HProlog Manual

Tim Yates

December 15, 2011

Contents

Chapter 1

About HProlog

HProlog is a interpreter and compiler for a subset of Prolog. HProlog is written in Haskell. Its features include:

- Full and correct support for resolution of logical goals, including recursive rules and nested structures
- Negation as a failure
- A compiler targeting a simplified version of the *Warren Abstract Machine* (WAM), as defined in Hasan Ait-Kaci's reconstruction[1]
- Lists and numbers (but unfortunately no math at this point)
- User-defined operators

[1] http://wambook.sourceforge.net/wambook.pdf

Chapter 2

Using HProlog

2.1 Setup

HProlog is distributed as a *cabal package*,[1] a commomly-used distribution system for Haskell programs. To build it, you will need to install the **Haskell Platform**.[2] Once that is installed, you can compile HProlog by running the following commands inside the directory where you unpacked the HProlog distribution:

```
$ cabal configure
$ cabal build
```

After that, the HProlog binary will be located at `dist/build/prolog/prolog`. You can copy it into the base directory for convenience:

```
$ cp dist/build/prolog/prolog .
```

2.2 Usage

When you first start HProlog, the program presents an interactive prompt where you can run queries and built-in commands:

```
$ ./prolog
?-
```

You can read in files using the *consult/1* command. Alternatively, you can specify files to consult through the command line:

[1] http://www.haskell.org/cabal/
[2] http://hackage.haskell.org/platform/

```
$ ./prolog file1 file2
?-
```

When you are finished with your session, press *Control+D* to quit.

2.3 Examples

The following sections contain example sessions using the files provided in the `test/` directory.

2.3.1 Family Tree Database

```
$ cd test/
$ ../prolog
?- consult(family_trees).
true.
?- parent_child(bill, ted).
true.
?- parent_child(ted, bill).
false.
?- parent_child(Who, bob).
Who = bill ? ;

Who = mary ? ;

false.
?- ancestor_descendent(kim, Whom).
Whom = george ? ;

Whom = mary ? ;

Whom = ted ? ;

Whom = bob ? ;

false.
?- ancestor_descendent(Who, ted).
Who = bill ? ;

Who = mary ? ;
```

```
Who = george ? ;

Who = susan ? ;

Who = dave ? ;

Who = kim ? ;

false.
?- ^D
```

2.3.2 List Processing

```
$ cd test/
$ ../prolog
?- consult(lists).
true.
?- member(What, [a,b,c]).
What = a ? ;

What = b ? ;

What = c ? ;

false.
?- append([a,b,c], [d,e,f], What).
What = [a,b,c,d,e,f] ? ;

false.
?- append(What, [d,e,f], [a,b,c,d,e,f]).
What = [a,b,c] ? ;

false.
?- reverse([a,b,c,d]).
What = [d,c,b,a] ? ;

false.
?- ^D
```

2.3.3 Compiling

```
$ cd test/
$ ../prolog
?- consult(lists).
true.
?- consult(family_trees).
true.
?- consult(crazy_structures).
true.
?- compile(everything).
true.
?- ^D
```

At the end of this session, there should be a file named `everything.wam` in the `test/` directory. It will contain WAM instructions in text format for all the predicates defined in all three test files.

2.4 Tests

Aside from the examples given above, HProlog also has unit tests for its parser and unification engine. To run these tests (from within the main HProlog directory):

```
$ ghci -isrc
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :load Prolog.Test
[1 of 5] Compiling Prolog.Data      ( src/Prolog/Data.hs, interpreted )
[2 of 5] Compiling Prolog.Parser    ( src/Prolog/Parser.hs, interpreted )
[3 of 5] Compiling Prolog.Compiler  ( src/Prolog/Compiler.hs, interpreted )
[4 of 5] Compiling Prolog.Interpreter ( src/Prolog/Interpreter.hs, interpreted )
[5 of 5] Compiling Prolog.Test      ( src/Prolog/Test.hs, interpreted )
Ok, modules loaded: Prolog.Test, Prolog.Data, Prolog.Parser, Prolog.Interpreter, Prolog.Com
*Prolog.Test> runTestTT test_parser
Loading package transformers-0.2.2.0 ... linking ... done.
Loading package bytestring-0.9.1.10 ... linking ... done.
Loading package mtl-2.0.1.0 ... linking ... done.
Loading package parsec-3.1.1 ... linking ... done.
```

```
Loading package HUnit-1.2.2.3 ... linking ... done.
Loading package array-0.3.0.2 ... linking ... done.
Loading package containers-0.4.0.0 ... linking ... done.
Cases: 29  Tried: 29  Errors: 0  Failures: 0
Counts {cases = 29, tried = 29, errors = 0, failures = 0}
*Prolog.Test> runTestTT test_unification
Cases: 17  Tried: 17  Errors: 0  Failures: 0
Counts {cases = 17, tried = 17, errors = 0, failures = 0}
*Prolog.Test> :quit
Leaving GHCi.
```

Note: As of this writing, there is a bug in operator parsing that will fail one of the parser tests. The only problem the bug causes is an inability to enforce non-associativity of operators. It should be fixed, but for now, it won't cause any problems in normal usage of the program.

