Automated theorem proving for the two variable fragment in the First Order Logic

Author: Patrick Cătălin Alexandru Sava

Supervisor: Dr. Ian Pratt-Hartmann

Abstract

First-order logic has few fragments which are very interesting in terms of decidability. One of them is the two-variable fragment with no function symbols. Even though the two-variable fragment could be very rich in terms of potential discoveries, there are few papers written in this regard by now. One of them is Hans de Nivelle and Ian Pratt-Hartmann’s paper of 2001 which, by no coincidence, represents namely the goal of the project. Being the first of its kind, this paper describes a decision procedure for the two-variable fragment with equality (prior works describe only the decision procedure for the case without equality). Implementing this algorithm embodies the project’s aim, nonetheless, the roadmap for achieving this objective involved moreover the implementation of a theorem prover for the general case.

Acknowledgements

Firstly, the project would not have been possible without the continuous help and support of my supervisor, Dr. Ian Pratt-Hartmann. I am very grateful for the satisfying experience of working together on such a challenging, yet rewarding project. I would like to highlight my appreciation for all of the excellent advice I was receiving from him throughout the entire project.

Secondly, there is no doubt that my self-development, knowledge and capabilities were constantly enhanced by the world of Computer Science which has been an indispensable part of my life ever since. Thus, I would like to extend my gratitude to the programming community, my teachers and friends who have been a part of this great journey.

Finally, I would like to thank my family (especially my mother, my grandmother and my partner, Yoana) for all of their trust, support and love during all of these university years and not only.

[**Introduction**](#_x02d7y3csfxn) **6**

[Background](#_orz03eqchgg4) 6

[Motivation](#_8k4aygsvrx9q) 6

[Project aim](#_ax4updv1mq93) 6

[Project Roadmap](#_r2ujw6ayztbf) 6

[Methodology](#_oizawpbvlrdq) 7

[Report Structure](#_2oarfns66zbh) 7

[Impact of Covid-19](#_xwhv18ieh44j) 7

[**Context**](#_9zd57b4duhd5) **8**

[First-Order Logic](#_dy6konftaet3) 8

[Definitions](#_djdrgmdkxmng) 8

[Operators](#_2vspvqs2p7le) 8

[Special case: equality](#_wnin7kxxh15l) 9

[Quantifiers](#_fi1ejy2x2bsw) 9

[Model](#_b2ymtzfccxmi) 9

[Validity and Satisfiability](#_fmzc8s7c2bok) 9

[Soundness and Completeness](#_tu40xscopkem) 9

[Decidability](#_4f0rsqujbia0) 10

[Clausal Normal Form (CNF)](#_r0vwamua472s) 10

[Basic Reduction](#_jj13vywtq8bx) 11

[Skolemization](#_soolxzd7jy5g) 11

[Automated Theorem Proving](#_f3q4g59ixn98) 12

[Unification](#_c8vaumr6n4pg) 12

[Resolution rule](#_aqodhlrppqkq) 12

[Factoring rule](#_4p8p51x1sbok) 13

[Refinements](#_ifehuyerqdp6) 13

[Tautology Removal](#_2uipkhcq6lh9) 13

[Subsumption](#_1tbv1tqm568p) 13

[Unification within the same clause, on literals having the same sign](#_77q9rj31lmho) 13

