CMPT 777 Formal Verification Project Report (Theorem Prover for First-order Logic)

Jimmy Chen Chen
Khang Le
Nazanin Yousefian

November 2023

Contents

1 Overall Design

We decided to use Rust to implement the prover because it supports algebraic data type which is being used to model the language of first-order logic (the full version). It's also a safe and practical language, providing many correctness guarantees without sacrificing flexibility. Lastly, we leverage Rust's powerful macro system simplify the process of writing first-order formulas as algebraic data types; this helps with testing.

1.1 Code Structure

The main components of the code consists of clausal form conversion, unification, and resolution. They are made modular to allow effective collaboration among teammates. The algorithm we are using is called *Resolution Refutation*. To implement this, we have **is_valid(Formula)** which negates the input formula and converts it to an *equisatisfiable* clausal form. This will return a set of clauses (a set of formulas which are all atomic formulas or the negation of them) which we will perform resolution on. If it derives an empty clause, we conclude that the negated formula is unsatisfiable, meaning the original formula is valid. However, it may run forever, so we set a limit to stop it if it runs for too long. In this case, it can't tell us the validity of the formula. This is because first-order logic, although is sound and complete, is not decidable; it's semi-decidable.

2 Running the Software

After installing Rust, you can clone the repository and run the code directly (which builds it too).

`cargo run -- --help`	output a help menu
`cargo run -- formulas.txt`	run the software and check the validity of formulas in a file

There's a file called formulas.txt at the root of the project that contains some example formulas.

2.1 Syntax of Formulas

We've implemented a parser that parses first-order formulas. The operator precedence is encoded in the order in which the operators appear in the following grammar, from lowest to highest.

Syntax

$\langle formula \rangle$ ::= $p(t_1, ..., t_n)$ where $p \in Pred$, t_i are terms, and n is the arity of p
| $\langle formula \rangle$ `<->` $\langle formula \rangle$
| $\langle formula \rangle$ `->` $\langle formula \rangle$
| $\langle formula \rangle$ `\/` $\langle formula \rangle$
| $\langle formula \rangle$ `/\` $\langle formula \rangle$
| $\sim \langle formula \rangle$
| $forall\ v.\ \langle formula \rangle$ where $v \in Var$
| $exists\ v.\ \langle formula \rangle$ where $v \in Var$
| ($\langle formula \rangle$)

$\langle term \rangle$ ::= c where $c \in Const$
| v where $v \in Var$
| $f(t_1, ..., t_n)$ where $f \in Func$, t_i are terms, and n is the arity of p

$$Pred = \{p, q, r, s, t\}$$

$$Func = \{f, g, h, i, j, k, l, m, n\}$$

$$Const = \{a, b, c, d, e, o\}$$

$$Var = \{u, v, w, x, y, z\}$$

2.2 Parsing Limitation

Our parser requires that $\sim$, *exists*, and $forall$ are separated by parenthesis. E.g. $\sim forall\ x.\ p(x)$ must be translated to $\sim (forall\ x.\ p(x))$.

2.2.1 Example

$$\neg(\exists y.\forall z.(p(z,y) \leftrightarrow \neg\exists x.(p(z,x) \wedge p(x,z))))$$

Must be translated to

$$\sim (exists\ y.\ (forall\ z.\ (p(z,y) \leftrightarrow \sim (exists\ x.\ (p(z,x) \wedge p(x,z))))))$$

3 Clausal Form Conversion

3.1 High-level Approach

Given a formula, there are five steps to convert it to an equisatisfiable clausal form. The steps include: 1. remove free variables from the formula. 2. Convert the result to Prenex Normal Form. 3. Convert the result to Skolem Normal Form. 4. Convert the formula to Conjunctive Normal Form. 5. Drop quantifiers and write the formula as a set of clauses. Each Step is described in the following section.

3.2 Removing Free Variable

To remove free variables from the formula, we need to first detect free variables. Then, for each variable, we need to add an existential quantifier at the beginning of the formula. To this end, we iterate the formula keeping track of the variables that are in the scope of a quantifier, and add those which do not satisfy this condition with an existential quantifier at the beginning of the formula.

3.3 Converting to Prenex Normal Form

To convert a formula to Prenex Normal Form, there are three steps:

1. **Convert to Negation Normal Form**: First, we need to eliminate *implies* and *iff* and replace them with *and*, *or*, or *negation* logical connectives. This is done as follows:

   $$F \rightarrow G \Leftrightarrow \neg F \vee G$$

   $$F \leftrightarrow G \Leftrightarrow (\neg F \vee G) \wedge (F \vee \neg G)$$

   Then we need to push negations in through the following rules (first 2 are DeMorgan's Laws):

   $$\neg(F \wedge G) \Leftrightarrow \neg F \vee \neg G$$

   $$\neg(F \vee G) \Leftrightarrow \neg F \wedge \neg G$$

   $$\neg\neg F \Leftrightarrow F$$

   $$\neg\forall x.\varphi \Leftrightarrow \exists x.\neg\varphi$$

   $$\neg\exists x.\varphi \Leftrightarrow \forall x.\neg\varphi$$

2. **Rename Quantified Variables**: Each quantified variable which is repetitive, gets replaced with a unique fresh variable.

