First-Order Logic Resolution Theorem Prover In Haskell

Charles Chiou

April 29, 2001

Abstract

Historically, functional programming languages such as ML had been designed with intentions for usage in theorem proving and expert systems. In this report, I present an implementation of a simple theorem prover in first-order logic using Haskell. Three major parts of this system include: the parser, the knowledge base, and the resolution-based inference engine. Most ideas were taken out of the AI text book by Russel & Norvig[1]. The algorithms and implementation details will be discussed in this report.

1 Introduction

In this project, I implemented a mini theorem prover in first-order logic using Haskell. Several aspects of this project that are interesting and employed concepts and techniques introduced in the course. They include parsing, state monads, IO, pattern matching, and inference rules, etc.

2 Intepreting Language in First-Order Logic

2.1 The Grammar

The production of a valid FOL sentence is described by the following in BNF notation:

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \\
 & | & Sentence\, Connective \\
 & | & Quantifier\, Variable, \ldots\ Sentence \\
 & | & \neg\, Sentence \\
 & | & (Sentence) \\
AtomicSentence & \rightarrow & Predicate(Term, \ldots) \\
 & | & Term = Term
\end{array}
$$

$$
\begin{array}{rcl}
Term & \rightarrow & Function\,(Term) \\
 & | & Constant \\
 & | & Variable \\
Connective & \rightarrow & \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \\
Quantifier & \rightarrow & \forall \mid \exists \\
Constant & \rightarrow & A \mid X \mid John \mid \ldots \\
Variable & \rightarrow & a \mid x \mid s \mid \ldots \\
Predicate & \rightarrow & Before \mid HasColor \mid Raining \mid \ldots \\
Function & \rightarrow & Mother \mid LeftLegOf \mid \ldots
\end{array}
$$

There are actually more features in FOL such as the unique existential quantifier $!\exists$, etc. However, a (appropriately large) subset of FOL was implemented so that the system can handle interesting inferences.

2.2 Monadic Combinator Parser For FOL

The data structure needed to represent an FOL sentence is largely derived from the grammar above. The following data types are defined:

$$
\begin{array}{rcl}
data\,Sentence & = & Connective\,Sentece\,Connective\,Sentence \\
 & | & Quantifier\,Quantifier\,[Variable]\,Sentence \\
 & | & Not\,Sentence \\
 & | & Predicate\,Predicate\,[Term] \\
 & | & Equal\,Term\,Term \\
data\,Term & = & Function\,Function\,[Term] \\
 & | & Constant\,Constant \\
 & | & Variable\,Variable \\
 & | & SkolemFunction\,Int\,[Term] \\
 & | & SkolemConstant\,Int\,[Term] \\
data\,Connective & = & Imply\,Equiv\,And\,Or \\
data\,Quantifier & = & ForAll\,Exists \\
type\,Constant & = & String \\
type\,Variable & = & String \\
type\,Predicate & = & String \\
type\,Function & = & String
\end{array}
$$

The *AtomicSentence* data structure is droppped because *Sentence* is sufficient once the *Predicate* and *Term* constructors are added. In the data type *Term*, two constructors: *SkolemFunction* and *SkolemConstant* are added. The parser does not make use of them, however, functions that convert a raw sentence to conjunctive normal form and implicative normal form would rename some given function names or constants into a unique numbering so that they can be updated when inserted into the knowledge base.

A monadic combinator parser of the form $sentence :: String \rightarrow Parser\ Sentence$ was implemented to convert a raw string that a user types into a system into the a *Sentence* data structure.The standard library "ParseLib.hs" that is distributed with hugs is used for the building of this parser. For handling parse (syntax) errors, the function $parseFOL :: String \rightarrow Maybe\ Sentence$, is used at the top level.

The following is a list of keywords that are reserved: "TRUE", "FALSE", "NOT", "AND", "OR", "FORALL", "EXISTS" (as well as the all-lowercase and capitalized versions of each keyword). It was decided that "TRUE" and "FALSE" are not allowed to be part of a sentence inputed by the user since doing so is redundant.

When matching constants, predicates and functions, the names must begin with an uppercase letter, while variables must begin with a lowercase letter.