[Depth-ordered resolution](#_e7rwvwdaw2yp) 14

[Two-Variable fragment](#_gioufz95cz1z) 14

[Without equality](#_xq9n8zjme282) 15

[With equality](#_8lahq96r0pgp) 15

[**Design**](#_xxpev9n2lwjw) **17**

[**Implementation**](#_gfjtingyzhcz) **18**

[Environment](#_l1vcfc3nbwqf) 18

[Front-end](#_d5si54q21z42) 18

[Tokenizer](#_cn3wt633rot0) 19

[Parse Tree](#_l6i6odkebjow) 19

[Intermediate Representation](#_e3c2dgf365go) 19

[The approach for disambiguating the given formulas](#_vrbua8pgemx8) 19

[The approach for resolving the precedence](#_oja1m8ihg9eo) 19

[The approach for eliminating double Implications](#_ebd5oqbkz1ac) 19

[The approach for eliminating implications](#_hfjgzsolr963) 19

[Basic reduction](#_9xc26xr5lcjq) 19

[Skolemization](#_bxgow04gybs6) 19

[Back-end](#_61zrhg2p0mft) 19

[Clause Form](#_apo4ra7ygpa6) 19

[Basic Theorem Prover](#_5shpuwpg832v) 19

[Overview](#_gjbi0aoldqn4) 19

[Multithreading](#_s1hwhpfhlpg6) 19

[Depth-Ordered Theorem Prover](#_kqflx93o42gg) 19

[Two-Variable Theorem Prover](#_2qdj0scw4dg3) 19

[Without Equality](#_wmmcmnyjbk25) 19

[With Equality](#_e5cregib9bw3) 19

[**Evaluation**](#_wb83b0x8anmn) **19**

[Unit testing](#_xmp6tw7l5j9v) 19

[End-to-End testing](#_b1rwdin7qwlu) 20

[Using the problems found on tptp.org](#_mlszyojem44) 20

[Using general formulas](#_2bj5s1nzowqv) 20

[Testing against Vampire](#_gs6mdb6o2163) 20

[**Experiments**](#_w8eibc75csft) **20**

[**Reflection**](#_vct7big78d3b) **20**

[**Conclusion**](#_l2sgo3qyqxj8) **20**

[**Appendix 1**](#_witszixm8988) **20**

# 

# 

# 

# 

# Introduction

## Background

First-order logic is characterized by a strong richness in terms of expressivity. Currently the world is aware of its presence behind many fields such as Mathematics, Philosophy or Computer Science. In Computer Science it could be present as part of sophisticated solutions in the subfield of databases or the one of natural language processing.

In comparison with propositional logic, the first-order logic has a stronger power due to the predicates and the quantifiers. Symmetrically, higher-order logic has a stronger power than the first-order logic, but it does not provide a sound and complete proof calculus. [reference <https://en.wikipedia.org/wiki/Higher-order_logic>]

In essence, first-order logic lies at the intersection of high expressiveness with soundness and completeness and this is one of the reasons why it is valued by the research world.

## Motivation

During the course COMP24412 I became particularly attracted to using Prolog and Vampire. I have then realised that this is an area of research which I have not had the chance to explore well enough so I was looking for a third-year project liying in this sub-field.

At the same time, I was looking to grasp my knowledge in the sub-field of compilers, which the implementation of a theorem-prover was able to offer to me since I strongly preferred to write the front-end of it by myself.

Ultimately, the curiosity of seeing how well I could implement a piece of software which is going to compete directly with an absolut champion like Vampire, encouraged me to accept this challenge.

## Project aim

The main goal of the project was to implement the first ever decision procedure for the two-variable fragment with equality of the first order-logic which was priorly described in a paper written by Hans de Nivelle and Ian Pratt-Hartmann in 2001, but for which there is no evidence currently that it was actually implemented by someone else in the meantime. On the top of that, an additional goal of the project was to compare the algorithm with Vampire and to produce some satisfiability charts in the case in which the input is a randomly generated formula.

## Project Roadmap

The project roadmap was quite well defined:

* Semester one
  + The first six weeks (including week 0) were dedicated to familiarising myself more to the topic, that meaning mostly reading papers, books and additional materials
  + The next three weeks (up to week 8) were mostly invested in writing the parser for the formulas
  + The last weeks up to the Christmas holiday were spent coding a general theorem prover
  + The Christmas holiday and the exam session were dedicated for testing the existing functionality, coding the depth-ordered theorem prover and the two-variable theorem prover for the case without equality
* Semester two
  + The first three weeks were spent on reading some papers, coding the two-variable theorem prover for the case with equality, optimizing, testing and debugging the code
  + The next three to four weeks were mostly invested in optimizing, testing and debugging the code. On the top of that, two random generators for formulas and some scripts of comparing my work against Vampire were written.

