MANCHESTER
1824
The University of Manchester

Automated Theorem Proving for the Two - Variable Fragment in First Order Logic

Patrick-Cătălin-Alexandru Sava

Supervisor - Dr. Ian Pratt-Hartmann

April 2021

Abstract

The two-variable fragment with no function symbols is a subset of first-order logic, known for being decidable. In the two-variable fragment of first-order logic only two variables, for instance, x and y may appear. This report presents the implementation of three theorem provers, specifically focusing on a resolution-based theorem-prover for the two-variable fragment. Currently, there are no other theorem provers precisely for this fragment. Thus, this two-variable theorem prover is the project's main contribution. The project utilises several heuristics tailored specifically for the two-variable fragment, draws on existing literature and the author's own research. The comprehensive testing benchmark employed therein serves as a solid framework for further works on the topic. Finally, the impact of varying different parameters on satisfiability is studied when using well-known techniques of generating random formulas.

Acknowledgements

The project would not have been possible without the continuous help and support of my supervisor, Dr. Ian Pratt-Hartmann. I am grateful for working with him on such a challenging and rewarding project. Thus, I would like to extend my appreciation for the excellent advice I was receiving from my supervisor throughout the entire project.

Further, I would like to thank the programming community, my teachers and friends who facilitated my self-development, knowledge and capabilities. I am especially grateful to my dear friends Cosmin Andrei Carabet, Andrei Barbuță and Mihai-Ionuț Enache for their help and encouragement.

Finally, I would like to thank my family (especially my mother, my grandmother and my partner, Yoana) for all of their support and love during all of these years.

Contents

1 Introduction

1.1 Background

First-order logic is a branch of logic characterized by the use of quantified variables over non-logical objects, allowing sentences to contain variables. It is present in many fields such as Mathematics, Philosophy or Computer Science. In Computer Science, first-order logic manifests as a part of solutions in Databases and Artificial Intelligence by representing a different way of modelling the knowledge. Furthermore, it is a solid tool for automated theorem proving. Being a sound formal system, first order logic always provides the guarantee that solely true things are inferred from ground facts.

Compared to its counterparts, such as higher-order logic, first-order logic offers a range of deductive systems which are both sound and complete. One of them is the two-variable fragment with equality and no function symbols, containing at most two variables.

1.2 Motivation

In 1975, Mortimer (1975) proved that the two-variable fragment with equality has the finite model property which implies that any satisfiable formula has a finite model. This result also suggests that the two-variable fragment is decidable. Moreover, the author demonstrated that if the finite model property holds, the model has an exponential size relative to the size of the formula.

This discovery has further led to the establishment of various two-variable fragment decision procedures. The paper of Hans de Nivelle and Ian Pratt-Hartmann published in 2001 is the first one presenting an algorithm for the entire fragment. Firstly, one of the main contributions of this project is to implement the decision procedure described in the paper of De Nivelle and Pratt-Hartmann (2001) and thus to deliver an automated theorem prover specialized in solving problems of this fragment only.

Secondly, an additional contribution of the project is a reusable, extensive benchmarking system. This system has the ability of measuring the performance of such a theorem prover, testing both its functionality and correctness and evaluating additional heuristics.

1.3 Project Aim

The main goal of the project was the implementation of the decision procedure, specifically for the two-variable fragment with equality of first order-logic. This procedure was priorly described in the paper of De Nivelle and Pratt-Hartmann (2001), published in Gore et al's "Automated Reasoning" (2001). To the best of the author's knowledge there is no evidence that the procedure has been implemented beforehand. Currently, there are theorem-provers for the general case (e.g. Vampire (2021)), yet none of them provides an implementation solely for the case of the two-variable fragment. Besides, an additional contribution of the project is a study on the probability distribution of satisfiability when the input is a randomly generated formula. Finally, a reusable testing framework which can be used to benchmark the performance of similar theorem provers is presented.

1.4 Project Roadmap

The project roadmap was as follows:

- Semester One
  - The first six weeks (including week 0) were dedicated to familiarising myself more with the topic, reading papers, books and additional materials.
  - The next three weeks (up to week 8) were 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 to 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 several 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 invested in optimizing, testing and debugging the code. Moreover, two random generators for formulas and some scripts for comparing my work against Vampire were written.

1.5 Methodology

The implementation process was robust since Week 0, being done gradually and following closely software engineering practices such as self-code review, using version control (Git) and system design. Furthermore, the principles of object-oriented programming (such as encapsulation and abstraction) were employed which facilitated a good organisation of the codebase.

The communication with my supervisor was going beyond the weekly timetabled tutorials by actively exchanging emails during the week when an aspect of the project needed either further clarifications or refinement. My supervisor suggested keeping track of everything in a log book, which proved to be very useful, not only for writing this report, but also for organizing thoughts and ideas better. Regularly updating the log book proved to be extremely useful given the size of the project.

1.6 Report Structure

First, the report provides a background on first-order logic.The following section reiterates through the basic notions of first-order logic and the formal tools which will be part of the algorithm. The section afterwards explains the design decisions, including, but not limited to, the high-level presentation of the two-variable theorem prover. Afterwards, the implementation and its low-level details are introduced. Following this, the report presents methods of evaluating the project and discusses experiments and reflections. The final section concludes the report.

1.7 Impact of COVID-19

The COVID-19 pandemic did not create major disruptions to the project. This was due to the nature of the project as it was independent of University equipment, but also due to the excellent communication with my supervisor. I am grateful that the overall quality of the project was unaffected by this matter.

2 Context

From now on, it will be assumed that the reader is familiar with first-order logic. This section will iterate through all the aspects of first-order logic which are of interest.

In this section some of the most important concepts for this dissertation are defined. The definitions employed are standard across the literature, for example found in Leitsch (1997).

2.1 First-Order Logic

First-order logic is more expressive than propositional logic due to the presence of quantifiers and more verbose predicates. In particular, those predicates have arguments, which can be constants, variables or even functions. These are defined in the next subsection.

In the sections thereafter, it will be referred only to first-order logic. Thus, any reference to logic from this point onwards refers to first-order logic.

2.1.1 Definitions

The highest level of detail that will be used throughout this report is the logic formula. It can be any string accepted by the grammar of first-order logic.

Definition 2.1 (*Predicate, Atom, Literal*)

A predicate *is a n-ary symbol containing n terms as arguments. Conversely, the* arity *of a predicate is the number of arguments. The* sign *of a predicate is positive if it is not negated, and negative otherwise. An* atomic formula *or an* atom*, is a formula containing only a positive predicate. A formula containing only a positive predicate or only a negative predicate is called* literal.

Definition 2.2 (*Term*)

A term *is either a function, a variable or a constant. A* function *is a fixed (or pre-defined) mapping from a variable or a constant to a variable or a constant. A* variable *is a mapping from a variable to another variable (including itself) or to a constant. A* constant *is a mapping exclusively to itself.*

Due to the fact that function symbols break the decidability (the reasoning is explained in the later subsections), the input will be restricted so as to not contain such terms.

2.1.2 Operators

The operators which are of interest to this project are: double implication ($\iff$), implication ($\implies$), negation ($\neg$), and ($\wedge$), or ($\vee$), equality ($=$) and inequality ($\neq$).

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

The operator <u>and</u> is referred to as conjunction and the operator <u>or</u> as disjunction.

2.1.2.1 Special Case: Equality

Until the presentation of the two-variable theorem prover algorithm, both the equality and inequality are neglected. This is mostly because dealing with equality (and inequality) in the general case (outside of the two-variable fragment) is out of the scope of the project as it involves different techniques such as paramodulation.

2.1.3 Quantifiers

The quantifiers are the following:

- for all ($\forall$)
- there exists ($\exists$)
- there exists an unique ($\exists!$)

The uniqueness quantifier will not be of interest until the last subsection of this section. Moreover, the grammar does not contain this symbol and it is encountered only at an abstract level when the algorithm tackles equality.

2.1.4 Precedence

In terms of operator precedence, the conventions of Vampire (2021) are adopted. This allows for random generation of formulas compatible with both Vampire and the theorem prover. The conventions are as follows:

- the round brackets enforce precedence in the same way they do in the majority of mathematical systems
- the operator <u>not</u> 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 <u>and</u> has a strictly higher precedence than the operators corresponding to *or*, *implication* and *double implication*
  - analogously, the operator <u>or</u> has a higher precedence than the (double) implications and the implication precedes the double implication

It is important to note that the implication is right associative.

2.1.5 Validity and Satisfiability

Definition 2.3 (*Domain of discourse*)

The domain of discourse *is the set of entities over which the variables from a first-order logic formula range.*

Definition 2.4 (*Model*)

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.*

Definition 2.5 (*Validity and Invalidity*)

A formula is valid *iff it is true in all of the possible models. Conversely, a formula is* invalid *if and only if there exists at least one model in which it is false.*

Definition 2.6 (*Satisfiability and Unsatisfiability*)

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.*

As a consequence, an unsatisfiable formula is invalid. Moreover a formula is satisfiable iff its negation is invalid. This interconnection between satisfiability and validity resonates well with the goal of the theorem proving, to determine whether a given formula is unsatisfiable, satisfiable or valid (theorem).

2.1.6 Decidability

It is reasonable to produce a theorem prover only for sound proof systems. Soundness guarantees that only true things are inferred from ground facts. However, this is necessary but not sufficient in order to guarantee that the theorem prover terminates, or that it is able to infer all possible true things from the initial set of ground facts. For that to be the case, the proof system needs to be complete as well (all of the possible true things can be inferred from the initial set of axioms).

The satisfiability problem for first-order logic is known to be undecidable. However, this is not the case for some fragments which are part of first-order logic, for instance, the monadic one or the two-variable fragment (Gradel et al, 1997).

The two-variable fragment is sound and complete and hence decidable. Mortimer (1975) presented this result, showing that the two-variable fragment has the finite model property, further implying that it is decidable.

2.2 Clausal Normal Form (CNF)

In order to approach the automated theorem proving, a normalized form to deal with the formulas is needed. Firstly, the input format has to be emphasised. It is assumed that the input contains N formulas, the first $N-1$ of them forming the knowledge base and the N-th one representing the query (the formula for which satisfiability has to be verified assuming that all of the formulas from the knowledge base are axioms).

Second, reducing everything to a single formula is desirable. This is 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. The resulting formula will be acknowledged thereafter as ϕ.

As described in Leitsch (1997, p.12-18), ϕ can be transformed into a sub-language of logic called conjunctive normal form (also referred to as clausal normal form). The final form will be a conjunctions of disjunctions, in which no quantifiers are present and will be referred to as θ . Even though ϕ may not have contained function symbols, the resulting θ may contain such and this is explained in the next subsections.

2.2.1 Basic Reduction

The following seven steps are repeated until the formula does not change anymore:

- break double implications in implications
- break implications in disjunctions
- push the operator <u>not</u> further on conjunctions
- push the operator <u>not</u> further on disjunctions
- simplify two consecutive <u>not</u> operators to nothing
- push the operator <u>not</u> past universal quantifiers
- push the operator <u>not</u> past existential quantifiers

Let ϕ' be the resulting formula after all steps above have been exhausted. It is clear that ϕ' contains only quantifiers, disjunctions, conjunctions, brackets and literals.

2.2.2 Skolemization

The name of this procedure comes after the Norwegian mathematician Thoralf Skolem (Leitsch, 1997). The intuition behind this procedure is that the existentially quantified variables can be reduced to a brand-new function. This function's arguments are all of the universally quantified variables which have in their scope the respective existentially quantified variable. The definition of 'scope' is the same as in programming (an analogy can be drawn with the scope of local variables in C++). Obviously, if there are no such universally quantified variables, the function becomes a constant, precisely a Skolem constant. Otherwise, it remains a function, specifically a Skolem function. Hence, all occurrences of existentially quantified variables are replaced with Skolem functions or Skolem constants. Afterwards, all quantifiers (both existential and universal) can be disposed of, since from now on, the terms on sub-types (functions, variables, constants) will be differentiated. It is thus observed that the quantifiers are redundant now.

For simplicity it is assumed for any variable x in ϕ' that all of the variables which are in the scope of x have unique names. For instance, a substring like ∀x∀y∀x or one like ∀x∃y∃y would never be present in ϕ'. The approach for resolving this issue will be further described in the implementation details.

Let ϕ'' be the resulting formula. Lastly, in order to transform ϕ'' in θ, the distributivity laws over conjunctions and disjunctions are performed. Considering the preparation for the next step, θ is transformed into a set of disjunctions (or clauses) since it is uniquely determined which is the operator between its clauses (the operator <u>and</u>).

Even though the distributivity laws are explicitly used, the way they will be implemented differs from the traditional way. This is due to the fact that implementing the distributivity laws naively may lead to an exponential growth of the size of the formula.

2.3 Automated Theorem Proving

In this subsection the report will present the core ideas and tools behind the automated theorem proving. All of these are part of the project's codebase.

2.3.1 Unification

The key of the automated theorem proving is in the ability of the software to detect whether two literals are the same if the sign is ignored. In other words, they are similar predicate-wise, term-wise and order-wise. (i.e. $P(f(x))$ and $\neg P(f(x))$ respect that while $P(f(x))$ and $P(f(y))$ do not).

Definition 2.7 (*Substitution*)

A substitution *is a mapping from the set of variables to the set of terms. Moreover, it is possible to compose two substitutions in the same way two functions are composed.*

An invalid substitution *is one in which there exists a mapping from a variable to a term containing the same variable (it can be either itself or a function which has the variable nested in one of its arguments). A* valid substitution *is a substitution which is not invalid.*

Definition 2.8 (*Unifiable terms, atoms, literals and the most general unifier*)

Two terms are unifiable *if there exists a valid substitution such that after applying substitution the terms become identical.*

Two n-ary predicates $P(t_1, t_2, \ldots, t_n)$ and $Q(r_1, r_2, \ldots, r_n)$ are unifiable *if P and Q are identical and t_i and r_i are unifiable for $1 \leq i \leq n$.*

Two literals are unifiable *if their corresponding predicates (ignoring the sign) are unifiable.*

Let $P(t_1, t_2, \ldots, t_n)$ and $Q(r_1, r_2, \ldots, r_n)$ be two n-ary unifiable predicates. Let $S_1, S_2, \ldots, S_n$ be the valid substitutions for the terms t_i and r_i with $1 \leq i \leq n$. Let S be the result of the composition of $S_1, S_2, \ldots, S_n$. If the application of S unifies P and Q (makes P and Q identical), then S is the most general unifier *of P and Q. Note that most general unifiers for literals are referred to in the same way as for predicates since the signs do not matter.*

Having explained unification, the two core rules for theorem proving: resolution and factoring will be presented thereafter.

2.3.2 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 literals in C. Similarly, the same can be defined on a literal L or on an atom A, resulting in $\{L\}S$ and $\{A\}S$, respectively.

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

Definition 2.9 (*Resolution rule, resolvent, resolved atom*)

Let C_1 and C_2 be two clauses that have in common two unifiable literals L_1 (part of C_1) and L_2 (part *of C_2) of opposite signs. Let S be the most general unifier of L_1 and L_2. Thus, $C_1 \wedge C_2 \implies \{C_1 \backslash L_1 \wedge C_2 \backslash L_2\}S$. This transformation is referred to as the* resolution rule.

$\{C_1 \backslash L_1 \wedge C_2 \backslash L_2\}S$ is referred to as the resolvent *of C_1 and C_2, while the atom A which is common to L_1 and L_2 is referred to as the* resolved atom.

2.3.3 Factoring rule

Definition 2.10 (*Factoring rule*)

Let C be a clause and L be a literal occuring more than once in C. If L occurs k times in C identically, then $k-1$ occurrences of L can be removed from C. This transformation is referred to as the factoring rule.

Once the resolution and factoring rules have been implemented, a basic theorem prover is obtained. Since the satisfiability problem for first-order logic is undecidable, the program will never terminate. If it terminates, this means that it either reaches saturation - the case when there is no new unique clause inferred by the current set of clauses, or it derives the empty clause. The former case implies that the clause C is counter satisfiable while the latter implies satisfiability.

From now on the report will present a few refinements which enhance the speed of the basic theorem prover and ultimately facilitate the design of the theorem prover for the two-variable fragment.

2.3.4 Refinements

2.3.4.1 Tautology Removal

Definition 2.11 (*Tautology Removal*)

Let C be a clause containing two literals L_1 and L_2 which are identical except for their signs (which are opposite). This clause is obviously true in any model, so it can be removed from the current set of clauses. This procedure is referred to as tautology removal.

2.3.4.2 Subsumption

Definition 2.12 (*Subsumpion*)

Let C_1 and C_2 be two clauses such that the set of literals of C_1 is included in the set of literals of C_2. Then, C_1 is regarded as a "more general" clause than C_2. Therefore C_2 can be dropped since it is subsumed *by C_1.*

This strategy is sensible because at any point it is assumed that all clauses from the current set of clauses are axioms. Hence, the intuition behind subsumption is that C_2 is no longer needed in the set as C_1 was implying it regardless and since C_1 has to be true, then C_2 certainly is true.

2.3.4.3 Unification within the same clause on literals with the same sign

This is a heuristic which was created during the implementation process. The results were promising as it enhanced the speed of a few tests used against the implementation. Intuitively, a clause C and two unifiable literals of the same sign L_1 and L_2 which are part of the clause, yield the option of unifying the literals L_1 and L_2 and applying the factoring rule further. Thus, a less general clause is obtained, which is of shorter size than the former.

2.3.4.4 Depth-ordered resolution

Definition 2.13 (*Ground Term*)

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 with nested only function symbols or/and constants.*

Definition 2.14 (*Term depth, atom depth, literal depth*)

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

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).*

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 the* depth of a literal *is defined as the depth of its corresponding atom, since the sign does not matter for the calculation.*

As described, the satisfiability problem for first-order logic is undecidable. Therefore, it follows that the basic theorem prover may not halt in some cases. One of the reasons why this may happen is the unbounded nesting of function symbols (currently, there is no mechanism that prevents expansions like $P(f(f(f(f(f(\dots)))))))$.