In fact, as will be described in later sections, the interactive system that accepts commands from the user also uses monadic combinator parser of the form $parseCmd :: String \rightarrow Maybe\ Cmd$ is used to parse user commands (although its syntax is much simpler.)

2.3 Convert Sentences to Conjunctive Normal Form

After a raw input string has been converted to *Sentence*, it needs to be converted into a normal form, which is also of the type *Sentence*. However, it is much easier to make inferences using resolution in normal form (conjunctive or implicative). The following functions were impleneted to convert an arbitrary sentence into one in CNF:

1) $eliminateImplication :: Sentence \rightarrow Sentence$

This function gets rid of the $\Rightarrow$ connective by replacing $p \Rightarrow q$ with $\neg p \vee q$

2) $moveNotInwards :: Sentence \rightarrow Sentence$

Use de Morgan's law to move the $\neg$ symbol onto atoms (not sentences), replace:

$$\begin{array}{rcl} \neg(p \vee q) & becomes & \neg p \wedge \neg q \\ \neg(p \wedge q) & becomes & \neg p \vee \neg q \\ \neg\forall\, x, p & becomes & \exists x\, \neg p \\ \neg\exists\, x, p & becomes & \forall\, x\, \neg p \\ \neg\neg p & becomes & p \end{array}$$

3) $standardizeVariables :: Sentence \rightarrow Sentence$

Rename variables that are quantified more than once in a sentence, eg: rename $\exists\, x\, Q(x)$ to $\exists y\, Q(y)$ in the original sentence $\forall\, x\, P(x)\ \vee\ \exists\, x\, Q(x)$. This is used to avoid confusions when dropping quantifiers. (NOT implemented).

4) $moveQuantifiersLeft :: Sentence \rightarrow Sentence$

At this stage, we can safely collect all the quantifiers and move them to the left hand side of a sentence.

5) $skolemize :: Sentence \rightarrow Sentence$

Eliminate $\exists$ quantifiers by replacing the quantified variables with skolem constants or skolem functions. In addition, any $\forall$ quantifiers on the left hand side of the sentence may be dropped because any variable left in the sentence must be universally quantified.

6) $distributeAndOverOr :: Sentence \rightarrow Sentence$

Perform the following convertion:

$$\begin{array}{ccc} (a \wedge b) \vee c & becomes & (a \vee c) \wedge (b \vee c) \\ a \vee (b \wedge c) & becomes & (a \vee c) \wedge (b \vee c) \end{array}$$

as well as flatenning nested conjunctions and disjunctions.

7) $toCNFSentence :: Sentence \rightarrow Sentence$

This is basically the top level function that use the listed helper function as pipeline for transforming a raw sentence into one in conjunctive normal form

2.4 Convert Sentences to Implicative Normal Form

Once *Sentence* is in conjunctive normal form, it is easy to convert it in implicative normal form. The function $toINFSentence :: Sentence \rightarrow Sentence$ does exactly that. It first calls $toCNFSentence$ and then collects positive literals and negative literals into two list and puts the negative literals on the left hand side, positive literals on the right hand side of the $\Rightarrow$ connectiv, where the literals on the LHS are connected conjunctions and the literals on the RHS are connected by disjunctions.

$$\neg a \vee \neg b \vee c \vee d \quad becomes \quad a \wedge b \Rightarrow c \vee d$$

2.5 Handling Skolemization

When assignin names of skolem functions or constants, they must not be names that have not appear in the knowledge base before. However, since the normal form conversion functions were written without references to a working knowledge base, it is not possible to check for this name collision. Instead, this problem is deferred until the sentence is actually inserted into the knowledge base. Therefore, an integer counter is used to identify the order of skolem functions or constants within a sentence, which the knowledge base can later use as a relative reference for assigning actual names.

3 Knowledge Base

We use a state monad to represent a knowledge base. A set of useful functions are then used for manipulating and querying the states.