## Methodology

One of the most important aspects of the project was definitely the time management. Coordinating with my supervisor while both researching and implementing the theorem prover was challenging and required a good amount of prioritization. My supervisor suggested keeping track of everything I am doing in a log book, which I found to be very useful not only for writing now this report, but also for organizing my thoughts and ideas better. Even though I did not feel it as being useful at all at the beginning, this idea proved as being brilliant in time, especially when the size of the codebase for the project hit 5000 lines of code and even remembering what my own lines of code are doing was quite problematic.

## Report Structure

The structure of the report is chosen in a way in which the reader is presented the context first, then the high-level idea of the project and further details of smaller granularity. Hence, the next section is reserved for reiterating the basic notions of first-order logic and the formal tools which are going to be part of the algorithm. The next section is going to be dedicated to the design decisions, included but not limited to the high-level presentation of the two-variable theorem prover. It follows the section dedicated to the implementation, which indeed underlines the low-level details of it. Further there is a section dedicated to the methods of evaluating the project, one dedicated to the experiments, one dedicated to the reflection and finally the one which concludes the report.

## Impact of Covid-19

Fortunately, I am in a position in which I could say that the whole situation caused by Covid-19 did not create any major disruptions, and that’s mostly because my project was completely independent of any University equipment. The communication with my supervisor was excellent and I feel grateful that the overall quality of the project was not affected by this matter.

# Context

For the following sections I will assume that the reader is familiar with the first-order logic, even though I will do a high-level iteration on all of the aspects which are of interest.

## First-Order Logic

As mentioned before, first-order logic is having a stronger expressiveness than the propositional logic due to the presence of quantifiers and more verbose predicates, in the sense that they could have arguments, which could be constants, variables or even functions. I will define these in the next subsection.

From now on, I will be referring only to first-order logic, so it could be assumed that any reference to logic from this report refers to first-order logic.

### Definitions

The highest level of granularity I will be using throughout this report will be the logic formula, which could be any string accepted by the grammar of the logic described in Appendix 1.

**Definition 1.** A predicate is a n-ary symbol containing n terms as arguments. Conversely the arity of a predicate is the number of arguments.

**Definition 2**. The sign of a predicate is positive if it is not negated and negative otherwise.

**Definition 3**. An atomic formula or an atom, is a formula containing only a positive predicate.

**Definition 4**. A formula containing only a positive predicate or only a negative predicate is called literal.

**Definition 5.** A term is either a function, a variable or a constant.

**Definition 6.** A function is a fixed (or pre-defined) mapping from a variable or a constant to a variable or a constant.

**Definition 7**. A variable is a mapping from a variable to another variable (including itself) or to a constant.

**Definition 8.** A constant is a mapping exclusively to itself.

Due to the fact that function symbols break the decidability (the reason is explained in the later subsections), we will restrict the input to do not contain such terms.

### Operators

With the risk of repeating the basics, the operators in which we will be interested are: double implication (⇔), implication (⇒), negation (ㄱ), and (∧), or (∨), equality (=) and finally inequality (!=).

Round brackets are considered as well part of the operators, since they improve the clarity in the majority of cases, especially when used together with quantifiers.

We sometimes refer to the operator and as conjunctions and to the operator or as disjunction.

#### Special case: equality

The reader could assume that up to the point when the two-variable theorem prover algorithm is presented, both the equality and inequality are not part of our language. The reason for which we will proceed this way is mostly because dealing with equality (and inequality) in the general case (that is, outside the two-variable fragment) is out of the scope of the project, involving different techniques such as paramodulation.

### Quantifiers

The quantifiers with which we are going to deal are the following:

* for all (∀)
* there exists (∃)
* there exists an unique (∃!)

The uniqueness quantifier will not be of interest up to the last subsection of this section. Moreover, the grammar will not contain this symbol and we will encounter it only at an abstract level, when we will be dealing with equality.

