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) **5**

[Background](#_orz03eqchgg4) 5

[Motivation](#_8k4aygsvrx9q) 5

[Project aim](#_ax4updv1mq93) 5

[Project Roadmap](#_r2ujw6ayztbf) 6

[Methodology](#_oizawpbvlrdq) 6

[Report Structure](#_2oarfns66zbh) 6

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

[**Context**](#_9zd57b4duhd5) **7**

[First-Order Logic](#_dy6konftaet3) 7

[Definitions](#_djdrgmdkxmng) 7

[Operators](#_2vspvqs2p7le) 8

[Special case: equality](#_wnin7kxxh15l) 8

[Quantifiers](#_fi1ejy2x2bsw) 8

[Precedence](#_dqleivz3fmzk) 8

[Model](#_b2ymtzfccxmi) 9

[Validity and Satisfiability](#_fmzc8s7c2bok) 9

[Soundness and Completeness](#_tu40xscopkem) 9

[Decidability](#_4f0rsqujbia0) 9

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

[Basic Reduction](#_jj13vywtq8bx) 10

[Skolemization](#_soolxzd7jy5g) 10

[Automated Theorem Proving](#_f3q4g59ixn98) 11

[Unification](#_c8vaumr6n4pg) 11

[Resolution rule](#_aqodhlrppqkq) 12

[Factoring rule](#_4p8p51x1sbok) 12

[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) 13

[Two-Variable fragment](#_gioufz95cz1z) 14

[Without equality](#_xq9n8zjme282) 14

[With equality](#_8lahq96r0pgp) 15

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

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

[Environment and technology stack](#_l1vcfc3nbwqf) 18

[Front-end](#_d5si54q21z42) 19

[Tokenizer](#_cn3wt633rot0) 19

[Parse Tree](#_l6i6odkebjow) 20

[Intermediate Representation](#_e3c2dgf365go) 21

[The approach for disambiguating the given formulas](#_vrbua8pgemx8) 21

[Basic reduction](#_xa71vjnr2u09) 21

[The approach for resolving the precedence](#_oja1m8ihg9eo) 21

[The approach for eliminating double Implications](#_ebd5oqbkz1ac) 22

[The approach for eliminating implications](#_hfjgzsolr963) 24

[The approach for pushing the operator not](#_o5k49jl3p72w) 24

[Skolemization](#_bxgow04gybs6) 25

[Simplified Clausal Normal Form](#_abdm34xrxnfz) 26

[Back-end](#_61zrhg2p0mft) 28

[Clausal Normal Form and Unification](#_apo4ra7ygpa6) 28

[Term Unification](#_vmh3sd9x2znq) 28

[Literal Unification](#_p6mijx9wvw29) 30

[Clause Unification](#_vrcfqfte9hhj) 30

[Basic Theorem Prover](#_5shpuwpg832v) 32

[Factoring Rule](#_gjbi0aoldqn4) 32

[Subsumption Rule](#_c7rxg2pyk711) 34

[Resolution Rule and Theorem Proving](#_qwoqx2j13lw2) 34

[Depth-Ordered Theorem Prover](#_kqflx93o42gg) 35

[Two-Variable Theorem Prover](#_2qdj0scw4dg3) 36

[Without Equality](#_wmmcmnyjbk25) 36

[With Equality](#_e5cregib9bw3) 36

[**Evaluation**](#_wb83b0x8anmn) **36**

[Unit testing](#_xmp6tw7l5j9v) 36

[End-to-End testing](#_b1rwdin7qwlu) 36

[Using the problems found on tptp.org](#_mlszyojem44) 36

[Using general formulas](#_2bj5s1nzowqv) 36

[Testing against Vampire](#_gs6mdb6o2163) 36

[**Experiments**](#_w8eibc75csft) **36**

[**Reflection**](#_vct7big78d3b) **36**

[**Conclusion**](#_l2sgo3qyqxj8) **36**

[**Appendix 1**](#_witszixm8988) **36**

[**References**](#_j1y5fgxid3et) **36**

# 

# 

# 

# 

# 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 (2001), published in Gore et al’s “Automated Reasoning” (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.

### Precedence

In terms of the operator precedence we have adopted the following conventions which should be the same as the ones of Vampire (we have decided that for convenience, to be easier to generate random formulas which mean the same thing for both Vampire and our theorem prover). These conventions are the following:

* the round brackets enforce precedence in the same way they do in the majority mathematical systems
* the operator not and both the existential and universal quantifiers are applying strictly to the next:
  + predicate, if they precede one
  + subformula enclosed in a pair of round brackets, if they precede the corresponding open bracket
* the operator and has a strictly higher precedence than the operators corresponding to or, implication and double implication
  + this will happen analogous for operator or and the (double) implications, and for the implication and the double implication

A very important aspect which should be taken into consideration as well is the fact that the implication is right associative.

### 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 (Wilfrid, 1977, p. 242).

**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 Leitsch (1997, p.12-18), 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 Leitsch (1997, p. 99), 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 (Leitsch, 1997, p. 99).

The depth-ordered resolution is sometimes referred as ordered resolution and it is proved as being complete (Joyner Jr. et al., 2002).

