CMPT 477 Final Project Report: Automated Theorem Prover

Ethan MacDonald, Neil Doknjas, Jayden Brown,
Bruce Choi, Syed Humza Shah

Design

The prover supports full first-order logic. The user inputs a first-order logic formula as a text file. The module Parser.hs parses this text into a Formula abstract syntax tree. The datatypes Term, Formula, Literal, Clause, ClausalForm, and Subst are defined in Datatypes.hs. Note that ClausalForm = [Clause] = [[Literal]]. The validity of the Formula is then decided using first-order resolution. First, the Formula must be negated. The proceeding resolution steps are performed by a series of modules. FreeVariables.hs removes all free variables from the Formula by existentially quantifying them. NegationNormalForm.hs converts the Formula to negation normal form. Renamer.hs renames all quantified variables to ensure none share the same name. MoveQuantifiers.hs moves all quantifiers to the front of the Formula. Now, the Formula is in prenex normal form. Skolemize.hs converts the Formula to skolem normal form, where existential quantifiers are dropped and their variables replaced by skolem functions. ClausalFormConverter.hs converts the Formula to conjunctive normal form, and returns it as a ClausalForm. This module uses Tseitin's transformation (adapted to FOL) for the CNF conversion. Lastly, Resolution.hs attempts to find a resolution refutation. That is, it uses unifiers to derive new resolvents from the Clauses in the ClausalForm until an empty Clause is derived or no more unique resolvents can be derived. In the case of the former, the program will output "Valid!", and, in the case of the latter, "Invalid!". Since determining validity of first-order logic formulae is only semi-decidable, the resolution process is only guaranteed to terminate when the user inputs a valid formula. Otherwise, it may never terminate. Resolution.hs is assisted by MGU.hs, which provides a most general unifier for a given pair of Literals.

Implementation

We implemented our project in Haskell; as the functional paradigm is useful when working with recursive data structures. Below are some implementation details for each of our modules:

- Datatypes.hs

  We define Literal (an atom or its negation) with a distinct datatype. This is cleaner and easier to work with than defining Literal as a Formula that is either a Predicate or a Not of a Predicate.

- Parser.hs

  We import Megaparsec for parsing. This made implementing much easier since the library provides parsers that can be combined to consume whitespace, consume alphabet strings, and parse inside of things like parentheses.

- FreeVariables.hs

  The module finds free variables by recursively keeping a set of variables bound to a quantifier. Importantly, all occurrences of a quantifier's variable within its scope are set to bound, and variables of the same name which are outside of the quantifier's scope are not set to bound. After finding every free variable that is not in the bound set, the module existentially quantifies these variables by folding ThereExists over the set.

- NegationNormalForm.hs

  The NNF module simply recursively applies the negation formulae taught in class. It uses a polarity argument to "carry" the negation deeper into the formula, and when it reaches a predicate, it returns it with or without a prepended Not depending on the polarity.

- Renamer.hs

  At a high level, Renamer.hs goes through the formula while maintaining a mapping that records how many times each instance of the original variable name has been bound by a quantifier. When it encounters a variable that has not yet been renamed, it is renamed to a fresh identifier of the form "[quantificationCount]_[originalName]". This guarantees that each variable is uniquely renamed.

- MoveQuantifiers.hs

  MoveQuantifiers.hs proceeds in two phases. First, it traverses the formula and extracts all quantifiers into a tree structure, removing them from the main formula while preserving their scope. This tree captures the nesting and ordering of quantifiers. Second, the tree is flattened and folded back onto the formula, producing an equivalent formula in which all quantifiers are moved to the front with their scopes preserved (prenex normal form).

- Skolemize.hs

  The module removes existential quantifiers by replacing existentially quantified variables with fresh Skolem functions whose arguments are the universally quantified variables that appear before the relevant existential quantifier. After removing the existential quantifier, the new Skolem function replaces every occurrence of the variable it represents. Renamer.hs already renamed all quantified variables to be unique, so any potential issues that could arise from shadowing will be absent.

- ClausalFormConverter.hs

  First, the universal quantifiers are removed from the front of the Formula. It is assumed that the Formula was skolemized, so these should be the only quantifiers. We also create qvars, a list of the names of all quantified variables. Next, Tseitin's transformation is performed recursively, directly outputting a ClausalForm. In first order logic, atoms are predicates as opposed to propositional logic where atoms are variables. So, when we introduce an auxiliary predicate, its arguments must contain all of the free (formerly quantified) variables found in the subformula it represents. We do this the easy way: every auxiliary predicate will have qvars as its argument list. Each auxiliary predicate is named after a 16-bit integer. This is done using a name range. For example, if AND(F G) is encountered, the auxiliary predicate representing this conjunction is named minName. Now, auxiliary predicates representing subformulas of F can be given names in [minName + 1, mid], while those representing subformulas of G can be given names in [mid + 1, maxName].

- MGU.hs

  This module uses the Martelli-Montanari algorithm to find a most general unifier (MGU) for two Literals. From the two Literals, the algorithm creates a system of equations that relate the terms of the first Literal with the terms of the second. The system is stored as a list of term tuples (t, s) which each represent t = s. Then, four special transformations are appropriately and repeatedly applied to each tuple. The result will either be that the two Literals cannot be unified or that the list has reached a solved form (no more transformations can be applied). In this latter case, the MGU is the resulting list, with tuples now interpreted as mappings from variables to terms. Note that in our implementation, the signs of the input Literals are ignored. This allows the module to be used for both unification (opposite signs) and factoring (same signs).