3. **Move Quantifier to the Front**: For each quantified variable that is in the middle of the formula, we remove it and add it at the beginning of the formula.

3.4 Convert to Skolem Normal Form

In this step, all quantifiers are at the beginning of the formula. For each existential quantifier, we remove the quantifier from the formula and iterate the formula. If the quantified variable is in the scope of other variables, we replace it with a new function constant which takes all the variables of the scope as argument. If there are no such variables, we replace the variable with a fresh object constant.

3.5 Convert to Conjunctive Normal Form

In this step, we ignore all the quantifiers at the beginning of the formula and convert the inner formula to Conjunctive Normal Form. We first convert it to NNF and then we need to apply the two following replacements:

$$(F \wedge G) \vee H \Leftrightarrow (F \vee H) \wedge (G \vee H)$$
$$F \vee (G \wedge H) \Leftrightarrow (F \vee G) \wedge (F \vee H)$$

3.6 Derive Clauses

The final step consists of removing quantifiers at first. Then, each set of disjunctions separated by a conjunction is a clause. In the end, we need to rename variables in each clause so that no clause has a variable that is the same as the variables of other clauses.

4 Resolution Refutation

4.1 High-level Approach

The general algorithm for resolving a list of clauses is to do it pairwise, iteratively resolving pairs of clauses if possible. When a pair of clauses is resolved, new clauses are produced and added to the original list for further resolution. The process continues in a loop, with the updated list of clauses after each iteration, attempting to resolve a new pair of clauses until a termination condition is met. **First**, if an empty clause is derived during the resolution, we successfully show that there is a resolution refutation and, thus, prove the input formula is valid. **Second**, if there is no new clause obtained after resolving every possible pair of clauses, we conclude that there is no resolution refutation and, thus, the input formula is invalid. **Third**, if the resolution ends due to the time limit, the algorithm is unable to determine whether the input formula is valid or invalid.

4.2 Example

To better illustrate the working mechanism of our algorithm, let's try to find a resolution refutation of the following example:

$$C1 : \{p(x), p(y)\}$$
$$C2 : \{\neg p(a)\}$$

Loop 1:

1. Iteratively pick a pair of clauses from the list to do resolution. In this case, the only pair we have is: $\{C1, C2\}$.
2. From each chosen pair, iteratively pick one formula from each clause to do resolution (if possible). In this case, we have two possible scenarios for the pair $\{C1, C2\}$: `resolve`$(p(x), \neg p(a))$ and `resolve`$(p(y), \neg p(a))$. Note that `resolve` makes use of the MGU (most general unifier) algorithm.
3. `resolve`$(p(x), \neg p(a))$ and `resolve`$(p(y), \neg p(a))$ produce two new resolvents $C3 : \{p(y)\}$ and $C4 : \{p(x)\}$, respectively.

4. The updated list of clauses is as follows:

$$C1 : \{p(x), p(y)\}$$
$$C2 : \{\neg p(a)\}$$
$$C3 : \{p(y)\}$$
$$C4 : \{p(x)\}$$

Loop 2:

1. Iteratively pick a pair of clauses from the list to do resolution. In this case, the pairs we have are: $\{C1, C2\}$, $\{C1, C3\}$, $\{C1, C4\}$, $\{C2, C3\}$, $\{C2, C4\}$, and $\{C3, C4\}$.
2. Because we have already resolve $\{C1, C2\}$ before, and $\{C1, C3\}$, $\{C1, C4\}$, and $\{C3, C4\}$ cannot be resolved, we skip those pairs. This gives us the remaining pairs: $\{C2, C3\}$ and $\{C2, C4\}$.
3. From each chosen pair, iteratively pick one formula from each clause to do resolution (if possible). In this case, we have the following scenarios:
   - For the pair $\{C2, C3\}$: `resolve`$(\neg p(a), p(y))$
   - For the pair $\{C2, C4\}$: `resolve`$(\neg p(a), p(x))$
4. `resolve`$(\neg p(a), p(y))$ for the pair $\{C2, C3\}$ produce an empty clause $C5 : \{\}$. Similarly, `resolve`$(\neg p(a), p(x))$ for the pair $\{C2, C4\}$ also produce an empty clause $C6 : \{\}$. We found the empty clause. This shows that there is a resolution refutation and concludes the resolution.