## 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 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 De Nivelle and Pratt-Hartmann (2001). 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 De Nivelle and Pratt-Hartmann (2001) 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 De Nivelle and Pratt-Hartmann (2001), 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. In terms of algorithms, this component uses mostly ad-hoc algorithms (which will be further described) or depth-first search. 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 terms of the implementation, the reader will probably notice further that many of the methods implemented for reducing the formula are making assumptions on the structure of the input instead of enforcing that structure (this is because enforcing that would imply more work in terms of the resources than the respective method is supposed to do). The majority of the functionality, if not all, executes the reducing operations using a depth-first search. 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. Implementation-wise, this component makes the most usage of both ad-hoc algorithms and classical algorithms (breadth-first search, depth-first search, backtracking, 2-colorability). Moreover, sophisticated data structures (such as persistent data structures) are as well involved. 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 always 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 always contain lowercase letters only

As mentioned before, we will assume that the input formula does not contain function symbols, even though the CNF of it could end up having (due to the Skolemization).

Below it is presented a diagram which gives an idea about how the three aforementioned components interact with each other. The name of the subcomponents are as well revealed, in order to offer a hint to the reader in regards to what will follow in the next section.

{insert image here}

# Implementation

This section will be dedicated fully to the implementation details of the algorithms discussed in during the course of the previous sections.

## Environment and technology stack

As an operating system, both Windows 10 Pro and Ubuntu 20.04 were used. Ubuntu was actually run as a Windows subsystem, enabling the programmer to utilise both the Linux infrastructure and the Unix-like commands.

In terms of hardware, a computer having a 7th generation Intel i7 processor and 32GB of RAM memory was used. During the implementation process, the generous memory resources were quite useful, since an unoptimized version of the theorem prover was using more than 18GB of RAM memory on some input formulas which were saturating. After noticing that, the programmer optimized the code in such a way such that the amount of memory used by the theorem prover was drastically reduced (below 1GB of RAM memory on similar or the same tests).

In terms of programming languages used, the choice was to use C++ Standard 17 for the implementation (since the programmer had the most experience with this programming language but also for runtime speed C++ is offering). Standard 17 was a must since the implementation makes extensive use of std::variant, which is a type safe union (CPP Reference, 2021) and of std::execution (CPP Reference, 2021) which provides the execution policies which could be used when parallelizing code. However, the latter is not fully implemented by some C++ 17 compilers so I had to import it manually by linking my code with the libtbb library (Debian, 2021) which provides the implementation. In order to unit-test some parts of the code, the programmer used Google's C++ testing framework (Googletest, 2021). Other programming languages which have been used during the implementation of this project are Python3.8 and Unix Bash. The former was used mostly for writing input generators while the latter was used for End-to-End testing or for making comparison before our theorem prover and Vampire.

As a building system, CMake was strongly preferred due to the powerful set of options but also due to the fact that Google's C++ testing framework was available to build the easiest for CMake-based projects.

## Front-end

This section is going to present the algorithms behind the front-end. As probably noticed, the workflow of the (sub)components typically tends to get an input and produce an output. The front-end uses exactly this mechanism: the tokenizer will be receiving the raw input and produce an ordered list of tokens; further, the parse tree will be passed this list of tokens and will produce the initial parse tree.

### Tokenizer

As described before, the purpose of the tokenizer is to take the raw input and to parse it into tokens. For better understanding, we will define what a token could be:

* an atom
  + that would be any string matching the following: it starts with uppercase letter, then it has an arbitrary number of lowercase characters (possibly zero) and it further has a pair of round brackets within a set of variables (lowercase strings having a positive length) separated by commas is nested
  + since our convention is that predicates are always starting with capital letter and there are no other entities which could possibly start in the same way, the detection of atoms is trivial: each time when we see a capital letter we start scanning up to the first closed bracket
* an operator
  + this is trivial as well, since each operator could be detected with one symbol lookahead
    - the only two particular cases are happening for equality and inequality: they have to be replaced with their corresponding predicates (called “Equality” or “Inequality”)
* a quantifier
  + as for operators, this would be quite easy to detect as well: the current pointer in our raw input has to point to either @ or ? and be followed further by a positive number of lowercase letters (forming the variable)

Apart from detecting the tokens above and eventually reporting errors, the tokenizer will have to potentially add manually a conjunction in the ordered list of tokens. (the conjunction which needs to be added for preserving the logical equivalence for the inequality)

### Parse Tree

Our parse tree will look like a normal tree, with the mention that we will store some information in each node, which in our code is referred as entity. A node might not store any entity. For the nodes storing entities, these could encapsulate 4 types of entities:

* quantified variable (i.e. @x or ?y)
* literal (i.e. P(x) or ~R(f(x)))
* operator (and, or, not, implication, double implication; note that the tokenizer will have replaced both the equality and inequality with predicates by this point)
* normal form (we could ignore it for the moment because we will analyze it in the section dedicated to the intermediate representation implementation)

It could be also the right time to reveal the reasons for choosing to store our formula this way:

* for some nodes, the following property holds: the subtree rooted in each of them represents a subformula of our initial formula
* as you probably noticed above, the entity which encapsulates an operator does not include the brackets; this is because we can use the pre-order traversal on the tree for representing the precedence enforced by the brackets
  + hence, our design choice already reduced drastically the initial language for the formula; we managed to get rid of equality, inequality and brackets already

Now the only thing left is to do is to present the algorithm which makes the transition from the ordered list of tokens to the parse tree. We will present the high-level idea of the algorithm. Note that some details are intentionally omitted for the sake of simplicity:

* we will declare a stack, called fatherChain
* we will dynamically add the first node to the parse tree, which is the root
* we will push the root to fatherChain
* let tokens be the ordered list of tokens
* for each token in tokens
  + if the token is open bracket, an operator, a quantifier or a literal
    - let newNode be a new node added to the parseTree
    - add the newNode in the adjacency list of the node from the top of fatherChain
    - push newNode in the fatherChain
    - if the token was an operator, a quantifier or a literal, assign a new entity to the node newNode, encapsulating whichever type of entity it is
  + if the token is closed bracket
    - pop the top from fatherChain

This algorithm runs in O(|Tokens|) time and consumes O(|Tokens|) memory, where |Tokens| is the number of tokens.