As described in Leitsch (1997, p. 99), a binary relation $<_A$ is defined on the set of all atoms which has the following properties:

- is reflexive
- is transitive
- for any two atoms A and B and a substitution S, $A <_A B$ implies $\{A\}S <_A \{B\}S$

The depth-ordered theorem prover will work in the same way as the basic theorem prover, with the restriction that the resolution rule is applied on two clauses C_1 and C_2 only when there is no literal in L in the resolvent C (of C_1 and C_2) such that $B <_A L$, where B is the resolved atom of the resolution (Leitsch, 1997, p. 99).

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

2.4 Two-Variable fragment

This section discusses the two-variable fragment theorem prover. Since the algorithm is described at a theoretical level, it is assumed that where references are not in place and the difficulty of (sub)topics presented exceed the basics of logic, the credits for those contributions are of my supervisor.

For the next two subsections it is assumed that a logic formula is given in two variables. The formula is passed further as input to the two-variable theorem prover. Moreover, this formula is in the CNF format.

For simplicity it is assumed that the set of variables occurring in the formula is $\{x, y\}$.

2.4.1 Without equality

For the rest of this subsection, it is assumed that the logic formula does not contain equality or inequality. Following that, the algorithm will be revisited to account for equality in the next subsection. The steps of the algorithm are as follows:

1. Depth-ordered resolution on a formula is executed, only on the literals involving exactly two variables. Consequently, the resolved atom has exactly two different variables.

- If the empty clause is derived, the formula is satisfiable.
- Otherwise, the algorithm should halt by saturation since the ordered resolution is used.

2. If the algorithm did saturate, then a set of clauses remain. From those, all clauses containing literals in exactly two variables can be safely disposed of.

3. Following this, a set of clauses with the following property remains: if each clause is grouped on its set of distinct variables, the maximum number of groups per clause will be three. This is because in the worst case a group of the following will exist:

    - empty set (all literals in this group do not hold variables; they are ground instances as they are called)
    - the set $\{x\}$
    - the set $\{y\}$

    This whole procedure is referred to in De Nivelle and Pratt-Hartmann (2001) as the splitting rule and it is possible to be applied to a clause without overlapping variables, which is the case here.

4. Let C be the set of clauses and $|C|$ be its size. Further, $|C|$ group choices are made, one for each clause. In other words, at most $3^{|C|}$ auxiliary sets of clauses are built containing exactly one group from each clause. Further, the depth-ordered theorem prover is run on each of these auxiliary sets:

    - If all of these runs derive the empty clause, the input formula is unsatisfiable.
    - Otherwise, the depth-ordered theorem prover halts by saturation on at least one of these auxiliary sets. Hence, the input formula is satisfiable.

2.4.2 With equality

Before revisiting the algorithm discussed above, the report will present how to deal with:

- the equality
    - tackling the equality operator is trivial in this case, since the equality can be simply replaced with a special predicate of arity two called "Equality"
    - in case the equality operator occurs multiple times, all of these occurrences are replaced by the *same* predicate called "Equality"
- the inequality
    - the inequality is handled similarly to the equality with the consideration that the inequality operator will be replaced with a brand-new predicate of arity two called "Inequality"
        - as in the case of equality, different occurrences of the inequality are mapped to the same predicate called "Inequality"
    - in order to preserve the logical equivalence, a new clause in one variable has to be added to the CNF, containing the negated "Inequality" predicate (i.e.$\neg Inequality(x, x)$ has to be true)

The "Inequality" will be treated exactly as any other predicate in the set of clauses from now on. The "Equality" will instead be transformed during the execution of the algorithm.

The algorithm is revisited by mentioning the necessary changes:

- Step 1 performs resolution on literals involving exactly two variables, which are not the "Equality".
- Step 2 disposes of all clauses containing literals different from the "Equality" which are in exactly two variables.

- Before proceeding to step 3, the equality has to be expanded in the same way as described in Lemma 5 and Lemma 6 in De Nivelle and Pratt-Hartmann (2001). The report will present the high-level idea of this expansion, letting the group of the ground instances aside:
  - Let R be an arbitrary clause in two variables from the set of clauses, containing equality. Let T(x) represent all of the subformulas in variable x from R and U(y) - all of the subformulas in variable y from R. Hence, R is written in the following way:

    $$\forall x \forall y (T(x) \lor U(y) \lor x = y)$$

    The aforementioned formula is logically equivalent with the following:

    $$\forall x T(x) \lor \forall y U(y) \lor (\exists! x \neg T(x) \land \forall x (T(x) \iff U(y)))$$

    For the following formula:

    $$\exists! x T(x)$$

    there exists the following logical equivalence:

    $$\exists x (T(x) \land \forall y (T(x) \implies x = y))$$

    The problem which arises with the logical equivalence from above is that the algorithm enters in an infinite loop, since each attempt of replacing the equality results in a new formula containing equality again.

    The solution for this case is explained as part of Lemma 6 in De Nivelle and Pratt-Hartmann (2001) and 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, the binary predicates can be safely omitted if the expansion of the uniqueness quantifier is performed at this step (after the disposal from Step 2 took place). After Step 2, the formula will have only literals in one variable, which can be regarded as predicates of arity one. However, this has to be done with caution, since, for example, $P(constant, x)$ and $P(x, constant)$ represent different predicates of arity one, even though they share the same predicate name.

    After combining the observation above with Lemma 6 from De Nivelle and Pratt-Hartmann (2001), the following is attained:

    $$\exists! x \neg T(x)$$

    and

    $$T(e) \land \bigwedge_{P} (\forall x (T(x) \implies (P(x) \iff P(e))))$$

    are equisatisfiable, where $\underline{e}$ is a brand-new constant symbol.

- The only consideration before proceeding to Step 3 is to execute once more the clausal normal form algorithm on the resulting equisatisfiable formula, since this formula is not anymore a valid CNF.

This concludes the theoretical part of the project. The next section will present the big picture of the algorithms and how they fit together.

3 Design

The codebase has three main components: the front-end, the intermediate representation and the back-end. This differentiation was inspired by the Compilers literature, striving for consistency where possible.

The front-end is responsible for parsing the input and reporting any syntactic error. Furthermore, it also builds the parse tree of the input formula. It is important to note that the report will not be following precisely the definitions from the compiler terminologies as the parse tree is a hybrid between a parse tree and an abstract syntax tree. The reasoning behind this is because all operations done before the formula is in CNF are performed on this tree, which is changed dynamically. In terms of algorithms, this component uses mostly ad hoc algorithms (which will be further described) or depth-first search (Cormen et al, p. 603). This component is executing all operations in polynomial time.

The intermediate representation is mostly dealing with the parse tree, implementing all operations necessary for building the clausal normal form efficiently. In terms of the implementation, many of the methods implemented for reducing the formula make assumptions on the structure of the input, instead of enforcing that structure (enforcing the structure would imply more effort 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 operations are executed in polynomial time.

The back-end is the largest part of the codebase, containing the implementation for the unification, basic theorem prover, depth-ordered theorem prover and the two-variable theorem prover. Moreover, the majority of the work for the theorem prover is parallelized using multithreading. Implementation-wise, this component makes the most use of both ad hoc algorithms and variations of classical algorithms: breadth-first search (Cormen et al, 2009, p. 594), depth-first search, backtracking (Matuszek, 2002), 2-colorability (Cormen et al, 2009, p. 1103). Moreover, sophisticated data structures (such as persistent data structures) are involved. In terms of time complexity, the majority of the operations happen in polynomial time. An exception is the part involving the two-variable fragment which operates in exponential time due to the nature of the Cartesian product, implemented as a backtracking algorithm.

In terms of the language, a few changes have been made since the logic symbols are not present on (probably) none of the existing keyboard layouts. Hence, the following mapping is adopted: double implication (`"<->"`), implication (`"->"`), negation ($\sim$), and (`"^"`), or (`"|"`), equality (`"="`), inequality (`"!="`), universal quantifier (`"@"`) and finally the existential quantifier (`"?"`). Further, the following additional conventions are implemented:

- the terms and predicates always contain letters of the Latin alphabet only
- the predicates always start with uppercase letter, containing further only lower case letters
- all terms always contain lowercase letters only

As mentioned, it is assumed that the input formula does not contain function symbols, even though it is possible for its CNF to have such symbols in the end (due to the Skolemization)

Figure 1 below provides a representation on how the three aforementioned components interact with each other. Moreover, the names of some subcomponents of interest are shown.

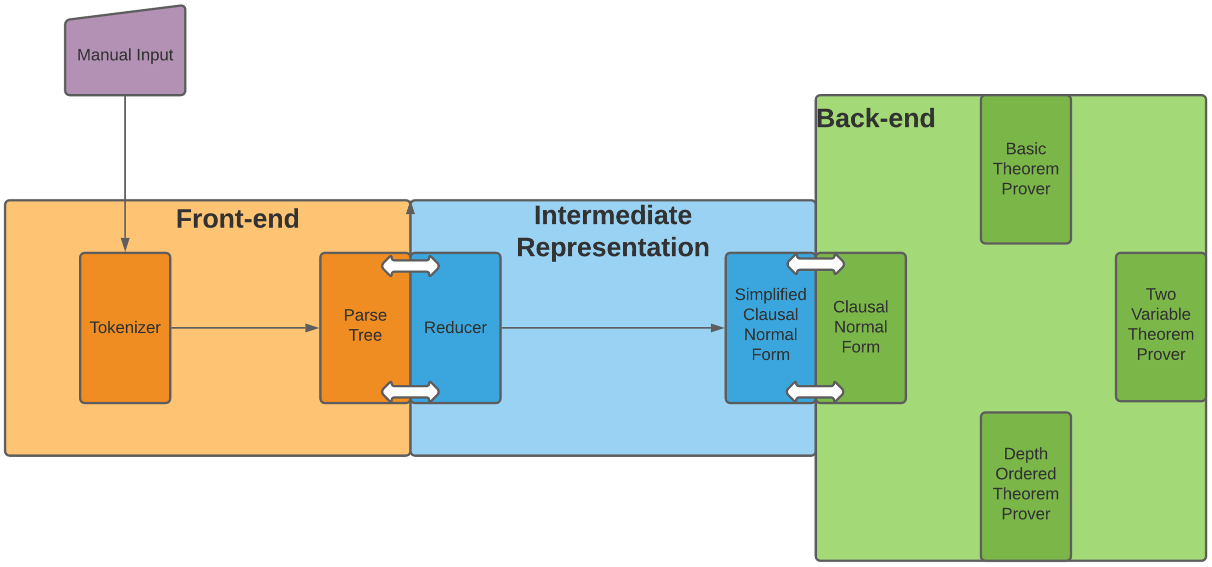
Manual Input
Front-end
Tokenizer
Parse Tree
Intermediate Representation
Reducer
Simplified Clausal Normal Form
Back-end
Clausal Normal Form
Basic Theorem Prover
Two Variable Theorem Prover
Depth Ordered Theorem Prover

Figure 1: Design Model

4 Implementation

This section explains fully the implementation details of the algorithms.

4.1 Environment and technology stack

Both Windows 10 Pro and Ubuntu 20.04 were used. Ubuntu was run as a Windows subsystem, enabling the usage of both the Linux infrastructure and the Unix-like commands.

In terms of hardware, a computer with a 7^{th} generation Intel i7 processor and 32GB of RAM memory was used. During the implementation process, the significant memory resources were fairly useful since an unoptimized version of the theorem prover was using more than 18GB of RAM memory on some input formulas which were saturating. Following this, the code was optimised in a way that the amount of memory used by the theorem prover was drastically reduced (below 1GB of RAM memory on similar or on the same tests).

In terms of programming languages, the choice was C++ Standard 17 for the implementation (due to the author's experience with this language and C++ runtime speed). Standard 17 was a must since the implementation makes extensive use of std::variant, a type safe union (CPP Reference, 2021) and of std::execution (CPP Reference, 2021). The latter provides the execution policies which can be used when parallelizing code. However, the latter is not fully implemented by some C++ 17 compilers , so it was manually imported by linking the code with the libtbb library (Debian, 2021) that provides the implementation. In order to unit-test some parts of the code, Google's C++ testing framework (Googletest, 2021) was used. Other programming languages which were beneficial during the implementation of this project were Python3.8 and Unix Bash. The former was used mostly for writing input generators, while the latter for End-to-End testing or for making comparisons between the theorem prover and Vampire.

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

4.2 Front-end

This section presents the algorithms behind the front-end. The front-end uses the following mechanism: the tokenizer receives the raw input and produces an ordered list of tokens; further, this list of tokens is passed to the parse tree, thus producing the initial parse tree.

4.2.1 Tokenizer

The purpose of the tokenizer is to take the raw input and parse it into tokens. A token can be:

- an atom
  - any string matching the following: it starts with an uppercase letter and it has an arbitrary number of lowercase characters (possibly zero). Further, it has a pair of round brackets which contain a comma-separated set of variables (those variables are lowercase strings with a positive length)
  - since the convention adopted is that predicates always start with capital letter and there are no other entities which start in the same way, the detection of atoms is trivial: each time when a capital letter is detected, scanning up to the first closed bracket is conducted
- an operator
  - this is trivial since each operator is detected with one symbol lookahead
    * the only two particular cases occur for equality and inequality: they have to be replaced with their corresponding predicates (called "Equality" or "Inequality")
- a quantifier
  - as for operators, this is easily detectable: the current pointer in the raw input has to point to either "@" or "?" and be followed by a positive number of lowercase letters (forming the variable)

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

4.2.2 Parse Tree

The parse tree appears like a normal tree, with the consideration that some information may be stored in each node. In the code this information is referred to as an entity. A node may not store any entity. These nodes will be initially used solely for simulating the precedence, as it will be described later. For the nodes storing entities, these can encapsulate 4 types of entities:

- quantified variable (i.e. "@x" or "?y")
- literal (i.e. $P(x)$ or $\sim R(f(x)))$
- 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 (is ignored for the time being because it will be analyzed it in the section describing the intermediate representation implementation)

The reasons why the formula is stored this way are as follows:

- for some nodes, the following property holds: the subtree rooted in each of them represents a subformula of the initial formula
- the entity which encapsulates an operator does not include the brackets; this is because the pre-order traversal on the tree is used for representing the precedence enforced by the brackets

- hence, the design choice already reduced drastically the initial language of the formula: equality was removed, alongside with inequality and brackets

Next, the algorithm which makes the transition from the ordered list of tokens to the parse tree is presented with its high-level idea. Some details are intentionally omitted for the simplicity:

Algorithm 1 Transition Algorithm

```
    Input : $Tokens$, the ordered list of tokens
    Output: None (void function)
 1: function buildParseTree($Tokens$)
 2:     $fatherChain \leftarrow$ empty stack
 3:     the first node is added to the parse tree          ▷ the first node is the root
 4:     push the root in fatherChain
 5:     for $token$ in $Tokens$ do
 6:         if if $token$ is an open bracket, an operator, a quantifier or a literal then
 7:             $top \leftarrow$ the top of fatherChain        ▷ but it is not popped off the stack
 8:             $newNode \leftarrow$ a new node added to the parseTree
 9:             add $newNode$ in the adjacency list of $top$
10:             push $newNode$ in the fatherChain
11:             if $token$ is an operator, a quantifier or a literal then
12:                 assign a new entity
13:         else if $token$ is closed bracket then
14:             pop the top from fatherChain
```

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

4.3 Intermediate Representation

Following this, the functionality of the intermediate representation is shown. This component is represented in the code under the name "Reducer" and acts as a friend class (CPP Reference, 2021) for the class Parse Tree (previously presented). The name "Reducer" is consistent with its functionality: keep reducing the parse tree until the root holds the CNF.

4.3.1 The approach for disambiguating the given formulas

As in many programming languages, it is perfectly legal to have two variables with the same name, one nested in the scope of another, both being bound by different quantifiers. This may be problematic later on unless it is addressed now. Hence, a depth-first search maintaining an accumulator with all of the variables which have been encountered so far is run. Note that the accumulator never evicts elements (only its size increases). If the algorithm is currently at a node holding a quantified variable, the respective variable should be added in the accumulator. In case the variable already exists in the accumulator, it has to be mapped to a brand-new variable which does not exist in the accumulator. The mapping will persist while traversing the entire subtree rooted in this node. Moreover, while going down the traversal, any occurrence of the variable which has been mapped at this step is substituted with its corresponding mapping (otherwise, it is either a constant or a free variable and remains unchanged).

In order to implement this part efficiently, the accumulator was modelled as an std::unordered_set (CPP Reference, 2021), in order to allow for inserting and searching in constant time.
The time complexity of this part is $\mathcal{O}(N)$, where N is the number of nodes the parse tree has by the time of traversal.

4.3.2 Basic reduction

This section presents all tools needed, in order to achieve the "Basic reduction", discussed in the theoretical part of the report.

4.3.2.1 The approach for resolving the precedence

Another challenging aspect of the parse tree is dealing with the precedence of the remaining operators. As mentioned earlier, the pre-order traversal takes care of the precedence enforced by the brackets. This is the main tool for dealing with the precedence for the other operators as well.

The first case, the one of operator and is discussed therein. The remaining cases are treated analogously as the main idea holds. It will now be assumed that the algorithm is at an arbitrary node and its children have various types of entities (literals, operators, quantifiers, including none).