Chapter 3

Technical Description

3.1 Architecture

HProlog is roughly divided into a parser, interpreter, and compiler units as shown in Figure 3.1. The job of each of these units is described in the following sections.

3.2 Parser

The parser is defined in `src/Prolog/Parser.hs`. Its job is to transform the concrete representation of Prolog rules into a list of rule data structures. We can divide this task into two levels: parsing rules, and parsing terms.

3.2.1 Rules

Consider the following input:

```
foo(X) :- bar(X, Y).
foo(a).
bar(Z, Z).
```

This will be transformed into a list of data structures of the form:

$$\textbf{DefiniteClause}\ h\ [g_1, g_2, \ldots, g_n],$$

where h is the representation of the head of the clause, and g_n is the representation of goal n in the body. Facts (heads with no body) are represented in the same form, but the list of goals is empty.

Queries and directives (rules with no heads) are represented in the form:

$$\textbf{GoalClause}\ [g_1, g_2, \ldots, g_n].$$

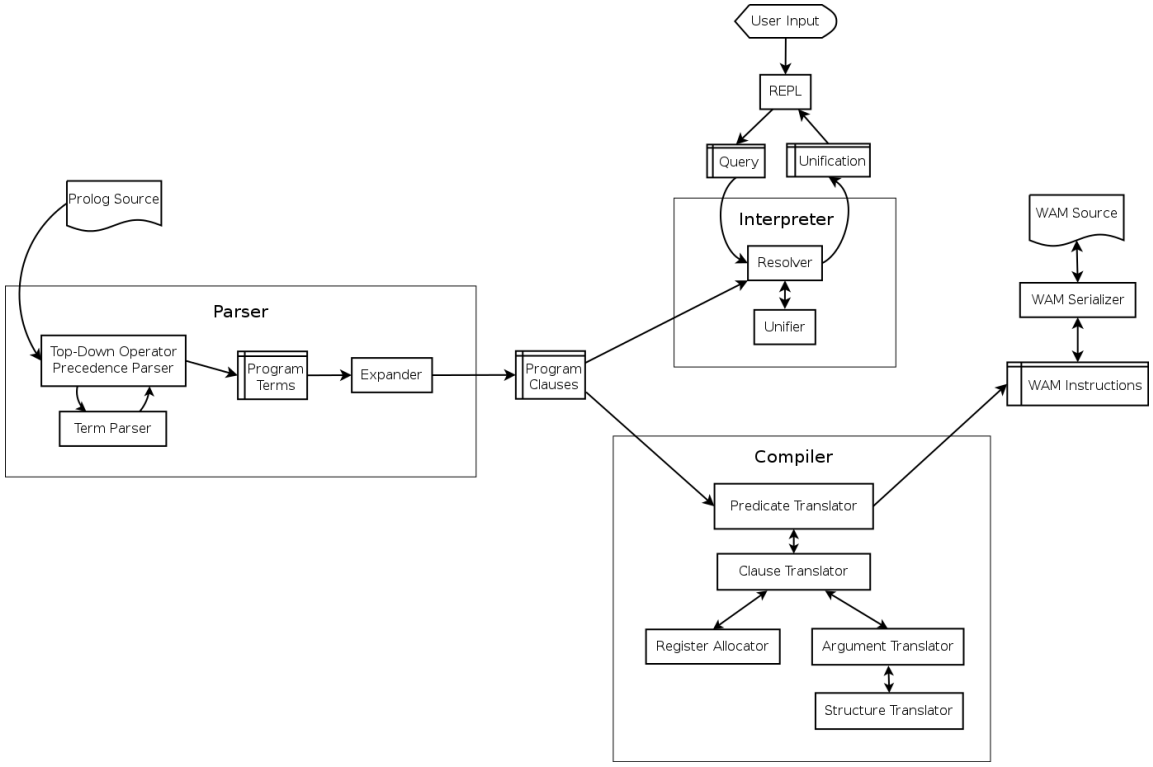

Figure 3.1: **Architecture diagram**. This version is slightly simplified. The interpreter is actually involved in parsing program clauses, so that directives in the source file (such as *op/3* definitions and *consult/1* directives) can be executed as they are read. Also, the compiler is run by the interpreter as a built-in predicate. The internal architectures are still accurate.