## Intermediate Representation

We will now present the functionality of the intermediate representation, which we expect to seem much more complex to the reader, since quite a lot of time invested in the implementation of the project was dedicated to polishing this component. This component is represented in the code under the name “Reducer”, which acts as a friend class for the class Parse Tree (which we have just presented). The name “Reducer” is consistent with its functionality, keep reducing the parse tree until it holds the CNF in the root.

### The approach for disambiguating the given formulas

Similarly to what happens in many programming languages, it is perfectly legal to have two variables with the same name, one nested in the scope of another, being bounded by different quantifiers. This might represent a challenge for us later unless we take action now. Hence, what we will do would be to run a depth-first search maintaining an accumulator with all of the variables which have been seen so far. If we are currently in a node holding a quantified variable, we should add the respective variable in the accumulator. In case, the variable already exists in the accumulator, we will have to map it to a brand-new variable (which does not exist in the accumulator). Furthermore, we will have to remember that at this step we have added a mapping in order to remove it after having traversed the entire subtree rooted in this node. Moreover, while going down the traversal, any occurence of the variable which we have mapped at this step will have to be substituted with the corresponding mapping.

In order to implement this part efficiently, we modeled the accumulator as an std::unordered\_set (CPP, 2021). in order to be able to insert and search in constant time.

The time complexity of this part is O(N), where N is the number of nodes our parse tree is having by the time of traversal.

### Basic reduction

This section will be presenting all of the tools needed in order to achieve the “Basic reduction” discussed in the theoretical part of the report.

#### The approach for resolving the precedence

Another interesting and challenging aspect of our parse tree is how do we deal with the precedence on the remaining operators. As already mentioned earlier, the pre-order traversal takes care of the precedence enforced by the brackets. This is our main tool of dealing with the precedence for the other operators as well.

We will discuss the first case, the one of operator and. The remaining cases could be treated analogously as the main idea does not change at all. We will assume that we are at an arbitrary node in our parse tree, and the neighbours of our parse tree are having various types of entities (literals, operators, quantifiers, including not having any).

Before proceeding, there are few observations we need to make about our parse tree, assuming that it represents a syntactically valid logic formula:

* the subtree rooted in a neighbour having no entity associated represents a syntactically valid logic formula
* the order in which the neighbours occur is very important, since it uniquely determines the formula
* no two consecutive neighbours could have operators as entities, because otherwise our formula would be invalid syntactically

Having these observations in mind, we could deal with the precedence by adding new nodes in the parse tree, which would be equivalent with adding parentheses. Precisely we will replace some of the current neighbours by brand-new nodes having into their adjacency list only the node they are replacing.

For resolving the case for the operator and we need to take maximal groups of consecutive neighbours with the property that the only operator occurring in the multiset of their entities is the operator and. For each group, we will add all of the nodes belonging to it in the adjacency list of a brand-new node, by preserving the relative order between them. Finally, we will replace each group of nodes in the original adjacency list by the brand-new node corresponding to the respective group.

After successfully doing this, our parse tree will have the precedence for the operator and resolved, so we could move on to the operator or. The procedure will be identical, since we will not interfere with the operator and at all. We can resolve the case for implication and double implication analogously.

Even though resolving precedence might seem complex, the way we implemented it was quite straightforward, by using four calls to a depth-first search function, one for each operator.

The time complexity of this part is O(N), where N is the number of nodes our parse tree is having by the time of traversal.

#### The approach for eliminating double Implications

After we have an unambiguous formula in our parse tree, the next step would be to reduce it to an even simpler sub-language: one which does not contain double-implications. It is well known that the double implication could be splitted into a conjunction of two implications, but the problem which would arise if we would proceed this way is that the size of our formula could grow exponentially (i.e. P(x) ⇔ (R(x) ⇔ Q(x)) will further transform in P(x) ⇔ ((R(x) ⇒ Q(x)) ∧ (Q(x) ⇒ R(x))), finally becoming (P(x) ⇒ ((R(x) ⇒ Q(x)) ∧ (Q(x) ⇒ R(x)))) ∧ ((R(x) ⇒ Q(x)) ∧ (Q(x) ⇒ R(x))) ⇒ P(x)); as we can notice, the initial formula has one occurence of Q, the next one has two while the final one has four).

In order to prevent this issue from happening, we will be always replacing the deepest occurence of a double implication with a brand-new predicate (for the purpose of this section the reader could assume that the replacement of a double implication consists of replacing the operator, the left-hand side and the right-hand side, respectively). In order to preserve the logical equivalence while doing that, we will have to add a new conjunction to our formula, containing a double implication between the brand-new predicate introduced and the subformula it is replacing. Repeating this procedure until we do not have a double implication operator nested in a subformula which is further part of a double implication will finally produce a logical equivalent formula on which it is safe to apply the expansion described in the first paragraph without having to worry about any exponential blow-up.