Before proceeding, there are few important observations about the parse tree, assuming that it represents a syntactically valid logic formula:

- the subtree rooted in a child with no entity associated represents a syntactically valid logic formula
- the order in which the children occur is very important since it uniquely determines the formula
- no two consecutive children have operators as entities, because otherwise the formula would be invalid syntactically

Considering the above points, the precedence is handled by adding new nodes in the parse tree which is equivalent to adding parentheses. Precisely, some of the current children are replaced by brand-new nodes, having into their adjacency list only the node they are replacing.

In order to resolve the case for the operator and, maximal groups of consecutive children with the property that only the operator and occurs in the multiset of their entities need to be obtained. For each group, the algorithm will add all nodes belonging to it in the adjacency list of a brand-new node, by preserving the relative order between them. Finally, each group of nodes in the original adjacency list is replaced by the brand-new node, corresponding to the respective group. This is illustrated in Figure 2 and Figure 3 below:

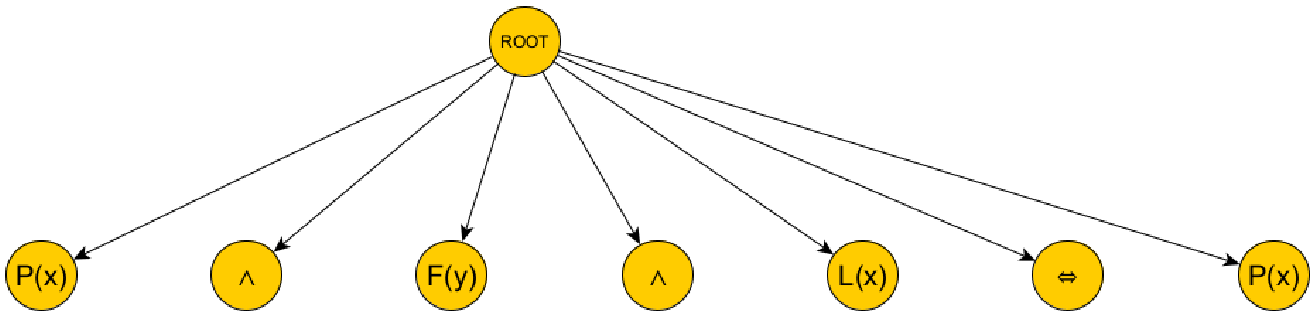

Figure 2: Before Resolving Precedence for Operator and

Following this, the precedence for the operator and is resolved in the parse tree. Then, the algorithm proceeds with the operator or. The procedure is similar, since the operator and is not handled at all. The cases for implication and double implication are resolved analogously.

We implement precedence semantics by doing four calls to a depth-first search function, one for each operator.

The time complexity of this part is $\mathcal{O}(N)$, where N is the number of nodes the parse tree contains by

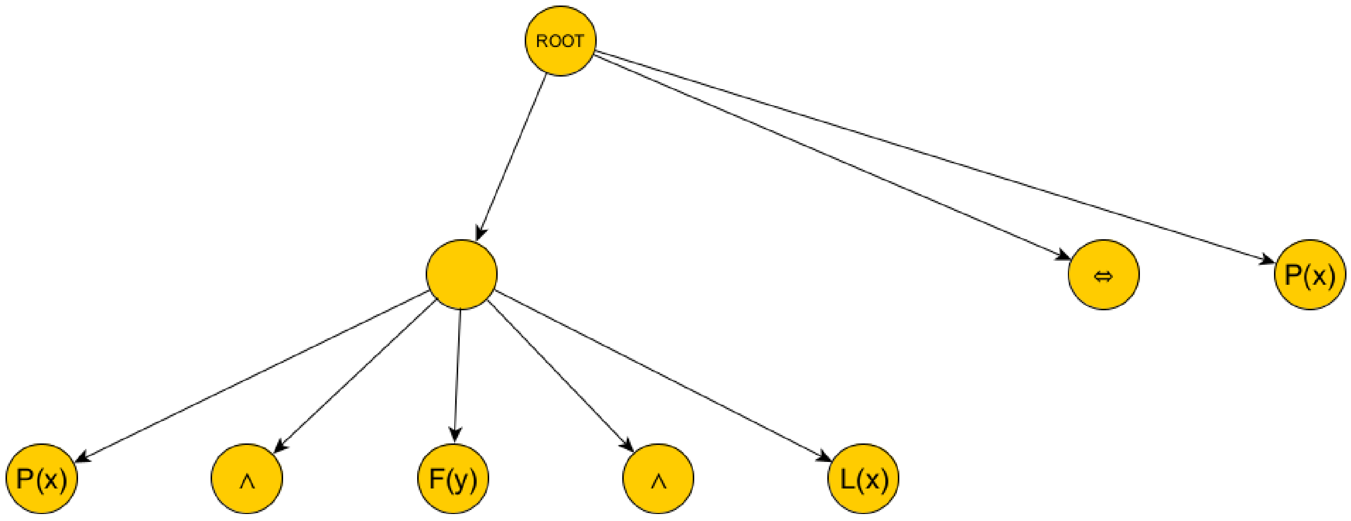

Figure 3: After Resolving Precedence for Operator and

the time of traversal.

4.3.2.2 The approach for eliminating double implications

After an unambiguous formula in the parse tree is reached, the next step is to reduce it to a simpler sub-language: one which does not contain double-implications. The double implication can be split into a conjunction of two implications, yet the problem which arises if this approach is taken, is that the size of the formula may grow exponentially (ie $P(x) \iff (R(x) \iff Q(x)$ will further transform in $P(x) \iff ((R(x) \implies Q(x)) \land (Q(x) \implies R(x)))$, finally becoming $(P(x) \implies ((R(x) \implies Q(x)) \land (Q(x) \implies R(x)))) \land (((R(x) \implies Q(x)) \land (Q(x) \implies R(x))) \implies P(x))$; notably, the initial formula has one occurrence of Q, the next one has two, while the final one has four)

In order to prevent this, the deepest occurrence of a double implication may be replaced with a brand-new predicate (for the purpose of this section it is assumed that the replacement of a double implication consists of replacing the operator, the left-hand side and the right-hand side, respectively). To preserve the logical equivalence while performing the above, a new conjunction has to be added to the formula, containing a double implication between the brand-new predicate introduced and the subformula it is replacing. This procedure is repeated until the amount of double implications which are in the scope of one another is at most two. It is safe to apply the naive expansion described in the first paragraph on the resulting formula without facing the risk of an exponential growth (see complexity analysis)

Implementing the replacement of the double implications with predicates is easily performed on the parse tree:

Algorithm 2 Replacement of the double implications with predicates

```
1: all nodes are visited using a depth-first search
2: for every detected occurrence of the double implication do
3:     if is the deepest in the current subtree then
4:         • the three nodes forming it are taken (left-hand side, double implication, right-hand side) and further
           appended 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
5:         • two brand-new nodes will be created in the parse tree, both holding the same brand-new predicate
           as an entity; these two nodes will be referred to as $T$ and $P$
6:         • the three nodes forming the initial double implication will be now replaced with $T$
7:         • the nodes $C$, $P$, $D$, $R$ (in this order) will be appended at the end of the adjacency list of the root; $C$
           and $D$ are brand-new nodes holding the operator and and the double implication, respectively; these
           four nodes simulate the appending of the conjunction needed for preserving the logical equivalence
```

The naive approach of eliminating the double implications is detailed further. This approach does not lead to an exponential growth of the number of literals since the maximum amount of double implications nested in the scope of one another is at most two, after having performed the algorithm from above.

Algorithm 3 The expansion of the double implications with predicates - Naive Approach

```
1: all nodes are visited using a depth-first search
2: for every detected occurrence of the double implication, containing the nodes $L$, $D$, $R$ do
3:     • create a deep copy of node $L$ in node $L'$
4:     • create a deep copy of node $R$ in node $R'$
5:     • create a brand-new node $I$, holding an implication as an entity
6:     • change the entity held on $D$ to an implication
7:     • create a brand-new node $C$, holding the operator and replace the nodes $L$, $D$, $R$ with the nodes $L$,
       $D$, $R$, $C$, $R'$, $I$, $L'$ in this order
```

As there is a polynomial number (with respect to the size of the formula) of double implications in the parse tree, the complexity of performing this transformation is polynomially bounded. The optimization for replacing the double implications with brand-new predicates will require a quadratic amount of nodes to be added in the worst case. Conversely, performing the naive expansion will only double the size of the entire tree in the worst case.

Hence, the time complexity of this reduction is $\mathcal{O}(N^2)$, where N is the number of nodes in the parse tree.

4.3.2.3 The approach for eliminating implications

In the previous section, the approach of eliminating double implications was presented. Hence, the only operator left to be eliminated is the implication which is easier to perform. In fact, the implication from the left-hand side to the right-hand side is equivalent to the disjunction between the left-hand side negated and the right-hand side, which does not double the number of occurrences of any predicate that is part of any of the sides.

The algorithm is therefore:

Algorithm 4 Eliminating implications

```
1: all nodes are visited using a depth-first search
2: for for every node $Q$ with three consecutive children matching the pattern left-hand side (referred as $L$),
   implication (referred to as $I$), right-hand side (referred to as $R$) do
3:     create a brand-new node $K$, holding the operator not as an entity
4:     add $L$ in the adjacency list of $K$ (as being the only direct son)
5:     replace the occurrence of $L$ in the adjacency list of $Q$ with $K$
6:     change the operator node $I$ is holding to disjunction
7:     leave $R$ unchanged
```

Since all operations, which are executed during the search of an implication, are performed efficiently in $\mathcal{O}(1)$, the time complexity for this part is $\mathcal{O}(N)$, where N is the number of nodes the parse tree contains by the time of traversal.

4.3.2.4 The approach for pushing the operator not

The only task remaining is to push the operator not as far as possible. It has to be pushed past quantifiers, disjunctions of conjunctions. Moreover, simplifying consecutive occurrences has to be implemented.

Due to the convenient resulting simplified language of the parse tree, the algorithm proceeds as follows:

Algorithm 5 Pushing the operator not Algorithm

```
1: all nodes are visited using a depth-first search
2: for for every node Q the assigned entity is the operator not do
3:     for every child of Q, depending on the entity held by the child do
4:         if no entity then
5:             transforms in the operator not
6:         else
7:             if holds operator then
8:                 switch operator do
9:                     case not
10:                        transforms in no entity
11:                    case and
12:                        transforms in or
13:                    case or
14:                        transforms in and
15:            else if holds quantifier then
16:                switch quantifier do
17:                    case existential
18:                        transforms into universal and moreover, an intermediary brand-new node between the
                           child and its subtree is created, holding the operator not
19:                    case universal
20:                        transforms into existential and moreover, an intermediary brand-new node between
                           the child and its subtree is created, holding the operator not
21:            else if holds literal then
22:                the sign is flipped and the operator not will not go past it
23:            else                                                          ▷ holds normal form
24:                do nothing, because by this point the parse tree does not have any node holding this
                   type of entity
```

One aspect which has to be highlighted is that all of the changes described above have to happen before going down the depth-first search traversal (that is, before the recursive call). Hence, implementing the same changes, but using a breadth-first search, would have been more natural.

Since each node is visited 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 $\mathcal{O}(N)$, where N is the current number of nodes from the parse tree.

4.3.3 Skolemization

Once only conjunctions are present and disjunctions and the negations are attached to the predicates, Skolemization is implemented:

Algorithm 6 Skolemization

```
1: all nodes are visited 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
2: a std::map (CPP Reference, 2021) is maintained (called $M$)    ▷ containing the mapping for the
                                                                    existentially quantified variables to their
                                                                    corresponding functions (and their respective
                                                                    arguments)
3: if the current node $Q$ holds an universally quantified variable as an entity then
4:     $V \leftarrow$ the variable
5:     $V$ will be pushed to the stack
6:     $V$ will be further popped off the stack after the traversal of all subtrees rooted in the children of Q
       has been fully performed
7: else if the current node holds an existentially quantified as entity then
8:     $K \leftarrow$ the variable
9:     a brand-new function symbol $F$ is created
10:    an entry in the map $M$ is added 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)
11: else if the current node holds a literal as an entity then
12:    iterate through its terms and for each variable which has entry in $M$, its occurrence will be replaced
       with the corresponding value from the map (that is, the Skolem function or constant)
```

After the algorithm has been performed, all universal quantifiers can be disposed of. This will not affect the correctness since during the execution of the Skolemization, the algorithm keeps track of the types of terms (a detail which was omitted in the presentation of the algorithm for simplicity). Having designed a mechanism to uniquely determine the type of the terms, enables the removal of the universal quantifiers without losing any information.

The report will now present the time complexity of the implementation for Skolemization. The operations performed on the map M are done efficiently, in a logarithmic complexity. However, the amount of variables stored in M can 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 (N^2) (where N is the number of nodes from the parse tree). Hence, the time complexity of the algorithm is $\mathcal{O}(N^2 log_2 N)$.

4.3.4 Simplified Clausal Normal Form

After having performed all of the algorithms described in this section, the construction of the clausal normal form is finalized.

First, the CNF is retrieved in a temporary format - the simplified clausal normal form. This data structure model does not support a term nest higher than two, but this is enough for now since the initial formula cannot contain function symbols. The data structure model will acquire a different format after its retrieval (this will be detailed in the section dedicated to the back-end).

The algorithm which is presented as part of this section may appear complex. However, the reason behind the design decision is to avoid exponential growth while merging the CNFs. If this is done naively, by computing the Cartesian product of two CNFs, the size of the resulting CNF evidently faces exponential growth.
The report will now present the algorithm which merges two arbitrary CNF (called "merge(A, B)") which contain in between an operator (either conjunction or disjunction; both of the cases will be discussed):

Algorithm 7 CNF Distributive Law Optimized Algorithm

```
    Input: A and B, two CNFs, each stored as an array of arrays containing literals
    Output: C, the resulting CNF

    Assumptions: let toBeAdded be a global set of disjunctions, which need to be
    appended to the formula as conjunctions
 1: function MERGE($A$, $B$)
 2:     if the operator between $A$ and $B$ is a conjunction then
 3:         create $C$ as a concatenation of $A$ and $B$
 4:         return $C$
 5:     else
 6:         if either of $A$ or $B$ are empty then
 7:             return the other one
 8:         else
 9:             if both the sizes of $A$ and $B$ are 1 then
10:                 $C \leftarrow$ [[A[0][0], B[0][0]]]     ▷ containing a single element, the concatenation of the
                                                  first (and only) element of $A$ and the first (and only)
                                                  element of $B$
11:                 return $C$
12:             else
13:                 for each $i$ in [0, 3] do
14:                     if $i \leq 1$ then
15:                         let $D$ be a reference to $A$     ▷ any modification of $D$ propagates to $A$ as well
16:                     else
17:                         let $D$ be a reference to $B$     ▷ any modification of $D$ propagates to $B$ as well
18:                     if all of the elements from $D$ are of size 1 then
19:                         create a brand-new predicate $F$
20:                         for each element $P$ of $D$ do
21:                             add $(\neg F \lor P)$ in toBeAdded
22:                         $D$ becomes $[[F]]$, where a pair of square brackets represent an array
23:                     else
24:                         for each element $P$ of $D$ do
25:                             create a brand-new predicate $F_i$
26:                             add $(\neg F_i \lor P_1 \lor P_2 \ldots \lor P_j - 1 \lor P_j)$ in toBeAdded, where $j$ is the size of $P$
27:                             $P$ becomes $F$
28:                 MERGE($A, B$)
```

Finally, after the algorithm has been completed, all of the elements from the set *toBeAdded* have to be appended to the resulting CNF. The approach of appending those elements is described later, after the presentation of the algorithm.

Following the algorithm of merging two CNFs on a conjunction or on a disjunction, the algorithm for retrieving CNF from the parse tree is:

Algorithm 8 CNF Merge Algorithm Parse Tree

```
 1: all nodes are visited using a depth-first search
 2: when the recursive call for a node $P$ (arbitrary node) is about to return, do    ▷ after executing the algorithm for all of the nodes, the CNF of the whole parse tree will reside in its root
 3:     $mergeWithChildren \leftarrow$ False
 4:     if $P$ does not hold any entity then
 5:         assign an empty CNF (i.e. [ ]) to $P$
 6:         $mergeWithChildren \leftarrow$ True
 7:     else if $P$ holds a literal $L$ then
 8:         assign a CNF containing only $L$ (i.e. [[L]]) to $P$
 9:         $mergeWithChildren \leftarrow$ True
10:     if $mergeWithChildren$ is True then
11:         $curEntity \leftarrow$ the entity of $P$
12:         for each children $son$ of $P$ do    ▷ the node P will hold as an entity the CNF corresponding for its whole subtree
13:             $childEntity \leftarrow$ the entity of $son$
14:             $curEntity \leftarrow$ MERGE(curEntity, childEntity)
```

As described before, the disjunctions from *toBeAdded* have to be appended as conjunctions to the CNF retrieved from the parse tree. This is done by concatenating the former to the latter.

By analysing the complexity of the merge algorithm, it is concluded that the algorithm introduces a polynomially (in respect of the size of the CNF) bounded amount of additional conjunctions. In the worst case, the time complexity of this part will be $\mathcal{O}(N)$ is the number of nodes from the parse tree.

The depth-first search algorithm runs in polynomial time, precisely $\mathcal{O}(N^2)$. Even though much more sophisticated solutions may have been implemented (such as one involving the small-to-large trick (CP-Algorithms, 2021), optimizing the time complexity to $\mathcal{O}(Nlog_2N)$ overall), it was considered that this was unnecessary, since that solution represents only a micro-optimization (being dominated by $\mathcal{O}(N^2log_2N)$, from the *Skolemization* step).

Hence, the overall time complexity for the retrieval of the Simplified Clausal Normal Form is $\mathcal{O}(N^2)$.

Appendix 1 represents the majority of operations described in this section.

4.4 Back-end

This section discusses the theorem proving part of the project. First, the data structure chosen for representing the clause form is described. Following that, the implementation of the three theorem provers (basic, depth-ordered and two-variable) is explained in detail.