3.1 State Representattion

In the knowledge base, we want to be able to keep track of the following:

- Normalized FOL sentences and its "module" association.
- Skolem variable assignment counter.
- A list of modules.

$$\begin{aligned}
type\,State &= ([(ImplicativeNormalForm, Int)], SkolemCount, [FolModule]) \\
type\,SkolemCount &= Int \\
type\,FolModule &= (String, Int)
\end{aligned}$$

3.1.1 Representing FOL sentences in implicative normal form

The function $toINFSentence :: Sentence \rightarrow Sentence$ converts a sentence in implicative normal form of type $Sentence$. However, the following type is better for storage and making inferences in the knowledge base:

$$\begin{aligned}
data\,ImplicativeNormalForm &= INF\,[NormalSentence]\,[NormalSentence] \\
data\,NormalSentence &= NFNot\,NormalSentence \\
&\mid NFPredicate\,Predicate\,[Term] \\
&\mid NFEqual\,Term\,Term
\end{aligned}$$

Similar data types have also been defined for conjunctive normal form representation.

Then, the function $toINF :: Sentence \rightarrow ImplicativeNormalForm$ performs the neccessary conversion. After which, it can be stored into the knowledge base state monad.

3.1.2 Skolem variable count

This is simply an integer counter that is incremented each time a variable has been assigned. The variable name strings are appended with these numbers to ensure that there are no duplicates.

3.1.3 An FOL Module

To faciliate easier "logic debugging", every sentence inserted into the KB is associated with a "logic module", which can be seen as nothing more than a grouping of related sentences together. A file containing FOL sentences can be loaded into the KB with a load command. When they are inserted into the KB, an integer tag is applied (the tuple $(ImplicativeNormalForm, Int)$). Since sentences can be entered through the command line niterface, they are tagged as "USER" modules.

3.2 Interactive System and State Transition

The interactive system module (interactive.hs) is part of of the hugs distribution. Exactly how it works is a little mysterious to me. However, there are enough examples so that I can use it effectively. The process of the interactive system is:

1. Display prompt
2. Get user command string
3. Parse command string and possibly display syntax error
4. Call the command processor interactive functions, which performs output and repeat this cycle.

Inside each command processor function, we need to pass in the current state, call the $runState :: State \rightarrow StateMonad\,a \rightarrow (a, State)$ function and pass the new resultant state into the next cycle. When the system is started, the $zeroKB$ state, which is the initial state, is passed in.

3.3 Commands

The following commands are supported in the user interface:

- Tell <sentence> - Inserts a sentence into the KB.
- Ask <sentence> - Proves a sentence (True, False).
- Delete [index] - Deletes an entry from the KB.
- Load <file> - Loads a file as a module into the KB.
- Unload <file> - Unloads a previously loaded module.
- Reset - Resets the KB to empty.
- Show - Displays the contents of the KB.
- Quit - Quits this program.
- Help - Displays a help message.

3.4 Command Parser

The command parser constructed using monadic combinator parser library just like the FOL sentence parser.

3.5 Managing Modules and IO

The commands "load" reads in contents of a file and parses each line. If valid FOL sentences are present, then they are inserted into the knowledge base as a module named with the filename. Currently, it does not support comments in the files that are to be loaded. However, parse errors simply ignores the line. The "unload" command takes a module name (or equivalently, the filename) as an arguments and looks up the table and deletes each enrty in the knowledge base that matched the index.

4 Making Inferences in First-Order Logic

In the file "Resolution.hs", the following data types are defined:

$$\begin{aligned} type\, Subst &= [(Variable, Term)] \\ type\, SetOfSupport &= [Unification] \\ type\, Unification &= (ImplicativeNormalForm, Subst) \end{aligned}$$

As an option, it is possible to make inferences in conjunctive normal form or implicative normal form. This implementation chose the latter since a few other popular FOL inference engines written in LISP make inferences in implicatibe normal form too.

4.1 Unification

The unify function takes two atomic sentences p, and q, and return a a substitution that makes p, and q "look" the same. If there are no substitutions that are consistent with the knowledge base, then the unification fails.

$$Unify(p, q) = \theta\, where\, Subst(\theta, p) = Subst(\theta, q)$$

The function $\begin{array}{c} tryUnify :: [NormalSentence] \to [NormalSentence] \to \\ Maybe\,(([NormalSentence], Subst), ([NormalSentence], Subst)) \end{array}$ does exactly that. It takes two lists of sentences and tries to unify them as much as possible and return the substitutions made with each sentence .