Implementing the replacement of the double implications with predicates would be quite trivial to perform on the parse-tree:

* we visit all the nodes using a depth-first search
* whenever we detect an occurrence of the double implication we check whether it is the deepest in the parse tree and if it is, we do the following
  + we will take the three nodes forming it (left-hand side, double implication, right-hand side) and we will append them to the adjacency list of a brand-new parse tree node, called R; note that appending the three nodes refers to moving the entire subtrees rooted in them
  + we will create two brand-new nodes in our parse tree, both holding the same brand-new predicate as an entity; we will refer to these two nodes as T and P
  + we will now replace the three nodes forming the initial double implication with T
  + finally we will append at the end of the adjacency list of the root the nodes C, P, D, R in this order; C and D are brand-new nodes holding the operator and and the double implication, respectively; these four nodes simulate the appendance of the conjunction needed for preserving the logical equivalence
    - we omitted here some details which take care of the precedence for the sake of simplicity

We will detail as well the algorithm of getting rid completely of double implications having performed the algorithm above by this point:

* we visit all the nodes using a depth-first search
* whenever we detect an occurence of the double implication, having the nodes L, D, R we do the following
  + we create a deep copy of node L in node L’
  + we create a deep copy of node R in node R’
  + we create a brand-new node I holding an implication as an entity
  + we change the entity held on D to an implication
  + we create a brand-new node C holding the operator and
  + we will now replace the nodes L, D, R with the nodes L, D, R, C, R’, I, L’ in this order
    - we omitted here some details which take care of the precedence for the sake of simplicity

Because there is a polynomial number (in respect with the size of the formula) of double implications in the parse tree, the complexity of performing this transformation is polynomially bounded. In fact, if N is the number of nodes in our parse tree, we would expect it to fourfold in the worst case. (it would double for performing the optimization for replacing the double implications with brand-new predicates and it would double further once more for performing the expansion)

Hence, the time complexity of this reduction is O(N), with N defined above.

#### The approach for eliminating implications

In the previous section we have shown how to eliminate the double implications. Hence, the only operator left which we need to eliminate is the implication. Due to its nature, we will not have to face the same challenges as for the double implication. The reason for this is because the implication from left-hand side to the right-hand side is equivalent with the disjunction between the left-hand side negated and the right-hand side, which does not double the number of occurrences of any predicate which is part of any of the sides.

The algorithm is therefore straightforward:

* we visit all the nodes using depth-first search
* whenever we find for a node Q three consecutive neighbours matching the pattern left-hand side (referred as L), implication (referred as I), right-hand side (referred as R) we do the following:
  + create a brand-new node K, holding the operator not as an entity
  + add L in the adjacency list of K (as being the only direct son)
  + replace the occurrence of L in the adjacency list of Q with K
  + change the operator node I is holding to disjunction
  + leave R unchanged

Since all of the operations which are executed when finding an implication could be performed efficiently in O(1), the time complexity for this part is O(N), where N is the number of nodes our parse tree is having by the time of traversal.

#### The approach for pushing the operator not

The only outstanding task after having implemented all of the previous approaches is to push the operator not as far as possible and by that we mean that it has to be pushed past quantifiers, disjunctions of conjunctions. Moreover, simplifying consecutive occurrences has to be done as well.

Due to the very convenient sub-language of which our parse tree is now part, we can go ahead by just implementing the algorithm as it is:

* we visit all the nodes using depth-first search
* whenever we are in a node of which assigned entity is the operator not we iterate through its neighbours and do the following depending on the entity held by the neighbour:
  + no entity transforms in the operator not
  + operator
    - not, transforms in no entity
    - and, transforms in or
    - or, transforms in and
  + quantifier
    - existential transforms in universal and moreover, an intermediary brand-new node below the neighbour and its son is created, holding the operator not
    - universal transforms in existential and moreover, an intermediary brand-new node below the neighbour and its son is created, holding the operator not
  + literal
    - the sign gets changed and the operator not will not go past it
  + normal form
    - nothing, because by this point our parse tree does not have any node holding this sort of entity
* one aspect which has to be highlighted is that we perform all of the changes described above before going down the depth-first search traversal
  + hence, implementing the same changes but using a breadth-first search would have probably been more natural

Since we visit each node exactly once and the changes which take place to the entities of the nodes are executed in constant time, the time complexity for this approach is O(N), where N is the current number of nodes from the parse tree.

### Skolemization

Once we have only conjunctions, disjunctions and the negations are sticked to the predicates, we could proceed in implementing the Skolemization:

* we visit all the nodes using depth-first search, by maintaining a stack for the universally quantified variables which are lying on the path from root to the current node; moreover, we will also maintain a std::map [reference <https://en.cppreference.com/w/cpp/container/map>] (called M) containing the mapping for the existentially quantified variables to their corresponding functions (and their respective arguments)
* in case the current node holds as an entity an universally quantified variable, we will push the corresponding variable to the stack; when we will be done with visiting the subtrees rooted in all of its neighbours we will pop that variable off the stack
* in case the current node holds as an entity an existentially quantified variable K, we will create a brand-new function symbol F and we will add an entry in the map M for the key K, with the value corresponding to a tuple holding F and a copy of the current state of the stack (which holds the variables within K nests)
* in case the current node holds a literal as an entity, we would iterate through the terms of it and for each variable having an entry in the map M, we will replace its occurrence with the corresponding value from the map (that is, the Skolem function or constant)

After the algorithm is performed, we could get rid also of all universal quantifiers. This will not affect the correctness since during the execution of the Skolemization we kept track of the types of terms (a detail which was omitted in the presentation of the algorithm for the sake of simplicity). Having a mechanism to uniquely determine the type of the terms enables us to execute the removal of the universal quantifiers without losing any information.

We will now analyze the time complexity of our implementation for Skolemization. The operations performed on the map M will be done efficiently, on a logarithmic complexity. However, the amount of variables stored in M could be quite large in the worst case scenario (i.e. a formula which has a very high number of nested quantifiers), precisely a number of variables bounded by N2 (where N is the number of nodes from the parse tree). Hence, the time complexity of our algorithm is O(N2log2N).