4.4.1 Clausal Normal Form and Unification

Since the theorem proving might require complex representations of the literals from the CNF, the way of representing the clausal normal form is changed. Instead of regarding it as an array of arrays, as used and described in the intermediate representation, a much more convenient way of storing (in respect of the unification) is required.

Hence, the CNF now encapsulates an array of clauses. A higher level of granularity is needed, so that each clause encapsulates an array of literals. Finally, a literal stores behind its representation an array of terms. A term can be quite complex, following the definition (for example, a function symbol having nested other terms). A term will further represent an array of terms. For instance, $(P(x, f(g(x))) \vee R(x, y)) \wedge (Q(x, y, z) \vee P(f(w(t(x), y)), z)$ is modelled as follows using the aforementioned considerations:

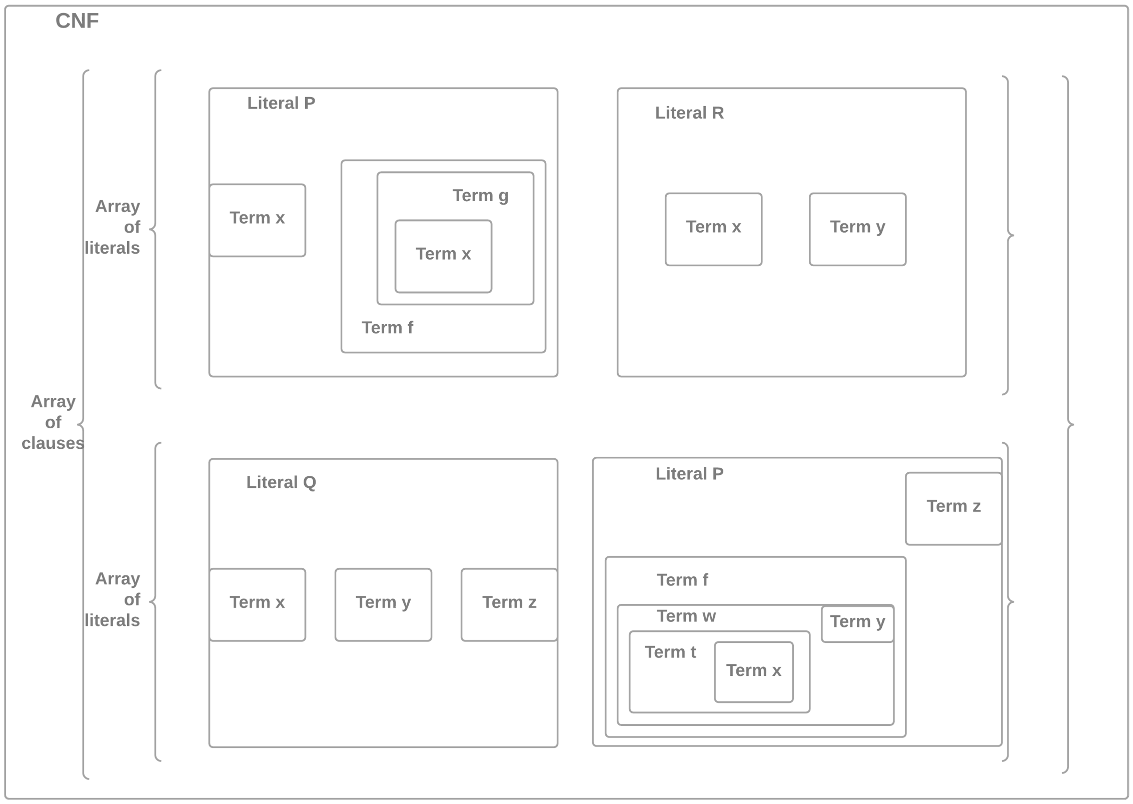

Figure 4: $(P(x, f(g(x))) \vee R(x, y)) \wedge (Q(x, y, z) \vee P(f(w(t(x), y)), z)$

In order to present the unification, a bottom-up approach is utilized (i.e. the algorithm first unifies between terms, then between literals and finally between clauses; the last part is later connected with the basic theorem prover).

4.4.1.1 Term Unification

The first tool which is described as part of this section is the recursive function called *"findPartialSubstitution"*. It takes as an input two terms, A and B. The function attempts to unify A with B and return true if the terms are already the same, false if it fails, and a substitution in case the algorithm is able to continue. In the case of failure, an attempt to unify B with A is performed. Finally, if this is also unsuccessful, A and B are not unifiable. The algorithm for *findPartialSubstitution* is the following:

Algorithm 9 Find Partial Substitution

```
    Input: A, B two terms
    Output: True if the terms are the same, a substitution if this is still not determined, False otherwise
 1: function findPartialSubstitution($A, B$)
 2:     if $A = B$ then
 3:         return True
 4:     else
 5:         if $A$ is constant then
 6:             return False
 7:         if $A$ is a variable, called $V$ then
 8:             if $B$ contains $V$ then
 9:                 return False
10:             else
11:                 return $\{V, B\}$        ▷ signifying that V is mapped to B (which can be a function, a constant or another variable)
12:         if $A$ is a function symbol then
13:             if $B$ is not a function symbol then
14:                 return False
15:             else
16:                 if $A$ and $B$ do not have the same function name then
17:                     return False
18:                 else
19:                     $N \leftarrow$ the size of the argument list for $A$        ▷ since $A$ and $B$ share the function name, this has to be the same for both
20:                     for each $i$ in $[0, N-1]$ do
21:                         ${T_i}^1 \leftarrow i^{th}$ argument of $A$        ▷ which is a term
22:                         ${T_i}^2 \leftarrow i^{th}$ argument of $B$        ▷ which is a term
23:                         $result \leftarrow$ findPartialSubstitution(${T_i}^1, {T_i}^2$)
24:                         if $result$ is not True then
25:                             return $result$
26:                 return True
```

To establish the time complexity of the aforementioned algorithm, the function *treeSize* is introduced, 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 $\rightarrow$ 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 it is possible to write K as g$(t_1, t_2, \ldots, t_n)$; therefore, $treeSize(K) = 1 + \sum_{i=1}^{n} treeSize(t_i)$

Having defined the *treeSize*, the time complexity for findPartialSubstitution is established: $\mathcal{O}(M^2)$, where M is the minimum between $treeSize(A)$ and $treeSize(B)$. This is owing to the fact that at each call of the function, in order to check if A and B are identical, the tree of each of A and B is traversed. For highly nested A and B, the algorithm performs this check as many times as there are nodes in the tree of the term with the lowest number of nodes.

The second tool which is presented is a function called "augmentUnification", which takes as an input two terms, A and B. This function is simpler than the previous one, representing solely a "wrapper" for *findPartialSubstitution*. Its purpose is to call *findPartialSubstitution* for A and B. In case of failure, it calls the same function but for B and A (so the arguments are just reversed). Finally, if both of the calls fail, the function returns false. Conversely, if both of the calls return true, this function returns true. Otherwise, the function returns a substitution (revisit the previous algorithm for better understanding). An analogy can be drawn between this function and a Python generator - if it does not return false at the initial call, it keeps returning substitutions until either of true or false is reached; if false is reached after a couple of substitutions returned, this implies that the operations have to be rolled back. The function *augmentUnification* has the same time complexity as *findPartialSubstitution*.

Finally, the third tool is a function called "createDeepCopy", taking as input one term. This function creates a deep copy of the input and returns it. The implementation uses a variation of the depth-first search. The report will not present its details since the algorithm does not contain anything sophisticated. This function is required in the next subsection. The time complexity for it is $\mathcal{O}(R)$, where R is the value of the *treeSize* for the input term.

4.4.1.2 Literal Unification

In the previous section three tools (functions) were discussed which are used in the back-end. In order to avoid confusion, the functions *"augmentUnification"* and *"createDeepCopy"* are referred to as *"augmentUnificationTerm"* and *"createDeepCopyTerm"* from now on.
The report will present the function *"augmentUnification"* which applies the algorithm of unification on literals. This receives as an input two literals. Similarly, after this section, this function is referred to as *"augmentUnificationLiteral"* for clarity.
This function does the following:

Algorithm 10 augmentUnification

```
    Input: $L_1$, $L_2$ two literals.
    Output: True if the literals were unified.
 1: function AUGMENTUNIFICATION($L_1$, $L_2$)
 2:     if the predicate-names for $L_1$ and $L_2$ are different then
 3:         return False
 4:     else
 5:         $N \leftarrow$ the arity of $L_1$          ▷ Note that both $L_1$ and $L_2$ have to be of the same arity
 6:         for each $i$ in $[0, N-1]$ do
 7:             ${T_i}^1 \leftarrow i^{th}$ term of $L_1$
 8:             ${T_i}^2 \leftarrow i^{th}$ term of $L_2$
 9:             $result \leftarrow$ AUGMENTUNIFICATIONTERM(${T_i}^1, {T_i}^2$)
10:             if result is False then
11:                 return False
12:             else if result is substitution then
13:                 return $result$
14:     return True
```

The time complexity of this function is $\mathcal{O}(E + \sum_{i=1}^{N} M_i)$, where E is the maximum between the length of the predicates, N is the number of arguments the predicate has and M_i is defined as being the minimum between the values of *treeSize* for the i_{th} pair. Similarly to the previous section a function called *"createDeepCopyLiteral"* is introduced. This function creates a deep copy of a given literal. The complexity of this function is $\mathcal{O}(F + \sum_{i=1}^{N} R_i))$, where F is the length of the corresponding predicate, N is the number of terms (arguments) the predicate has and R_i corresponds to the *treeSize* value of the i_{th} term.

4.4.1.3 Clause Unification

This functionality was implemented using C++ templates (CPP Reference, 2021) and it represents the core of theorem proving. Moreover, this part makes use of most modern C++, std::variant (CPP Reference, 2021) and lambda expressions (CPP Reference, 2021) which are heavily adopted.

The function called "attemptToUnify" is described further. It is a templated function and receives 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 why *attemptToUnify* is a templated function (since the types of the functions have to be passed as template arguments).

At this point, the algorithm behind *attemptToUnify* is presented. Some of the details are omitted for simplicity. The algorithm does the following:

Algorithm 11 attemptToUnify

Input: F and S clauses, literalCheck and resolventCheck boolean functions

Output: a list storing all of the possible clauses derived using the resolution rule from the clauses F and S

```
1:  function ATTEMPTTOUNIFY($F$, $S$, $literalCheck$, $resolventCheck$)
2:      $result \leftarrow []$
3:      for each literal $X$ in $F$ do
4:          for each literal $Y$ in $S$ do
5:              if if LITERALCHECK($X, Y$) returned True then
6:                  $F' \leftarrow$ a deepCopy of $F$
7:                  $S' \leftarrow$ a deepCopy of $S$
8:                  if $F'$ and $S'$ have common variables then
9:                      rename the variables such that they do not have common variables anymore
10:                 while AUGMENTUNIFICATION($F', S'$) returns a substitution do
11:                     apply the substitution on both $F'$ and $S'$
12:                 if AUGMENTUNIFICATION($F', S'$) returns true then
13:                     remove $X$ from $F'$ and $Y$ from $S'$ by concatenating $S'$ to $F'$
14:                     if RESOLVENTCHECK($X, F'$) returns True then
15:                         append $F'$ to $result$
16:     return $result$
```

Note that the application of substitution represents only a mapping from one term to another one and it can be implemented in $\mathcal{O}(\sum_{i=1}^{N} R_i)$ for an N-ary predicate, with R_i corresponding to the value of *treeSize* for the i_{th} term of it.

Similarly, the step of renaming all variables from F' and S' can be implemented using the application of multiple substitutions subsequently. This is achieved by retrieving all distinct variables from F' and S' in two sets, $V_{F'}$ and $V_{S'}$, respectively. Once these two sets have been obtained, their intersection has to be computed, and then a new variable (which is not part of neither of $V_{F'}$ and $V_{S'}$) is mapped for each variable which is part of the intersection. This assignment is performed on only one of the clauses F' or S'. The complexity of this step is $\mathcal{O}(\sum_{i=1}^{N} R_i + \sum_{i=1}^{N} Q_i + C \times max(\sum_{i=1}^{N} R_i, \sum_{i=1}^{N} Q_i))$, assuming that C is the cardinal of the intersection of $V_{F'}$ and $V_{S'}$ and R and Q correspond 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 no criteria is applied to choose the clause which will be substituted when assigning new variables for those variables which are part of the intersection).

Finally, the complexity of the *attemptToUnify* algorithm is $\mathcal{O}(|F| \times |S| \times LC \times RC \times (\sum_{i=1}^{N} R_i + \sum_{i=1}^{N} Q_i + (max(V_{F'}, V_{S'}) + C) \times max(\sum_{i=1}^{N} R_i, \sum_{i=1}^{N} Q_i)))$, where:

- $|F|$ and $|S|$ are the size of F and S, respectively
- LC is the number of operations *literalCheck* performs for a pair of literals in the worst case (this cannot be determined precisely since this function is passed as a parameter to the *attemptToUnify*, so its complete implementation is unknown)
- RC is the number of operations *resolventCheck* performs 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 the time complexity analysis is the same as for the variable renaming step, with the consideration that in the worst case, an amount of variables equal to the the maximum between the cardinals of $V_{F'}$ and $V_{S'}$ has to be renamed

4.4.2 Basic Theorem Prover

This section of the report presents the details of the basic theorem prover, the algorithm which is behind all of the theorem proving algorithms implemented as part of this project.

This part is by far the most convoluted. Because of that, the report will attempt to present a simplified version of the implementation of the algorithms.

This component makes use of multithreading and it is also persistent (clauses may be dynamically appended and popped while a timeline of these operations is maintained). For the persistence part, the algorithm keeps track of a timestamp named *"currentTimestamp"* (in fact, this is more complex on the implementation side, but presenting the mechanism this way allows for a solid understanding).

Since the basic theorem prover may not terminate (due to the fact that there is no control on the term depth), the complexity analysis in some cases is omitted.

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 encountered so far (by maintaining a set of clauses called "HSet"). Because it is possible for the clauses to be quite large, a hashing function which relabels the variables was used (i.e. $\sim P(x, f(x), y, x, g(y))$ becomes $\sim P(v_1, f(v_1), v_2, v_1, g(v_2)))$. For simplicity, the implementation of the hashing function will not be described.

4.4.2.1 Factoring Rule

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

This part is executed on one single clause. Hence, multiple clauses are treated independently. This is the reason why the programmer decided to make use of multithreading here, by parallelizing this procedure.

Further, this procedure is described using a top to bottom approach (the high-level overview first):

Algorithm 12 Factoring rule implementation

```
Input clause: $C$, passed by reference. AllClauses and Hset are global variables.
Output: None (void function)
function
    if $C$ is marked as deleted then return
    $H \leftarrow$ current hash of $C$
    removeDuplicates(C)
    if $C$ was changed then
        $H' \leftarrow$ new hash of $C$
        remove $H$ from $HSet$
        insert $H'$ in $HSet$
        remove $H$ from $HSet$
        $H \leftarrow H'$
    perform isTautology(C)
    if $C$ is a tautology then
        mark $C$ as deleted and save additionally the timestamp of this moment
        return
    $unificationResult \leftarrow$ the list of clauses resulting from unifying two literals of the same sign from $C$
    for each $currentClause$ in $unificationResult$ do
        if $currentClause$ is not a tautology then
            perform removeDuplicates(currentClause)
            $currentHash \leftarrow$ hash of $currentClause$
            if $currentHash$ 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 the procedure finds such a pair, it reports that a tautology has been found.

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

- F and S will point to the same clause
- the step in which the intersection of $V_{F'}$ and $V_{S'}$ is enforced to be empty is omitted *literalCheck* returns True only if the two literals have the same predicate name and the same sign
- *resolventCheck* always returns True
- the time complexity is the same as for *attemptToUnify* with the consideration that C (the cardinal of the intersection of $V_{F'}$ and $V_{S'}$) is equal to 0

4.4.2.2 Subsumption Rule

The subsumption is executed on a single clause, in the same way as the factoring rule, 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 can be parallelized.

Algorithm 13 Subsumption rule implementation

```
    Input clause: $C$
    Output: None (void function)
1: function SUBSUMPTION(C)
2:     if $C$ is marked as deleted then
3:         return
4:     $CHS \leftarrow$ the ordered set of the hashes for its literals    ▷ i.e. for the clause $(\neg P(x) \lor L(x, y) \lor R(z, y, x))$, the corresponding set is $\{L(v_1, v_2),\ \neg P(v_1),\ R(v_3, v_2, v_1)\}$
5:     for each clause $E$ which has not been deleted do
6:         compute $EHS$
7:         if $EHS$ is included in $CHS$ then
8:             • $C$ is subsumed by $E$
9:             • let $T$ be the current timestamp
10:            • $C$ is marked as deleted and $T$ is remembered    ▷ i.e. the timestamp at which it got marked as deleted
11:        if $CHS$ is included in $EHS$ then
12:            • $E$ is subsumed by $C$
13:            • let $T$ be the current timestamp
14:            • $E$ is marked as deleted and $T$ is remembered    ▷ i.e. the timestamp at which it got marked as deleted
```

The algorithm for subsumption works for ground clauses and it may work as well for clauses which are 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 may not match the relabelling from another clause. However, a correct implementation for non-ground clauses may be exponential, which outweighs the benefits.

4.4.2.3 Resolution Rule and Theorem Proving

Next, the report presents the resolution rule together with the previous two rules, as part of the basic theorem proving. Similarly to the subsumption and factoring rules, this rule is implemented as well using multi-threading

The algorithm for basic theorem proving is now presented. It makes use of the previously discussed function, *attemptToUnify*. The algorithm is the following:

Algorithm 14 Basic Theorem Prover implementation

Input: AllClauses and Hset are global variables. The implementation for the *literalCheck* returns true when applied on two literals which have the same predicate names and opposite signs. The implementation for *resolventCheck* returns always True.

Output: True if the empty clause is derived.

