Resolution-Based Automated Theorem-Prover

Carter J. Bastian

March 9, 2016

Introduction

This project is the design and implementation of a resolution-based automated theorem-prover for first-order logic. An emphasis was places on building a theorem-prover that is refutation-complete under first order logic – software that, given infinite time and computational resources, is capable of proving any provable theorem in first-order logic [1]. Because of the intended result's breadth of capability, the design of the solution had to be both flexible with regards to the queries it accepts and rigorously strict with the methods it uses to prove them.

This document focuses mostly on the design of the theorem prover and a high-level approach to it's implementation. It begins with a theoretical look at the theorem-prover's proof method – binary resolution paired with factoring – and the sub-problems necessitated by the goal of completeness, continues with a look at the implementation of said prover, and finishes with an examination of the results of the project. Note that, for brevity, many implementation details will be marginalized or skipped entirely so that the focus can remain on those design solutions relevant to the larger problem of automated theorem proving.

Automated Theorem-Proving

In general, automated theorem-proving is the process of applying one or more inference methods to a set of known (provided) logical sentences in order to prove another sentence. Throughout this document and the inline documentation of the theorem-prover's source code, the set of provided sentences will be referred to as the "Knowledge Base" (KB), and the sentence to be proved will be referred to as the "Query."

First-Order Logic

First-order logic, which builds on the declarative, compositional, context-independent, and unambiguous nature of propositional logic, is a formal language capable of representing objects, relations, and functions in addition to the logical connectives of propositional calculus. Whereas propositional calculus allows for proof-methods that are guaranteed to complete in finite time, the nature of functions necessitates a possibly infinite set of clauses which can be derived from any sufficiently-complex knowledge base.

As will be seen , the goal of completeness in first-order logic necessitates the solution of a series of sub-problems. For example, in order to properly convert to conjunctive normal form in first-order logic, one must take additional steps to remove universal and existential quantifiers. These steps are covered throughout this document.

One important example of such a step is the process of unification — the process of finding a most-general substitution that makes two different first-order logical expressions look identical. This process is used both in resolution as the test for whether or not two disjoint clauses can be omitted from a pair of clauses to

[1] Of course, under Gödel's Incompleteness Theorem, this does not mean that the prover can prove any true sentence in first-order logic, but rather that if a proof by contradiction for a set of sentences exists, it will be derived. On the other hand, in the event that no such proof exists, the theorem-prover may run infinitely and never complete. This trade off is inherent in the nature of the problem, and therefore is inescapably present in the solution.

form a new resolvent, and also in factoring as the test for whether a set of disjoint clauses are redundant. However, because the unification algorithm is both complicated and unnecessary for the comprehension of first-order resolution, it will not be covered here.

Binary Resolution

Binary resolution is an inference method which can be applied to two conjunctive clauses in a knowledge base to form a new, unique clause. The resolution inference rule is as follows:

$$\frac{l_1 \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_n}{\textbf{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee l_l \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

Where

$$\textbf{UNIFY}(l_i, \neg m_j) = \theta$$

Notice that, by using only binary unification, this inference method specifies binary resolution. As such, in order to guarantee the completeness of the theorem-prover, the results must be factored so as to remove multiple copies in a resolvent that can be unified.

The full algorithm for first-order resolution is listed below:

Algorithm 1 Proving a Theorem in First-Order Logic with Binary Resolution

Precondition: KB and Q are in Conjunctive Normal Form

```
1  function RESOLUTION(KB, Q)
2      clauses ← Set of clauses in KB ∧ ¬Q
3      new ← {}
4      loop
5          for all Pairs C_i, C_j in clauses do
6              resolvents ← RESOLVE(C_i, C_j)
7              if resolvents contains the empty clause then return True
8              new ← new ∪ resolvents
9          if new ⊆ clauses then return False
10         clauses ← clauses ∪ new
11
12 function RESOLVE(C_1, C_2)
13     resolvents ← {}
14     for D_i ∈ C_1 do
15         for D_j ∈ C_2 do
16             if UNIFIABLE(D_i, ¬D_j) then Add FACTOR((C_1 − D_i) ∧ (C_2 − D_j)) to resolvents
       return resolvents
```

Conjunctive Normal Form

As is seen above, resolution requires the knowledge base to be in conjunctive normal form, as described in the textbook. The series of steps taken to convert the knowledge base to conjunctive normal form is as follows:

Eliminate Implications and Equivalence Uses the logic provided on page 253 of the textbook (AIMA).

Move Negation Inwards Ensures that $\neg$ occurs strictly in literal clauses.

Standardize Variables Ensures that each named variable is used in only one quantifier.