### Simplified Clausal Normal Form

After having performed all of the algorithms described in this section, we are in a position of finalizing the construction of the clausal normal form.

Before continuing, we have to establish that we will attempt to build the simplified version of the clausal normal form. That is, we will retrieve the CNF in a convenient format for now, but we will change the way of representing it after the retrieval (this is going to be detailed in the section dedicated to the back-end). Hence, we will model the CNF as an array of arrays (each containing literals). This representation is completely accurate assuming that between the inner arrays there are conjunctions and between the elements of an inner array are lying disjunctions.

We will now present the algorithm which merges two arbitrary CNF (we will call it “merge(A, B)”) having between them an operator (either conjunction or disjunction; we will discuss both of the cases):

* declare toBeAdded globally, as being a set of disjunctions which need to be appended to our formula as conjunctions
* let A and B be two CNFs, each stored as discussed, as an array of arrays containing literals
* if the operator between A and B is a conjunction, create C as a concatenation of A and B and return C
* otherwise the operator is a disjunction and we have three cases:
  + if either of A or B are empty, return the another one
  + the trivial case, both the sizes of A and B are 1
    - create C having a single element, the concatenation of the first (and only) element of A and the first (and only) element of B
    - return C
  + otherwise, this means that either A or B violate the condition from above
    - we will therefore apply the following algorithm on each of them twice:
      * let be D the input of the algorithm, passed by reference (i.e. the modifications on it are visible in the calee)
      * if all of the elements from D are having size 1, then:
        + create a brand-new predicate F
        + for each element P of D add (ㄱF ∨ P) in toBeAdded (note that what we are actually adding is an implication)
        + D becomes [[F]], where a pair of square brackets represent an array
      * otherwise, we do the following:
        + for each element P of D

create a brand-new predicate Fi

add (ㄱFi ∨ P1 ∨ P2 … ∨ Pj-1 ∨ Pj) in toBeAdded, where j is the size of P (note that we have added in implication)

P becomes F

* + - * after we have applied the algorithm twice, we can call once more merge(A, B)

Finally, after the algorithm is completed, all of the elements from the set toBeAdded have to be appended as actual conjunctions to the resulting CNF. We will come back to this later, after we will have all of the CNF built, apart from this detail.

After having presented the algorithm which merges two CNFs on conjunction or on disjunction, we can now move ahead in detailing the algorithm for retrieving CNF from the parse tree:

* we will visit all of the nodes, using a depth-first search
* when we are about to come back from the recursive call for a node P, we are going to make some changes to the entity held by it:
  + if it does not hold any entity, it will now hold an empty CNF (i.e. [ ])
  + if it holds a literal L, it will now hold a CNF containing only that literal (i.e. [[L]])
  + if we are in any of the two cases presented above, we will iterate once more through the neighbours of P, calling subsequently the merge function between its corresponding CNF and the CNF of each of its neighbours which are holding one as well
    - finally, the node P will hold as an entity the CNF corresponding for the its whole subtree
* after executing the algorithm for all of the nodes, the CNF of the whole parse tree will reside in its root

As described before, we have now to append the disjunctions from toBeAdded as conjunctions to the CNF we have retrieved from our parse tree. This could be done straightforwardly by concatenating the former to the latter.

Hence, we have shown how the retrieval of the CNF is done. However, a good question one would ask will be why we have proceeded in such a convoluted manner (easier implementations being trivial to spot at a glance). The reason which justifies our complex implementation is the avoidance of an exponential blow-up (as in the case of the double implications). While merging the CNFs, we are actually applying the distributivity laws efficiently. If we would do that naively, by doing the cartesian product of the two CNFs, the size of the CNF will evidently face exponential growth.

Analysing the complexity of the merge algorithm, we could conclude that it introduces a polynomially (in respect with the size of CNF) bounded amount of additional conjunctions. In the worst case, the time complexity of this part will be O(N), where N is the number of nodes from the parse tree.

The depth-first search algorithm runs in polynomial time, precisely O(N2). Even though the programmer could have came with much more sophisticated solutions (such as one involving the small-to-large trick, optimizing the time complexity to O(N log2N) overall), it was considered that it is not necessary, since that solution will represent only a micro-optimization (being dominated by O(N2log2N), from the Skolemization step).

Hence, the overall time complexity for the retrieval of the Simplified Clausal Normal Form is O(N2).

## Back-end

During the course of this section we will discuss the theorem proving part of the project. We will start by first describing the data structure chosen for representing the clause form. After that, we will detail the implementation of the three theorem provers (basic, depth-ordered and two-variable).

### Clausal Normal Form and Unification

Since the theorem proving might require complex representations of the literals from the CNF, we are going to change the way of representing the clausal normal form. Instead of seeing it as an array of arrays, as used and described in the intermediate representation, we would need a much more convenient way of storing (in respect with the unification).

Hence, the CNF will now encapsulate an array of clauses. A higher level of granularity is needed, so each clause will encapsulate an array of literals. Finally, a literal will store behind its representation an array of terms. A term could be quite complex following the definition (i.e. for example a function symbol having nested other terms). With the risk of creating confusion, a term will further represent an array of terms. The figure from below will hopefully help the reader in gaining a better understanding.

{insert image here}

In order to present the unification, we will use a bottom-up approach (i.e. we will present the algorithm from terms, then the transition to literals and finally the transition to clauses; the last part will be later plugged in the basic theorem prover).