```
1: function
2:     repeat ← a boolean initialized with True
3:     while repeat is True do
4:         repeat ← False
5:         for all clauses C in AllClauses do          ▷ this step creates separate threads in the actual implementation
6:             if C is marked as deleted then
7:                 continue
8:             for all clauses G in AllClauses do
9:                 if G is marked as deleted then
10:                    continue
11:                unificationResult ← attemptToUnify(C, G, literalCheck, resolventCheck)
12:                if unificationResult is not empty then
13:                    for all of the clauses J from unificationResult do
14:                        if the clause J is a tautology then
15:                            continue
16:                        perform removeDuplicates on J
17:                        H ← hash of J
18:                        if H exists in HSet then
19:                            continue
20:                        else
21:                            insert H to HSet
22:                            if J is the empty clause then
23:                                return True
24:                            append J to AllClauses
25:                            repeat ← True
26:    return False                                     ▷ saturation was reached
```

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

4.4.3 Depth-Ordered Theorem Prover

The depth-ordered theorem prover adds heuristics for the term depth on top of the basic theorem prover described above. These heuristics will guarantee the termination when used in the context of the two-variable fragment and they further result in a different implementation for the *resolventCheck*. Hence, solely the implementation of this function is detailed in this section.

As before, a top-to-bottom approach for presenting the *resolventCheck* function is utilized. The following implementation uses a function called *isAOrdering*, which is detailed later.

Hence, the *resolventCheck* function does the following:

Algorithm 15 resolventCheck implementation for the depth-ordered theorem prover

```
    Input: $L$ and $C$, the resolved literal and the clause, respectively
    Output: True if there is no literal $T$ in $C$ such that $L <_A T$
1: function resolventCheck($L$, $C$)
2:     $LC \leftarrow$ the set of all literals which are part of the clause $C$
3:     for each literal $T$ in $LC$ do
4:         if isAOrdering($L, T$) is True then
5:             return False
6:     return True
```

Algorithm 16 isAOrdering function implementing $<_A$

```
    Input: $A$ and $B$, two literals
    Output: True if $A <_A B$
1: function isAOrdering($A$, $B$)
2:     $AMax \leftarrow$ be the maximum depth of a variable in $A$
3:     $BMax \leftarrow$ be the maximum depth of a variable in $B$
4:     if $A$ does not have variables then
5:         return False
6:     if $B$ does not have variables then
7:         return False
8:     if $AMax \leq BMax$ then
9:         return False
10:    $AVars \leftarrow$ be the set of variables from $A$
11:    $BVars \leftarrow$ be the set of variables from $B$
12:    for each variable $V$ in $AVars$ do
13:        $AV \leftarrow$ the maximum depth of $V$ in $A$
14:        $BV \leftarrow$ the maximum depth of $V$ in $B$
15:        if $AV \geq BV$ then
16:            return False
17:    for each variable $V$ in $BVars$ do
18:        $AV \leftarrow$ the maximum depth of $V$ in $A$
19:        $BV \leftarrow$ the maximum depth of $V$ in $B$
20:        if $BV \leq AV$ then
21:            return False
22:    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 of the entire depth-ordered theorem prover is difficult to compute (even though the termination is guaranteed). However, computing the complexity for the *resolventCheck* and *isAOrdering* function is achievable.

The time complexity for the *resolventCheck* is $\mathcal{O}(\sum_{i=1}^{|C|} A(L, LC_i))$ where $|C|$ represents the size of the clause set C and $A(L, LC_i)$ represents the maximum number of operations (in the worst case) consumed on running the *isAOrdering* function on the literal L and the i^{th} literal from the clause C.

Since the definition of $A(L, LC_i)$ does not yet have the desired level of verbosity, the time complexity for $A(L, W)$ is further defined, where both L and W are literals, the size of their arities being LA and WA, respectively. Finally, the time complexity for $A(L, W)$ is $\mathcal{O}(\sum_{i=1}^{N} L_i + \sum_{i=1}^{N} W_i)$, where L_i and W_i are the values corresponding to the *treeSize* function for the i_{th} term of the literal L and i^{th} term of the literal W, respectively.

4.4.4 Two-Variable Theorem Prover

This section presents the two-variable fragment theorem prover. The explanation begins with describing the variable renaming approach and continues with explaining the first version which assumes that the formula to be proved does not contain equality. Finally, the second version for the case with equality is detailed.

4.4.4.1 Variable renaming

Before presenting the algorithms, the approach of renaming the variables is as follows:

Algorithm 17 variable renaming procedure for the two-variable fragment

```
    Input: $C$, the set of clauses
    Output: True/False
 1: function
 2:     $G \leftarrow$ be an empty graph, to which nodes and edges are dynamically added
 3:     for each clause $Q$ in $C$ do
 4:         $LVMax \leftarrow$ maximum number of distinct variables across all of the literals of the clause $Q$
 5:         if $LVMax$ is greater than 2 then
 6:             return False                ▷ i.e. the given formula is not part of the two-variable fragment
 7:         else
 8:             for each literal $L$ in $Q$ do
 9:                 $V \leftarrow$ the set of variables occurring in $L$
10:                 if $V$ has size 2 then
11:                     $V_1 \leftarrow$ first variable
12:                     $V_2 \leftarrow$ second variable
13:                     draw an edge from the node corresponding to $V_1$ to the node corresponding to $V_2$ in $G$
14:                 if $V$ has size 1 then
15:                     add a node corresponding to the variable from $V$ in $G$
16:     return True
```

Eventually, if the algorithm above returns True, the algorithm proceeds by 2-colouring the graph G. Since the graph built by the algorithm aforementioned is bipartite, 2-colouring is performed in linear time (in respect of the size of the graph, which is bounded by the number of distinct variables occurring in the given formula, further bounded by the size of the input), using the depth-first search algorithm. As detailing this algorithm is out of the project scope, the exact implementation will be omitted.

4.4.4.2 Without Equality

Due to the fact that the two-variable theorem prover inherits from the basic theorem prover, the first step of defining it presents the implementations of the *literalCheck* and *resolventCheck*.

The implementation of the *literalCheck* is as follows:

Algorithm 18 literalCheck implementation for the two-variable fragment theorem prover

```
    Input: $F$ and $S$, two literals
    Output: True/False
 1: function literalCheck($F$, $S$)
 2:     if $F$ and $S$ do not have opposite signs then
 3:         return False
 4:     if $F$ and $S$ do not have the same predicate names then
 5:         return False
 6:     $V_F \leftarrow$ the variables contained in $F$
 7:     $V_S \leftarrow$ the variables contained in $S$
 8:     if either of $|V_F|$ and $|V_S|$ has the size 2 and none of $F$ and $S$ represent the equality (i.e. their predicate
    names are not called Equality) then
 9:         return True
10:     else
11:         return False
```

It is important to note that even though *literalCheck* checks for equality, during the course of this section, equality is never encountered. The reason to define this check now is to avoid redefining this function again in the following section.

The implementation of the *resolventCheck* is the same as the one for the depth-ordered theorem prover. The function *resolventCheck* can be implemented identically as for the basic theorem prover, however that does not guarantee termination.

The time complexity for the *literalCheck* function is $\mathcal{O}(F^p + S^p + \sum_{i=1}^{N} FT_i + \sum_{i=1}^{N} ST_i)$, where F^p and S^p are the lengths of the predicate names for F and S, respectively and FT_i and ST_i are the values corresponding to the *treeSize* function for the i^{th} term of the literal F and i^{th} term of the literal S, respectively.

Now, the basic theorem prover is executed using the aforementioned *literalCheck* and *resolventCheck* implementation. It either derives the empty clause or reaches saturation. In case the former happens, a contradiction has been found and hence, the algorithm reports validity for the given formula. Otherwise, all clauses containing literals in two variables (excepting the Equality predicate) are removed and the algorithm continues.

Let D be a brand-new instance of the depth-order theorem prover which, as mentioned, is a persistent data structure, being able to efficiently perform appendance and removal from the set of clauses. Let C be the set of clauses after the disposal of those mentioned above. The next step of the algorithm is a backtracking procedure doing the following:

Algorithm 19 Backtracking procedure for the two-variable fragment

Input: the set of clauses C (which is 0-indexed), the index R (initially 0) of the currently processed clause and the depth-order theorem prover D (which is passed by reference)

Output: False if it derives the empty clause in all of the cases

```
1: function BACKTRACKING($C$, $R$, $D$)
2:     if $R$ is equal with the size of $C$ then
3:         if run $D$ returns True then                         ▷ $D$ holds all clauses added so far
4:             return True
5:         else
6:             return False
7:     else
8:         group the literals from the $R^{th}$ clause on the set of variables occurring in that respective literal
9:         for each such set $S$ do
10:            $W \leftarrow$ a new clause containing all of the literals from $S$
11:            append $W$ to $D$, and remember the timestamp $T$ at which $W$ was added
12:            if BACKTRACKING($C$, $R + 1$, $D$) returns True then              ▷ recursive call
13:                return True
14:            revert $D$ to the timestamp $T$      ▷ that will not only remove $W$ from $D$, but will also
                                                      remove all of the clauses inferred past $W$
15:    return False
```

Since the set of the variables is either the empty set, x or y, the time complexity of the algorithm is $\mathcal{O}(3^{|C|} * DTP)$, where $|C|$ is the size of the clause set and DTP is the amount of operations the depth-ordered theorem prover performs in the worst-case for a given set of clauses.

4.4.4.3 With Equality

Assuming that the formula contains equality, the algorithm has to resolve the equality after the disposal was executed and before the initial call of the backtracking.

The approach for implementing this part involves the conversion of the set of clauses C into a string and subsequently appending to that string the implications described by the Lemmas 5 and 6 in De Nivelle and Pratt-Hartmann (2001). After performing these concatenations, the string is passed further as an input to the Clausal Normal Form algorithm. The output of the latter will overwrite C.

Since this part implements exclusively string manipulations consistent with either the Lemmas 5 and 6 from De Nivelle and Pratt-Hartmann (2001) or with what has been presented earlier in the theoretical part of this report, it is omitted for simplicity.

5 Evaluation

This section is dedicated to the evaluation part of the project and highlights the main methods of asserting the correctness of the implementation.

5.1 Unit testing

Google's testing framework was used in order to unit-test several functionalities from the front-end component. However, in the early stages of the development, it was noticed that the value unit-testing brings is not as large as the end-to-end testing one. This is because the functionality of the theorem proving is distributed across multiple components and the testing should focus more on the communication between these components, rather than on the implementation of each component separately. If the former fails, the latter might not fail whereas if the former passes, the latter will pass with a very high chance.

5.2 End-to-End testing

End-to-end testing is a black-box testing technique of verifying the functionality of the code. For the purpose of the project, this translates into expecting an exact output for a specific input.

5.2.1 Using the problems found on tptp.org

The first method of achieving this testing was using the problems available on tptp.org. Even though the archive of problems from tptp.org is generous in terms of size, after some filtering, only less than a hundred of these problems represented valid inputs for the use-case of the implementation. This occurred because only tests containing not more than two variables, not containing function symbols in the input or special operators like the arithmetic ones were of interest.

The final archive of tests contained only 47 problems, the majority of them being part of the "SYN" (possibly from "syntax") problem set.
Even though 47 test-cases is not a large amount, this part of the testing detected most of the mistakes which happened during the implementation process. For automating this part, a Linux Bash script was produced and used (this shall be revisited later).

5.2.2 Using general formulas

A general formula generator was written in order to achieve end-to-end testing. Even though the formulas generated represented a sound way of detecting potential bugs, those were insufficient. The reason why was because the implementation of this generator involved a recursive function whose calls were randomly weighted. Although the generated formula was a valid first-order logic formula, identifying whether it is a theorem or not, was practically impossible without doing the same for a third-party software which acts as a verifier. Vampire may have been that software, but generating the same formula for the theorem prover as for Vampire represented a challenge as well. Moreover, even if a lot of effort would have been invested in finding a way of resolving these issues, the generator could not have been adapted in such a manner, so as to generate a formula with an exact number of variables. This would have implied that large formulas of the two-variable fragment would have been produced only probabilistically. Finally, since the two-variable fragment is known as being decidable only when it does not contain function symbols, the generator should have been resilient enough to address this as well. Approaches tackling all of the above issues, excluding the last one, exist. However, the last problem is probably too complex to be solved in polynomial time (since keeping track of the number of variables and the potential arising function symbols, while generating a totally unrestricted first-order logic formula, is infeasible).

Therefore, the idea of using the general formula generator for black box testing was abandoned. Fortunately, having a third-party verifier proved itself as being a solid starting point and the exact strategy is described in the next section.

5.3 Testing against Vampire

This section presents a test-generator. The contributions for the ideas behind the generator (where not referenced) are of my supervisor.

In order to execute end-to-end testing using Vampire, a definition of the Scott's normal form (Otto, 2001) is presented:

$$\forall x \forall y(\alpha \vee x = y) \wedge \bigwedge_{i=1}^{C}(\forall x \exists y(\beta_i \wedge x \neq y))$$

In the formula above, α is a formula in CNF containing A clauses and β_i is an atomic formula. All atoms occurring in the aforementioned form have precisely the arity two.

In order to adapt the Scott's normal form as a starting point for a test-generator for two-variable fragment formulas, the following conventions are adopted:

- the predicate names occurring in β_i can occur in α and vice versa
  - hence, there is a global set Q (of cardinality $|Q|$) of predicate names which includes all names from α and β_i from which the predicate names are then picked
- each of the A clauses of will have exactly L literals, with L being chosen randomly from the range $[LMIN, LMAX]$, where $LMIN$ and $LMAX$ are parameters for the generator representing the minimum and the maximum number of literals per clause, respectively

Having defined the Scott's normal form, generating a formula in the two-variable fragment is as follows:

Algorithm 20 Scott's Normal Form random generator

```
    Input: $A$, $C$, $P$, $LMIN$, $LMAX$
    Output: a string, containing a two-variable fragment formula in Scott's Normal Form
    Assumptions: let $Q$ be the global set of predicate names previously described
 1: function GENERATERANDOMSCOTTNORMALFORM($A, C, P, LMIN, LMAX$)
 2:     $L \leftarrow$ a number, randomly picked from the range $[LMIN, LMAX]$
 3:     $Q \leftarrow$ P randomly-generated unique predicate names    ▷ containing letters of the Latin alphabet and being generated in an increasing order of their lengths (i.e. first all predicate names of size 1, then all predicate names of size 2 and so on)
 4:     $F \leftarrow$ a string initialized with "$\forall x \forall y$("
 5:     $\alpha \leftarrow$ an empty set of strings
 6:     for $A$ times do
 7:         $W \leftarrow$ an empty list of literals
 8:         for $L$ times do
 9:             $R \leftarrow$ an empty string
10:             $B \leftarrow$ a random boolean
11:             if $B$ = True then
12:                 append negation to $R$
13:             $U \leftarrow$ a random predicate name from $Q$
14:             append $U$ to $R$
15:             $B' \leftarrow$ a random boolean
16:             if $B'$ = True then
17:                 append "(x, y)" to $R$
18:             else
19:                 append "(y, x)" to $R$
20:             append $R$ to $W$
21:         join all elements of $W$ on disjunction and append the result to $\alpha$    ▷ i.e. "¬Pa(x, y)", "Pb(y, x)" becomes $\neg Pa(x, y) \vee Pb(y, x)$
22:     join all the elements of $\alpha$ on conjunction and append the result to $F$
23:     append "$\vee x = y) \wedge$" to F
24:     $\beta \leftarrow$ an empty string
25:     for $C$ times do
26:         $\beta_i \leftarrow$ a string initialized with "($\forall x \exists y$("
27:         $K \leftarrow$ a predicate name at random from $Q$
28:         $B'' \leftarrow$ a random boolean
29:         if B = True then
30:             append "(x, y)" to $\beta_i$
31:         else
32:             append "(y, x)" to $\beta_i$
33:         append "$\wedge x \neq y)) \wedge$" to $\beta_i$
34:         append $\beta_i$ to $\beta$
35:     remove the last character from $\beta$
36:     append $\beta$ to $F$
37:     return $F$
```

The algorithm from above runs in polynomial time in respect of $A * LMAX + C$, assuming that the lengths of the predicate names are reasonably small in comparison with the parameters, which is the case since the following function is exponential:

$$t(x) = \text{the number of unique predicate names of length, smaller or equal with } x$$

Finally, the test-generator from above has the following advantages:

- it can be adapted to both the theorem prover and Vampire (it does not use any particularity of either of their languages), assuming that they have the same operator precedence which in theory happens, but extra brackets are added for safety
- it enforces the number of variables (this is valuable since all of the implemented theorem provers can be tested using it)
- when negated, it does not add any function symbols (it adds only constant), which allows the testing on the two-variable fragment without function symbols
- it tests as well the front-end and the intermediate representation when the formula contains equality, which would happen in most of the cases of interest

Finally, a script which generates random two-variables formulas is implemented. This script is able to perform end-to-end testing as the output of Vampire is considered correct. Such a script was actually implemented as part of the project and in 25,000 randomly generated formulas it detected the same verdict for both Vampire and the two-variable theorem prover.

5.4 Stress testing

In order to optimize the theorem prover, its core infrastructure (involving the factoring, resolution and subsumption rules) was multi-threaded. A challenge of this design aspect was related to testing, precisely how to ensure that at different runs, the theorem prover does not have different outputs in terms of meaning (i.e. derived empty clause versus reaching saturation).

In addressing this issue, the 47 tests from tptp.org were used. The script running those tests for each theorem prover (basic, depth-ordered and two-variable) was run 2,000 times and the results were checked for each test-suite of the 47 tests in part. None of them failed, as all of them were consistent with the expected output.

