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

Author: Patrick Cătălin Alexandru Sava
Supervisor: Dr. Ian Pratt-Hartmann

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 specifically 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.

// {start} future ideas, do not review

- FOL → 2 variable fragment; use project outline on Manch website (http://studentnet.cs.manchester.ac.uk/ugt/year3/project/projectbookdetails.php?projectid=41364)
    - The *satisfiability problem for first-order logic*, sometimes known as the *Entscheidungsproblem*, is the following problem. Given a formula of first-order logic, determine whether or not it is satisfiable (i.e. represents a logically possible situation). The problem was first posed by D. Hilbert and W. Ackermann in 1928. It has been known since the work of A. Church and A. Turing in 1936 that this problem is undecidable. However, when the input formula is restricted to various fragments of first-order logic, the problem becomes decidable. One such fragment is the *two-variable fragment*, the fragment consisting of those formulas of first-order logic in which only the variables *x* and *y* appear.
    - There are various decision methods for the two-variable fragment, but the simplest is probably to use resolution theorem proving. The aim of this project is to implement a resolution-based theorem-prover for the two-variable fragment in Java, starting from scratch.

- no paper mention in abstract
- what you did in the report: implemented multiple theorem solvers (3), focusing 2-variable fragment
- what is the contribution → provide an implementation using a bunch of different heuristics tailored specifically for this objective, fragment, used literature + own implementation ; of the project ; what do you bring for further research etc currently there are no tools which implement exactly THIS thing

- compiled a comprehensive testing benchmark (some self made, other from the research community) for the 2 variable fragment case ; if someone wants to continue similar research → your work is a good framework
- no references to Vampire , tools etc
- different types of randomly generated formulas ; probability of being satisfiable / unsatisfiable
- generator gets multiple parameters and creates statistics which show the distribution of satisfiability and non-satisfiability when ranging those parameters
- we have studied the impact on satisfiability of different parameters in some randomly generated formulas

// {end} future ideas, do not review

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.

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.

Introduction

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.

Motivation [do not review, work in progress]

- why two variable fragment is interesting?
    - talk about decidability and soundness
    - emphasize on the contribution
        - no theorem solvers which try to tailor the implementation to this subset of first order logic
        - tried to provide such implementation an a good benchmark for measuring the performance

In 1975, Mortimer [reference] 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 [HDN and IPH] published in 2001 being 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 [HDN and IPH, 2001] and hence 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.

Project Aim

- mention something about the testing part/framework for two-variable fragment

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 Hans de Nivelle and Ian 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 [reference vampire]), 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 as well.

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 some papers, coding the two-variable theorem prover for the case with equality, optimizing, testing and debugging the code
    - The next three to four weeks were 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.

Methodology

- steps followed during the implementation process
- let the log book there
- followed software engineering principles

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

Apart from the weekly timetabled tutorials, the communication with my supervisor was going beyond that by actively exchanging emails during the week each time when an aspect of the project needed either further clarifications or refinement. My supervisor suggested keeping track of everything I am doing in a log book, which I found to be very useful, not only for writing this report, but also for organizing my thoughts and ideas better. Even though I was reluctant at the beginning, regularly updating the log book proved to be extremely useful as the project grew in size.

Report Structure

First, the report provides a background on first-order logic.The following section reiterates 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.

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.

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 we aim to define some of the most important concepts for this dissertation. The definitions employed are standard across the literature, for example one could look at [reference Leitsch].

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.

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 the first-order logic.

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

Operators

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

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

The operator and is referred to as conjunction and the operator or as disjunction.

Special Case: Equality

Until the presentation of the two-variable theorem prover algorithm, both the equality and inequality will be 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.

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 will not contain this symbol and it will be encountered only at an abstract level when the algorithm tackles equality.

Precedence

In terms of operator precedence, the conventions of Vampire [reference here] are adopted. This allows for the random generation of formulas that imply the same thing for 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 not and both the existential and universal quantifiers are applying strictly to the next:
    - predicate, if they precede one
    - subformula enclosed in a pair of round brackets, if they precede the corresponding open bracket
- the operator and has a strictly higher precedence than the operators corresponding to or, implication and double implication
    - analogously, the operator or 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.

Validity and Satisfiability

Definition 3. The *domain of discourse* is the set of entities over which the variables from a first-order logic formula are ranging.

Definition 4. 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 5. A formula is *valid* iff it is true in all of the possible models. Conversely, a formula is *invalid* if and only if it is false in all models.

Definition 6. 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, a formula is invalid iff it is unsatisfiable. Moreover a formula is satisfiable iff its negation is invalid. This interconnectivity between satisfiability and validity is well connected with the goal of the theorem proving, that is to determine whether a given formula is invalid, satisfiable or valid (theorem).

Decidability

A theorem prover would make sense to be produced only for sound proof systems. By soundness it is understood that only true things are inferred from ground facts. However, this is necessary but not sufficient in order to have the guarantee that the theorem prover terminates, or that it is able to infer all of the possible true things from the initial set of ground facts. For that to be the case, the proof system would have to be complete as well (all of the possible true things can be inferred from the initial set of axioms).

The satisfiability problem for the first-order logic is known to be undecidable. However, this is not the case for some fragments which are part of first-order logic (as the monadic one or the two-variable fragment). [reference godel]

The two-variable fragment is sound, complete and hence decidable. The result is due to Mortimer who showed in 1975 that the two-variable fragment has the finite model property, which further implies that it is decidable. [reference mortimer]

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 underlined. 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 (that is, the formula of 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 easily done 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 F.

As described in Leitsch (1997, p.12-18), F 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 C. Even though F may not have contained functions symbols, the resulting C may contain such and this is explained in the next subsections.

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 not further on conjunctions
- push the operator not further on disjunctions
- simplify two consecutive not operators to nothing
- push the operator not past universal quantifiers
- push the operator not past existential quantifiers

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

Skolemization

The name of this procedure comes after the Norwegian mathematician Thoralf Skolem [Leitsch]. 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 will be 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 easily seen that the quantifiers are part of redundancy now.

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

Let F'' be the resulting formula. Lastly, in order to transform F'' in C, the distributivity laws over conjunctions and disjunctions are performed. Considering the preparation for the next step, C is transformed into a set of disjunctions (or clauses) since it is uniquely determined which is the operator staying between its clauses (this is the operator and). Hence, the name of clause form is better justified now.

Even though the distributivity laws are explicitly used, the way they will be implemented is going to differ 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.

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 codebase of the project.

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 the same 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 7. 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 (meaning that it can be either itself or a function which has the variable nested in one of its arguments). A *valid substitution* is obviously a substitution which is not invalid.

Definition 8. 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 appliance 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.

Resolution rule

Notation 1. Let C be a clause and S a substitution. Then {C}S is the resulting clause by applying the substitution S to all of the literals in C. Similarly, 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\L is the resulting clause obtained by removing L from C exactly once.

Definition 9. 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, $C_1 \wedge C_2 \Rightarrow \{C_1 \backslash L_2 \wedge C_2 \backslash L_2\}S$. This transformation is referred to as the *resolution rule*.

$\{C_1 \backslash L_2 \wedge C_2 \backslash L_2\}S$ 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*.

Factoring rule

Definition 10. 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 occurences 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 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 support the design of the theorem prover for the two-variable fragment.

Refinements

Tautology Removal

Definition 11. Let C be a clause containing two literals L_1 and L_2 which are identical except their signs (which are opposite). This clause is obviously true in any model, so it is possible to remove it from the current set of clauses. This procedure is referred to as *tautology removal*.

Subsumption

Definition 12. 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 in time 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 will certainly be true.

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, yields 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.

Depth-ordered resolution

Definition 13. 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 14. 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 its calculation.

As described, the satisfiability problem for first-order logic is undecidable. From that 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(….)))))))).

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 be working 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).

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 will be assumed that a logic formula is given in two variables. The formula is going to be 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}.

Without equality

For the rest of this subsection, it will be 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. Having said that, the resolved atom will have exactly two different variables.
    a. If the empty clause is derived, this means that the formula is satisfiable.
    b. 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 the set of distinct variables it has, the maximum number of groups per clause would be three. This is because, in the worst case, the group corresponding to following will exist:

a. empty set (all of the literals in this group do not hold variables; they are ground instances as they are called)
b. the set {x}
c. 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 which does not have 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.
    a. If all of these runs derive the empty clause, the input formula is unsatisfiable.
    b. Otherwise, the depth-ordered theorem prover halted by saturation on at least one of these auxiliary sets. Hence, the input formula is satisfiable.

With equality

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

- the equality
    - dealing with the equality operator is trivial in this case, since the equality can be simply replaced operator with a special predicate of arity two called “Equality”
    - in case the equality operator occurs multiple times, all of these occurrences will be replaced by the same predicate called “Equality”
- the inequality
    - dealing with the inequality is regarded similarly to dealing with 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 will be mapped to the same predicate called “Inequality”
    - in order to preserve the logical equivalence, a new clause in one variable has to be added, containing the negated “Inequality” predicate to the CNF (i.e. ¬ 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 changes that need to be make:

- 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 it is 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 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:

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

The aforementioned formula is logically equivalent with the following:

$$\forall x T(x) \vee \forall y U(y) \vee (\exists! x \neg T(x) \wedge \forall x (T(x) \Leftrightarrow U(y)))$$

For the following formula

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

there exists as well the following logical equivalence

$$\exists x (T(x) \wedge \forall y (T(x) \Rightarrow x = y))$$

The problem which arises with the logical equivalence from above is that the algorithm will enter into an infinite loop, since each attempt of replacing the equality would result 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 it involves the introduction of a brand new constant for each clause containing equality. The construction presented in the paper takes into account unary and binary predicates.

However, the binary predicates can be safely omitted if the expansion of the uniqueness quantifier is performed at this step (which is after the disposal from Step 2 took place). The formula will have after step 2 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) are representing different predicates of arity one, even though they share the same predicate name.

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

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

and

$$T(e) \wedge \bigwedge_P (\forall x (T(x) \Rightarrow (P(x) \Leftrightarrow P(e))))$$

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

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

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

Design

The codebase has three main components: the front-end, the intermediate representation and the back-end. This separation came as inspiration from the Compilers literature, striving for consistency where possible.