#### Term Unification

The first tool which will be described as part of this section is the recursive function called “findPartialSubstitution”. This is going to take as an input two terms, A and B. It will try to unify A with B and return true in case it succeeded, false in case it fails and the substitution in case it partially succeeded. In case of failure, we will try to unify B with A as well. Finally, in case of failure A and B are not unifiable. The algorithm for findPartialSubstitution is the following:

* if A and B are identical, then return true
* otherwise, check the following:
  + if A is a constant, return false
  + if A is a variable, called V
    - if B contains that variable, return false
    - otherwise return a tuple containing V and the whole term of B, signifying that V is mapped to B (which could be a function, a constant or another variable
  + if A is a function symbol, then
    - if B is not a function symbol, return false
    - otherwise, do the following:
      * if A and B are not having the same function name, return false
      * otherwise, for each position in the list of arguments of A and B, try to unify their corresponding terms (i.e. for f(x, y, z) and f(a, b, c) we will try first x with a, then if successful y with b and finally if successful z with c)
        + if the return value R for the current unification is true, continue to the next pair of terms (because this means that the pair of terms we have tried to unify was identical)
        + otherwise, return R
      * finally, if nothing was returned yet, return true

In order to establish the time complexity of the aforementioned algorithm, we will have to introduce the function treeSize, having the domain in the set of terms (let it be called T) and the co-domain in the set of natural numbers (i.e. treeSize : T → Ｎ), with the following recursive definition for treeSize(K):

* if K is a variable or a constant, treeSize(K) is 1
* if K is a n-ary function symbol with the name g, then K could be written as g(t1, t2, …, tn); therefore,

Having defined the treeSize, we could now establish the time complexity for findPartialSubstitution: O(M2) where M is the minimum between treeSize(A) and treeSize(B). This is because at each call of the function the verification for equality of A and B traverses the tree of each of them and for highly nested A and B, we would perform this check as many times are nodes in the tree of term with lowest number of nodes.

The second tool which we will present is a function called “augmentUnification”, which takes as an input two terms, A and B. This function is way simpler than the previous one, representing just a “wrapper” for findPartialSubstitution. Its purpose is to call findPartialSubstitution for A and B. In case of failure, it will call the same function but for B and A (so the arguments are just reversed). Finally, if both of the calls failed, the function will be returning false. Conversely, if both of the calls returned true, this function will be returning true. Otherwise, the function will be returning a substitution (revisit the previous algorithm for better understanding). If the reader is familiar with the Python programming language, one could make an analogy between this function and a Python generator. (i.e. if it does not return false at the initial call, it will keep returning substitutions until either of true or false is reached; if false is reached after a couple of substitutions were returned, this means that we have to roll back the operations --- this will be explained later). The function augmentUnification has the same time complexity as findPartialSubstitution.

Finally, the third tool is a function called “createDeepCopy”, having as input one term. This function will create a deep copy of the input and return it. The implementation uses a variation of the depth-first search and we will not present the details of it (since the algorithm is trivial and does not contain anything sophisticated). We will need this function in the next subsection. The time complexity for it is O(R), where R is the value of the treeSize for the input term.

#### Literal Unification

In the previous section we have discussed three tools (or functions) we are going to use in the back-end. In order to avoid confusion, we will refer to functions “augmentUnification” and “createDeepCopy” as “augmentUnificationTerm” and “createDeepCopyTerm” from now on.

We are going to describe the function “augmentUnification”, which applies the algorithm of unification on literals. This gets as an input two literals. Similarly, after this section we are going to refer to it as “augmentUnificationLiteral” for better clarity.

This function will do the following:

* it will compare the literals predicate-wise
  + if the predicates are different, returns false
  + otherwise, it will call augmentUnificationTerm for each argument of the predicates
    - if we are at a pair of terms and the algorithm returns false, we will return false further
    - otherwise, if it returns true, we move to the next pair of terms
    - otherwise, if it returns a substitution, we return it
  + finally, if we did not return yet, we will return true

The time complexity of this function is O(), where E is the maximum between the length of the predicates, N is the number of arguments the predicate is having and Mi is defined as being the minimum between the values of treeSize for the ith pair.

Similarly with the previous section we will introduce a function called “createDeepCopyLiteral”. This function will create a deep copy of a given literal. The complexity of this function is O(), where F is the length of the corresponding predicate, N is the number of terms (arguments) the predicate is having and Ri corresponds to the treeSize value of the ith term.

#### Clause Unification

This functionality was implemented using C++ templates [reference https://en.cppreference.com/w/cpp/language/templates] and it represents the core of theorem proving. Once we will have detailed it, it would be reasonable to expect way shorter sections describing the theorem proving. It is also worth mentioning that this part makes use of most modern C++, std::variant (CPP Reference, 2021) and lambda expressions [reference <https://en.cppreference.com/w/cpp/language/lambda>] being heavily used.

We are going to describe the function called “attemptToUnify”, which is a templated function receiving as parameters the following:

* the first clause, F
* the second clause, S
* a function returning a boolean, called literalCheck
  + this function expects two literals as parameters
* a function returning a boolean, called resolventCheck
  + this function expects a literal and a clause as a parameter

In the actual implementation, both literalCheck and resolventCheck are implemented using lambda expressions, which justifies the reason for which attemptToUnify is a templated function (since the types of the functions have to be passed as template arguments).