### Model

**Definition 9.** The domain of discourse is the set of entities over which the variables from a first-order logic formula are ranging. [reference <https://en.wikipedia.org/wiki/Domain_of_discourse>]

**Definition 10.** A model (or an interpretation) of a formula refers to a mapping from each variable from the formula to an element of the domain of discourse. [reference <https://en.wikipedia.org/wiki/First-order_logic>]

### Validity and Satisfiability

**Definition 11.** A formula is valid iff it is true in all of the possible models. Conversely, a formula is invalid if and only if it is false in all models. [reference wilfried book]

**Definition 12.** A formula is satisfiable iff there exists at least one model in which it is true. Conversely, a formula is unsatisfiable if and only if it is false in all models. [reference wilfried book]

As a consequence, a formula is invalid iff is unsatisfiable. Moreover a formula is satisfiable iff its negation is invalid.

### Soundness and Completeness

Let the knowledge base be the set of axioms we are having initially. This could be seen also as a set of true hypotheses or formulas.

As we prove true things using the knowledge base, these will be added to it.

Some deductive systems of first-order logic are sound and complete [reference https://en.wikipedia.org/wiki/First-order\_logic]. The intuition behind soundness is that only true things can be inferred from the knowledge base (as an analogy one could think to the definition of the implication, that a true fact can imply only another true one) while the one behind completeness is that all of the true things which could be possibly derived from the knowledge base will be eventually being reached. (as an analogy one could imagine a graph containing initially one node for each axiom, in which nodes and edges are getting dynamically added; when adding a node containing a brand new fact (derived using only the existing nodes of the graph at that point), undirected edges from the nodes corresponding to the facts which inferred the latter are added as well; when time tends to infinity, the graph will contain all of the facts which could possibly be derived from the knowledge base)

### Decidability

**Definition 13.** A decision problem is a problem which could be reduced to a yes-no question.

By decidability in logic we refer to the existence of an effective method for approaching a decision problem [reference <https://en.wikipedia.org/wiki/Decidability_(logic)>].

For example, the satisfiability problem of first-order logic is undecidable [reference <https://en.wikipedia.org/wiki/First-order_logic>], whereas the same problem for the two-variable fragment without function symbols, but with the existence of uniqueness quantifiers is decidable [reference <https://en.wikipedia.org/wiki/Two-variable_logic>]. Consequently, the two-variable logic has to be sound and complete.

## Clausal Normal Form (CNF)

For approaching the automated theorem proving, we will need a more convenient form to deal with the formulas. Firstly, time has come to underline the input format. We will assume that the input will contain N formulas, the first N - 1 of them forming the knowledge base, and the N-th one representing the query (that is, the formula of which satisfiability has to be verified assuming that all of the formulas from the knowledge base are axioms).

Secondly, it would be better to reduce everything to a single formula. This could be done easily by negating the N-th formula, wrapping each formula in a pair of round brackets and then adding N - 1 conjunctions between the N formulas. We will refer further to the resulting formula as F.

As described in [reference Leitsch book], we could transform F into a sub-language of logic called conjunctive normal form (sometimes being referred as clausal normal form). The final form will be a conjunctions of disjunctions, in which no quantifiers are present. We will refer to this final form as C. Even though F might not have contained functions symbols, C could contain and this is explained in the next subsections

### Basic Reduction

We will repeat the following seven steps until the formula does not change anymore:

* break double implications in implications
* break implications in disjunctions
* push the operator not further on conjunctions
* push the operator not further on disjunctions
* simplify two consecutive not operators to nothing
* push the operator not past universal quantifiers
* push the operator not past existential quantifiers

Let F’ be the resulting formula after all of the steps above have been exhausted. It is obvious that F’ contains only quantifiers, disjunctions, conjunctions, brackets and literals.