The front-end is responsible for parsing the input, reporting any syntactic error. Furthermore, it also builds the parse tree of the input formula. It is important to note that the report would 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 of the operations done before the formula is in CNF are performed on this tree, which in turn is changed dynamically. In terms of algorithms, this component uses mostly ad-hoc algorithms (which will be further described) or depth-first search [reference Cormen]. This component is executing all of the 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, the reader will notice that many of the methods implemented for reducing the formula make assumptions on the structure of the input instead of enforcing that structure (this is because enforcing that 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 of the 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 [reference Cormen], depth-first search, backtracking [reference Cormen], 2-colorability [reference Cormen]). 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 happens 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 (~), 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 of the 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).

The diagram below gives an idea about how the three aforementioned components interact with each other. The names of the subcomponents are shown, in order to offer a hint to the next section.

{insert image here}

Implementation

This section explains fully the implementation details of the algorithms.

Environment and technology stack

Both Windows 10 Pro and Ubuntu 20.04 were used. Ubuntu was actually 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 generous memory resources were quite useful, since an unoptimized version of the theorem prover was using more than 18GB of RAM memory on some input formulas which were saturating. 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 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 are Python3.8 and Unix Bash. The former was utilised 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 strongly preferred due to the powerful set of options but also due to the fact that Google's C++ testing framework was available to build the easiest for CMake-based projects.

Front-end

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

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 further by a positive number of lowercase letters (forming the variable)

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

Parse Tree

The parse tree appears like a normal tree, with the consideration that some information may be stored in each node, which information in the code is referred to as an entity. A node may not store any entity as 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 ~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 moment because it will be analyzed it in the section dedicated to 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 for 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 sake of simplicity:

- an empty stack called *fatherChain* is declared
- the first node (which is the root) gets added to the parse tree
- the root gets pushed into *fatherChain*
- let *tokens* be the ordered list of tokens
- for each *token* in *tokens*
    - if the *token* is open bracket, an operator, a quantifier or a literal
        - let *newNode* be a new node added to the parseTree
        - add the *newNode* in the adjacency list of the node from the top of *fatherChain*
        - push *newNode* in the *fatherChain*
        - if the *token* is an operator, a quantifier or a literal, assign a new entity to the node *newNode*, encapsulating whichever type of entity it is
    - if the *token* is closed bracket
        - pop the top from *fatherChain*

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

Intermediate Representation

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

The approach for disambiguating the given formulas

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 bounded by different quantifiers. This may be challenging later on unless it is addressed now. Hence, a depth-first search maintaining an accumulator with all of the variables which have been seen so far is run. Note that the accumulator never evicts elements (its size only 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 occurence 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 stays the same).

In order to implement this part efficiently, the accumulator was modelled as an std::unordered_set (CPP, 2021), in order to be able to insert and search in constant time.

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

Basic reduction

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

The approach for resolving the precedence

Another 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 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 dealt with by adding new nodes in the parse tree which is equivalent with 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 the only operator occurring in the multiset of their entities is the operator and ,need to be obtained. For each group, the algorithm will add all of the nodes belonging to it in the adjacency list of a brand-new node, by preserving the relative order

between them. Finally, each group of nodes in the original adjacency list is replaced by the brand-new node corresponding to the respective group.

Following this, the parse tree has the precedence for the operator and resolved. Then, the algorithm proceeds with the operator or. The procedure is identical, since the operator and is not interfered with at all. The case for implication and double implication are resolved analogously.

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

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

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 splitted 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 (i.e. $P(x) \Leftrightarrow (R(x) \Leftrightarrow Q(x))$ will further transform in $P(x) \Leftrightarrow ((R(x) \Rightarrow Q(x)) \wedge (Q(x) \Rightarrow R(x)))$, finally becoming $(P(x) \Rightarrow ((R(x) \Rightarrow Q(x)) \wedge (Q(x) \Rightarrow R(x)))) \wedge ((R(x) \Rightarrow Q(x)) \wedge (Q(x) \Rightarrow R(x))) \Rightarrow P(x))$; notably, the initial formula has one occurence of Q, the next one has two while the final one has four).

In order to prevent this, the deepest occurence 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. Repeating this procedure until there is no double implication operator nested in a subformula which is further a part of a double implication will finally produce a logical equivalent formula. It is safe to apply the expansion described in the first paragraph on the resulting formula without the risk of an exponential explosion (see complexity analysis).

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

- all the nodes are visited using a depth-first search
- whenever an occurrence of the double implication is detected, it is checked whether it is the deepest in the current subtree and if it is, proceed as follows:
    - 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

    - two brand-new nodes will be created in parse tree, both holding the same brand-new predicate as an entity; these two nodes will be referred to as T and P
    - the three nodes forming the initial double implication will be now replaced with T
    - finally 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 appendance of the conjunction needed for preserving the logical equivalence
        - some details which deal with the precedence were omitted for simplicity

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

- all nodes are visited using a depth-first search
- whenever an occurence of the double implication is detected, containing the nodes L, D, R, the following is performed:
    - create a deep copy of node L in node L'
    - create a deep copy of node R in node R'
    - create a brand-new node I holding an implication as an entity
    - change the entity held on D to an implication
    - 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
        - some details which take care of the precedence were omitted for the sake of simplicity

Because 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. In fact, if N is the number of nodes in the parse tree, it would be expected to quadruple in the worst case. In other words, it will double for performing the optimization for replacing the double implications with brand-new predicates and it will double further once more for performing the expansion.

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

The approach for eliminating implications

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

The algorithm is therefore straightforward:

- all the nodes using depth-first search are visited

- whenever the algorithm finds for a node Q three consecutive children matching the pattern left-hand side (referred as L), implication (referred as I), right-hand side (referred as R) proceed as follows:
    - create a brand-new node K, holding the operator not as an entity
    - add L in the adjacency list of K (as being the only direct son)
    - replace the occurrence of L in the adjacency list of Q with K
    - change the operator node I is holding to disjunction
    - leave R unchanged

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

The approach for pushing the operator not

The only outstanding 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 done as well.

Due to the convenient resulting simplified language of the parse tree, implementing the algorithm is performed:

- all nodes are visited using depth-first search
- whenever the algorithm located in a node of which assigned entity is the operator not it iterates through its children and do the following depending on the entity held by the child:
    - no entity transforms in the operator not
    - operator
        - not, transforms in no entity
        - and, transforms in or
        - or, transforms in and
    - quantifier
        - existential transforms into universal and moreover, an intermediary brand-new node between the child and its subtree is created, holding the operator not
        - universal transforms into existential and moreover, an intermediary brand-new node between the child and its subtree is created, holding the operator not
    - literal
        - the sign gets changed and the operator not will not go past it
    - normal form
        - nothing, because by this point the parse tree does not have any node holding this sort 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 O(N), where N is the current number of nodes from the parse tree.

Skolemization

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

- 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; moreover, a std::map [reference https://en.cppreference.com/w/cpp/container/map] is maintained (called M) containing the mapping for the existentially quantified variables to their corresponding functions (and their respective arguments)
- in case the current node Q holds an universally quantified variable V as entity, V will be pushed to the stack; V will be further popped off the stack after the traversal of all the subtrees rooted in the children of Q has been fully performed
- in case the current node holds as an entity an existentially quantified variable K, a brand-new function symbol F is created and 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)
- in case the current node holds a literal as an entity, the algorithm iterates through the terms of it and for each variable which has entry in the map 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 kept 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 will be 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 $O(N^2 \log_2 N)$.

Simplified Clausal Normal Form

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

It has to be noted that the simplified version of the clausal normal form will be attempted. That is, the CNF is retrieved in a convenient format for now, but the way of representing it will be changed after the retrieval (this will be detailed in the section dedicated to the back-end). Hence, the CNF will be modelled for now as an array of arrays (each containing

literals). This representation is completely accurate, assuming that between the inner arrays there are conjunctions and between the elements of an inner array disjunctions are laying.

The algorithm which will be presented as part of this section might look at a glance very complex. One would be easily able to come up with a simpler implementation. However, the reason behind the design decision (of choosing the following approach) is the avoidance of an exponential explosion while merging the CNFs. If this would be done naively, by doing the Cartesian product of two CNFs, the size of the resulting CNF will evidently face 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):

- declare toBeAdded globally, as being a set of disjunctions which need to be appended to the formula as conjunctions
- let A and B be two CNFs, each stored as discussed, as an array of arrays containing literals
- if the operator between A and B is a conjunction, create C as a concatenation of A and B and return C
- otherwise the operator is a disjunction and there are three cases:
    - if either of A or B are empty, return the another one
    - the trivial case, both the sizes of A and B are 1
        - create C containing a single element, the concatenation of the first (and only) element of A and the first (and only) element of B
        - return C
    - otherwise, this means that either A or B violate the condition from above
        - therefore, apply the following algorithm on each of them <u>twice</u>:
            - let be D the input of the algorithm, passed by reference (i.e. the modifications on it are visible in the calee)
            - if <u>all of the elements</u> from D are having size 1, then:
                - create a brand-new predicate F
                - for each element P of D add ($\neg F \vee P$) in toBeAdded (note that an implication is added)
                - D becomes [[F]], where a pair of square brackets represent an array
            - otherwise, perform the following:
                - for each element P of D
                    - create a brand-new predicate F_i
                    - add ($\neg F_i \vee P_1 \vee P_2 \dots \vee P_{j-1} \vee P_j$) in toBeAdded, where j is the size of P (note that an implication is added)
                    - P becomes F
            - after the algorithm has been applied twice, call once more 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 will be described later, after the completion 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 presented:

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

As described before, 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 with the size of CNF) bounded amount of additional conjunctions. In the worst case, the time complexity of this part will be O(N), where N is the number of nodes from the parse tree.

The depth-first search algorithm runs in polynomial time, precisely $O(N^2)$. Even though much more sophisticated solutions might have been implemented (such as one involving the small-to-large trick, optimizing the time complexity to $O(N \log_2 N)$ overall), it was considered that it is not necessary, since that solution represents only a micro-optimization (being dominated by $O(N^2 \log_2 N)$, from the Skolemization step).

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

An example of all operations described in this section is provided in the Appendix 1.

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 expressed in detail.

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 seeing it as an array of arrays, as used and described in the intermediate representation, a much more convenient way of storing (in respect with 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 will store 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. The figure below will help the reader in gaining a better understanding.