We will now present the algorithm behind attemptToUnify. Some of the details might be omitted, for the sake of simplicity and clarity. The algorithm does the following:

* declares an array of clauses, called result; this is going to return all of the possible clauses derived using the factoring rule from the clauses F and S
* for each literal X in F
  + for each literal Y in S
    - if literalCheck of X and Y returned true, then
      * assign to F’ a deepCopy of F
      * assign to S’ a deepCopy of S
      * make sure that F’ and S’ do not have common variables
        + in case they do, rename the variables in such a way such that they do not anymore
      * while augmentUnification applied on F’ and S’ returns a substitution
        + apply the substitution on both F’ and S’
      * finally if augmentUnification returns true, then
        + commit this unification, by removing X (which in the meantime might have gotten substituted) from F’ and Y from S’ (analogous as for X) and concatenating S’ to F’
        + append F’ to result, only if resolventCheck of X and F’ succeeded

Note that the appliance of substitution represents only a mapping from one term to another one and it could be implemented in O() for an N-ary predicate, with Ri corresponding to the value of treeSize for the ith term of it.

Similarly, the step of renaming all of the variables from F’ and S’ could be implemented using the subsequent appliance of multiple substitutions. We could do that by only retrieving all of the distinct variables from F’ and S’ in two sets, VF’ and VS’, respectively. Once we have these two sets, we have to compute the intersection of them, and then assign for each variable which is part of the intersection a brand-new variable (which is not part of neither of VF’ and VS’). This assignment will be performed on only one of the clauses F’ or S’. The complexity of this step is O(), assuming that C is the cardinal of the intersection of VF’ and VS’ and R and Q corresponds to F’ and S’, respectively. The first sum corresponds to the traversal of F’ (in order to retrieve all of the variables), the second part corresponds to the traversal of S’ (analogous as for F’) and the last part corresponds to the number of variables which will be substituted multiplied by the size of the term trees of F’ and S’ (since we do not apply any criteria of choosing the clause which will be substituted when assigning new variables for those which are part of the intersection).

Finally, the complexity of the attemptToUnify algorithm is O( ), where:

* |F| and |S| are the size of F and S, respectively
* LC is the number of operations literalCheck is performing for a pair of literals in the worst case (this cannot be determined clearer since this function is passed as a parameter to the attemptToUnify, so we do not know its complete implementation)
* RC is the number of operations resolventCheck is performing for a literal and a clause in the worst case (as for the literalCheck function, this can be only determined at the runtime, since the function is passed as a parameter of attemptToUnify)
* the last part of our time complexity analysis is the same as for the variable renaming step, with the mention that in the worst case, we will have to rename in addition the maximum between the cardinals of VF’ and VS’

### Basic Theorem Prover

During the course of this section we will present the details of the basic theorem prover, which is the algorithm which stays behind all of the theorem proving algorithms implemented as part of this project.

This part is by far the most convoluted from the entire project. Because of that, we will try to present a simplified version of the implementation of the algorithms (for the sake of simplicity and clarity).

This component makes use of multithreading and it is also persistent (in the sense that clauses could be dynamically appended and popped while a timeline of these operations is maintained). For the persistence part, we will be keeping track of a timestamp named “currentTimestamp” (in fact, this is more complex on the implementation side, but presenting the mechanism this way will give a solid enough understanding to the reader).

Because the basic theorem prover might not terminate, and hence, the termination is not in our control, we might sometimes decide to skip the complexity analysis.

Apart from the current set of clauses (named “AllClauses”), which is always maintained chronologically (in order to not break the persistence mechanism), the basic theorem prover keeps track of all of the clauses seen so far (by maintaining a set of clauses called “HSet”). Because the clauses could be quite large, a hashing function which relabels the variables was used (i.e. ~P(x, f(x), y, x, g(y)) becomes ~P(v1, f(v1), v2, v1, g(v2))). Again, we try to reduce the risk of overloading the reader with implementation details, so we will not describe the implementation of the hashing function.

#### Factoring Rule

This algorithm implements the factoring rule described in the theoretical part of this report. For better integration of the multithreading, the programmer decided to implement the tautology removal and the clause unification (with itself) as part of this procedure.

As it could be easily noticed, this part is executed on one single clause. Hence, multiple clauses could be treated completely independent. This is the reason why the programmer decided to make use of multithreading here, by parallelizing this procedure.

We will further describe this procedure using a top to bottom approach (i.e. the high-level overview first):

* let C be the input clause
* if C is marked as deleted, then return
* let H be the current hash of C
* perform removeDuplicates procedure on C
  + if C was changed then
    - compute H’ as the new hash of C
    - remove H from HSet
    - insert H’ to HSet
    - assign H’ to H
* perform isTautology for the clause C
  + if C was detected as being a tautology
    - mark C as deleted and save additionally the timestamp of this moment; then return
* let unificationResult be a list of clauses
* assign to unificationResult the list of all clauses resulted by unifying two literals of the same sign from C
* iterate through unificationResult
  + if the current clause currentClause is not a tautology then
    - perform removeDuplicates on currentClause
    - compute currentHash as the hash of currentClause
    - if currentHash is does not exist in HSet then
      * append currentClause in the AllClauses

The procedure removeDuplicates takes as input a clause and removes literals from it such that a literal does not occur multiple times.

The procedure isTautology takes as input a clause and searches for two atoms L and L’ such that they are identical apart from the fact that they have opposite signs. If it finds such a pair it will then report that a tautology has been found.

Finally, the procedure unification result is a variation of the function attemptToUnify. This variation could be regarded as an actual call to the attemptToUnify procedure with the following mentions:

* F and S will point to the same thing
* the step in which the intersection of VF’ and VS’ is enforced to be empty will be omitted
* literalCheck will return true only if the two literals are having the same predicate name and the same sign
* resolventCheck will always return true
* the time complexity will be the same as for attemptToUnify with the mention that C (the cardinal of the intersection of VF’ and VS’) will be equal to 0

#### Subsumption Rule

The subsumption is executed on a single clause, in the same way as the factoring rule does, independently from the other clauses from AllClauses. Similarly, since this isolation is enforced by the logic of the algorithm, the code corresponding to the implementation could be parallelized.

