Haskell port of "Handbook of Practical Logic and Automated Reasoning"

Sean McLaughlin

May 28, 2015

1 Introduction

This document describes a port of the Objective Caml code supporting John Harrison's logic textbook "Handbook of Practical Logic and Automated Reasoning" to Haskell.

The code is written in literate Haskell style. We kept the file names of the original mostly intact, though use Haskell naming conventions. As an experiment, in some places we inlined the book text. We think this is a nice way to present the code, but didn't have the patience to do it systematically. Perhaps we will do this when the text is finalized.

One minor difference is in the moderate use of type classes which makes some of the functionality much cleaner, esp. the manipulation of polynomials, which can be achieved through Haskell binary operators instead of explicit term manipulation.

To interact with the port, we wrote our own interpreter for reasons given in the next section.

1.1 Interpreter

While Haskell and OCaml are quite similar, the port is complicated by the mode of interaction used in Harrison's code. Harrison uses the OCaml top level interpreter as a basis for experimentation. Additionally, his syntax for formulas includes

```
A \/ B /\ C
```

for "A or B and C". The backslash characters pose a difficulty. For example:

```
$ ocaml
        Objective Caml version 3.09.3

# let s = "A \/ B";;
Warning X: illegal backslash escape in string.
val s : string = "A \\/ B"
```

As you see, in Ocaml strings can only contain backslashes when they are "escaped" with yet another backslash. Indeed, as in most other languages (including Haskell), backslash is used to type unprintable or extended characters which are translated by the OCaml parser to the relevant ASCII or Unicode symbols. Note that OCaml only issues a warning when you type such a backslash, but this is still a problem:

```
$ ocaml
        Objective Caml version 3.09.3

# "A /\ B";;
- : string = "A / B"
```

An escaped space is certainly not what you meant here! When we try this experiment using GHCi (the interpreter of GHC, a Haskell compiler) we get an error.

```
$ ghci
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude> "A \/ B";
<interactive>:1:5: lexical error in string/character literal at character ' '
```

Harrison avoids the difficulty by using the Camlp4 preprocessing tool. Using Camlp4, new types of quotations can be invented, and parsed however you like. In particular, he can treat backslash as a simple backslash and not as an escape character. The default Camlp4 syntax for quotations is `<<string>>`, which Harrison uses in his examples:

```
# <<A /\ B>>;;
- : fol formula = <<A /\ B>>
# <<A \/ B>>;;
- : fol formula = <<A \/ B>>
```

Having no such tool for GHC, how do we proceed? There are a number of options.

1. Change the syntax of the examples to something the top level can parse, for instance

   ```
   "A & B | C"
   ```

2. Escape every backslash

   ```
   A /\\ B \\/ C
   ```

3. Write your own top level. Then you can parse a string any way you like, as Camlp4 does.

We decided upon the third option so as to keep the syntax close to the original. This means, however, that we can not do experiments in the same way Harrison does. He simply types commands at the OCaml top level

```
# resolution <<(exists y. forall x. P(x, y)) ==> forall x. exists y. P(x, y)>>;;
1 used; 1 unused.
- : bool list = [true]
```

We write

```
$ atp
> resolution -f <<(exists y. forall x. P(x, y)) ==> forall x. exists y. P(x, y)>>
<<(exists y. forall x. P(x, y)) ==>
  (forall x. exists y. P(x, y))>>
0 used; 2 unused.
1 used; 1 unused.
Solution found!
Computation time: 0.000 sec
```

The `-f` means "parse what follows as a formula". We also have a large file of stored formulas that can be referenced by name. For example, Pelletier's problems can be used as

```
> resolution p1
<<p ==> q <=> ~q ==> ~p>>
Solution found!
Computation time: 0.000 sec
```

As a result, we can't experiment quite as freely in Haskell as in OCaml. For instance, trying a different normalization function in a decision procedure involves killing the top level, recompiling, and starting the top level again. However, we do think writing your own top level has advantages of its own.

2 The ATP Interpreter

Experimenting with the code is done through the *ATP interpreter*. After building the interpreter:

```
$ make
ghc  -fwarn-deprecations -fwarn-dodgy-imports -fwarn-hi-shadowing -fwarn-implicit-prelude -fwarn-incompl
[ 1 of 33] Compiling UnionFind        ( UnionFind.lhs, UnionFind.o )
[ 2 of 33] Compiling Lib              ( Lib.lhs, Lib.o )
...
Linking atp ...
```

you can start it by typing

```
$ atp
>
```

Now you are ready to issue commands. If you just type `<return>` or `help` you will get a list of all available interpreter commands.

```
> help
Possible commands:
   === Util ===

      help               : this message
      echo               : echo the inputs
      bug                : bug of the moment

   === Parsing ===

      parseE             : parse an arithmetic expression and print it
      parseP             : parse a propositional formula and print it
      parseT             : parse a first order term and print it
      parseF             : parse a first order formula and print it

   === Propositional formulas ===

      nnf                : negation normal form
      cnf                : conjunctive normal form
      dnf                : disjunctive normal form
      defcnf             : definitional cnf
      truthtable         : show propositional truth table

   === Propositional decision procedures ===
```

```
   tautology           : Tautology checker via truth tables
   dp                  : Davis-Putnam procedure (propositional)
   dpll                : Davis-Putnam-Loveland-Logemann procedure (propositional)

=== First order formulas ===

   showFol             : show first order test formula
   pnf                 : prenex normal form
   skolemize           : skolem normal form

=== Basic Herbrand methods ===

   gilmore             : Gilmore procedure
   davisputnam         : Davis-Putname procedure

=== Unification ===

   unify               : unify two terms

=== Tableaux ===

   prawitz             : Prawitz procedure
   tab                 : Analytic tableaux procedure
   splittab            : Analytic tableaux procedure

=== Resolution ===

   basicResolution     : Basic resolution procedure
   resolution          : Resolution with subsumption
   positiveResolution  : Postive resolution
   sosResolution       : Set-of-support resolution

=== Prolog ===

   hornprove           : Basic horn clause prover using backchaining
   prolog              : Prolog interpreter

=== MESON ===

   basicMeson          : Basic Meson procedure
   meson               : Optimized Meson procedure

=== Equality ===

   ccvalid             : Congruence closure validity
   rewrite             : Rewriting
   bmeson              : Meson with equality elimination
   paramodulation      : Paramodulation

=== Decidable problems ===
```

```
      aedecide            : Decide AE problems
      dloQelim            : Dense Linear Orders
      integerQelim        : Presburger arithmetic
      nelopInt            : Nelson Oppen

>
```

When a new algorithm is implemented, the top level is extended by adding a clause to `Main.lhs`. For example, when resolution was finished, we added

```
> import qualified Resolution
```

to the preamble and

```
> resolution :: Command
> resolution = ("resolution",
>                "Resolution with subsumption",
>                runfol Resolution.resolution)
```

to the list of first order logic algorithms. If one wishes to extend the Haskell atp code with their own algorithms, it is fairly easy to see how to add it to the interpreter.