### Skolemization

The name of this procedure comes after the Norwegian mathematician Thoralf Skolem [reference <https://en.wikipedia.org/wiki/Skolem_normal_form>]. The intuition behind this procedure is that we could always see the existentially quantified variables as the result of a brand-new function which has as arguments all of the universally quantified variables having in their scope the former. By “scope” we refer to the same meaning as the one from programming. (i.e. one could make an analogy with the scope of local variables in C++) Obviously, if there are no such universally quantified variables, our function becomes a constant, precisely a Skolem constant. Otherwise, it remains a function, precisely a Skolem function. Hence, we will replace all of the occurrences of existentially quantified variables with Skolem functions or Skolem constants. Afterwards, we can safely dispose of all of the quantifiers (both existential and universal). The reason for being able to do so is because we will differentiate from now on the terms on sub-types (functions, variables, constants) by keeping track of that while doing the Skolemization. It could be easily seen that the quantifiers are part of redundancy now.

For the sake of simplicity we could assume for any variable x in F’, all of the variables which are in the scope of x have unique names. (i.e. a substring like ∀x∀y∀x or one like ∀x∃y∃y would never be present in F’). The approach for resolving this issue will be further described in the implementation details.

Let F’’ be the resulting formula. The only thing which we will have to do to transform F’’ in C is to apply the distributivity laws over conjunctions and disjunctions in order to end up with C. Having in mind the preparation for the next step, we could transform into a set of disjunctions (or clauses), since it is uniquely determined which operator stays between them (the operator and). Hence, the name of clause form is probably better justified now.

Even though here we were using explicitly the distributivity laws, the way we are going to implement them is going to differ from the traditional way. This is because implementing the distributivity laws naively could lead to an exponential growth of the size of the formula.

## Automated Theorem Proving

In this subsection we are going to present the core ideas and tools behind the automated theorem proving. All of these are part of the codebase of the project.

### Unification

The key of the automated theorem proving stays in the ability of the software to detect whether two literals are the same if the sign would be ignored. By that we mean that they are the same predicate-wise, term-wise and order-wise. (i.e. P(f(x)) and ㄱP(f(x)) respect that while P(f(x)) and P(f(y)) does not).

**Definition 14.** We call substitution a mapping from the set of variables to the set of terms. Moreover we could compose two substitutions in the same way we could compose two functions.

**Definition 15.** An invalid substitution is one in which there exists a mapping from a variable to a term containing the same variable (and by that we mean that it could be either itself or a function having the variable nested in one of its arguments). A valid substitution is obviously a substitution which is not invalid.

**Definition 16.** Two terms are unifiable if there exists a valid substitution such that after its appliance they become the same.

**Definition 17.** Two n-ary predicates P(t1, t2, …, tn) and Q(r1, r2, …, rn) are unifiable if P and Q are identical and ti and ri are unifiable for 1 ≤ i ≤ n.

**Definition 18.** Two literals are unifiable if their corresponding predicates (letting aside the sign) are unifiable.

**Definition 19.** Let P(t1, t2, …, tn) and Q(r1, r2, …, rn) be two n-ary unifiable predicates. Let S1, S2, …, Sn be the valid substitutions for the terms ti and ri  with 1 ≤ i ≤ n. We will call S the result of the composition of S1, S2, …, Sn. If the appliance of S results unifies P and Q (that is, makes P and Q identical) then S is the most general unifier of P and Q. Note that we could refer to most general unifiers for literals in the same way we do for predicates since the signs do not matter.

Having explained unification, we are now ready to present the two core rules for theorem proving: resolution and factoring.

### Resolution rule

**Notation 1.** Let C be a clause and S a substitution. Then {C}S is the resulting clause by applying the substitution S to all of the literals in C. Similarly we can define the same on a literal L or on an atom A (i.e. that would result in {L}S and {A}S, respectively)

**Notation 2.** Let C be a clause and L be a literal which occurs in C. Then C\L is the resulting clause obtained by removing L from C exactly once.

**Definition 19.** Let C1 and C2 be two clauses having in common two unifiable literals L1 (part of C1) and L2 (part of C2) of opposite signs. Let S be the most general unifier of L1 and L2. Then C1 ∧ C2 ⇒ {C1\L2 ∧ C2\L2}S. {C1\L2 ∧ C2\L2}S is often referred as being the resolvent of C1 and C2, while the atom A which is common to L1 and L2 is referred as the resolved atom.