4.2 Resolution

The generalized resolution for disjunctions for literals p_i and q_i where $Unify(p_j, \neg q_k) = \theta$:

$$\frac{\begin{array}{c} p_1 \vee \cdots p_j \cdots \vee p_m, \\ q_1 \vee \cdots q_k \cdots \vee q_n \end{array}}{Subst(\theta, (p_1 \vee \cdots p_{j-1} \vee p_{j+1} \cdots p_m \vee q_1 \cdots q_{k-1} \vee q_{k+1} \cdots \vee q_n))}$$

The generalized resolution for implications for literals p_i , q_i, r_i, s_i where $Unify(p_j, q_k) = \theta$:

$$\frac{\begin{array}{c} p_1 \wedge \cdots p_j \cdots \wedge p_{n1} \Rightarrow r_1 \vee \cdots r_{n2} \\ s_1 \wedge \cdots s_{n3} \Rightarrow q_1 \vee \cdots q_k \cdots \vee q_{n4} \end{array}}{Subst(\theta, (p_1 \wedge \cdots p_{j-1} \wedge p_{j+1} \cdots p_{n1} \wedge s1 \wedge \cdots s_{n3} \Rightarrow r_1 \vee \cdots r_{n2} \vee q_1 \vee \cdots q_{k-1} \vee q_{k+1} \vee \cdots \vee q_{n4}))}$$

The function $resolution :: Unification \rightarrow Unification \rightarrow Maybe Unification$ takes two arguments of type Unification (containing the sentence in implicative normal form and a substitution). It tries to unify the two sentences (with the right hand side of the first sentence and the left hand side of the second sentence, as according to the inference rule) and return the result.

4.3 Demodulation

The following rule simplifies sentences in the KB that contain equalities once substutions has been found:

$$\frac{x = y, (\cdots z \cdots)}{(\cdots Subst(\theta, y) \cdots)}$$

The corresponding function is $demodulate :: Unification \rightarrow Unification \rightarrow Maybe Unification$

4.4 Proving using Refutation

In this proof strategy, we negate the sentence asked by the user and try to prove the question by contradiction. If we end up with $True \Rightarrow False$, then the proof is complete. In other words, suppose that we try to prove Q, then it must be that the knowledge base implies Q, so $(KB \wedge \neg Q \Rightarrow False) \Leftrightarrow (KB \Rightarrow Q)$.

In implementing the function
$prove :: SetOfSupport \rightarrow SetOfSupport \rightarrow [ImplicativeNormalForm] \rightarrow (Bool, String)$, we use the "set-of-support" strategy to guide the search (of possible unification and resolution steps) towards the proof. In whichwe have the implicative normal form sentence to be empty on both sides of the implication symbol, $INF\,[]\,[]$, *ie.* $True \Rightarrow False$.

The set-of-support is first constructed by negating the question sentence to be proved. At each step of resolution, a sentence from the set-of-support is combined with another sentence, and adds the resolvant into the set of support. The claim [1] is that if the set of support is much smaller than the knowledge base then the search space would be reduced dramatically. In practice, I have not tested the system on a really complex knowledge base, therefore there are no empical results on how this strategy is efficient compared to others. The pitfall is, however, that if the wrong set-of-supports are chosen, then th proof algorithm is incomplete.

One added bonus of using this strategy is that we can flatten the SetOfSupport data (list of unifications) and return the result of the proof as a string. This can be displayed for visual debugging or verification purposes.

