Propositional Theorem Proving Using Resolution in SML

Joe Snikeris
West Chester University of Pennsylvania
Spring 2009

Table of Contents

Introduction

This paper serves as documentation for a propositional theorem proving program written in the Standard ML (SML) programming language. Its secondary purpose is to provide the reader with an introduction to the resolution technique for proving theorems of the propositional logic. The program and this paper are the result of an independent research project by a senior-level undergraduate student in computer science. The intended audience is another undergraduate student in computer science with a basic understanding of mathematical logic.

Data Representation

The initial design of the representation for formulas included a separate datatype for each logical connective. For example, there was a separate datatype for propositional variables, negations, conjunctions, etc. This idea was scrapped because SML's pattern matching features make such a distinction useless. Instead, there is one datatype to represent all formulas, and it includes constructors for each of the logical connectives. This has the advantage that in order to check if a formula is a negation, for example, one only has to match on the pattern `(~ f)`. Additionally, it was discovered that SML allows constructors to immediately precede variables (with no space in between) so one can simply write `~f` to denote the negation of `f`. This is a more natural way to express negations, especially when `f` is a propositional variable.

The datatype includes five constructors to represent different propositional variables. These could obviously be extended if more than five variables are necessary. Also, an alternate implementation might represent propositional variables as strings so that the user could input propositions in sentence form instead of symbolically. Instead of the user making note of the fact that the propositional variable `P` represents some proposition like 'It is raining', the functionality could be extended so that the string, 'It is raining' is a propositional variable itself.

SML allows constructors to be either prefix or infix. This further allows formulas to be represented more naturally since conjunctions, disjunctions, conditionals, and biconditionals are usually written in infix notation. This is the same way that most arithmetic expressions are written. For example, to represent the disjunction of the propositional variables `P` and `Q`, one writes `P v Q` similarly to how, in arithmetic, one might write `x + y`. Of course, since negation only involves one variable, as it does in arithmetic, it has not been modified from the default prefix notation.

The symbols for the constructors representing the logical connectives were chosen to correspond as closely as possible to traditional notation. One exception is the symbol for a conditional. SML reserves `->` for representing function types, so an alternative had to be chosen. Since `-->` is conveniently the same number of characters as the biconditional symbol, `<->`, it was selected.

Conjunctive Normal Form

Propositional resolution requires that formulas be converted to **Conjunctive Normal Form** (CNF) before being resolved. To understand CNF, a couple definitions are useful:

- A **literal** is a formula that is either a propositional variable or the negation of one. For example, `P` and `~P` are literals while `P v Q` and `~(P v Q)` are not.

- A **clause** is a disjunction of literals. For example, `P v ~Q v R` is a clause. `P` and `~P` are also clauses since they are the disjunction of a single literal.

A formula in CNF has the following properties:

- It is made up of only conjunctions, disjunctions, negations, and propositional variables. A formula containing conditionals and biconditionals is not in CNF.
- All negations immediately precede propositional variables.
- The formula is a conjunction of clauses.

It is rare that a given formula just happens to be in CNF so it is extremely useful, and almost necessary, for any resolution-based theorem proving system to implement functions that convert formulas into CNF. Since conversion into CNF can in some cases lead to an exponential increase in the size of the formula, requiring the user to convert formulas to CNF by hand would be imprudent.

The conversion involves four steps performed sequentially. These steps are naturally represented as separate functions, and a final function composes the four of them into a function that converts a formula into CNF.

1. Removing Conditionals

The first step is to replace conditionals and biconditionals with their logical equivalents. A conditional of the form `P --> Q` gets converted to `~P v Q`. Biconditionals can be converted into one of two equivalent forms. For example, `P <-> Q` can be converted into:

```
(P & Q) v (~P & ~Q)
```

or

```
(P v ~Q) & (Q v ~P).
```

The former was chosen since its negation readily transforms into CNF. Because resolving a formula `f` involves transforming the negation of `f` into CNF, this choice might be more efficient. The function that implements this step is named 'remConds'.