### Factoring rule

**Definition 20.** Let C be a clause and L be a literal occuring more than once in C. If L occurs k times in C, then we could remove k-1 occurences of it from C.

Once we have implemented the resolution and factoring rules, we can safely claim that we have a basic theorem prover. Since the satisfiability problem for first-order logic is undecidable, our program could never terminate. If it terminates, this means that either it reached saturation (this is the case when there is no new clause inferred by the current set of clauses which has not been seen priorly) or it derived the empty clause. The former implies that the C was counter satisfiable while the latter implies the satisfiability.

From now on we will discuss a few refinements which could speed up the basic theorem prover and ultimately bring us closer to understanding the design of the theorem prover for the two-variable fragment.

### Refinements

#### Tautology Removal

**Definition 21.** Let C be a clause containing two literals L1 and L2 which are identical except their signs (which are opposite). This clause is obviously true in any model, so it can be dropped from the current set of clauses.

#### Subsumption

**Definition 22.** Let C1 and C2 be two clauses such that the set of literals of C1 is strictly included in the set of literals of C2. Then C1 could be seen as a “more general” clause than C2. Therefore C2 could be dropped since it is subsumed by C1.

This strategy makes sense because at any point in time we assume that all of the clauses from the current set of clauses are axioms. So the intuition behind subsumption is that C2 is no longer needed in the set since C1 was anyway implying it and since C1 has to be true, then C2 will be certainly true.

Note that we used “strictly” in the definition because if C1 and C2 are identical, we would anyway prefer not to add whichever came last in the set of clauses.

#### Unification within the same clause, on literals having the same sign

This is a heuristic I came up with during the implementation process. It looked really promising by that time, speeding up a few tests I was testing against my implementation. Intuitively having a clause C and two unifiable literals L1 and L2 which are part of the clause, we could unify them and then apply the factoring rule. This way we get a less general clause, which is going to have a shorter size than the former.

#### Depth-ordered resolution

**Definition 23.** A ground term is a term which does not contain variables. In other words, it is a constant, a function symbol or a function symbol having nested only function symbols or/and constants.

**Definition 24.** The depth of a variable and the depth of a ground term are both 0.

**Definition 25.** The depth of a function which is not a ground term is the maximum number of nested function symbols up to a variable. (i.e. f(x, g(y)) has depth 2, f(x, y) has depth 1 and f(f(x, g(y)) has depth 3).

**Definition 26.** The depth of an atom is the maximum depth of its terms (i.e. P(f(x, g(y)), f(x, y)) has depth 2, P(f(x, y)) has depth 1 and P(f(x, y), f(f(x, g(y))) has depth 3). Similarly we could define the depth of an literal as the depth of its corresponding atom, since the sign does not have any role in the calculation of it.

As described before, the satisfiability problem for first-order logic is undecidable. From that we could deduce that the basic theorem prover we have discussed might not halt in some cases. One of the reasons for which that would happen is the unbounded nesting of function symbols. (i.e. we don’t currently have any mechanism to avoid expansions like P(f(f(f(f(f(….)))))) to happen)

As described in [reference to the Leitsch99], we could define a binary relation <A on the set of all atoms which is having the following properties:

* is reflexive
* is transitive
* for any two atoms A and B and a substitution S, A <A B implies {A}S <A {B}S

**Definition 27.** The depth-ordered theorem prover will be working in the same was as the basic theorem prover, with the restriction that we will apply the resolution rule on two clauses C1 and C2 only when there is no literal in L in the resolvent C (of C1 and C2) such that B <A L, where B is the resolved atom of the resolution. [reference to the Leitsch101]

The depth-ordered resolution is sometimes referred as ordered resolution and it is proved as being complete. [reference https://people.mpi-inf.mpg.de/~hillen/documents/4\_OrderedResolution.pdf]

## Two-Variable fragment

We will further discuss the two-variable fragment theorem prover. Since this section describes the algorithm at a theoretical level, it could be assumed that wherever references are not in place and the difficulty of (sub-)topics presented exceed the basics of logic, the credits for them are owned by my supervisor. I am mentioning this because even though I have found the same notions in academic papers or books, I understood the best the principles of these notions during the tutorial classes.