The value this type of testing brought to the project was considerable, detecting many bugs during the implementation process, which other testing did not reveal. The best example is a bug related to a race condition which was detected in 1,000 runs, occurring just 6 times (6 out of 47,000).

6 Experiments

This section is dedicated to the experiments and the collection of statistics. It makes an extensive use of the parameters of the Scott's normal form test-generator (i.e. A, C, $LMIN$, $LMAX$, P). Since the aim of the project was to implement the decision procedure for the two-variable fragment, only experiments using the two-variable theorem prover and the aforementioned generator were completed.

6.1 Run time and memory consumption

The first experiment was to see how fast the theorem prover performs on random formulas. The major finding was that the theorem prover works well on formulas having 500-1000 characters (generally, this happens for values for A, C, $LMAX$ lower than 6). "Well" in this case implies that the same formula generated for both the theorem prover and Vampire is either proved or refuted within 30 seconds. For formulas of size larger than the size previously mentioned, the theorem prover runs within significantly more time. This happens mostly because the size of the clause form passed as an input to the backtracking algorithm exceeds 30, which makes the computation infeasible in a reasonable amount of time.

During this experiment it was observed that the memory consumption is perfectly consistent with the expectations: the amount of memory consumed is polynomially bounded by the size of the input.

6.2 Satisfiability distribution

The second experiment's purpose was to discover what is the satisfiability distribution when ranging the values of A and C, with $LMIN$, $LMAX$ and P fixed. The following three charts present the results.

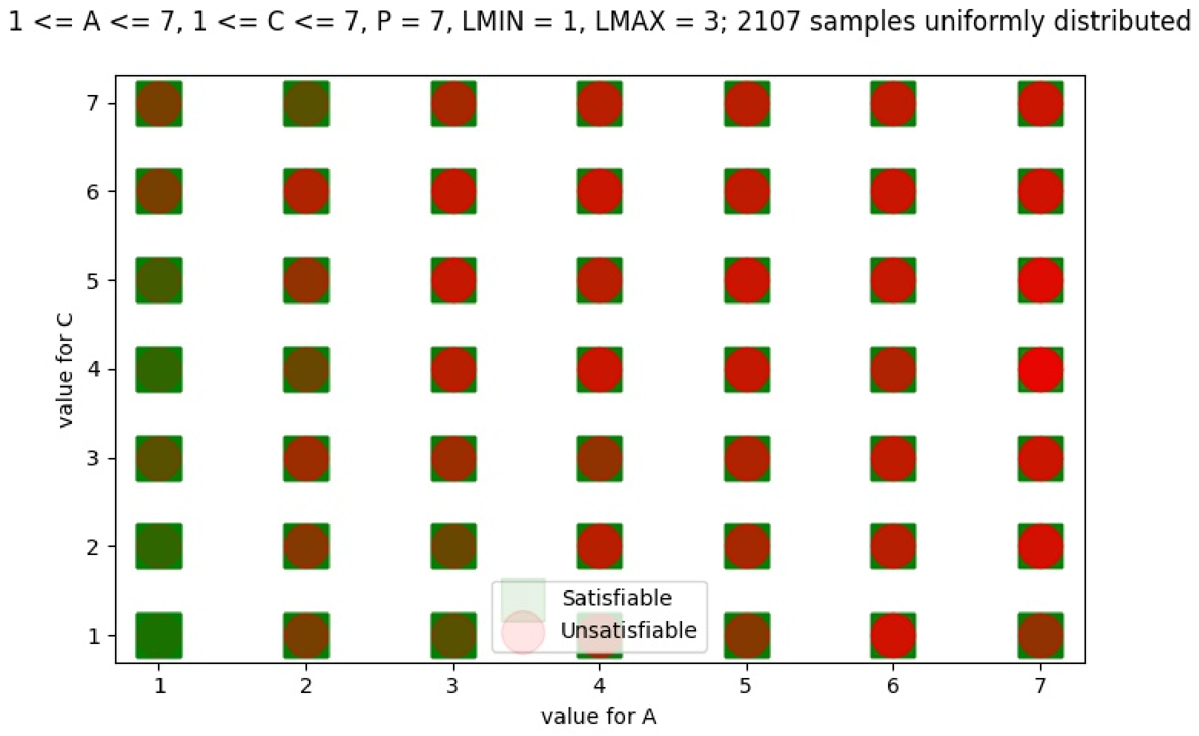

Figure 5: Satisfiability Distribution for fixed P; A and $C \in [1,7]$

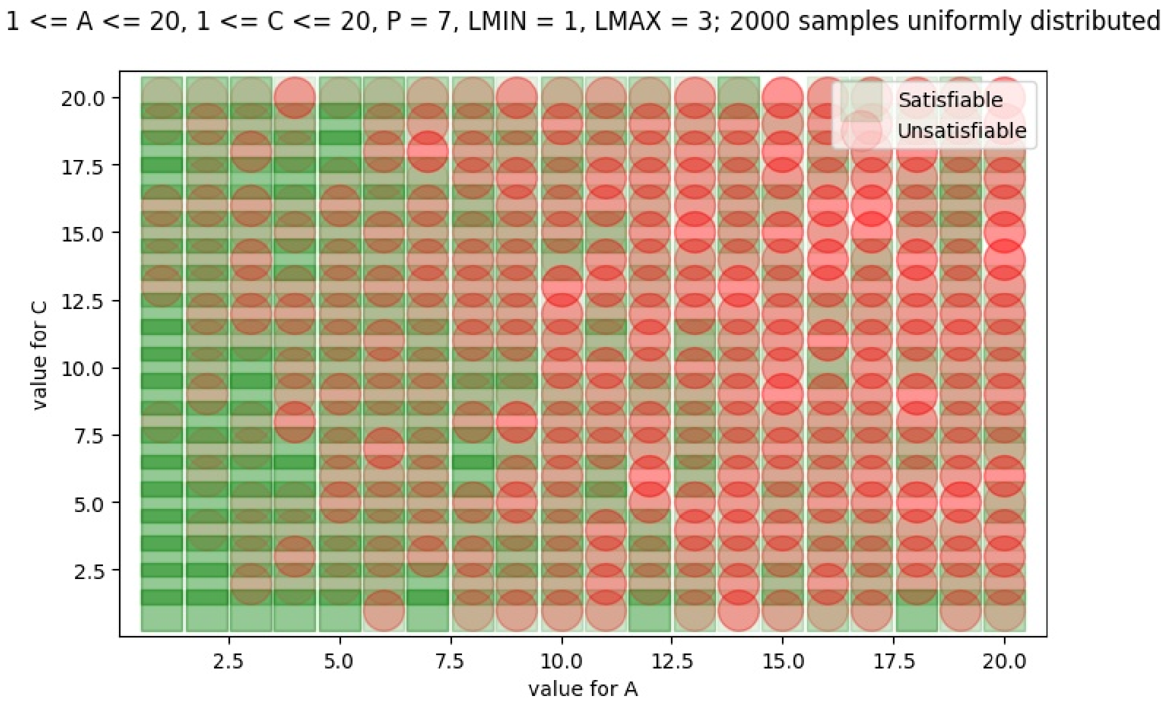

Figure 6: Satisfiability Distribution for fixed P; A and $C \in [1,20]$

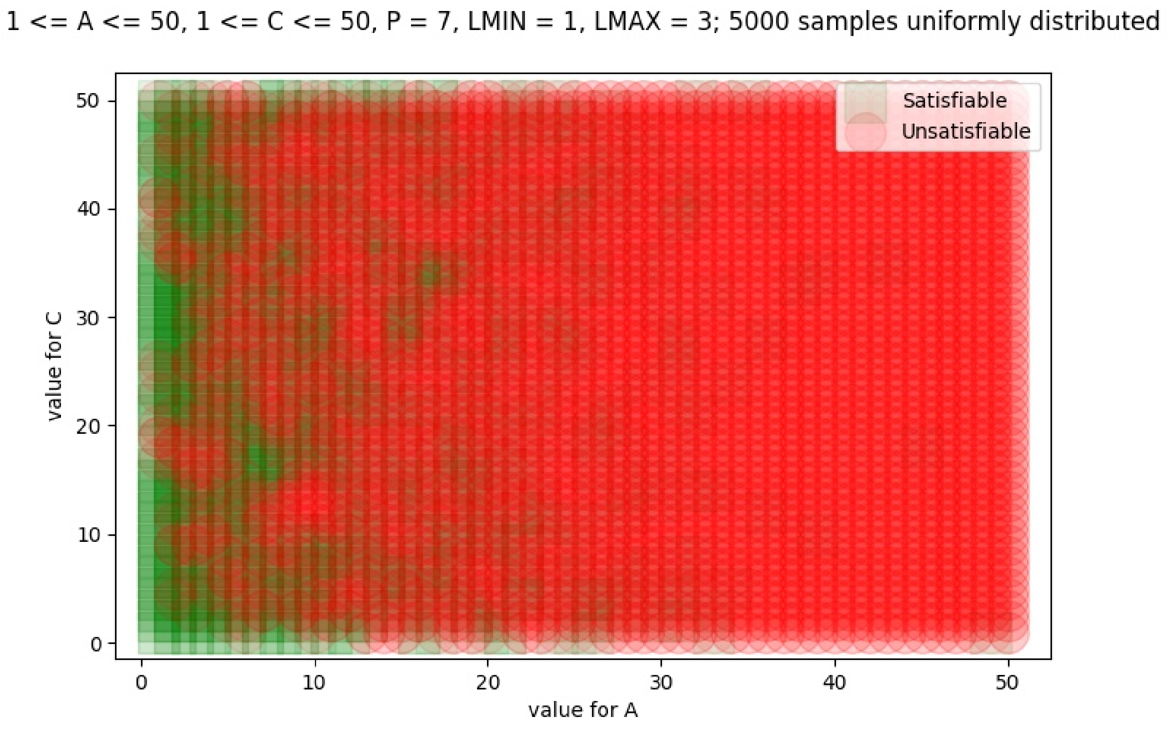

Figure 7: Satisfiability Distribution for fixed P; A and $C \in [1,50]$

All of the three charts are mutually consistent, suggesting that a lower value for A represents a higher probability of obtaining a satisfiable formula. Conversely, a higher value for A almost guarantees an unsatisfiable formula. However, a limitation is that these charts represent a fixed value of P and do not yield a clear idea about the probability of satisfiability. The next section is dedicated to the latter.

6.3 Probability of satisfiability

The purpose of the third experiment was to find the probability of satisfiability for various pairs of (P and C) when ranging the values of A. The next three charts present the findings.

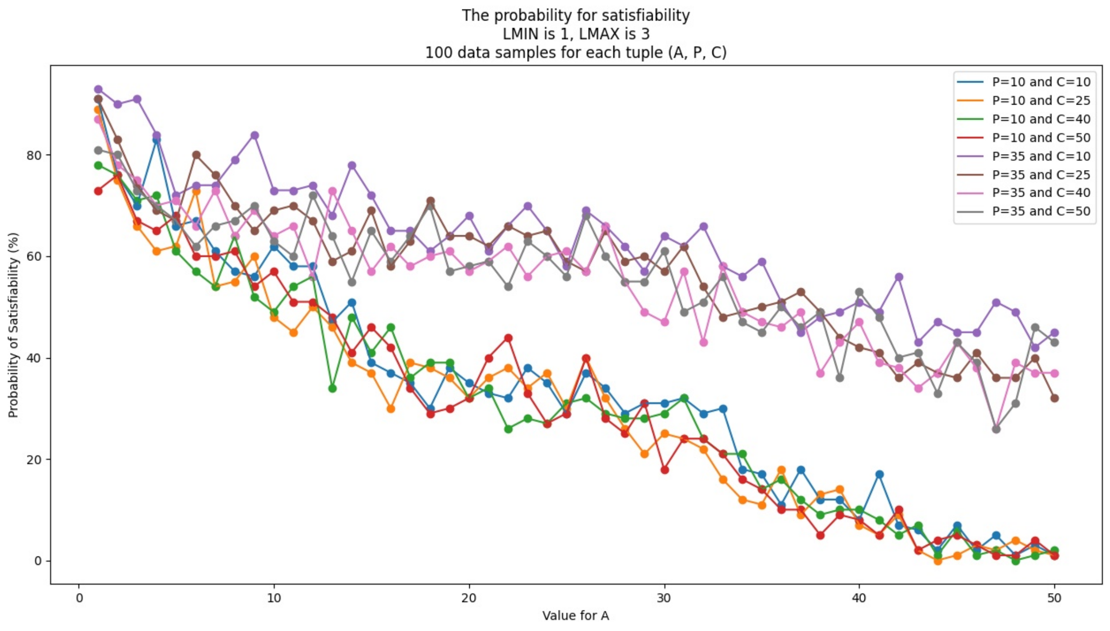

Figure 8: Probability of Satisfiability: first eight pairs A and C

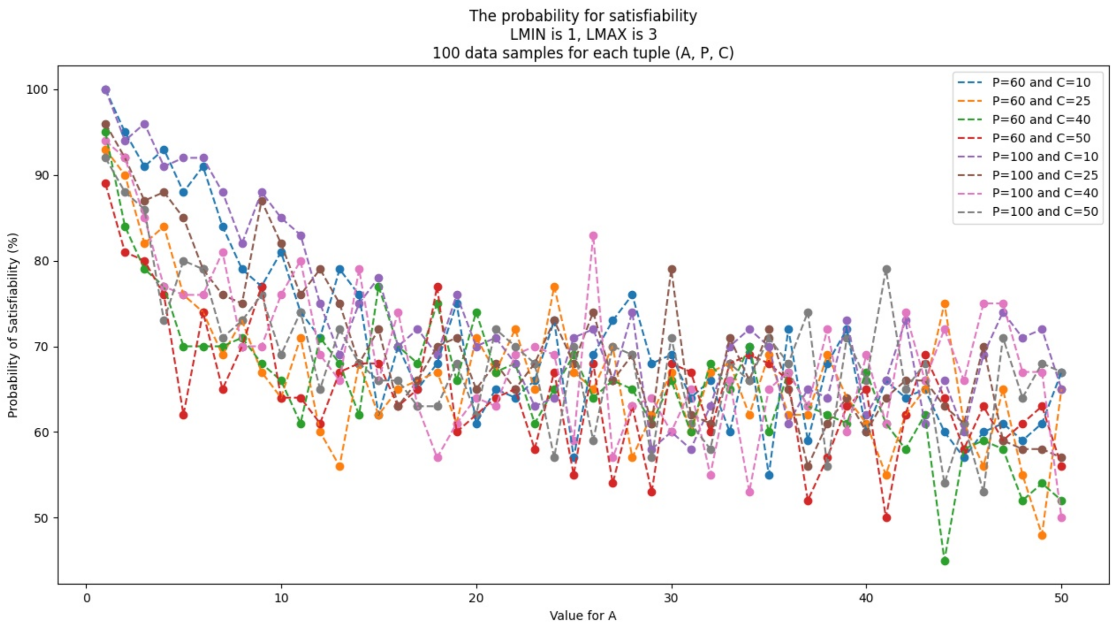

Figure 9: Probability of Satisfiability: last eight pairs A and C

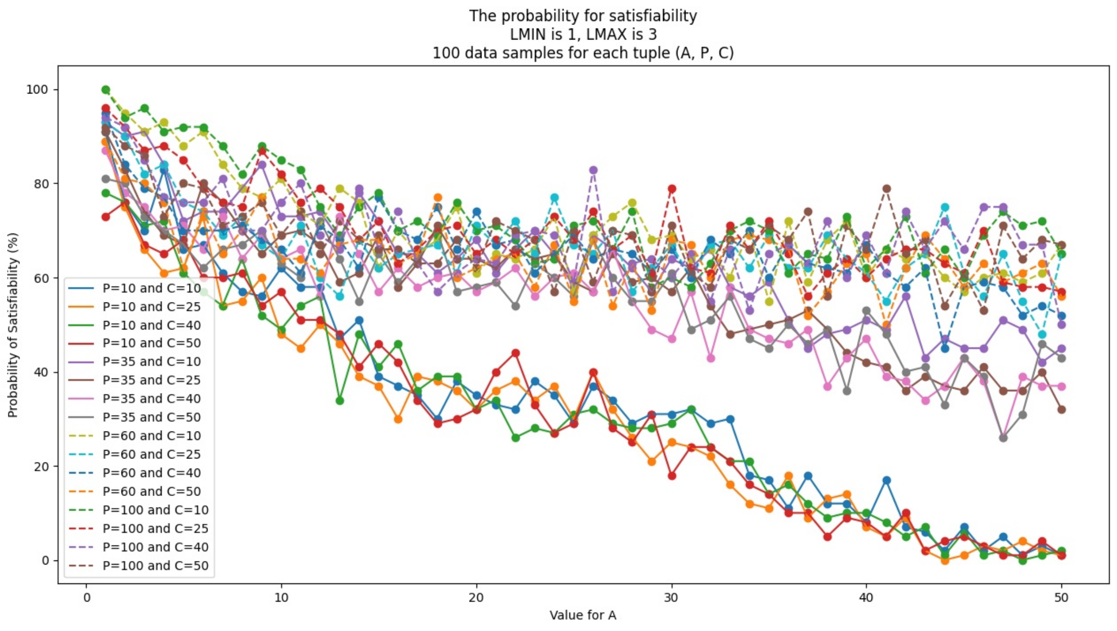

Figure 10: Probability of Satisfiability: all pairs A and C

The major findings of these charts are the following:

- the probability of satisfiability gradually decreases for larger values of A
- the value for C has a significantly lower impact on the probability value than the value of P
- the value of P and the probability of satisfiability are directly proportional

7 Conclusion

This work has successfully implemented the decision procedure in De Nivelle and Pratt-Hartmann (2001) for the two-variable fragment of first-order logic. Furthermore, it provides a reusable testing framework for formulas which are part of the two-variable fragment. In addition, statistics on randomly generated formulas are delivered.