2. Push Negations

The next step is to push negations inwards until they can only be found immediately preceding propositional variables. This is done through the use of De Morgan's laws:

- `~(P v Q) = ~P & ~Q`
- `~(P & Q) = ~P v ~Q`

It is important to note (in case this function is used outside the scope of this program) that this function assumes that there are no conditionals and biconditionals in the formula. Since De Morgan's laws do not apply to those operators, formulas that contain them must first be converted with the above mentioned function. This property holds for the remaining two functions as well. The function that implements this step is appropriately named 'pushNegations'.

3. Remove Double Negations

After pushing all negations inwards, situations often arise where a propositional variable is preceded by

two or more negations such as `~(~P)`. The function 'remDoubNeg' takes advantage of the fact that the double negation of a formula is always equivalent to the formula itself. This simplifies the formula in the same manner a person performing resolution by hand would.

4. Distribute

The final step of converting a formula into CNF is to distribute conjunctions over disjunctions until the formula is a conjunction of clauses as required. This operation is similar to multiplying two polynomials. For example,

`(P & Q) v (R & S)`

becomes

`(P v R) & (P v S) & (Q v R) & (Q v S).`

This step is implemented by the function 'distribute'.

Propositional Resolution

Propositional resolution is a theorem proving technique that is especially suited for automation. As opposed to more natural deduction systems, resolution requires only one rule applied over and over again to determine if a given formula is tautologous or not.

The Algorithm

The technique of resolution can best be described by an example of its use. Given two clauses `P v Q`, and `~P v R`, a third clause `Q v R` can be inferred. This is because if `P` is false, then `Q` must be true from the first clause. If `P` is true, then `R` must be true from the second clause. Since `P` must be either true or false, `Q` must be true or `R` must be true. Symbolically:

`((P v Q) & (~P v R)) --> (Q v R)`

Having two clauses, one containing a propositional variable and the other containing its negation, allows a new conclusion to be drawn. The new clause is simply the conjunction of the old clauses with the matching literals removed.

Surprisingly, this one rule, applied over and over again, is enough to prove whether or not any given formula is a theorem of the propositional logic. The strategy is to negate the theorem being proved and then to derive a contradiction from it using the resolution rule. This technique is known as proof by contradiction. By showing that the negation of the theorem leads to a contradiction, one is showing that it is impossible for the theorem to be false. A more detailed explanation of the process follows.

As previously mentioned, the first step is to negate the formula under consideration and put the result into CNF. The next step is to take each clause and insert it into a database of clauses. The database should have no duplicate clauses, and each clause should have no duplicate literals. Requiring this reduces the size of each clause and the size of the database without affecting the truth value of the database under any interpretation. Next, each clause in the database should be taken one at a time and an attempt should be made to resolve that clause with all the other clauses in the database. If two clauses do not resolve, the process should move on to the next clause. If they do resolve, the conclusion drawn from the resolution rule, or the resolvent, should be added to the database.

A contradiction occurs when both a propositional variable and it's negation occur as clauses in the database. The database can be viewed as a collection of conclusions that have been drawn from the initial formula. So if `P` and `~P` both occur as clauses in the database, it is equivalent to saying that `P` is true and `P` is false. This is clearly a contradiction since a propositional variable can only have one truth value. The resolution can halt here because the objective has been met: a contradiction has been concluded from the negation of the formula under consideration. Therefore, the formula is a theorem of the propositional logic. If an attempt has been made to resolve every clause in the database with the rest and a contradiction has not been encountered, then the formula under consideration is not a theorem.

The Implementation

Sets and a Set of Sets

After the formula has been negated and converted into CNF, the database must be built. A major design decision in the project involved deciding how best to represent the database in terms of the language being used. Given the rich collection of data structures in the SML/NJ library a set representation was settled on, in particular, a set implemented through the use of a binary tree. There were several reasons why this approach was chosen.

The most important is that sets already have the property that they cannot contain duplicates. As mentioned in the previous section, it is useful for the database to contain no duplicate clauses and for clauses to contain no duplicate literals. Because of this, I feel that representing the clauses as sets of literals and representing the database as a set of clauses (a set of sets) is the most natural representation.