5 Example Run: Did Curiosity Kill The Cat?

```
Main> main
Welcome to First Order Logic Prover!

Prover> load cat.kb
Loaded 6 entries from cat.kb:
Exists x (Dog(x) AND Owns(Jack,x))
ForAll x (Exists y (Dog(y) AND Owns(x,y)) => AnimalLover(x))
ForAll x (AnimalLover(x) => ForAll y (Animal(y) => NOT (Kills(x,y))))
(Kills(Jack,Tuna) OR Kills(Curiosity,Tuna))
Cat(Tuna)
ForAll x (Cat(x) => Animal(x))

Prover> show
1) [MOD:cat.kb] True => Dog(?SKC_1)
2) [MOD:cat.kb] True => Owns(Jack,?SKC_1)
3) [MOD:cat.kb] Dog(y) AND Owns(x,y) => AnimalLover(x)
4) [MOD:cat.kb] AnimalLover(x) AND Animal(y) AND Kills(x,y) => False
5) [MOD:cat.kb] True => Kills(Jack,Tuna) OR Kills(Curiosity,Tuna)
6) [MOD:cat.kb] True => Cat(Tuna)
7) [MOD:cat.kb] Cat(x) => Animal(x)
Prover> ask Kills(Jack,Tuna)
Result: *No*
Would you like to see the proof (y/n)?
Proof:
Kills(Jack,Tuna) => False
True => Kills(Curiosity,Tuna)
Animal(Tuna) AND AnimalLover(Curiosity) => False
Dog(y) AND Owns(Curiosity,y) AND Animal(Tuna) => False
Cat(Tuna) AND AnimalLover(Curiosity) => False
Owns(Curiosity,?SKC_1) AND Animal(Tuna) => False
Cat(Tuna) AND Owns(Curiosity,y) AND Dog(y) => False
Dog(y) AND Owns(Curiosity,y) AND Cat(Tuna) => False
AnimalLover(Curiosity) => False
Cat(Tuna) AND Owns(Curiosity,?SKC_1) => False
Owns(Curiosity,?SKC_1) AND Cat(Tuna) => False
Owns(Curiosity,y) AND Dog(y) => False
```

```
Dog(y) AND Owns(Curiosity,y) => False
Owns(Curiosity,?SKC_1) => False

Prover> ask Kills(Curiosity,Tuna)
Result: *Yes*
Would you like to see the proof (y/n)?
Proof:
Kills(Curiosity,Tuna) => False
True => Kills(Jack,Tuna)
Animal(Tuna) AND AnimalLover(Jack) => False
Dog(y) AND Owns(Jack,y) AND Animal(Tuna) => False
Cat(Tuna) AND AnimalLover(Jack) => False
Owns(Jack,?SKC_1) AND Animal(Tuna) => False
Dog(?SKC_1) AND Animal(Tuna) => False
Cat(Tuna) AND Owns(Jack,y) AND Dog(y) => False
Dog(y) AND Owns(Jack,y) AND Cat(Tuna) => False
AnimalLover(Jack) => False
Animal(Tuna) => False
Cat(Tuna) AND Owns(Jack,?SKC_1) => False
Cat(Tuna) AND Dog(?SKC_1) => False
Owns(Jack,?SKC_1) AND Cat(Tuna) => False
Owns(Jack,y) AND Dog(y) => False
Dog(?SKC_1) AND Cat(Tuna) => False
Dog(y) AND Owns(Jack,y) => False
Cat(Tuna) => False
Owns(Jack,?SKC_1) => False
Dog(?SKC_1) => False
True => False
Prover> quit
Bye!
```

6 Summary

The implementation works reasonably well. The functions of parsers, state monad knowledge base, and the interactive system work great. One flaw is that instead of composing monads in an elegant way in the file loading function (when IO and state monads are used), a more ugly hack of $runStateRun :: State \to StateMonad\, a \to (a, State)$, and $hugsIORun :: IO\, a \to Either\, Int\, a$ are used and some of the hidden primitive functions in Prelude had to be copied over. On the other hand, improvements can be made to the efficiency of the (CNF and INF) normalization functions. In particular, checking for redundant literals has not been implemented (*i.e.* reducing $p \wedge p$, $p \vee p$, $p \Rightarrow p$, and $p \Leftrightarrow q$ to p).

The most difficult and quirky part of the project is the algorithm for making resolutions and proving using refutation. Presently, it is able to handle relatively simple set of knowledge bases. How well (or badly) it would perform when the knowledge base is big and the complexity of the proof is higer is questionable. Refining the efficiency of the algorithm and perhaps use a different approach (other than choosing the set-of-support strategy) could be the next imrovement.

References

[1] Stuart Russel, *"Peter Norvig, Artificial Intelligence, A Modern Approach"*, Prentice-Hall. Inc. 1995

[2] Graham Hutton, Erik Meijer, *"Monadic Parser Combinator"*, Technical Report NOTICS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996

[3] L.C. Paulson, *"ML for the working programmer"*, Cambridge University Press 1991, 1996

[4] Paul Hudak, *"The Haskell School of Expression"*, Cambridge University Press 2000

[5] Mark P. Jones, "Mini Prolog Version 1.5g - A Simple Prolog Intepreter", Hugs Demo library, 1991, 1996