We will now describe the algorithm:

* let C be the input clause
* if C is marked as deleted, return
* let CHS be the ordered set of the hashes for its literals (i.e. for the clause (~P(x) ∨ L(x, y) ∨ R(z, y, x)), the corresponding the set is {L(v1, v2), ~P(v1), R(v3, v2, v1)})
* we will now iterate through all of the other clauses which have not been deleted
  + for each clause E
    - we compute EHS
    - if EHS is included in CHS, then C is subsumed by E
      * C gets marked as deleted and we remember the timestamp at which it got marked as deleted
    - if CHS is included in EHS, then E is subsumed by C
      * E gets marked as deleted and we remember the timestamp at which it got marked as deleted

The algorithm for subsumption works for ground clauses and it might work as well for clauses very dense in terms of variables. The reason for the latter is because relabelling the variables in the way described in one of the clauses might not match the relabelling from another clause. However, a correct implementation for non-ground clauses might be performed in exponential time, which would outweigh the benefits.

#### Resolution Rule and Theorem Proving

We will describe the resolution rule together with the previous two rules, as part of the basic theorem proving. Similarly with the subsumption and factoring rules, this rule could be implemented as well using multi-threading.

We will now present the basic theorem proving, which makes use of the previously discussed function, attemptToUnify. Hence, the algorithm is the following:

* literalCheck applied on two literals will return true if the corresponding two predicates are the same, but the literals are of opposite signs
* resolventCheck will return true no matter what is the input
* let repeat be a boolean initialized with true
* while repeat is true do
  + set the variable repeat to false
  + for all clauses C in AllClauses do the following (this step creates separate threads in the actual implementation):
    - if C is marked as deleted, then continue
    - for all clauses G in AllClauses do the following
      * if G is marked as deleted, then continue
      * let unificationResult be a list of clauses
      * assign to unificationResult the result of attemptToUnify, having as arguments C, G, literalCheck and resolventCheck
      * if unificationResult is not empty, then do
        + iterate through all of the clauses J from unificationResult

if the clause J is a tautology, then continue

perform removeDuplicates on J

let H be the hash of J

if H exists in HSet, then continue

otherwise do the following

insert H to HSet

if J is the empty clause, then

return true

append J to AllClauses

set the variable repeat to true

* finally, if we reach this point, return false (we have saturated)

This concludes the presentation of the basic theorem prover. The next section will be presenting the depth-ordered theorem prover, which is a very slight variation of the former.

### Depth-Ordered Theorem Prover

The depth-ordered theorem prover is identical with the basic theorem prover described in the previous section with the exception that it has a different implementation for the resolventCheck. Hence, we will discuss only the implementation of this function during the course of this section.

As before, we will use a top-to-bottom approach for presenting the resolventCheck function. The following implementation will make use of a function called isAOrdering, which is going to be detailed later.

Hence, the resolventCheck function does the following:

* let L and C be the input of the function, the resolved literal and the clause, respectively
* let LC be the set of all the literals which are part of the clause C
* for each literal T in LC
  + if isAOrdering applied on L and T (in this order, because the order matters) returns true then
    - return false
* return true

We will now define the function isAOrdering:

* let A and B be two literals, the input of the function
* let AMax and BMax be the maximum depth of a variable in A and B, respectively
* if A does not have variables, return false
* if B does not have variables, return false
* if AMax ≤ BMax, return false
* let AVars and BVars be the set of variables from A and B, respectively
* for each variable V in AVars
  + let AV be the maximum depth of V in A
  + let BV be the maximum depth of V in B
  + if AV ≥ BV, return false
* for each variable V in BVars
  + let AV be the maximum depth of V in A
  + let BV be the maximum depth of V in B
  + if BV ≤ AV, return false
* return true

The changes mentioned above are the only ones needed to transform the basic theorem prover into a depth-ordered theorem prover. The time complexity

### 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

# References

De Nivelle, H. and Pratt-Hartmann, I. (2001). ‘A Resolution-Based Decision Procedure for the Two-Variable Fragment with Equality’, *Automated Reasoning,* pp. 211-225.

Gore, R., Leitsch, A. and Nipkow, T. (2001) ‘Automated Reasoning’, Proceedings of the First International Joint Conference, IJCAR, Siena, Italy, 2001, pp.211-225

Wilfrid, H. (1977). *Logic.* 1st edn. Penguin.

Leitsch, A. (1997). *The Resolution Calculus.* 1st edn. Springer.

Joyner Jr., W.H., Fermueller, C.G., Leitsch, A., Hustadt, U. and Tammet, T. (2002). ‘Ordered Resolution’ [PowerPoint presentation]. Available at: <https://people.mpi-inf.mpg.de/~hillen/documents/4_OrderedResolution.pdf>

(Accessed: 7 April 2021).

CPP Reference (2021). *C++ Utilities Library: std::variant.* Available at: <https://en.cppreference.com/w/cpp/utility/variant> (Accessed: 7 April 2021)

CPP Reference (2021). *C++ Utilities Library: std::execution.* Available at: <https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t>

(Accessed: 7 April 2021).

Debian (2021). *Package: libtbb-dev (2020.3-1 and others).* Available at: <https://packages.debian.org/sid/libtbb-dev> (Accessed: 7 April 2021).

Googletest (2021). *Googletest.* Available at: <https://github.com/google/googletest>

(Accessed: 7 April 2021).

CPP Reference (2021). *C++ Utilities Library: std::unordered\_set.* Available at:

<https://en.cppreference.com/w/cpp/container/unordered_set>

(Accessed: 7 April 2021).