Also, the underlying tree representation of the database is also more efficient than a representation using SML's built in lists. To resolve two clauses, one must check if the negation of each element of the first is a member of the second. To accomplish this with plain lists requires a complete traversal of the second list for each element in the first in the worst case, that is, when the two clauses do not resolve. If the first list has length m, and the second has length n, this requires m*n comparisons. If each is represented by a tree, on the other hand, in the worst case one must still traverse the entire tree representing the first clause; however, each literal in the first will only need to be compared to O(log n) elements of the second clause due to the ordered nature of the tree. In this particular implementation, there can only be at most ten literals in each clause, so the performance gain might not be realized; however if the implementation was extended to allow an arbitrary number of literals as discussed earlier, these considerations might be more significant.

Creating the Database

Turning a formula in CNF into a database involves two functions. The first, called 'clauseToSet' takes a clause (as a formula), and adds each literal to a set. Of course, the clause should be a disjunction of literals, and nothing else. 'clauseToSet' raises an exception, 'NotAClause' if this is not the case.

The second function, 'cnfToDb', takes a formula in CNF, converts each clause into a set with 'clauseToSet' and unions these sets together to create a set of clauses that represents the database. 'cnfToDb' handles any 'NotAClause' exceptions that 'clauseToSet' might throw by printing an informative message, and treating the offending clause as if it were the empty clause.

Resolving Two Clauses

SML's 'option' type allows for a resolve function that both checks whether two clauses resolve, and returns their resolvent if they do. This is accomplished by walking through each element of the first clause and checking if its negation is a member of the second. If it is, the matching literals are removed and the union of the resulting two clauses is returned.

Resolving the Database

Resolving the entire database is accomplished by taking the first clause and attempting to resolve it with the remaining clauses in the database, adding resolvents to the database as they arise. One pass of this process is accomplished with the function 'resolveOnce'. Resolving the entire database results in a conclusion of whether or not the database is consistent; therefore, this function is named 'isConsistent'. It passes the first clause of the database to 'resolveOnce' until either the empty clause has been encountered, or there are no more clauses to be passed. In the former case, the database is not consistent since the empty clause is equivalent to an empty disjunction which is not satisfiable.

Top Level Function

The top level function, isTautology is simply the composition of the functions just described. It can be read as, "A formula is a tautology when its negation in CNF, taken as a set of clauses, is not consistent."

Implementation Issues

A number of issues were raised while implementing the program. The more interesting ones will be discussed.

Removing P v ~P from the Database

Clauses that contain a negation and its literal, can safely be removed from the database. These include clauses like `{P, ~P}` and `{R, Q, S ~Q}`. These clauses add no meaning to the database because their truth value is always true. In determining whether a set of sentences is consistent, the goal is to ensure that there is at least one interpretation under which the truth value of all sentence in the set is true. If one sentence of these sentences is always true, it can obviously be omitted from the set without affecting its consistency. In other words, given a clause that contains a literal and its negation, c, and a set of clauses G:

`G⊔{c}` is consistent if and only if G is consistent.

By omitting c from the set, less computation is required, and in the case of this implementation a particular bug is remedied. A description of this bug follows.

Suppose that the database contains the clauses `{P, Q}` and `{Q, ~Q}` among others, and suppose that the former of these is the next to be resolved through. `{P, Q}` is removed from the database and an attempt is made to resolve it with each remaining clause in the database.

`{P, Q}` and `{Q, ~Q}` do indeed resolve -- to `{P, Q}`. This becomes a problem if, `{P, Q}` happens to be added to the tree at a position where it will be resolved through before `{Q, ~Q}` does. For trees and sorted lists, this will be the case. If resolvents are always added to the "back" of the set, so to speak,

this is not an issue and the question of whether to remove `{P, ~P}` clauses is one of efficiency, not correctness.

FormulaSet.empty and FormulaSet.isEmpty

In experimenting with the different set representations available in SML/NJ's library a minor issue arose. In a prior version of this implementation, isConsistent checked if a clause was the empty clause by comparing it to FormulaSet.empty with the '=' operator. The original implementation used splay sets which are an equality type, so this comparison caused no issues. However, when switching to a sorted list or binary tree set implementation, equality type errors were raised. Strangely, although all three of these structures are described by the same signature, ORD_SET, one of them is an equality type while the other two are not. This issue was remedied by the use of the isEmpty function which is also described in the ORD_SET signature.