5 Most General Unifier

5.1 High-level Approach

The general algorithm for finding the MGU of n formulas (or atomic formulas, aka predicates) is to do it pairwise, linearly. After finding the MGU for two predicates, the unifier would have been updated. To unify the next pair, apply the unifier to both formulas in the pair before proceeding with the unification.

5.1.1 Example

For example, suppose we want to find the MGU of P1, P2, and P3. It first finds the MGU of P1 and P2. Then it applies this unifier to P1 and P3 and tries to find the MGU of the substituted P1 and P3. It could've picked P2 but it doesn't matter because P1 and P2 becomes the same after applying their MGU to them. Note that the MGU is updated (extended) at every iteration. At the end, we end up with an MGU of P1, P2, and P3.

5.2 Unification of Two Predicates

To unify two predicates, it first tries to line up the arguments (terms) of the predicates, recursively. Once the terms are lined up and stored in a worklist (call it `pairs_to_unify`, pairs of terms), it iterates over it. At every iteration, if one of them is a variable v, it maps that v to the other term t in the pair. Then it substitutes v with t in all the terms that occur in the unifier and in `pairs_to_unify`. Finally, it extends the unifier with this map. If none of them is a variable, then it tries to perform a term matching algorithm on them. During this process, lineup may occur. In such case, more terms will need to be unified later on so it extends `pairs_to_unify` with those extra pairs.

5.3 Term Lineup

It lines up the arguments of two predicates such that it creates a pair of terms for every i^{th} argument of the predicates. E.g. the first argument of the two predicates will form the first pair. Then it iterates over them, calling term matching for the elements of the pair. If any matching fails, it concludes the predicates can't be unified.

5.4 Term Matching

This is where it tries to match the terms and see if they can be made identical. It takes two terms as input (as well as `pairs_to_unify` to mutate it).

- If they are different object constants, it concludes the predicates can't be unified. E.g. a and b.
- One of them is a variable: if both terms are the same, nothing needs to be done. Otherwise, check that the other one doesn't contain the variable. If it does, then it adds the terms to `pairs_to_unify`; otherwise, it concludes the predicates can't be unified. E.g. x and f(x).
- If one of them is an object constant and the other is a function, it concludes the predicates can't be unified. E.g. a and f(a).
- Both are functions: if the functions are the same, then it recursively lines up the terms of the two functions; otherwise it concludes the predicates can't be unified. The same conclusion is made if the lineup fails. E.g. f(x) and g(a).

5.5 Example

Let's try to find the MGU of these 3 predicates if it exists.

$$P1 : p(x, f(x, g(z)))$$
$$P2 : p(h(a), f(z, g(y)))$$
$$P3 : p(h(w), f(z, g(v)))$$

Consider P1 and P2. It eventually lines up $(x, h(a))$, (x, z) and (z, y). It will iterate 3 times.

1. Create a map $m : x \mapsto h(a)$ and then substitutes x with $h(a)$ in the other pairs, resulting in $(h(a), z)$ and (z, y). Extends the unifier (empty at this point) with m.
2. Create a map $m : z \mapsto h(a)$ and the last pair becomes $(h(a), y)$. Extends the unifier with m.
3. Extends the unifier with $y \mapsto h(a)$.

Now it tries to unify P1 and P3. Before doing so, it applies the MGU of P1 and P2 to P1 and P3. The process is similar. After this, we end up with an MGU of P1, P2, and P3.

6 Conclusion

The result is a fully functioning theorem prover for first-order logic (with full language support), written in Rust. We have around 44 unit tests and around 12 end-to-end tests (in test_formulas.txt), ensuring the components as well as the whole software works as expected. The e2e tests also serve as evaluation and we've observed that the code can give an answer in under 1 second for all the formulas except for 2 unit tests and 1 e2e test. They might run forever because first-order logic is semi-decidable. So we've set the limit to 5 seconds for those unit tests (e2e tests have a default limit of 60 seconds). Note that these 3 tests were designed for this purpose. To run the unit tests, execute `cargo test` (finish in 5 seconds) and to run the e2e tests, execute `./e2e_test.sh` (finish in 60 seconds). They all pass.

7 Division of Work (required for the course)

Nazanin wrote the unit tests for the Clausal Form Conversion component, discussed with Jimmy regarding the algorithms for the 5 steps of the conversion, and contributed to that section of the report. Khang implemented the Resolution Refutation component, wrote the e2e tests and some unit tests for the software, contributed to that section of the report, and assisted Jimmy with various tasks. Jimmy wrote the rest of the report and the code. Aside from designing and implementing the algorithms for the Clausal Form Conversion and the Most General Unifier components, he also initiated, led, and finalized the project.