7.1 Applications

All of this work represents a starting point for further improvements and re-implementation of the same decision procedure. Such improvements can be done and tested with confidence via the comprehensive testing benchmark produced as part of this project.

On top of that, the implementation contains many optimizations which may be further reused in similar projects. Such projects may expand outside of the scope of the two-variable fragment (examples include the optimization for the double implications and the optimization for the distributive).

Finally, the implementation of the work is encapsulated as part of a reusable C++ library. The current plans are to make the code available freely by publishing it as an open-source project after graduation. The main reason behind this decision is to support the research community.

7.2 Limitations and Further Research

The performance of the implemented theorem prover is still lagging behind state of the art solutions (like Vampire). That being said, the heuristics added have greatly improved the execution time and the quality of the results. It remains hopeful that with more time, other heuristics could further bridge the gap with such tools. For example, a finding of this project is that the tree-pruning used for backtracking did optimize the two-variable theorem prover. It is still possible to further optimize this by adapting more sophisticated solutions for tree-pruning and memoizations.

Finally, using fuzz-testing for detecting potential errors represents as well a reasonable approach towards obtaining a more robust implementation. Tools such as undefined behavior sanitizers - UBSAN (The Clang Team, 2021) and address sanitizers - ASAN (The Clang Team, 2021) may be further used as well, for detecting potential memory errors. All of these tools can be easily integrated with the testing framework implemented as part of this project.

References

[1] Mortimer, M. (1975). 'On languages with two variables', *Zeit. fur Math. Logik und Grund. der Math*, 27, pp.135-140. [Online]. Available at: `https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19750210118` (Accessed: 7 April 2021).

[2] 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.

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

[4] Vampire (1997). *Vampire.* Available at: `https://vprover.github.io` (Accessed: 31 March 2021).

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

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

[7] Gradel, E, Kolaitis, P.G. and Vardi, M.Y. (1997). 'On the Decision Problem for Two-Variable First-Order Logic', *The Bulletin of Symbolic Logic*, 3(1), pp.53-69. [Online]. Available at: `https://www.jstor.org/stable/421196` (Accessed: 7 April 2021)

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein C. (2009). *Introduction to Algorithms.* 3rd edn. The MIT Press.

[9] 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: 12 April 2021).

[10] Matuszek, D. (2002). *Backtracking.* Available at: `https://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html` (Accessed: 12 April 2021).

[11] CPP Reference (2021). *C++ Language: Templates.* Available at: `https://en.cppreference.com/w/cpp/language/templates` (Accessed: 20 April 2021).

[12] CPP Reference (2021). *C++ Utilities Library: std::variant.* Available at: `https://en.cppreference.com/w/cpp/utility/variant` (Accessed: 20 April 2021).

[13] CPP Reference (2021). *C++ Functions: Lambda expressions.* Available at: `https://en.cppreference.com/w/cpp/language/lambda` (Accessed: 20 April 2021).

[14] CPP Reference (2021). *C++ Utilities Library: std::execution.* Available at: `https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t` (Accessed: 20 April 2021).

[15] CPP Reference (2021). *C++ Classes: friend declaration.* Available at: `https://en.cppreference.com/w/cpp/language/friend`. (Accessed: 20 April 2021).

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

[17] Googletest (2021). *Googletest.* Available at: `https://github.com/google/googletest` (Accessed: 20 April 2021).

[18] CPP Reference (2021). *++ Utilities Library: std::unordered_set.* Available at: `https://en.cppreference.com/w/cpp/container/unordered_set` (Accessed: 20 April 2021).

[19] CPP Reference (2021). *C++ Containers Library: std::map.* Available at: `https://en.cppreference.com/w/cpp/container/map` (Accessed: 20 April 2021).

[20] CP - Algorithms (2021). *Heavy light decomposition*. Available at: https://cp-algorithms.com/graph/hld.html (Accessed: 20 April 2021)

[21] Otto, M. (2001). ‘Two Variable First-Order Logic Over Ordered Domains’, *The Journal of Symbolic Logic*, 66(2), pp. 685-702. [Online]. Available at: https://www.jstor.org/stable/2695037 (Accessed: 20 April 2021).

[22] The Clang Team (2021). *UndefinedBehaviorSanitizer*. Available at: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (Accessed: 23 April 2021).

[23] The Clang Team (2021). *Address Sanitizer*. Available at: https://clang.llvm.org/docs/AddressSanitizer.htm (Accessed: 23 April 2021).

Appendices

A Parse Tree Example

For the formula $\neg(\exists x \forall y (F(x) \iff F(y)))$ the following changes are produced in the parse tree.

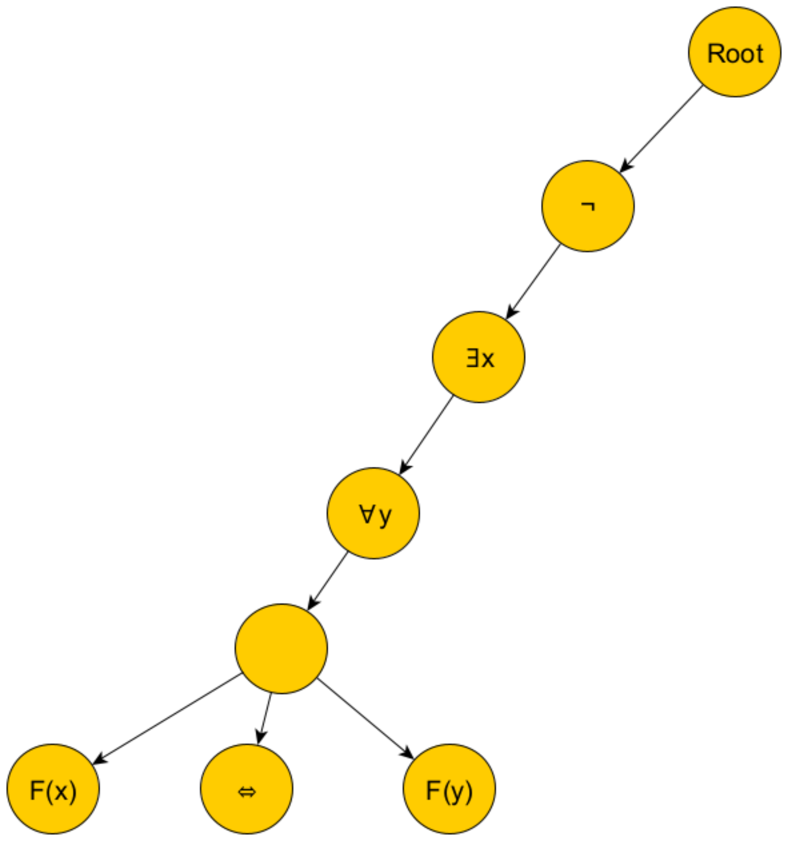

Figure 11: Initial state

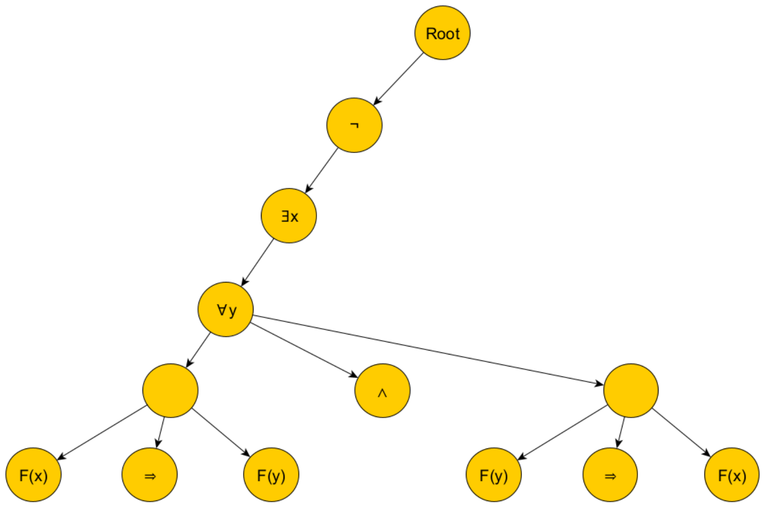

Figure 12: Double implication transforms in implication.

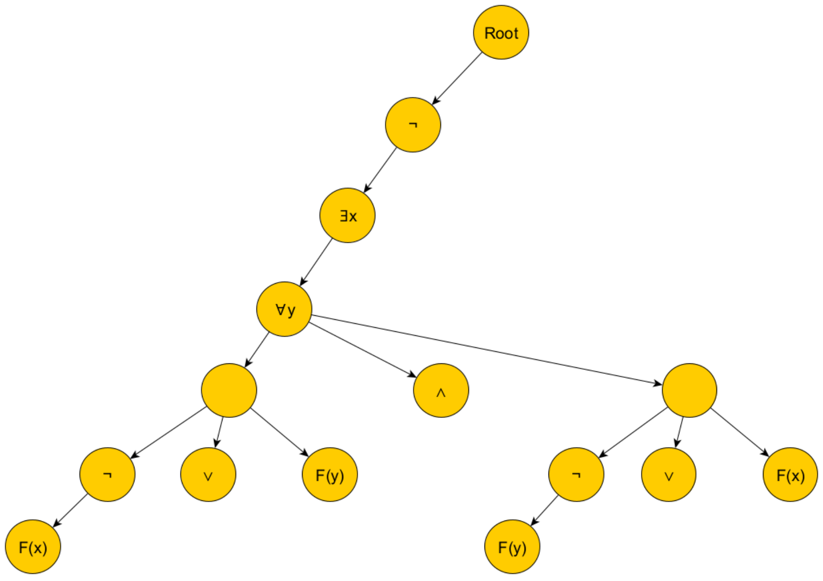

Figure 13: Implication transforms in disjunction.

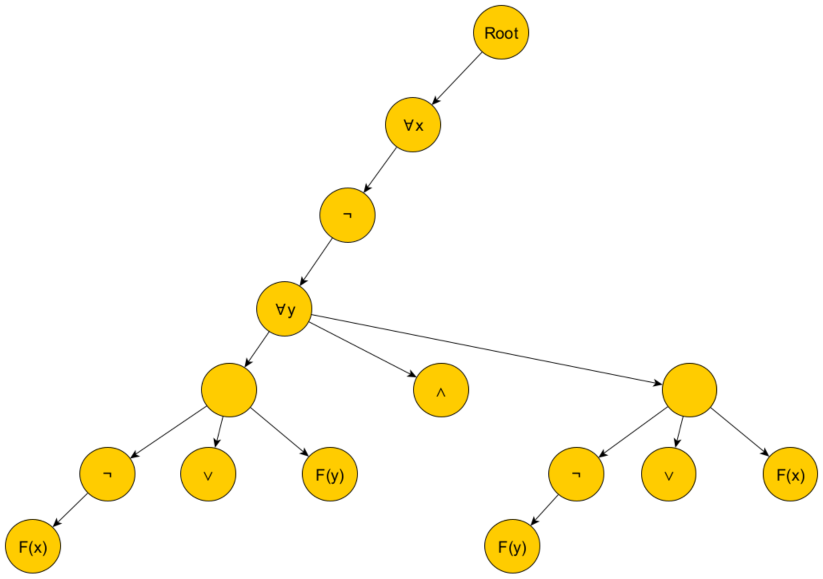

Figure 14: Negation is pushed past existential quantifier.

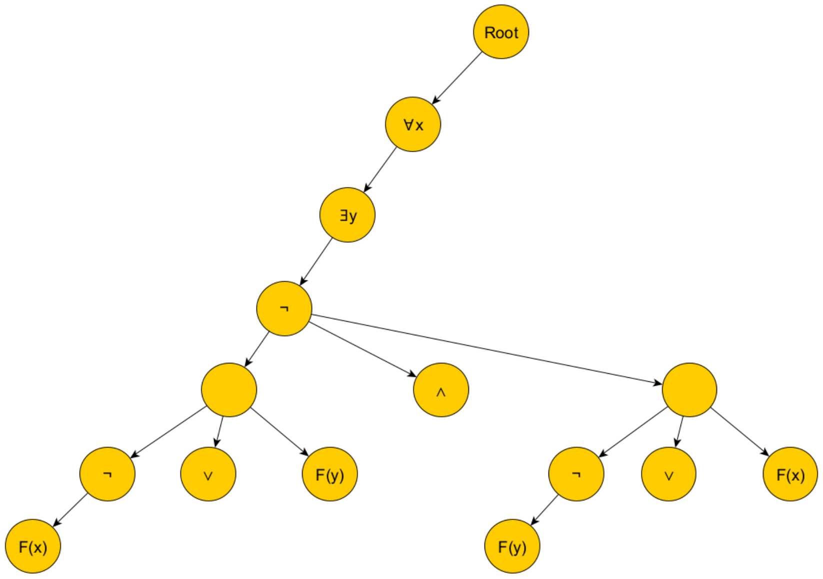

Figure 15: Negation is pushed past universal quantifier.

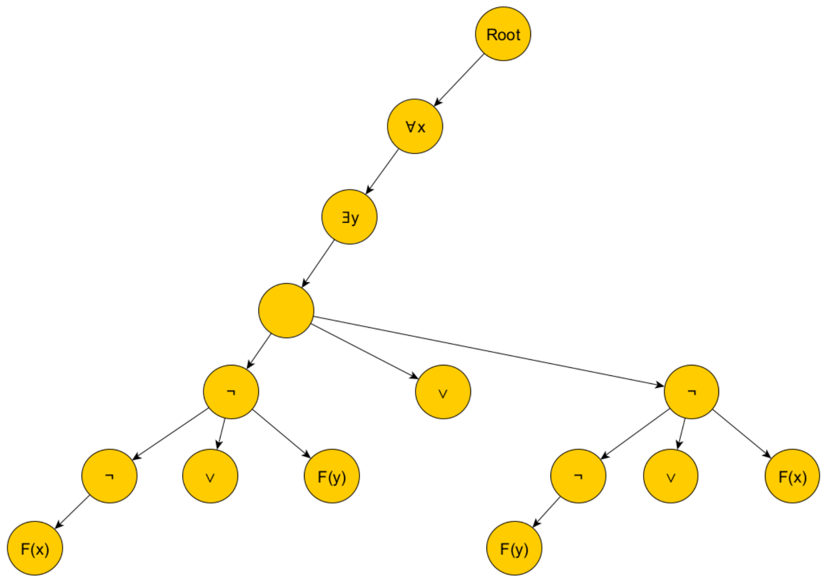

Figure 16: Negation is passed further and transforms conjunction in disjunction.

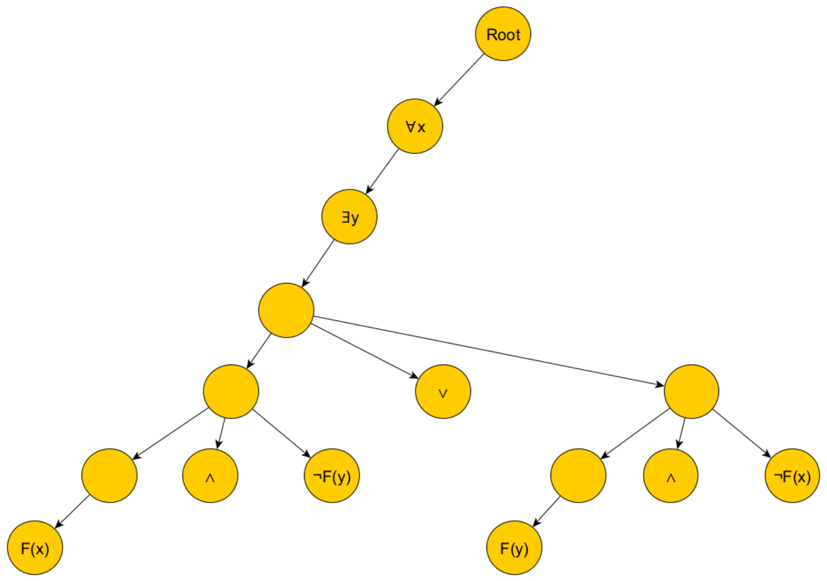

Figure 17: Negation is passed further, simplifying two consecutive negations to nothing and transforming disjunction in conjunction.

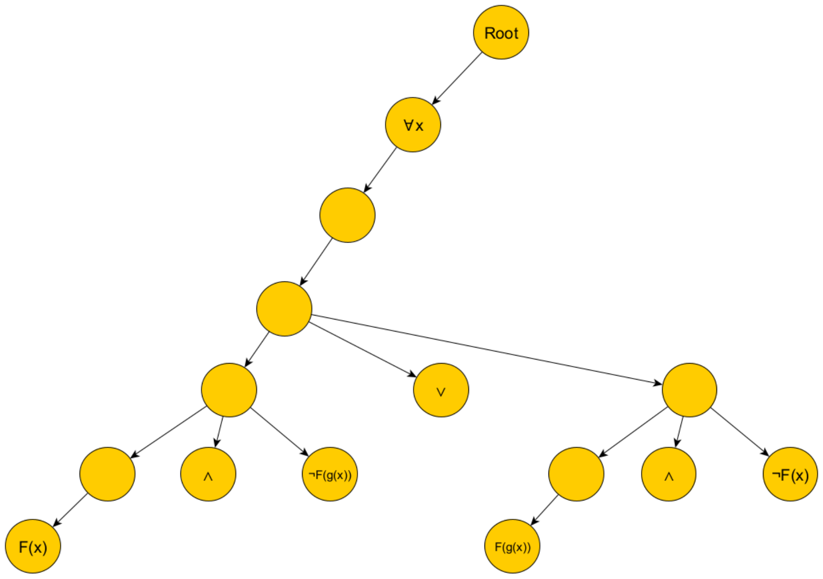

Figure 18: Skolemization is performed, by creating g, a brand new function symbol.