For the rest of the next two subsections we will assume that we are given a logic formula in two variables which is going to be passed further as input to the two-variable theorem prover. Moreover, this formula is in the CNF format.

For the sake of simplicity we could assume as well that the set of variables occurring in our formula is {x, y}.

### Without equality

We will start by adding a restriction to the logic formula: for the rest of this subsection, we will assume that it does not contain equality or inequality. We will further revisit the algorithm for the case with equality in the next subsection.

With the risk of breaking the consistency, we will be labeling the steps of the algorithm this time, in order to make it easier to revisit for the case with equality:

1. We are going to execute the depth-ordered resolution on our formula, only on the literals involving exactly two variables. That is, the resolved atom will always have exactly two different variables.
   1. If the empty clause is derived, this means that our formula is satisfiable.
   2. Otherwise, the algorithm should halt by saturation, since we use the ordered resolution.
2. If the algorithm did saturate then we will be left with a set of clauses. From them, we could safely dispose of all of the clauses containing literals in exactly two variables.
3. Now we are left with a set of clauses with the following property: if we would group each clause on the set of distinct variables it is having, the maximum number of groups we would get per clause would be three. This is because we can have in the worst case the group corresponding to the:
   1. empty set (i.e. all of the literals in this group do not hold variables; they are ground instances as they are sometimes referred)
   2. the set {x}
   3. the set {y}

This whole procedure is referred to in [reference HdN&IPH] as the splitting rule and it

could be applied on a clause which does not have overlapping variables, which is the case for ours.

1. Let C be our set of clauses and |C| be the size of it. Further, we will make |C| group choices, one for each clause. In other words, we will build at most 3|C| auxiliary sets of clauses containing exactly one group from each clause. We will be then running the depth-ordered theorem prover on each of these auxiliary sets.
   1. If all of these runs will derive the empty clause, this means that the input formula is unsatisfiable.
   2. Otherwise, that would imply that the depth-ordered theorem prover halted by saturation on at least one of these auxiliary sets. Hence, the input formula is satisfiable.

### With equality

Before revisiting the algorithm above, we will have to detail how to deal with:

* the equality
  + dealing with the equality operator is trivial in our case, since we can just replace the equality operator with a special predicate of arity two called “Equality”
  + in case the equality operator occurs multiple times, all of these occurrences will be replaced by the same predicate called “Equality”
* the inequality
  + dealing with the equality could be seen as a fairly similar process as dealing with equality with the mention that we will replace the inequality operator with a brand-new predicate of arity two called “Inequality”
    - as in the case of equality, different occurrences of the inequality will be mapped to the same predicate called “Inequality”
  + in order to preserve the logical equivalence, we will have also to add a new clause in one variable containing the negated “Inequality” predicate to our CNF (i.e.ㄱInequality(x, x) has to be true)

We could treat the “Inequality” like any other predicate in our set of clauses from now on, but we will further have to expand the “Equality” in our algorithm.

After we have set how to deal with these two operators, we can revisit the algorithm by mentioning the changes we need to make:

* Step 1 will now be doing resolution on literals involving exactly two variables, which are not the “Equality”.
* Step 2 will now dispose of all clauses containing literals different from the “Equality” which are in exactly two variables.
* Before proceeding to step 3, we have to expand the equality in the same way as it is described in the Lemma 5 and Lemma 6 in [reference HdN&IPH]. We will present the high-level idea of this expansion letting the group of the ground instances aside:
  + Let R an arbitrary clause in two variables from our set of clauses, containing equality. We will define T(x) as being all of the subformulas in variable x from R and U(y) being all of the subformulas in variable y from R. Hence, we can write R the following way:

∀x∀y(T(x) ∨ U(y) ∨ x=y)

The aforementioned formula is logically equivalent with the following:

∀xT(x) ∨ ∀yU(y) ∨ (∃!x ㄱT(x) ∧ ∀x(T(x) ⇔ U(y)))

For the following formula

∃!x T(x)

there exists as well the following logical equivalence

∃x(T(x) ∧ ∀y(T(x) ⇒ x = y))

The problem which arises with the logical equivalence from above is that our algorithm will enter into an infinite loop, since each attempt of replacing the equality would result into a new formula containing equality again.