{insert image here}

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

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:

- if A and B are identical, then return true
- otherwise, check the following:
    - if A is a constant, return false
    - if A is a variable, called V
        - if B contains that variable, return false
        - otherwise return a tuple containing V and the whole term of B, signifying that V is mapped to B (which can be a function, a constant or another variable)
    - if A is a function symbol, then
        - if B is not a function symbol, return false
        - otherwise, do the following:
            - if A and B do not have the same function name, return false
            - otherwise, for each position in the list of arguments of A and B, try to unify their corresponding terms (i.e. for f(x, y, z) and f(a, b, c) the algorithm will try first x with a, then if successful y with b and finally if successful z with c)
                - if the return value R for the current unification is true, continue to the next pair of terms (because this means that the pair of terms the algorithm tried to unify was identical)

    - otherwise, return R
- finally, if nothing was returned, 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 → 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, …, t_n); therefore, $treeSize(K) = \sum_{i=1}^{n} treeSize(t_i)$

Having defined the treeSize, the time complexity for findPartialSubstitution is now established: $O(M^2)$ where M is the minimum between treeSize(A) and treeSize(B). This is owing to the fact that at each call of the function the verification for equality of A and B traverses the tree of each of them and for highly nested A and B, the algorithm performs this check as many times as there are nodes in the tree of term with the lowest number of nodes.

The second tool which the report presents is a function called “augmentUnification”, which takes as an input two terms, A and B. This function is simpler than the previous one, representing just a “wrapper” for findPartialSubstitution. Its purpose is to call findPartialSubstitution for A and B. In case of failure, it 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). If the reader is familiar with the Python programming language, 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 means that the operations have to be rolled back --- this will be explained later). 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 is trivial and does not contain anything sophisticated. This function is required in the next subsection. The time complexity for it is O(R), where R is the value of the treeSize for the input term.

Literal Unification

In the previous section 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 it is referred to as “augmentUnificationLiteral” for clarity.

This function does the following:

- it compares the literals predicate-wise
    - if the predicates are different, returns false
    - otherwise, it calls augmentUnificationTerm for each argument of the predicates
        - if the algorithm is located at a pair of terms and augmentUnificationTerm returned false for them, return false
        - otherwise, if augmentUnificationTerm returned true, move to the next pair of terms
        - otherwise, if augmentUnificationTerm returned a substitution, return the respective substitution
    - finally, if the algorithm has not returned yet, return true

The time complexity of this function is 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 with 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 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.

Clause Unification

This functionality was implemented using C++ templates [reference https://en.cppreference.com/w/cpp/language/templates] and it represents the core of theorem proving. It is also important to note that this part makes use of most modern C++, std::variant (CPP Reference, 2021) and lambda expressions [reference https://en.cppreference.com/w/cpp/language/lambda] which are heavily used.

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 might be omitted, for simplicity. The algorithm does the following:

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

Note that the application of substitution represents only a mapping from one term to another one and it can be implemented in $O(\sum_{i=1}^{N} R_i)$ 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 done 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 will be performed on only one of the clauses F' or S'. The complexity of this step is $O(\sum_{i=1}^{N} R_i + \sum_{i=1}^{N} Q_i + C \times max(\sum_{i=1}^{N} R_i, \sum_{i=1}^{N} Q_i))$, 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 of choosing the clause which will be substituted when assigning new variables for those which are part of the intersection).

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

- |F| and |S| are the size of F and S, respectively

- LC is the number of operations literalCheck is performing 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 is performing for a literal and a clause in the worst case (as for the literalCheck function, this can be only determined at the runtime, since the function is passed as a parameter of attemptToUnify)
- the last part of the time complexity analysis is the same as for the variable renaming step, with the consideration that in the worst case, it has to be renamed in addition the maximum between the cardinals of $V_{F'}$ and $V_{S'}$

Basic Theorem Prover

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

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

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, 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 seen 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. ~P(x, f(x), y, x, g(y)) becomes ~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.

Factoring Rule

This algorithm implements the factoring rule described in the theoretical part of this 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):

- let C be the input clause
- if C is marked as deleted, then return
- let H be the current hash of C

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

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

The procedure isTautology takes as input a clause and searches for two atoms L and L' such that they are identical apart from the fact that they have opposite signs. If it finds such a pair, it 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 point to the same thing
- the step in which the intersection of $V_{F'}$ and $V_{S'}$ is enforced to be empty is be 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

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.

The algorithm is as follows:

- let C be the input clause
- if C is marked as deleted, return

- let CHS be the ordered set of the hashes for its literals (i.e. for the clause (~P(x) ∨ L(x, y) ∨ R(z, y, x)), the corresponding the set is {L(v_1, v_2), ~P(v_1), R(v_3, v_2, v_1)})
- iterate through all of the other clauses which have not been deleted
    - for each clause E
        - compute EHS
        - if EHS is included in CHS, then C is subsumed by E
            - C gets marked as deleted and the algorithm remembers the timestamp at which it got marked as deleted
        - if CHS is included in EHS, then E is subsumed by C
            - E gets marked as deleted and the algorithm remembers 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 might not match the relabelling from another clause. However, a correct implementation for non-ground clauses might be performed in exponential time, which outweighs the benefits.

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 with the subsumption and factoring rules, this rule is implemented as well using multi-threading.

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

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

        - insert H to HSet
        - if J is the empty clause, then
            - return true
        - append J to AllClauses
        - set the variable repeat to true
- finally, if this point is reached, return false (saturated was reached)

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

Depth-Ordered Theorem Prover

The depth-ordered theorem prover adds heuristics for the term depth on top of the basic theorem prover described in the previous section. 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 makes use of a function called isAOrdering, which is detailed later.

Hence, the resolventCheck function does the following:

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

The function isAOrdering is defined as follows:

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

The changes mentioned above are the only ones needed to transform the basic theorem prover into a depth-ordered theorem prover. The time complexity 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 $O(\sum_{i=1}^{|C|} A(L, LC_i))$ 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 $O(\sum_{i=1}^{N} L_i + \sum_{i=1}^{N} W_i)$, 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.

Two-Variable Theorem Prover

This section presents the two-variable fragment theorem prover. The explanation starts 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.

Variable renaming

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

- let C be the set of clauses
- let G be an empty graph, in which nodes and edges are dynamically added
- for each clause Q in C
    - let LVMax be the maximum number of distinct variables across all of the literals of the clause Q
    - if LVMax is greater than 2, then return false (i.e. the given formula is not part of the two-variable fragment)
    - otherwise:
        - for each literal L in Q
            - let V be the set of variables occurring in L
            - if V has size 2:
                - draw an edge from the node corresponding to the first variable to the node corresponding to the second variable in G
            - if V has size 1:
                - add a node corresponding to the variable from V in G

- finally, return true

Finally, 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 polynomial time (in respect with 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.

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:

- let F and S be the input literals
- if F and S do not have opposite sign, return false
- if F and S do not have the same predicate names, return false
- let V_F and V_S be the variables contained in F and S, respectively
- if either of $|V_F|$ and $|V_S|$ have the size 2 and none of F and S represent the equality (i.e. their predicate names is not called Equality), then return true
- otherwise 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 next section.

The implementation of the resolventCheck is the same as the one for the depth-ordered theorem prover. It is the same as for the basic theorem prover, but that does not guarantee termination.

The time complexity for the literalCheck function is $O(F^p + S^p + \sum_{i=1}^{N} FT_i + \sum_{i=1}^{N} ST_i)$, 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 will be 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:

- it receives as an input the set of clauses C (which is 0-indexed), the index current clause which is processed R (initially 0) and the depth-order theorem prover D (which is passed by reference)
- if R is equal with the size of C
    - run D (which holds all of the clauses added so far)
        - if it returned true (which means that saturation was reached), return true
        - otherwise, return false
- otherwise, group the literals from the R^{th} clause on the set of variables occurring in that respective literal
    - for each such set S,
        - create a new clause W containing all of the literals from S
        - append W to D, and remember the timestamp T at which W was added
        - run backtracking recursively on the tuple (C, R + 1, D)
            - if it returned true, return true
        - revert D to the timestamp T (that will not only remove W from D, but also all of the clauses inferred past W)
- finally, if this point is reached, return false

Since the set of the variables is either the empty set, {x} or {y}, the time complexity of the algorithm is $O(3^{|C|} * DTP)$, 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.

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 is surprising and 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 [reference HDNIPH]. 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 [reference HDNIPH] or with what was presented earlier in the theoretical part of this report, it is omitted for the sake of simplicity.

Evaluation

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

Unit testing

Google's testing framework was used in order to unit-test some functionality 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 the implementation of each component in part. If the former fails, the latter might not fail whereas if the former passes, the latter will pass with a very high chance.

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.

Using the problems found on tptp.org

The first method of achieving this kind of testing was to use 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 nor 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" (which possibly comes from "syntax") problem set.

Even though 47 test-cases is not a big 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).

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 of whose calls were randomly weighted. Even though the generated formula was a valid first-order logic formula, figuring out 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 might 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 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 to address this as well. Approaches tackling all of the issues above,

excluding the last one, exist. However, the last problem is probably too complex to be solved in polynomial time (because keeping track of the number of variables and the potential arising function symbols, while generating a totally unrestricted first-order logic formula, is infeasible).

Since inspecting how to couple the above-mentioned issues always lead to new challenges, 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.

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 [reference to M.Otto] 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
    - in other words, there is a global set Q (of cardinality |Q|) of predicate names which includes all of the names from α and β_i from which the predicate names are be picked
- each of the A clauses of α 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 now a formula in the two-variable fragment becomes trivial:

- A, C, LMIN, LMAX get set with a value
- let L be a number randomly pick from the range [LMIN, LMAX]
- let P be the desired value for |Q|
    - P gets set

        - Q gets populated with P unique predicate names (containing letters of the latin alphabet), in increasing order of their lengths (i.e. first all of the predicate names of size 1, then all of the predicate names of size 2 and so on)
- let F be a string initialized with "∀x∀y("
- let alpha be an empty set of strings
- repeat A times the following
    - let W be a list of literals
    - repeat L times the following
        - let R be an empty string
        - pick a boolean B at random
            - if true, append negation to R
            - otherwise let R unchanged
        - pick a predicate name U at random from Q
        - append U to R
        - pick a boolean B' at random
            - if true, append "(x, y)" to R
            - otherwise append "(y, x)" to R
        - append R to W
    - join all the elements of W on disjunction (i.e. {"¬Pa(x, y)", "Pb(y, x)"} becomes ¬Pa(x, y) ∨ Pb(y, x)) and append the result to alpha
- join all the elements of W on conjunction and append the result to F
- append "∨ x = y) ∧" to F
- let beta be an empty string
- repeat C times the following
    - let $beta_i$ be string initialized with "(∀x∃y(
    - pick a predicate name K at random from Q
    - pick a boolean B'' at random
        - if true, append "(x, y)" to $beta_i$
        - otherwise append "(y, x)" to $beta_i$
    - append "∧ x≠y)) ∧" to $beta_i$
    - append $beta_i$ to beta
- remove the last character from beta
- append beta to F
- return F

The algorithm from above runs in polynomial time in respect with 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) = 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.