Splay Sets

The splay tree representation for sets was initially chosen since splay trees have the property that recently accessed elements can be quickly accessed again. In the 'resolve' function, if the negation of a literal in the first clause is found to be a member of the second, that member is then deleted from both the first and second. In the case of the first deletion, regardless of which tree implementation is chosen, the literal will already be at the top of the tree and can be deleted in O(1) time. This is because the first literal is always chosen from the first clause. For the second clause, the matching literal may be anywhere in the tree. Since a splay tree rearranges itself so the most recently accessed element is at the root, the matching literal will always be at the root (since it was just accessed) and it can be immediately deleted without having to traverse the tree again. However, despite this, it was later discovered that this rearranging property is actually undesirable in this application.

Suppose that a clause is resolved through the database and there is at least one resolvent. The resolvent gets added to the database and a new clause is selected to be resolved through. In the case of a splay tree, the new clause selected will always be the last clause added to the database, that is, the last resolvent. For many formulas, this does not cause a problem. However, because of this property, situations can arise where the database oscillates between several states, the program never terminating. For an example of this behavior, see Appendix C.

Choosing a set representation that does not have the property that new elements are always the next to be chosen appears to remedy this issue.

Bugs

There are no known bugs at this time

Possible Improvements

- The majority of the data representation code could be contained in a structure to avoid cluttering the top-level environment with functions that are probably not useful to a user.
- The benchmarking of each set representation might be an interesting extension to this project since SML/NJ's library has several set implementations: sorted lists, binary trees, splay trees

and red-black trees. Also a linked list set structure could be implemented and benchmarked.

- The propositional variables could be represented as strings to allow users to enter formulas that represent real world situations more naturally.

Technical Information

The top level function is called 'isTautology'. It accepts any formula as a parameter and returns a `bool` indicating whether or not the given formula is a tautology.

There are several formulas included in the source for testing purposes, as well as a function to test them all. The formulas are included in a list, appropriately called 'formulas'. Each member of the list is a pair. The first element of the pair is a formula, and the second is a `bool` indicating whether or not the formula is in fact a tautology.

The function 'testAll' can be called with argument '0' to test every formula in this list. 'testAll' compares the result of 'isTautology' with the bool given in the pair to determine correctness.

The entire program is contained in a file called 'ptp.sml' that can be found in Appendix A.

Appendix A - Source Code

See attached - **ptp.sml**

Appendix B - Demonstrative Runs

See attached – **ptp.sml.out**

Appendix C – Splay Set Issue

The following is an example of a database that oscillates between three states due to the property that splay trees always store the most recently added element at the root.

State 1:

`{S~R~T}{S~R}{S~P~T}{S~P}{R~P~T}{P~S}{P~R~T}{P~R}{PS~R}{T}{~R}{~P}{~R~T}{~S}{~S~T}`

`{S~R~T}` is resolved through the database. Nothing new is added; however, the last resolvent is `{~R~T}`, which puts it at the root of the tree.

State 2:

`{~R~T}{~R}{~P}{T}{S~R}{S~P~T}{S~P}{R~P~T}{P~S}{P~R~T}{P~R}{PS~R}{~S}{~S~T}`

`{~R~T}` is resolved through the database. It resolves with `{R~P~T}` into `{~P~T}` which is added to the database at the root.

State 3:

`{~P~T}{~P}{T}{S~R}{S~P~T}{S~P}{R~P~T}{P~S}{P~R~T}{P~R}{PS~R}{~R}{~S}{~S~T}`

`{~P~T}` resolves with `{P~R~T}` and `{PS~R}` into `{~R~T}` and `{S~R~T}`, respectively. These are added to the database, which brings the database back to the state it was initially in:

State 1:

{S~R~T}{S~R}{S~P~T}{S~P}{R~P~T}{P~S}{P~R~T}{P~R}{PS~R}{~R}{~P}{T}{~R~T}
{~S}{~S~T}