolog programming language. In addition, this book also contains signifigant exposition on the inspiration behind the language desing of Bedelibry Prolog, as well as case studies for some potential uses cases for which Bedelibry Prolog would be particularly well-suited. Thus, not everyone will want to read this entire reference linearly. Some sections/chapters are optional.

1.2 Getting started with Bedelibry Prolog

To get up and running with Bedelibry Prolog, you must have a working installation of the appropriate version of the Haskell Platform on your machine. Bedelibry Prolog is currently built and tested using ghc version 8.4.4. To get Bedelibry Prolog up and running on recent version of Ubuntu Linux is as simple as running

```
> sudo add-apt-repository -y ppa:hvr/ghc
> sudo apt install ghc-8.4.4
> sudo apt install cabal-install
```

to install the appropriate version of the Haskell platform, and then running

```
> git clone https://github.com/Sintrastes/bli-prolog
> cd bli-prolog
> cabal install
```

to install the Bedelibry Prolog interpreter *blipl* and compiler *blic*. Passing a command-line argument of *–help* to either of these commands will yield the appropriate documentation on the command-line arguments to these commands. We will display these here for reference:

```
> blipl --help
bli-prolog interpreter v0.3.0, (C) Nathan Bedell 2019

options [OPTIONS] [GOALSTRING]

Common flags:
     --search'=SEARCH       Specify wether to use DFS, BFS, or Limit
     --program'=FILE        Prolog file with clauses
     --schema'=ITEM         Schema file
  -l --limit'=INT           Limit the number of solutions found
  -d --depth'=INT           Maximum depth to traverse when using lim
                            search
  -v --verbose'             Specify whether or not to use verbose ou
                            by default)
  -n --nocolor'             Turn off colors in the REPL.
  -j --json'                Specify whether or not json output forma
                            used for queries.
     --server'              Starts a REST server for processing bli
                            queries if set.
     --bedelibrymode'=ITEM  Sets the mode of interaction between bli
                            and the bedebliry server.
     --port'=INT            Port number to start the server.
     --burl'=ITEM           URL of the bedelibry server configured t
                            with bli-prolog.
  -? --help                 Display help message
  -V --version              Print version information
     --numeric-version      Print just the version number
```

```
> blic --help
bli-prolog compiler v0.3.0, (C) Nathan Bedell 2019.
   Visit https://github.com/Sintrastes/bli-prolog for more documentation

blic [source_file]                    Compiles the specified source file t
                                      in the same directory, with the same
                                      file, but without extension.
blic [source_file] [output_file]      Compiles the specified source file t
                                      [output_file].

Flags:
  --help            Displays this help screen.
  --dyn             Compiles the file, dynamically linknig any dependent
  --bytecode        Compiles to Bedelibry Prolog bytecode instead of link
                    a static executable.
  --no-typecheck    Skips the typechecking step.
  --only-typecheck  Only typecheck the given file, skipping the compilati
```

If you are interested in Bedelibry more generally, and not just Bedelibry Prolog, you will also want to install the *bli* command-line tool, and and instance of the Bedelibry server.

Chapter 2

Logic Programming

> One person might say "A proposition is the most ordinary thing in the world", and another, "A proposition – that's something very remarkable!" – And the latter is unable simply to look and see how propositions work. For the forms of the expressions we use in talking about propositions and thought stand in his way.
>
> Wittgenstein

Bedelibry Prolog is first and foremost a logic programming language. This means that the primary concept to deal with when programming in Bedelibry Prolog is that of a *predicate*. In Bedelibry Prolog's type system, predicates have type *pred*.