Stress testing

In order to optimize the theorem prover, the decision of making its core infrastructure (involving the factoring, resolution and subsumption rules) multi-threaded was made. A challenge of this design aspect was related to testing, precisely how to make sure 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 47 tests in part. None of them failed, all of them were consistent with the expected output.

The value this kind of testing brought to the project was solid, 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).

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.

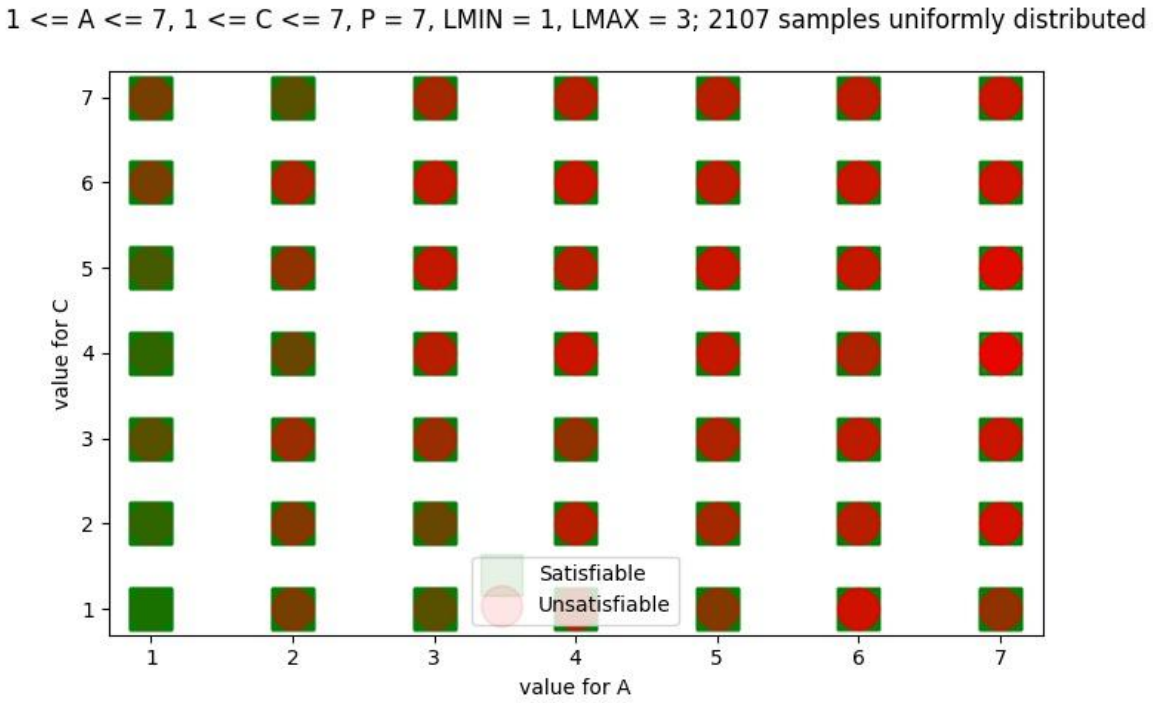
Run time and memory consumption

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). By "well" it is meant 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 clause form passed as an input to the backtracking algorithm exceeds 30 which makes the computation infeasible in a reasonable amount of time.

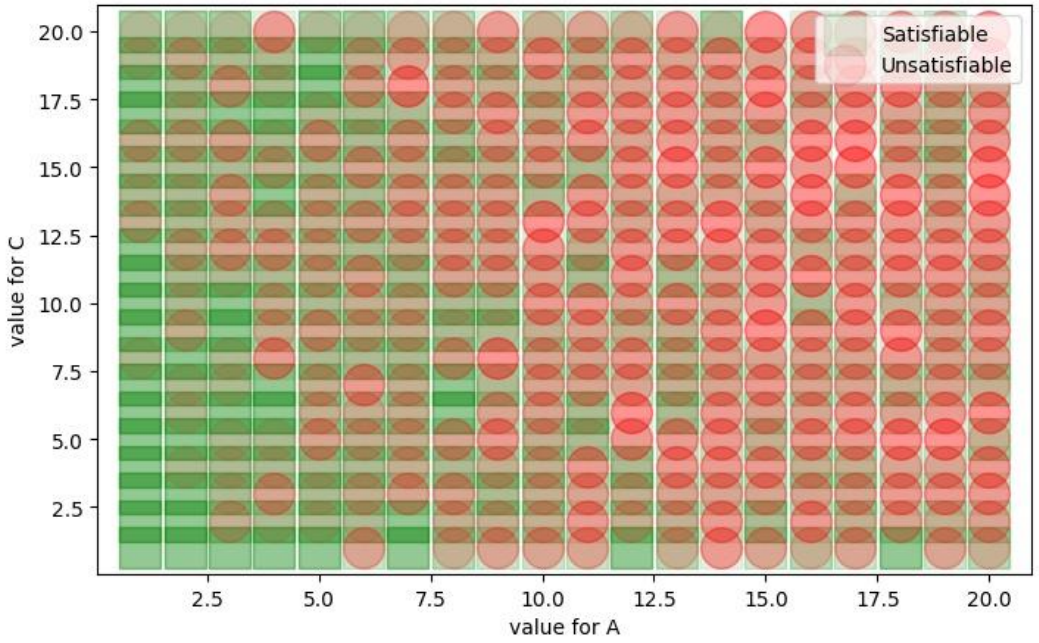
During this experiment it was observed that the memory consumption is perfectly consistent with the expectations: the amount of memory consumed is polynomially bounded by the size of the input.

Satisfiability distribution

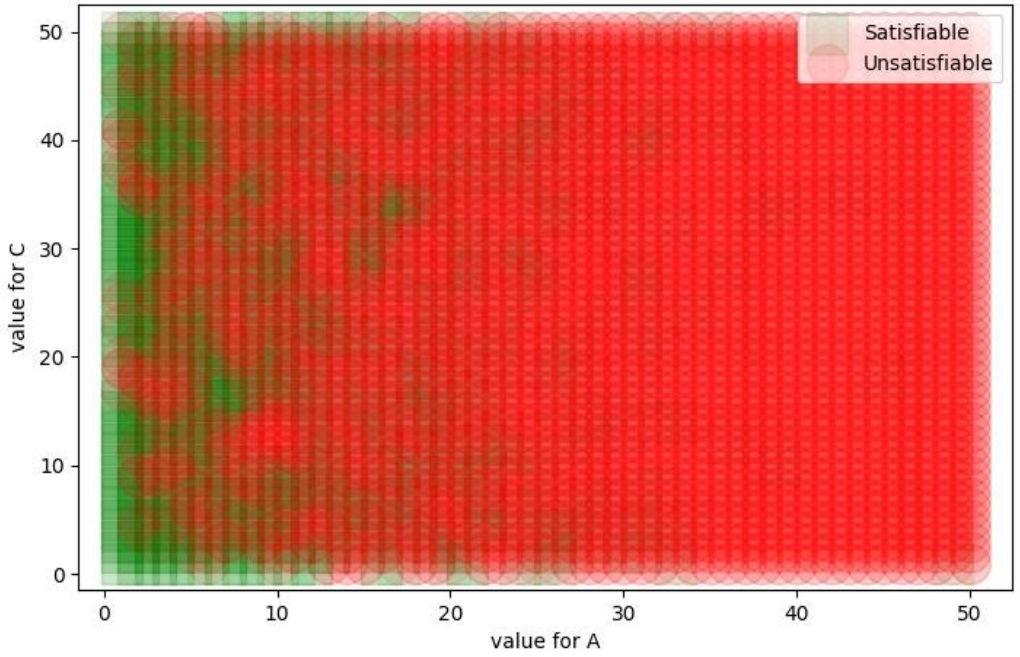
The second experiment's purpose was to discover what is the satisfiability distribution when ranging the values of A and C, with LMIN, LMAX and P fixed. The following three charts present the results.