Skolemize Removes existential quantifiers by replacing existentially quantified variables with functions of universally quantied variables in the same lexical scope.

Drop Univeral Quantifiers All variables are then assumed to be universally quantified.

Distribut $\vee$ over $\wedge$ Ensures that the final form of the knowledge base is a series of conjunctions of disjunct literals.

Design and Implementation of the Solution

The design of the software to implement this resolution strategy is separated into three parts, each of which is discussed in some detail below. Notice that the code was implemented in C for the sake of computational speed, reduced memory overhead, and compatibility with the flex and bison tools used to implement the first order logic compiler.

The First-Order Logic Compiler

In order to allow the user to create a knowledge-base and query, a logical compiler was implemented to specify the syntactical and grammatical language of first-order logic [2].

While the details of this implementation are beyond the scope of this document, the source code for the lexical analyzer can be found in `KBscan.l`and the Backus-Naur recursive syntax is defined in `parseKB.y`. The first-order logic (.fol file) language specification is documented in the program's `README`.

With these tools, any valid sentence in first-order logic can be expressed, and any query tested. Examples of this will be shown later in the document.

However, one implementation detail that is worth noting is that the logic compiler converts logical sentences to logical syntax trees. The specification of this datatype is listed in full below:

```
/*
 * The full enumeration of types of nodes in a Logic Syntax Tree.
 */
typedef enum {
              NULL_N, KB_ROOT_N, RELATION_N, FUNCTION_N, CONSTANT_N, VARIABLE_N,
              NEGATION_N, CONJUNCTION_N, DISJUNCTION_N, EQUALS_N,
              IMPLICATION_N, EQUIVALENCE_N, Q_EXISTS_N, Q_FORALL_N, SKOLEM_N
} lst_node_type;

/*
 * Structure for nodes of the logical syntax tree.  Uses the
 * left-child/right-sibling representation, so that each node can have
 * a variable number of children.
 */
typedef struct lst_node_struct *lst_node;
struct lst_node_struct {
  lst_node_type node_type;
  lst_node left_child, right_sib, parent;
  char *value_string;
};
```

[2] I call this a compiler since its structure and operation works very similarly to a compiler for other formal languages such as programming languages. Interestingly, the compiler converts the logical notation into a logical syntax tree, just as a code compiler converts source code into a abstract syntax tree. All further functions of the theorem prover can be seen as processing the syntax tree into various intermediate representations, analogous to the "middle-end" optimization process undergone by code compilers.

The Conjunctive Normal Form Transformer

The next aspect of the code is the section which applies the transformations necessary to convert the knowledge base to conjunctive normal form as above. This code is implemented in full as a series of functions in `transformations.c`which operate on logical syntax trees.

As an example, the code to remove implications for the logical syntax tree (the second step in the CNF transformation algorithm) is listed below;

```
lst_node remove_implies(lst_node root) {

  if (root->node_type == IMPLICATION_N) { // if root is implication
    // replace with disjunction
    root->node_type = DISJUNCTION_N;

    // insert negation before left child
    lst_node inserted_neg = create_lst_node(NEGATION_N);

    inserted_neg->parent = root;
    root->left_child->parent = inserted_neg;

    inserted_neg->left_child = root->left_child;
    inserted_neg->right_sib = root->left_child->right_sib; // Move the sibling up
    root->left_child->right_sib = NULL; // Cancel old link
    root->left_child = inserted_neg;

  }

  // recurse on each child
  for (lst_node curr = root->left_child; curr != NULL; curr = curr->right_sib)
    remove_implies(curr);

  return root;
}
```

As with the above function, all transformative algorithms are recursive functions operating directly on the compiler's tree representation of a sentence in first-order logic.

The Proof Engine

Finally, having converted to conjunctive normal form, the heavy lifting of resolution occurs in the methods defined in `resolution.c`. This includes the methods to perform resolution, resolve two conjunctive clauses, unify to disjoint clauses, and perform factoring.

To accomplish this, the following for-loop checks each clause for resolution to the empty clause (and thus proof by the Resolution Ground Theorem):

```
   // Loop through each clause in the clauses set
    for (cSet *c1 = clauses; c1 != NULL; c1 = c1->next) {
    if (c1->val == NULL)
      continue;
      // Loop through each subsequent clause in the clauses set
      for (cSet *c2 = c1->next; c2 != NULL; c2 = c2->next) {
        if (c2->val == NULL)
          continue;

        c1->val->right_sib = NULL;
        c2->val->right_sib = NULL;

        cSet *resolvents = resolve(c1->val, c2->val);