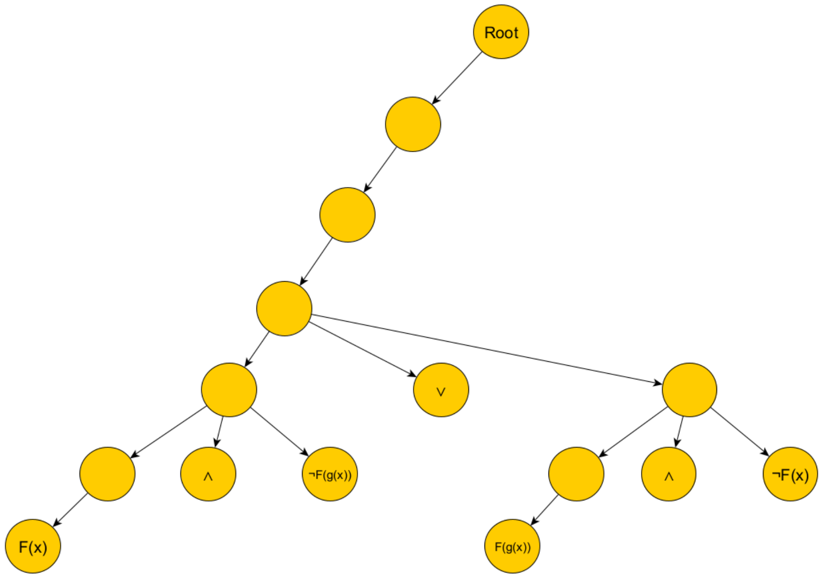

Figure 19: Universal quantifiers are removed.

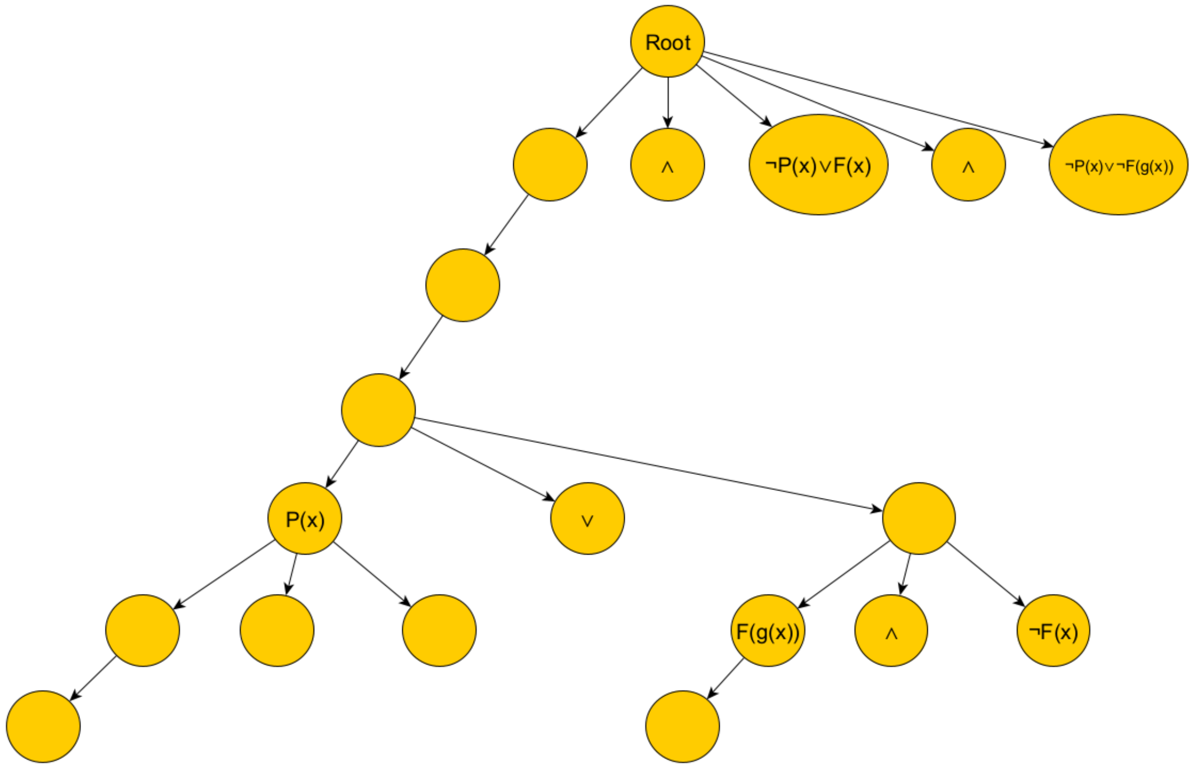

Figure 20: Left-hand side conjunction substituted by brand-new predicate P.

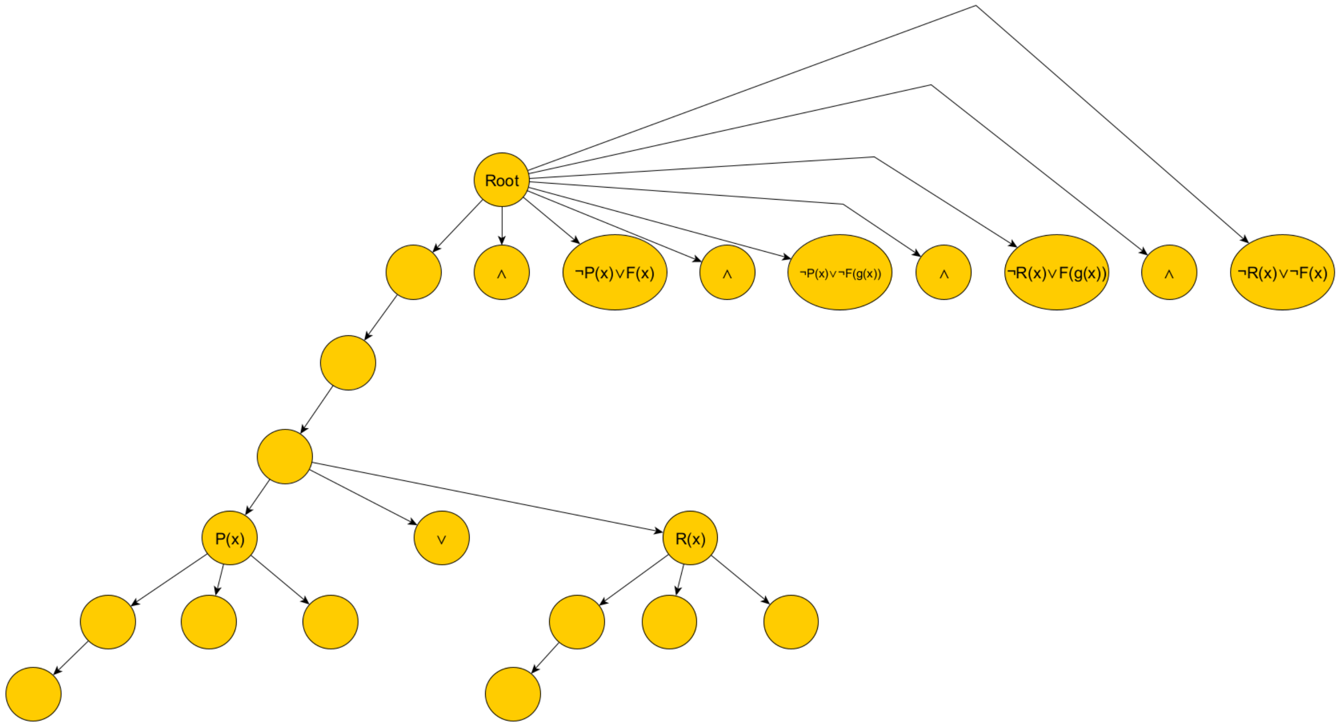

Figure 21: Right-hand side conjunction substituted by brand-new predicate R.

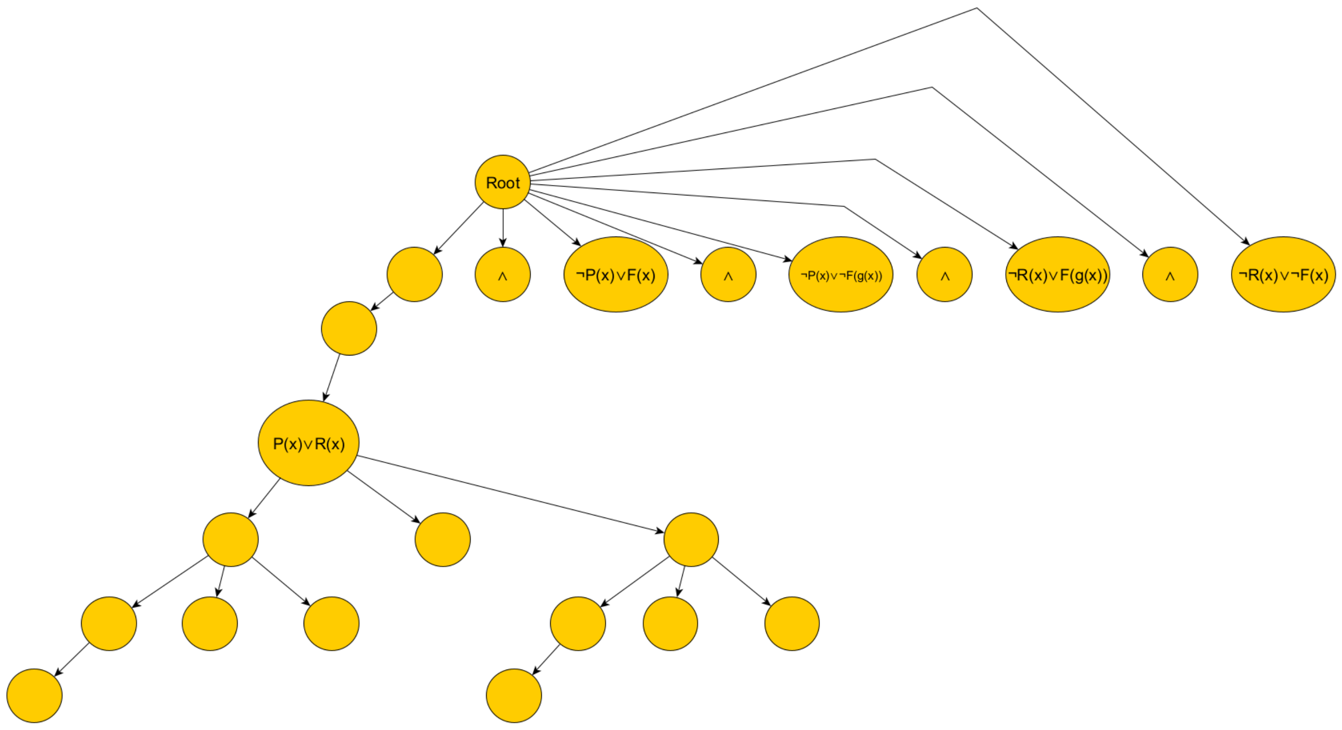

Figure 22: P and R merge in a single CNF.

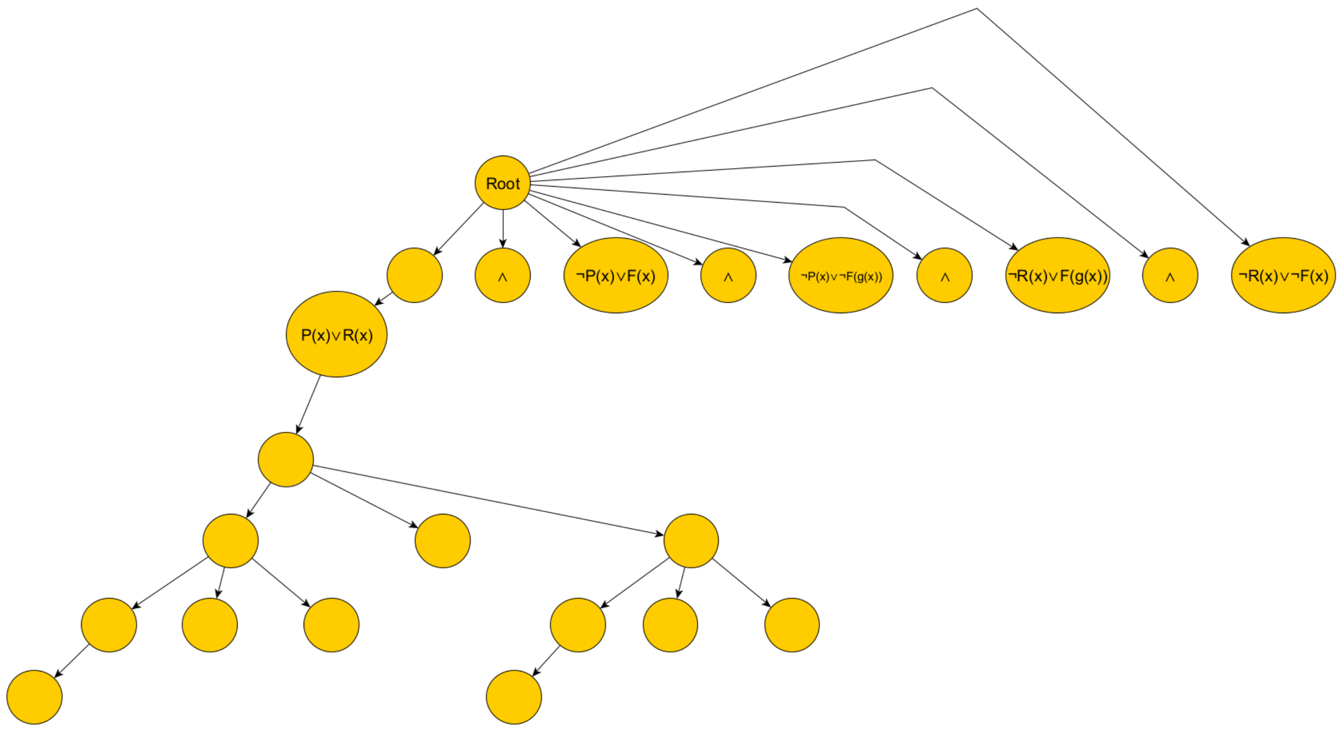

Figure 23: P and R are pushed up.

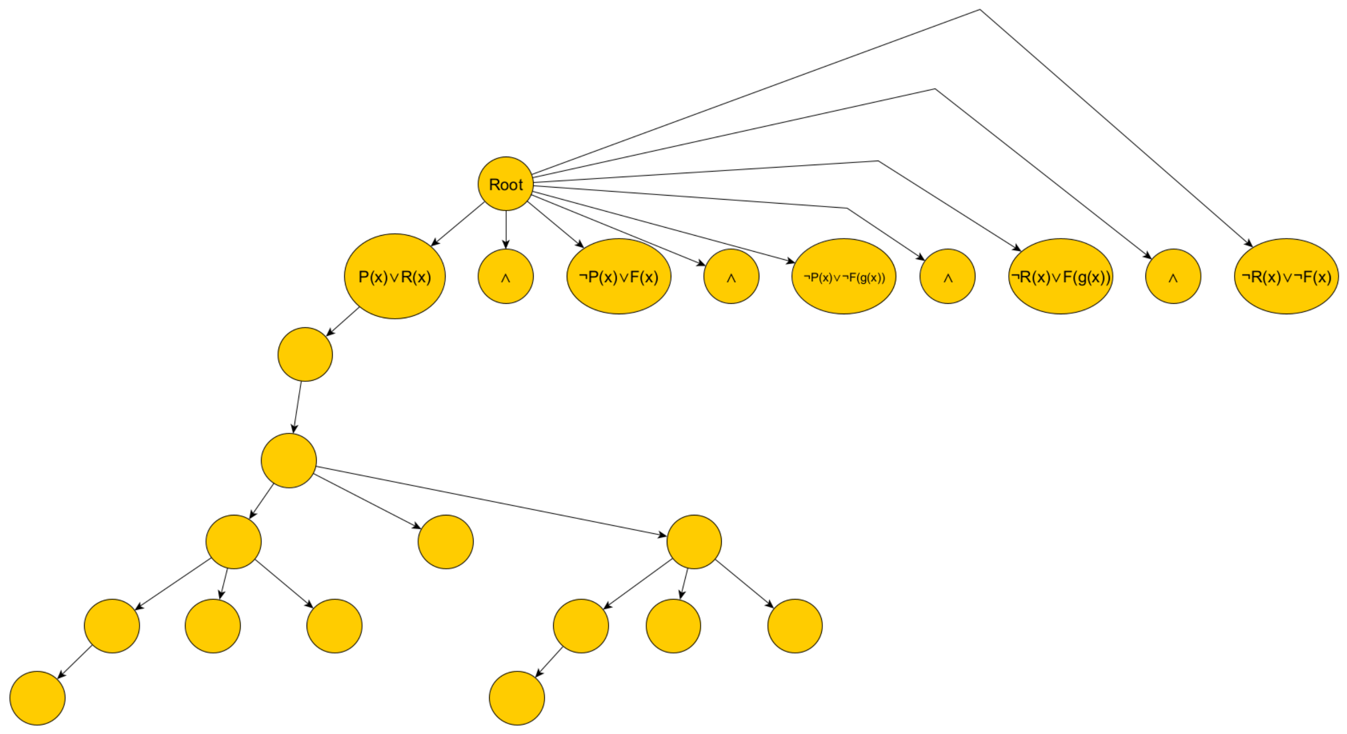

Figure 24: P and R are pushed up. The children of the root contain now the CNF. The next step will be to join on conjunction all of the clauses from the children, but this step will be skipped (because the size of the node for root will become too big to be drawn).