1 <= A <= 20, 1 <= C <= 20, P = 7, LMIN = 1, LMAX = 3; 2000 samples uniformly distributed



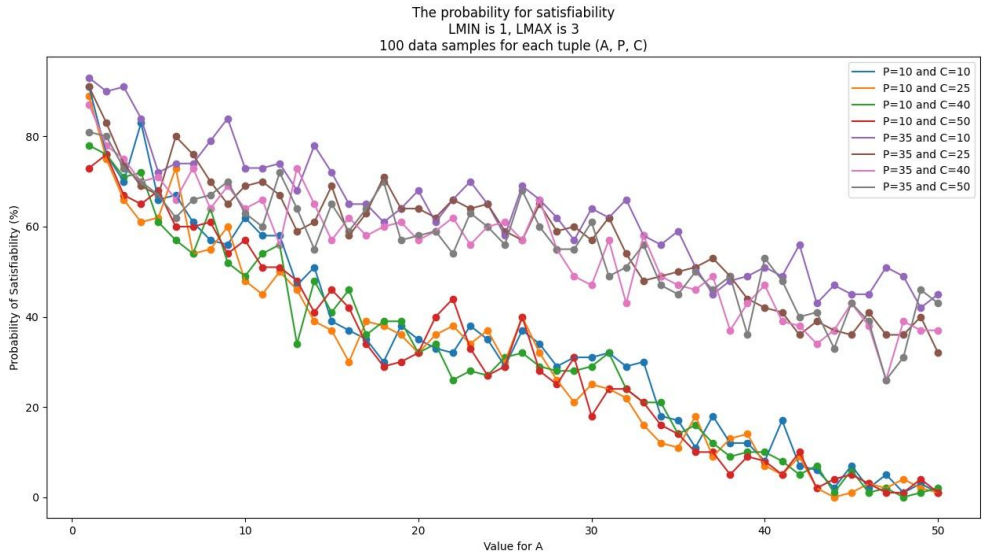
1 <= A <= 50, 1 <= C <= 50, P = 7, LMIN = 1, LMAX = 3; 5000 samples uniformly distributed

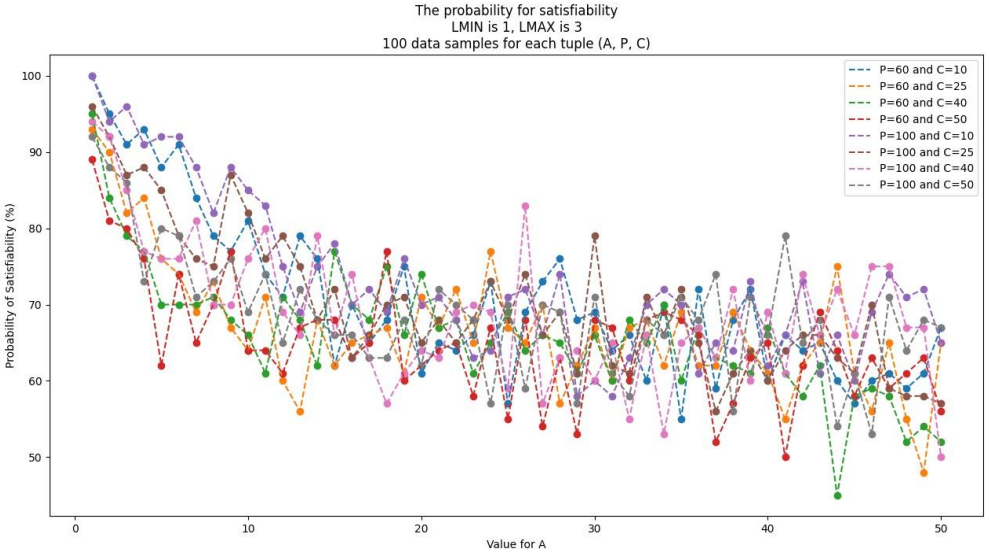


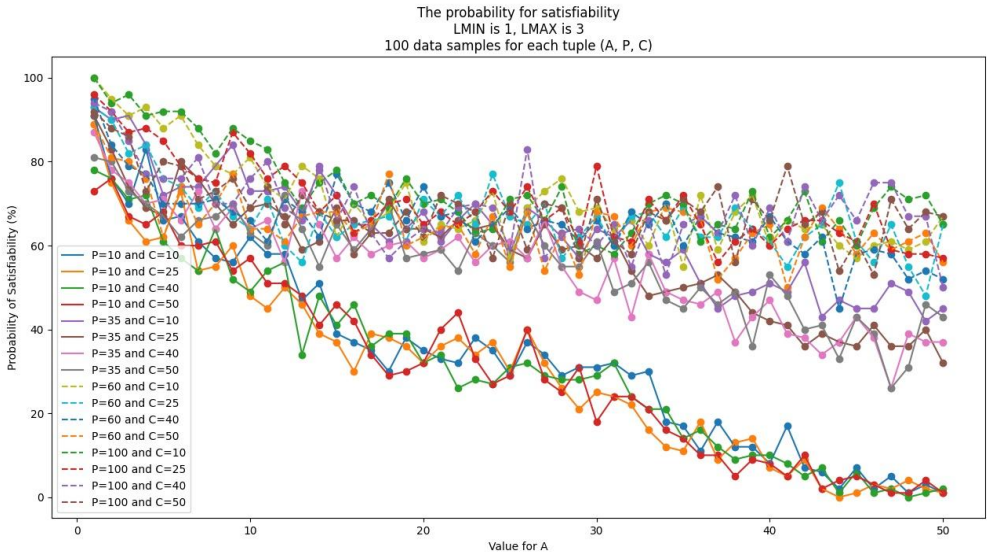
All of the three charts are consistent with one another, suggesting that a lower value for A represents a higher probability of obtaining a satisfiable formula. Conversely, a higher value for A almost guarantees an unsatisfiable formula. However, a limitation is that these charts represent a fixed value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

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.







The major findings of these charts are the following:

- the probability of the 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 one of the satisfiability are directly proportional

This concludes the section dedicated to the experiments.

Conclusion

This work successfully implements the decision procedure in [HDN&IPH] 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. Statistics on randomly generated formulas are as well delivered.

Applications

All of this work can represent a starting point for whoever would be willing to re-implement the same decision procedure or to benchmark his/hers against our implementation.

On the top of that, the implementation contains many optimizations which could be further reused in similar projects which are not necessarily tight to the two-variable fragment (examples include the optimization for the double implications and the optimization for the distributive).

Finally, one could find useful the approach chosen for storing and modelling the first-order logic formulas (using the parse tree). Possible further implementation might strive to join the three sub-components (front-end, intermediate representation and back-end) in a single

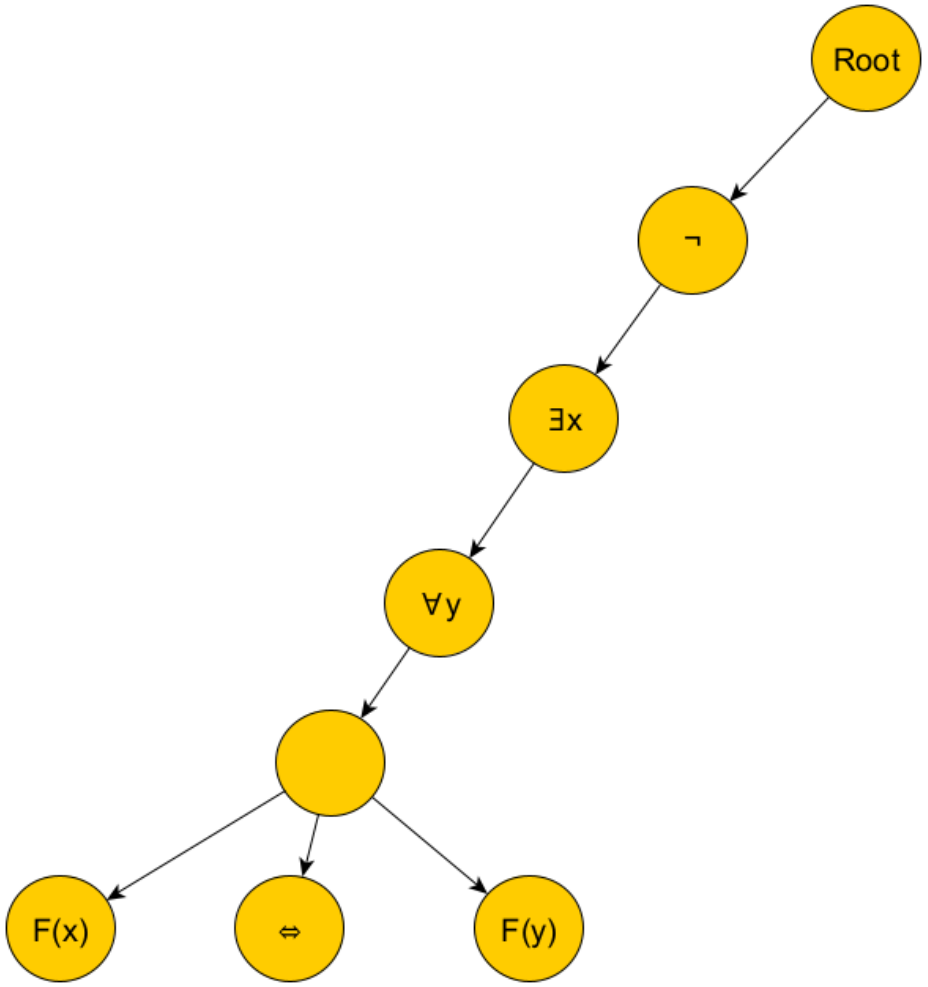
component, with the goal of integrating the functionality for the back-end in the same data structure as for the front-end and intermediate representation (note that this was achieved as part of this implementation).