```

```
15          // if resolvents contains the empty clause, return true
16          /* WARNING: MAKE SURE YOU PRINT THE END OF THE PROOF */
17          if (resolvents->val) {
18            for (cSet *resolved = resolvents; resolved != NULL; resolved = resolved->next)
19              if (inSet(new, resolved->val) == 0)
20                new = addSet(new, resolved->val);
21
22            //new = joinSet(new, resolvents);
23            if (resolvents->val->node_type == NULL_N) {
24              // Print the proof!
25              fprintf(fp, "Proof of query:\n");
26              for (cSet *clause = clauses; clause != NULL; clause = clause->next) {
27                if (clause->val) {
28  //                  print_lst(fp, clause->val, 0);
29                  print_clause(fp, clause->val);
30
31                  fprintf(fp, "\n");
32                  clause->val->right_sib = NULL;
33                }
34              }
35
36              fprintf(fp, "\n\nFINAL CLAUSES (from iteration %d)\n", i);
37              for (cSet *clause = new->next; clause != NULL; clause = clause->next) {
38                if (clause->val) {
39  //                  print_lst(fp, clause->val, 0);
40                  print_clause(fp, clause->val);
41
42                  fprintf(fp, "\n");
43                  clause->val->right_sib = NULL;
44                }
45              }
46
47
48              return 0;
49
50            }
51
52          }
53        }
54      }
```

A partial listing of the implementation of the function to resolve two conjunctive clauses follows. While it is a long method, this is simply due to the nature of it's operation. However, since it is vital to the operation of the proof engine, it will be listed anyway

```
1   for (cSet *d1 = disjuncts1; d1 != NULL; d1 = d1->next) {
2     for (cSet *d2 = disjuncts2; d2 != NULL; d2 = d2->next) {
3       lst_node d2neg;
4       if (d2->val->node_type == NEGATION_N) {
5         d2neg = d2->val->left_child;
6       } else {
7         d2neg = create_lst_node(NEGATION_N);
8       }
9
10      if ((unify(d1->val, d2neg, empty_sub()))->fail == 0) {
11        // create a new set from all the disjuncts in both clauses EXCEPT these two
12
13        /* OMITTED: looping through the clauses to add those not being left out*/
14
15        // Check if this new set is empty
16        if (!newSet    newSet->val == NULL) {
17          // Give it the empty clause
```

```
          newSet->val = create_lst_node(NULL_N);
          return newSet; // Fail early - this means we have proof
        } else {
          // Factor this new set
          newSet = factor(newSet);

          if (newSet) {
            lst_node rejoined = disjoin_clauses(newSet);
            print_lst(stdout, rejoined, 0);
            // Add this new node to the resolvents set
            cSet *newNode = generate_set();
            newNode->val = rejoined;

            resolvents = addSet(resolvents, rejoined);
          }

        }
      }
    }
  }
  return resolvents;
```

Results

In short, the first-order logic theorem-prover worked with resounding success. It was able to prove theorems of varying complexity in knowledge bases consisting of propositional logic, first-order logic, and even arithmetic logic.

For example, the following .fol first-order logic specification encodes basic arithmetic into formal logic. The query (listed last) requires that the theorem-prover prove the associative property of addition.

```
CS: 0;
RS: NatNum;
VS: n;
VS: m;
VS: a;
VS: b;
VS: c;
VS: d;
VS: e;
VS: f;
VS: w;
VS: x;
VS: y;
VS: z;
FS: S;
FS: Add;

x x = x;x y y = x => x = y;x y z x = y ^ y = z => x = z;

NatNum(0);
n NatNum(n) => NatNum(S(n));n ~(0 = S(n));
n m ~(m = n) => ~(S(m) = S(n));
m NatNum(m) => Add(0, m) = m;m n NatNum(m) NatNum(n) => Add(S(m), n) = S(Add(m,n));x y x = y => (NatNum(x)
    <=> NatNum(y));x y x = y => (S(x) = S(y));w x y z w = y x̂ = z => (Add(w,x) = Add(y,x));QS: a b
    c #d NatNum(a) \^ NatNum(b) \^ NatNum(c) \^ NatNum(d) \^
        (#e NatNum(e) \^ e = Add(a, b) \^ d = Add(e, c)) =>
        (#f NatNum(f) \^ f = Add(b, c) \^ d = Add(f, a));
```

Provided with this listing, it was able to prove associativity in only a few iterations of the resolution algorithm. The proof is provided in `out/associativity/proof.out`. Further results are all provided in the `out/`and `input/`directories together. There are example inputs specifying knowledge bases in propositional and first-order logic, as well as arithmetic contexts. Furthermore, full documentation of how to read and interpret the results, as well as how to run custom tests is provided in the `README`.