3.2.2 Terms

The next problem is how to represent the terms in the head and body of rules themselves. A *term* is one of:

- An *atom*: `a`, `foo`, `'with Quotes!'`, `-->`
- A *variable*: `X`, `SomeVar`
- A *number*: `123`
- A *compound term*: `f(a,b)`, `p(X, h(f(a), b))`

These are represented in the following forms:

- **Atom** *a*

- **Variable** v
- **Number** n
- **CompoundTerm** $f\ [t_1, t_2, \ldots, t_n]$

where:

- a, v, and f are the string representations of the atom, variable, and functor, respectively,
- n is the integer represented by the number token, and
- t_n is the nth subterm of the compound term.

Because compound terms contain other terms, the overall structure of parsed terms is a tree.

A final issue is how to deal with operators. HProlog supports user-defined operators, which are simply functors of arity 1 or 2 that are written in prefix, postfix, or infix notation. For example, the expression `a :- b` is really a compound term with functor *:-/2*, and can also be written as `:-(a, b)`. HProlog uses a *top-down operator precedence parser* to parse operations, which are then transformed into their term representation.

3.3 Interpreter

The interpreter is defined in `src/Prolog/Interpreter.hs`. Its job is to find logical solutions to queries using rules defined in a program. For example, consider the program:

```
parent_child(bill, ted).
parent_child(bill, bob).
parent_child(mary, ted).
parent_child(mary, bob).
parent_child(george, mary).
parent_child(susan, mary).

female(mary).
female(susan).
male(bill).
male(ted).
male(bob).
male(george).

mother_child(Mother, Child) :- female(Mother), parent_child(Mother, Child).
father_child(Father, Child) :- male(Father), parent_child(Father, Child).
```

Some queries that could be performed on this program include:

- `?- mother_child(susan, ted).` – Is `susan` the mother of `ted`?
- `?- father_child(Who, mary).` – `Who` is the father of `mary`?

To properly match these queries to rules in the program, we need two pieces: unification and resolution.

3.3.1 Unification

Unification is the process of substituting variables in two terms so that they match. For instance, `f(X, b)` can be unified with `f(g(a), Y)` by setting $X = g(a)$ and $Y = b$, so that both terms are equal to `f(g(a), b)`.

Unification is essentially the process of walking two term trees simultaneously and matching variables in one tree to the corresponding term in the other tree. We also have to obey a few rules:

- A variable can only have one substitution. We cannot unify `f(X, X)` $\sim$ `f(a, b)`, because that would require setting $X = a$ and $X = b$ at the same time.
- Only variables can be substituted. We cannot unify `f(a)` $\sim$ `f(b)` by substituting $a = b$.
- A variable cannot unify with a compound term that it occurs in (*occurs check*). We cannot unify `X` $\sim$ `f(a, X)`, because that would produce a cyclic term.

In many cases, unification is impossible, so we have to handle failure appropriately.

3.3.2 Resolution

The heart of Prolog is *resolution*. Resolution is an logical inference rule that can be used to solve the satisfiability problem for Horn formulas.

Consider a query:

```
?- g1, g2, ..., gn.
```

where g_n are independent goals in the query. We can determine whether all the goals are true by trying to prove any of them wrong. If none of them can be proved wrong, then they are all true. Turning this into logical form:

$$\begin{aligned} &\neg(g_1 \wedge g_2 \wedge \ldots \wedge g_n) \\ &= \neg g_1 \vee \neg g_2 \vee \ldots \vee \neg g_n \end{aligned}$$

Now assume we have a rule:

```
g1 :- h1, h2, ..., hn
```

We can represent this in logical form as:

$$\begin{aligned} & g_1 \leftarrow h_1 \wedge h_2 \wedge \ldots \wedge h_m \\ = & \, g_1 \vee \neg(h_1 \wedge h_2 \wedge \ldots \wedge h_m) \\ = & \, g_1 \vee \neg h_1 \vee \neg h_2 \vee \ldots \vee \neg h_m \end{aligned}$$

If we assume that both our goal and this rule are true, then we have:

$$\begin{aligned} & (g_1 \vee \neg h_1 \vee \neg h_2 \vee \ldots \vee \neg h_m) \wedge (\neg g_1 \vee \neg g_2 \vee \ldots \vee \neg g_n) \\ = & \, (g_1 \vee \neg(h_1 \wedge h_2 \wedge \ldots \wedge h_m)) \wedge (\neg g_1 \vee \neg(g_2 \wedge \ldots \wedge g_n)) \end{aligned}$$