The solution for this case is explained as part of the Lemma 6 in [reference HdN&IPH] and it involves the introduction of a brand new constant for each clause containing equality. The construction presented in the paper takes into account unary and binary predicates.

However, if looking at which point in the algorithm we are currently, we could safely skip taking into consideration the binary predicates, since we cannot have any. Our formula will have after the disposal from step 2 only literals in one variable, which could be safely seen as predicates of arity one. (even though we have to be cautious in doing that, since, for example, P(constant, x) and P(x, constant) are representing different predicates of arity one, even though the share the same predicate name)

Combining the observation above with the Lemma 6 from [reference HdN&IPH], we have the following:

∃!x ㄱT(x)

and

T(e) ∧ ∧P ∀x(T(x) ⇒ (P(x) ⇔ P(e)))

are equisatisfiable, where e is a brand-new constant symbol.

* The only thing left to do before safely proceeding to step 3 would be to execute once more the clausal normal form algorithm on the resulting equisatisfiable formula, since this formula is not anymore a valid CNF.

The two subsections from above conclude the second section of the report, which was dedicated to the theoretical part of the project. The next section will present the big picture of the algorithms and how they fit together.

# Design

The codebase is having three main components: the front-end, the intermediate representation and the back-end.

The front-end is responsible for parsing the input, reporting any syntactic and some semantic errors. Furthermore, it also builds the parse tree of the input formula. With the risk of creating confusions, we would not be following precisely the definitions from the compiler terminologies, our parse tree being a hybrid between what a parse tree and an abstract syntax tree are. The reason for that is because all of the operations up to the point in which our formula is in CNF are carried out on this tree, which changes it dynamically. This component is executing all of the operations in polynomial time.

The intermediate representation is mostly dealing with the parse tree, implementing all of the operations necessary for building the clausal normal form efficiently. In the same way as the front-end, all of the operations are executed in polynomial time.

The back-end is probably the biggest part of the codebase, holding the implementation for the unification, basic theorem prover, depth-ordered theorem prover and the two-variable theorem prover. Moreover, the majority of work for the theorem prover is parallelized using multithreading. In terms of time complexity, the majority of the operations are happening in polynomial time, excluding the part involving the two-variable fragment which happens in exponential time, due to the nature of the Cartesian product which is implemented as a backtracking algorithm.

In terms of the language, there we have made few changes since the logic symbols are not present on (probably) none of the existing keyboard layouts. Hence, we will have the following mapping: double implication (<->), implication (->), negation (~), and (^), or (|), equality (=), inequality (!=), universal quantifier (@) and finally the existential quantifier (?). On the top of that, we have adopted the following conventions:

* the terms and predicates will use contain letters of the latin alphabet only
* the predicates will always start with uppercase letter, containing further only lower case letters
* all of the terms will contain lowercase letters only

# Implementation

## Environment

## Front-end

Because the presence of function symbols is implying the undecidability for the satisfiability problem of both first-order logic and the two-variable fragment, we will assume that the input formula for our

### Tokenizer

### Parse Tree

## Intermediate Representation

### The approach for disambiguating the given formulas

### The approach for resolving the precedence

### The approach for eliminating double Implications

### The approach for eliminating implications

### Basic reduction

### Skolemization

## Back-end

### Clause Form

### Basic Theorem Prover

#### Overview

#### Multithreading

### Depth-Ordered Theorem Prover

### Two-Variable Theorem Prover

#### Without Equality

#### With Equality

# Evaluation

## Unit testing

## End-to-End testing

### Using the problems found on tptp.org

### Using general formulas

## Testing against Vampire

# Experiments

# Reflection

# Conclusion

# Appendix 1