If you have ever taken a course in formal logic, predicates in Bedelibry Prolog behave similarly to normal *propositions* in classical logic. A predicate is some *fact* that can either hold (in which case we say the predicate is *true*), or not hold (in which case we say it is *false*). To declare predicates in Bedelibry prolog (which always start with a lowercase letter), we use the syntax

```
rel p.
```

This is a declaration that p is a *relation* (which is the same thing as a predicate, except it can have an arbitrary number of arguments)

with no arguments – i.e. a *proposition*. If a declaration like the above has been made in a Bedelibry Prolog file that has been loaded into the REPL with no other facts declared about p, then typing the query

```
p.
```

```
> No solutions found.
```

This illustrates one important difference from classical logic in Bedelibry Prolog – we make an *open-world* assumption about facts by default. In other words, we have declared that p is a proposition, and we have not yet delcared that p is a *fact*, but this does not mean that p is false – only that we cannot determine that p is true.

If we add a declaration

```
p.
```

```
> True.
```

This is the basic type of query that we can make in Bedelibry Prolog – some combination of predicates, relations and proposions, which can either be *True*, or which we cannot determine to be true with our current state of knowledge.

The default argument of predicates and relations in Bedelibry Prolog is

```
entity
```

```
x: entity.
```

If we have a *predicate* or relations

```
rel p/1.
p(x).
```

2.1 Facts and rules

2.2 Types

In the last two sections we learned about the basic type system in Bedelibry, and how to declare facts and rules.

2.3 Assertions

> And is there also not the case where we play, and we make up the rules as we go along? And even where we alter them – as we go along.
>
> Wittgenstein

2.4 Metainterpreters

> But what does a game look like that is everywhere bounded by rules? whose rules never let a doubt creep in, but stop up all the gaps where it might? – Can't we imagine a rule regulating the application of a rule; and a doubt which it removes – and so on?
>
> Wittgenstein

Chapter 3

Functional Programming

Chapter 4

Imperative and Object-Oriented Programming

Chapter 5

Bedelibry

5.1 Syntax

> Our language can be seen as an ancient city: a maze of little streets and squares, of old and new houses, and of houses with additions from various periods; and this surrounded by a multitude of new boroughs with straight regular streets and uniform houses.
>
> Wittgenstein

36. Where our language suggest a body and there is none: there, we should like to say, is a spirit

But how many kinds of sentence are there? Say assertion, qeustion, and command? – There are *countless* kinds; countless different kinds of use of all the things we call "signs", "words", "sentences". And this diversity is not something fixed, given once and for all, but new types of language, new language-games, as we might say, come into existence, and others become obsolete and get forgotten.

5.2 Dealing with ambiguity

> And do we know any more ourselves? Is it just that we can't tell others exactly what a game is? – But this is not ignorance. We dont know the boundaries because none have been drawn.
>
> Philosophical Investigations, remark 69.

> And likewise the kinds of number, for example, form a family. Why do we call something a "number"? Well, perhaps because it has a – direct – affinity with several things that have hitherto been called "number", and this can be said to give it an inndirect affinity with other things that we also call "numbers". And we extend our concept of number as in spinning a thread we twist fibre on fibre. And the strength of the thread resides not in the fact that some one fibre runs through its whole length, but in the overlapping of many fibres.
>
> Philosophical Investigation, remark 67.

5.2.1 Sense disambiguation

> Consider this example: if one says "Moses did not exist", this may mean various things... 79
>
> Philosophical Investigation, remark 67.

5.2.2 Case study: Random quotation generator

fortune is a unix command-line utility which generates random epigrams from a preset set of epigrams stored on the user's filesystem.

```
> fortune | cowsay
 -------------------------------------
/ You'll feel much better once you've \
\ given up hope.                      /
```

```
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Charming! The user of fortune can of course cofigure these epigrams in various ways, and can add/remove different epigrams by editing the relevant files that fortune uses in its configuration.

To illustrate some of the features of Bedelibry prolog that we've introduced so far, let's make our own version of fortune, but one which instead prints random quotes from Wittgenstein which we have marked in Bedelibry that we like. We'll call our applicaton *wittquote*.

```
/* LANGUAGE BedelibryMode */

%
% _*_ wittquote.bpl _*_
%

using ontology_literature.
using philosoper_quotes.
using system_io.
using control_random.

?- {
  quote(?X), written_by(?X, wittgenstein), likes(me, ?X).
  random_select(?X).
  print_ln(?X).
}
```