Now notice that if g_1 is true, then $\neg g_1$ would be false and $\neg(g_2 \wedge \ldots \wedge g_n)$ would have to be true. If g_1 were false, then $\neg(h_1 \wedge \ldots \wedge h_n)$ would have to be true. In other words, one of the non-g_1 terms must be true no matter what g_1 is, so we can eliminate g_1 altogether and get:

$$\begin{aligned} & \neg(h_1 \wedge h_2 \wedge \ldots \wedge h_m) \wedge \neg(g_2 \wedge \ldots \wedge g_n) \\ = & \, \neg h_1 \vee \neg h_2 \vee \ldots \vee \neg h_m \vee \neg g_2 \vee \ldots \vee \neg g_n \end{aligned}$$

This last step is "resolution" proper. We now have a *new* set of goals, and we can repeat the procedure on this new set. We repeat until we either eliminate all the variables, proving our negation false and the original goals true, or until we have no rules left to resolve with, proving our negation true and the original goals false.

In summary, the steps of resolution are:

1. Negate the original goal clause.
2. Find a rule to unify with. If no rules unify, then fail.
3. Resolve against that rule to generate a new goal clause.
4. If nothing is left, succeed. Otherwise, repeat from step 2.

This is the basic algorithm used by HProlog, except that it also has to deal with the question of which rule to unify with when there are several possible alternatives. It handles this by (lazily) taking all possible paths and concatenating all the results into a single list. The result takes the same space complexity as so-called "backtracking" algorithms, but in a much more straightforward manner.

3.4 Compiler

The compiler is defined in `src/Prolog/Compiler.hs`. Its job is to take the rules in a program and translate them to a series of instructions for the Warren Abstract Machine (WAM). The details of the WAM are much too complicated to lay out here, but they can be found in *Warren's Abstract Machine: A Tutorial Reconstruction* by Hasan Ait-Kaci.[1] The version of the WAM targeted by HProlog is the one laid out in chapters 1-3 of that book. It does not include the many optimizations in chapter 4.

A simplified view of the job of the compiler is to take the rules and define them as callable procedures. These procedures are passed arguments through predefined registers. A rule of the form:

```
p(a1, a2, ..., an) :- q1(b1, b2, ..., bm), q2(...), ..., qn(...).
```

does the following:

1. Allocate space on the stack to store variables
2. Extract the arguments $a_1, \ldots, a_n$ of p/n and pull them onto the stack.
3. Pull the arguments $b_1, \ldots, b_m$ of q_1/m from the stack and put them in registers, and call q_1/m
4. Do the same for the rest of the goals.

While arguments, which contain references to terms in memory, are being moved from the stack to registers, their values are being unified. If unification fails, the whole rule fails. If more rules are possible, then the machine will try the other alternatives.

The compiler has to determine the right instructions in the right order to make this happen. Some examples of instructions are:

```
allocate 5
get_variable Y4 A1
get_value Y4 A2
put_variable Y3 A2
put_structure f/2 A3
unify_value X4
unify_value Y2
deallocate
```

where terms like $X1$, $A2$, and $Y4$ denote temporary registers, argument registers, and stack locations, respectively.

Some of the complications the compiler has to deal with include:

[1] http://wambook.sourceforge.net/wambook.pdf

- Assigning variables (and partially constructed structures) to appropriate registers.
- Deciding whether to keep variables in the stack or in temporary registers.
- Ordering the construction of nested terms so that they are constructed before the terms that contain them.

Chapter 4

Reference

The following built-in commands are available for you to use in HProlog:

consult(+Filename)
Read the Prolog source file *"<Filename>.pl"* into the current session.

compile(+Filename)
Compile all the predicates defined in the current session into WAM code, and dump the compiled output to *"<Filename>.wam"*.

not Goal
Negation as a failure: try to resolve *Goal*. Fail if a resolution is found, otherwise succeed.

true
Succeed without triggering any unification.

fail
Fail the current rule immediately.

op(+Precedence, +Type, +Symbol)
Define a new operator *Symbol* with precedence *Precedence* and fixity and associativity defined by *Type*. Valid values for *Type* are:

fx A non-associative prefix operator.

fy A right-associative prefix operator.

xf A non-associative postfix operator.

yf A left-associative prefix operator.

xfx A non-associative infix operator.

xfy A right-associative infix operator.

yfx A left-associative infix operator.