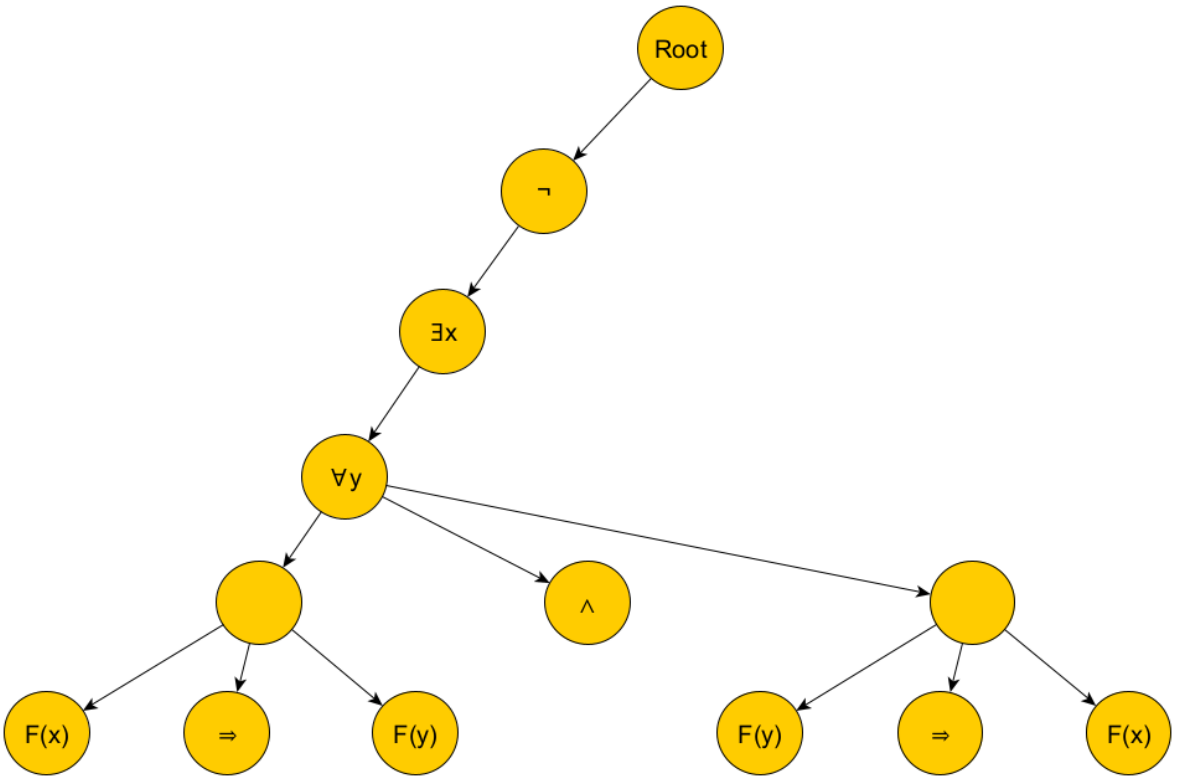
Limitations and Further Research

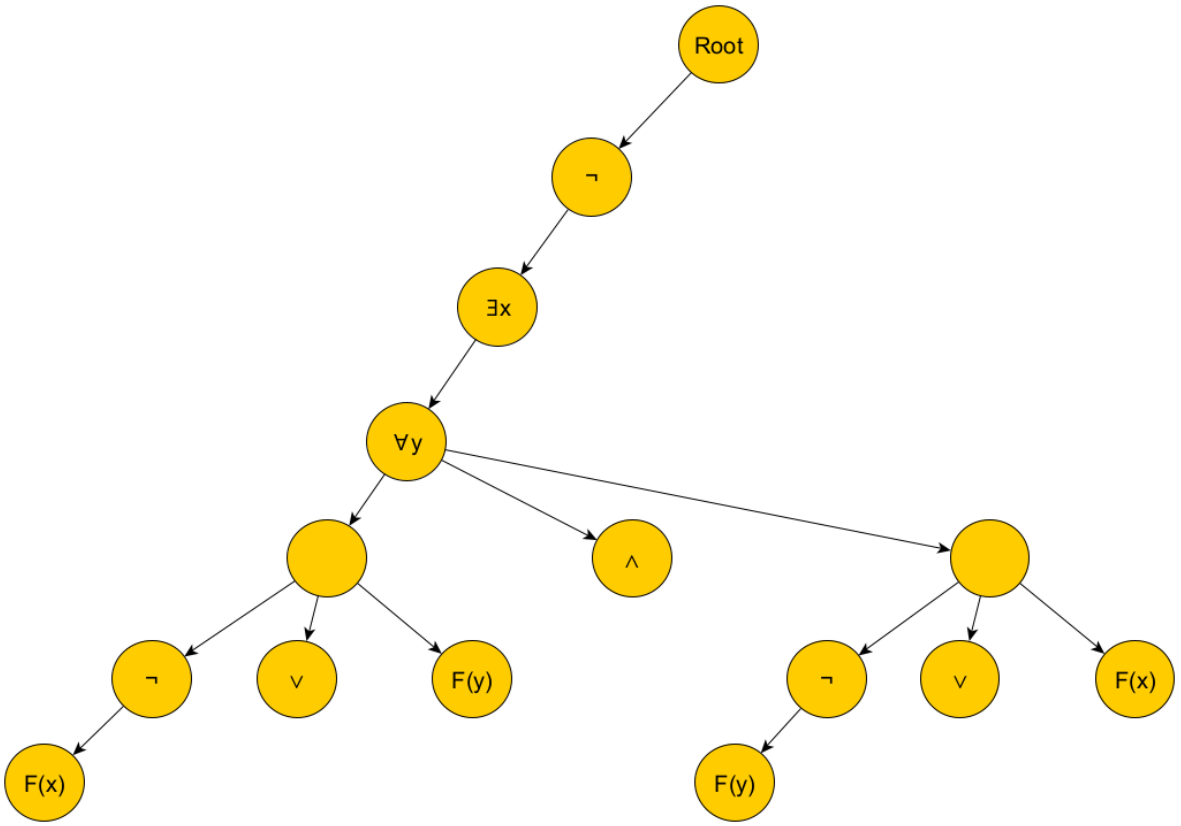
The difference between the run-times for randomly generated formulas between the produced theorem-provers and the state of art theorem provers (e.g. Vampire) is still considerable. Even though the amount of time in which the formers were developed is significantly smaller than for the latters, implementing further heuristics on the top of the existing implementation is surely not hopeless in achieving solid optimizations. Furthermore, using fuzz-testing for detecting potential bugs represents as well a reasonable path towards obtaining a more robust implementation. Moreover, even though simple tree-pruning was used for the implementation of backtracking, implementing other various pruning techniques might speed-up the execution time of the algorithm. Finally, changing the format of the language for the input formula to a standardized one (such as the one of Vampire) would make the entire testing-process less error-prone.

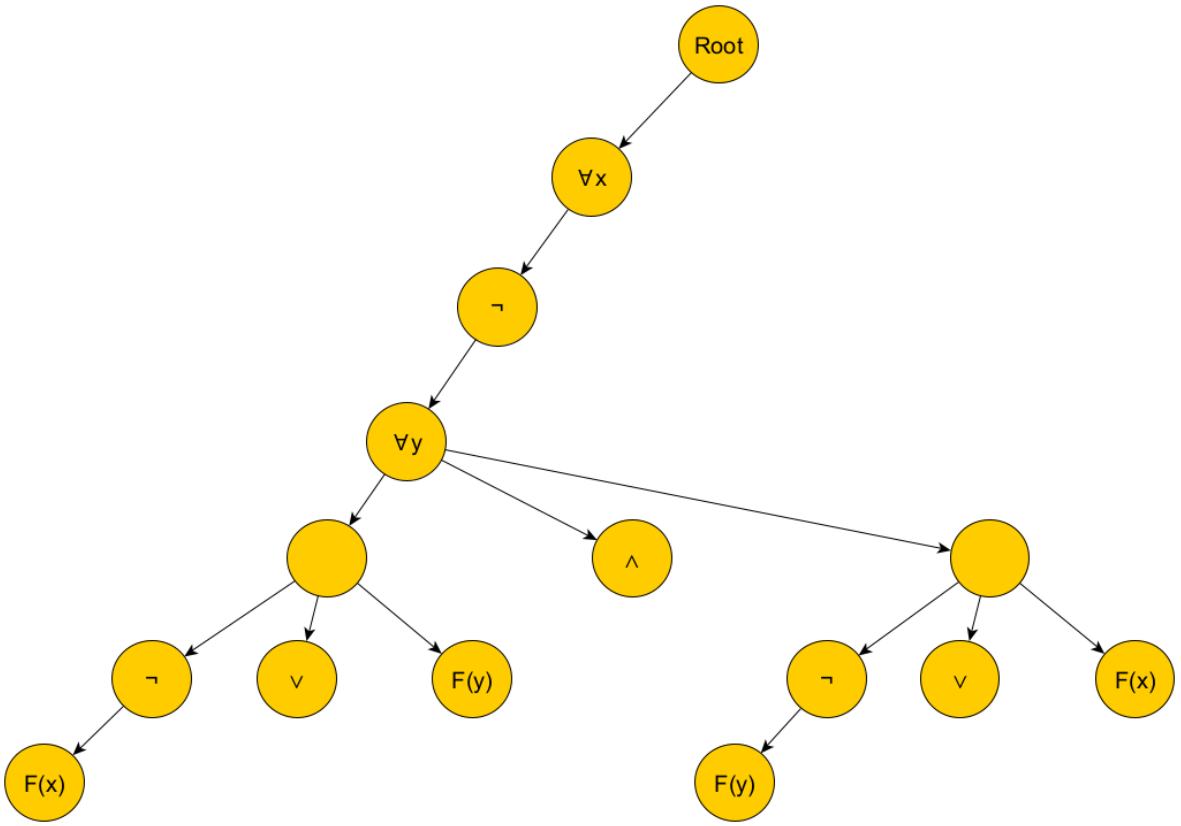
Appendix 1

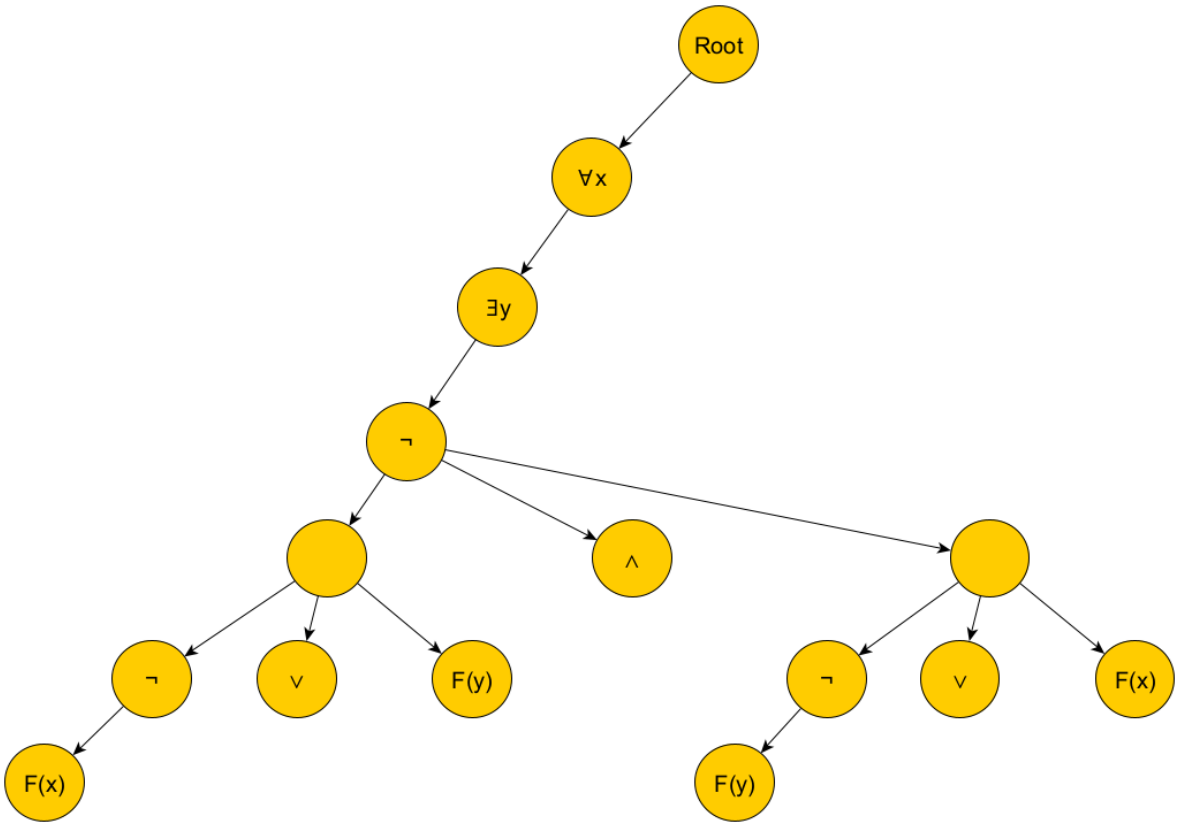
For the formula $\neg(\exists x \forall y(F(x) \leftrightarrow F(y)))$ the following changes are produced in the parse tree:

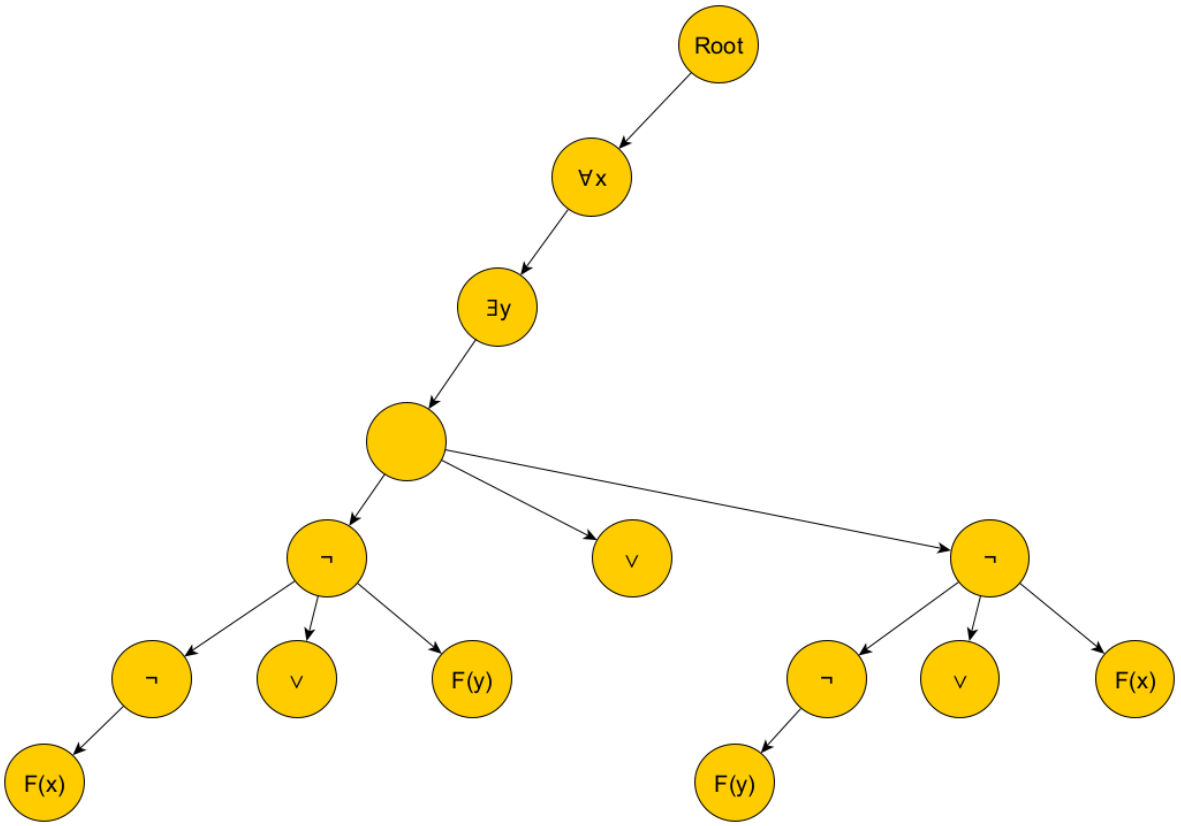

Root
¬
∃x
∀y
F(x)
⇔
F(y)

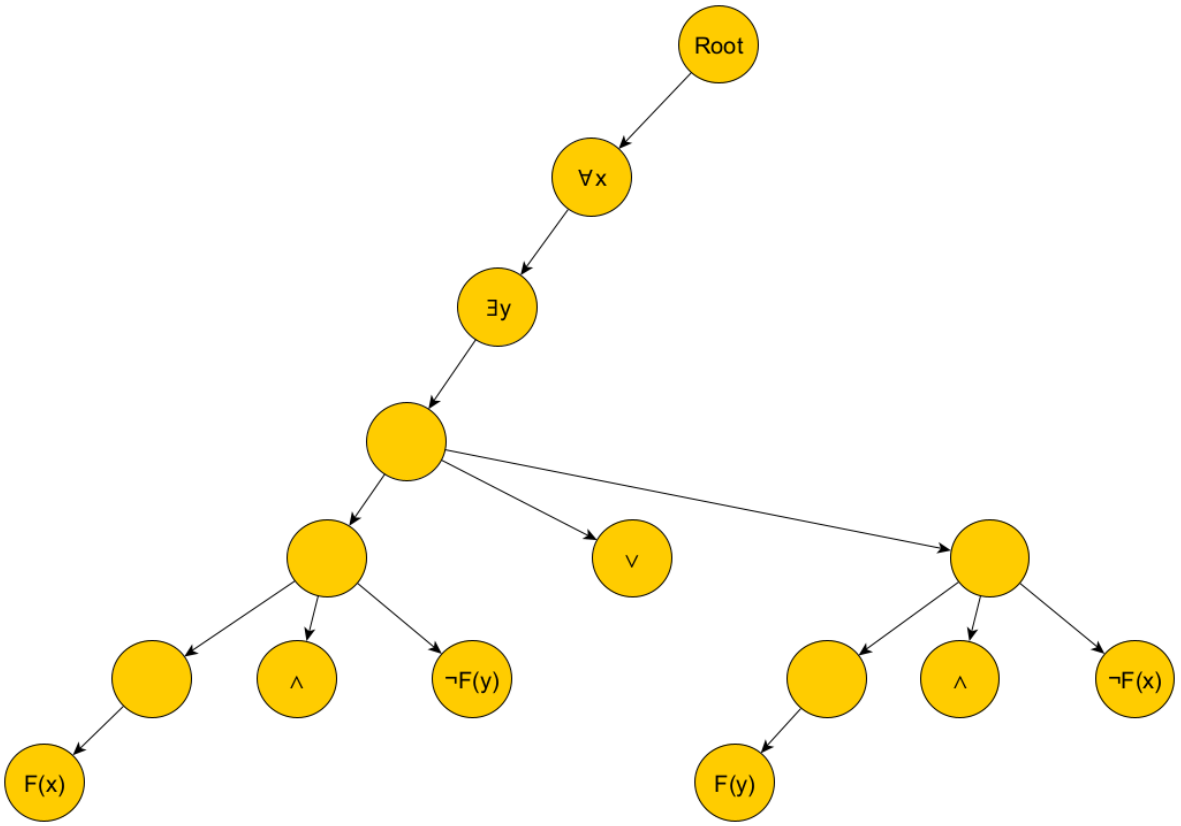

Root
¬
∃x
∀y
∧
F(x)
⇒
F(y)
F(y)
⇒
F(x)

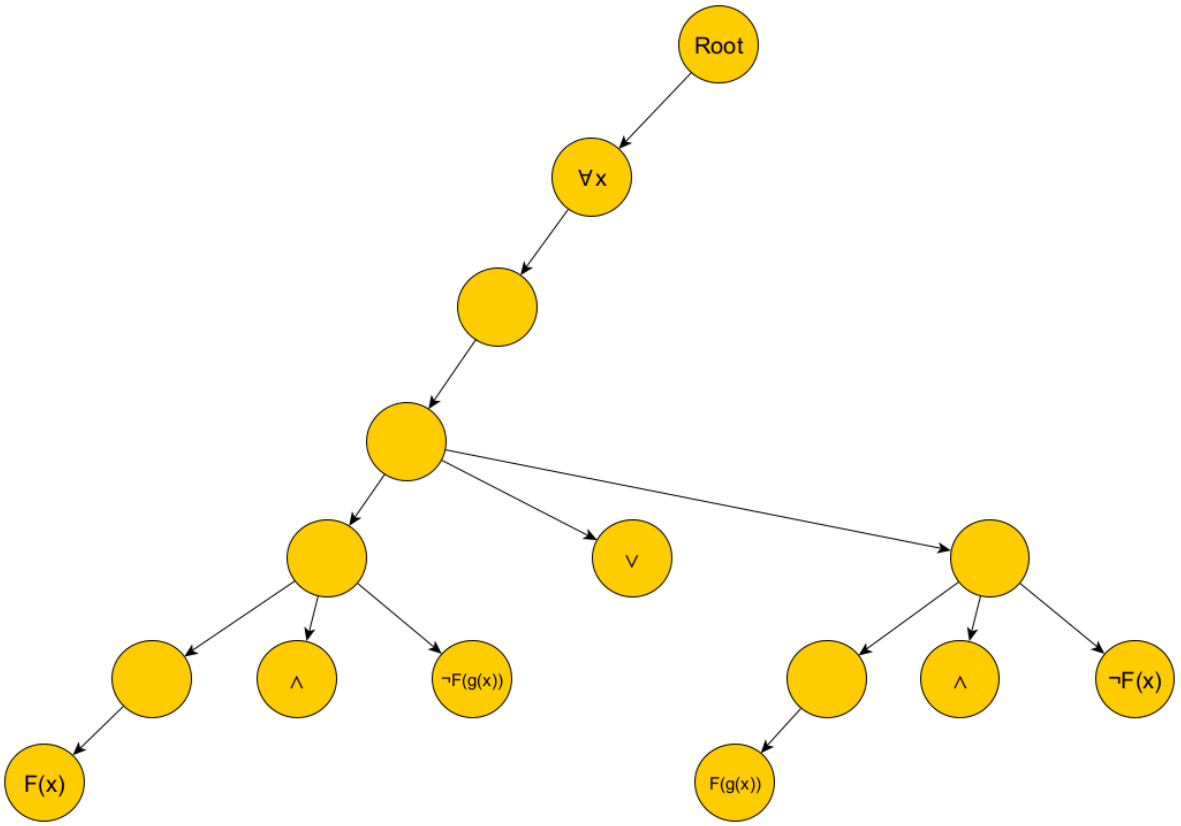

Root
¬
∃x
∀y
∧
¬
∨
F(y)
F(x)
¬
∨
F(x)
F(y)