- Resolution.hs

  This module performs resolution logic as described in class lectures on the list of Clauses in clausal form. Three important concepts are implemented. First, the binary resolution inference rule, which resolves two Clauses when they have a pair of oppositely-signed, unifiable Literals between them. Once resolved, a new resolvent Clause is created with the remaining Literals. Set-logic is used to prevent redundant resolutions. Second, the module has first-order factoring. If two unifiable, same-sign, same-Clause Literals exist, factoring will simplify the Clause by applying an MGU of the two Literals. Lastly, tautological Clauses are removed throughout the resolution process. A tautological Clause is one containing two oppositely-signed Literals that are unifiable. These Clauses are removed because they are trivially true and, therefore, will not contribute to deriving an empty Clause.

Results

We tested some basic propositional logic theorems including a De Morgan Law, Modus Ponens, Modus Tollens, idempotence of conjunction, and transitivity of implies. These were all determined to be valid. We also tested the formulas found in the files `input0.txt` through `input10.txt`. The formula contained in `input2.txt` is invalid and terminates because the resolvent search space is finite. Every other file contains a valid formula and terminates. We also tested `good.txt`. If factoring were not implemented, the program would not terminate for this file despite it containing a valid formula. Our test files can be found in `\tests`.

Though our prover will eventually verify any valid formula, it can be extremely slow for certain formulae. An example of this is `HS.txt`, which contains our "transitivity of implies" theorem. It is a simple looking formula that is obviously valid. However, it took our program about 5 hours to prove this. So, in a future version, we may want to implement more resolution strategies such as unit preference, set of support, or subsumption to improve the program's efficiency.

Instructions

The user provides a text file containing one first-order logic formula F which they would like to verify. If F is valid, the program will eventually terminate with "Valid!". If F is invalid and the resolvent search space is finite, then the program will terminate with "Invalid!". If F is invalid and the resolvent search space is infinite, then the program will run forever.

- Input formatting:

Type	Syntax	Description
Term	`VAR[name]`	Variable called `name`.
	`OBJ[name]`	Object called `name`.
	`FUNC[name](terms)`	Function called `name`. `terms` is a sequence of Terms separated by spaces.
Formula	`TRUE`	Logical true.
	`FALSE`	Logical false.

	PRED[name](terms)	Predicate called name. terms is a sequence of Terms separated by spaces.
	AND(F G)	Logical F ∧ G. F and G are Formulas.
	OR(F G)	Logical F ∨ G. F and G are Formulas.
	NOT(F)	Logical ¬F. F is a Formula.
	IMPLIES(F G)	Logical F → G. F and G are Formulas.
	IFF(F G)	Logical F ↔ G. F and G are Formulas.
	A{x}(F)	For all x, F. x is the name of a variable and F is a formula.
	E{x}(F)	There exists an x such that F. x is the name of a variable and F is a formula.

- Notes on syntax:

  Keywords are case sensitive.
  Names are strictly strings of uppercase and lowercase alphabet characters.
  Whitespace does not matter. Feel free to put newlines and spaces anywhere other than in the middle of a name or keyword string.
  If your input is not well-formed, the program will hopefully fail with some error message.

- Executing the program:

  An executable for x86_64 Windows systems is provided in the build folder. Using Command Prompt, navigate to \build and enter the following:

  ```
  ATP_x86_64_Win.exe [Input File Path]
  ```

  For other systems, the project needs to be compiled. We suggest using Cabal. Once you have installed Haskell and Cabal, open your shell in \AutomatedTheoremProver and enter the following to build and run:

  ```
  cabal run AutomatedTheoremProver -- [Input File Path]
  ```

- Source code:

  https://github.com/JE-MacDonald/AutomatedTheoremProver

Division of Work

- Ethan MacDonald:

  Implemented Parser.hs and ClausalFormConverter.hs.
  Wrote most of Main.hs.
  Designed the syntax for the user input.
  Created the datatypes Term, Formula, Literal, Clause, and ClausalForm.

- Neil Doknjas:

  Implemented FreeVariables.hs and Skolemize.hs.

- Jayden Brown:

  Implemented NegationNormalForm.hs, Renamer.hs, and MoveQuantifiers.hs.
  Wrote Show overrides for the Formula and Term datatypes.

- Bruce Choi:

  Implemented MGU.hs.
  Modified Renamer.hs to add support for variable shadowing.
  Created Subst type (to represent substitutions).
  Added I/O functionality for Main.hs

- Syed Humza Shah:

  Implemented Resolution.hs.

References

CMPT477 Lecture Slides. Professor Yuepeng Wang. Simon Fraser University. 2025.

Wolfram Alpha CNF Converter: www.wolframalpha.com/input?i=CNF+(v+<->+(p+and+q))

Haskell Libraries Documentation: https://downloads.haskell.org/ghc/latest/docs/libraries/

Artificial Intelligence: A Modern Approach, Fourth Edition, Global Edition. Stuart J. Russell, Peter Norvig. Pearson Education Limited. 2022. Chapter 9.5: Resolution. Accessible: http://lib.ysu.am/disciplines_bk/efdd4d1d4c2087fe1cbe03d9ced67f34.pdf

Wikipedia: Algebraic Data Type: https://en.wikipedia.org/wiki/Algebraic_data_type

Implementing Unification Algorithms in Haskell. Yue Li. University of Dundee. September 2015. Accessible: https://yuelipicasso.github.io/Reports/ThesisMainText.pdf

Tree Proof Generator: https://www.umsu.de/trees/

An Efficient Unification Algorithm. Alberto Martelli, Ugo Montanari. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2. April 1982. Sections 1-2. Accessible: https://dl.acm.org/doi/pdf/10.1145/357162.357169