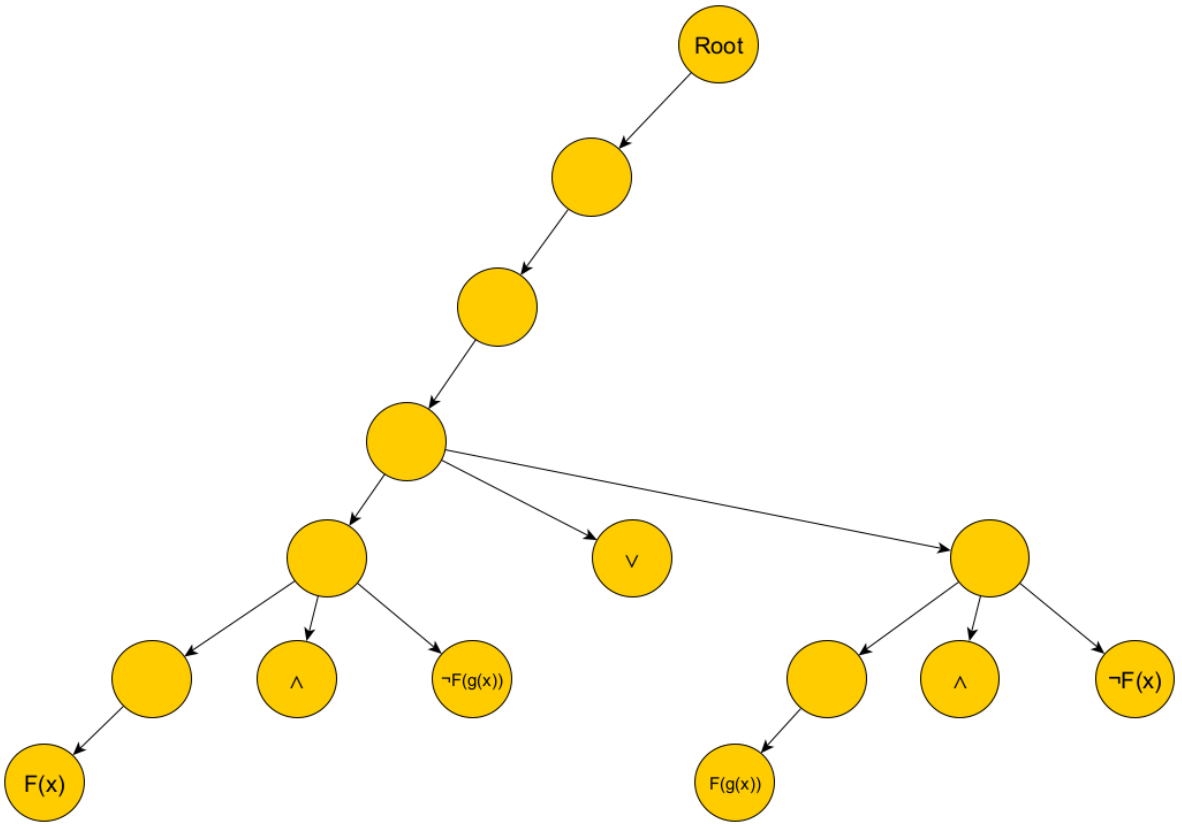

Root
∀x
¬
∀y
∧
¬
∨
F(y)
F(x)
¬
∨
F(x)
F(y)

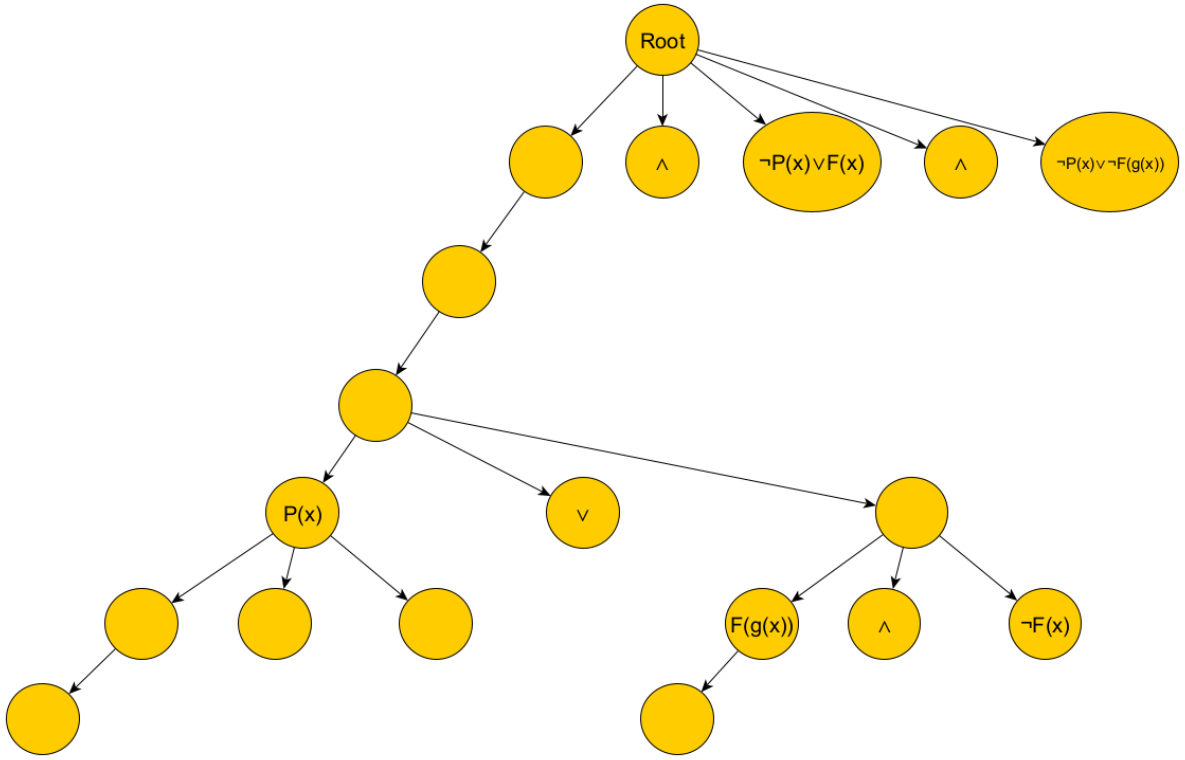

Root
∀x
∃y
¬
∧
¬
∨
F(y)
F(x)
¬
∨
F(x)
F(y)

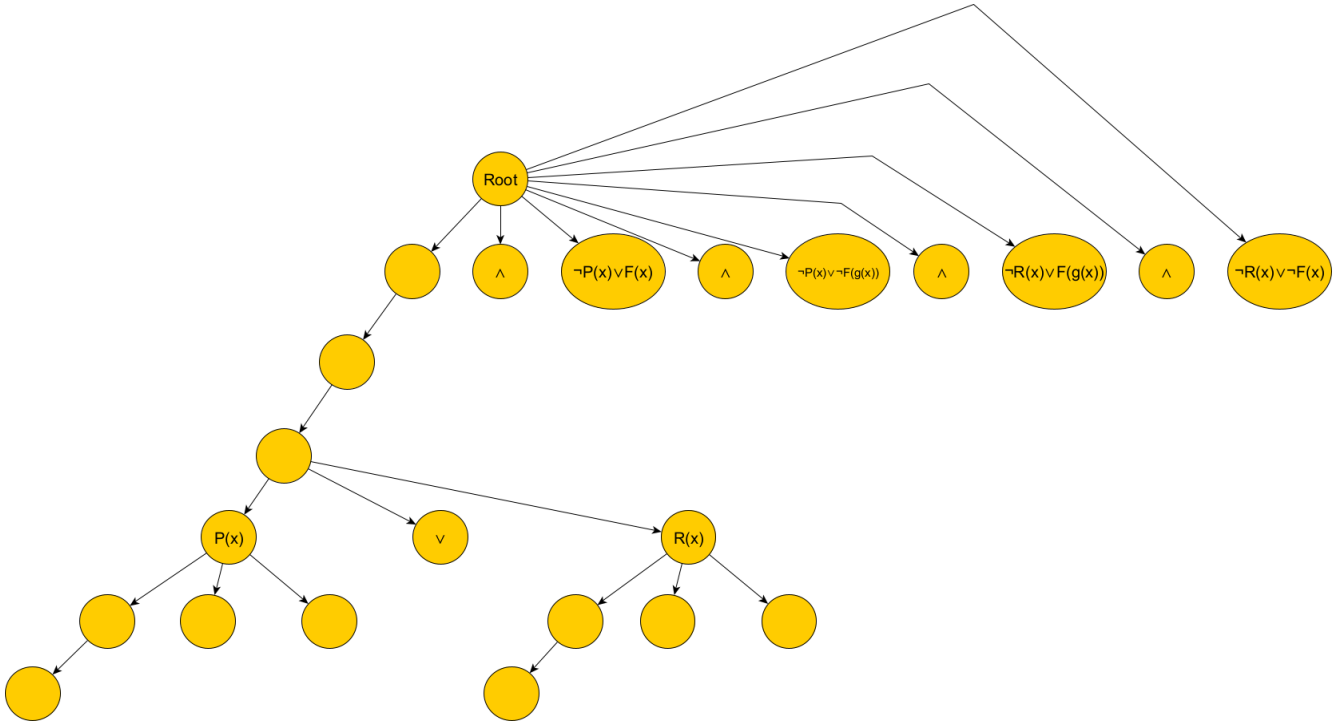

Root
∀x
∃y
¬
∨
¬
¬
∨
F(y)
F(x)
¬
∨
F(x)
F(y)

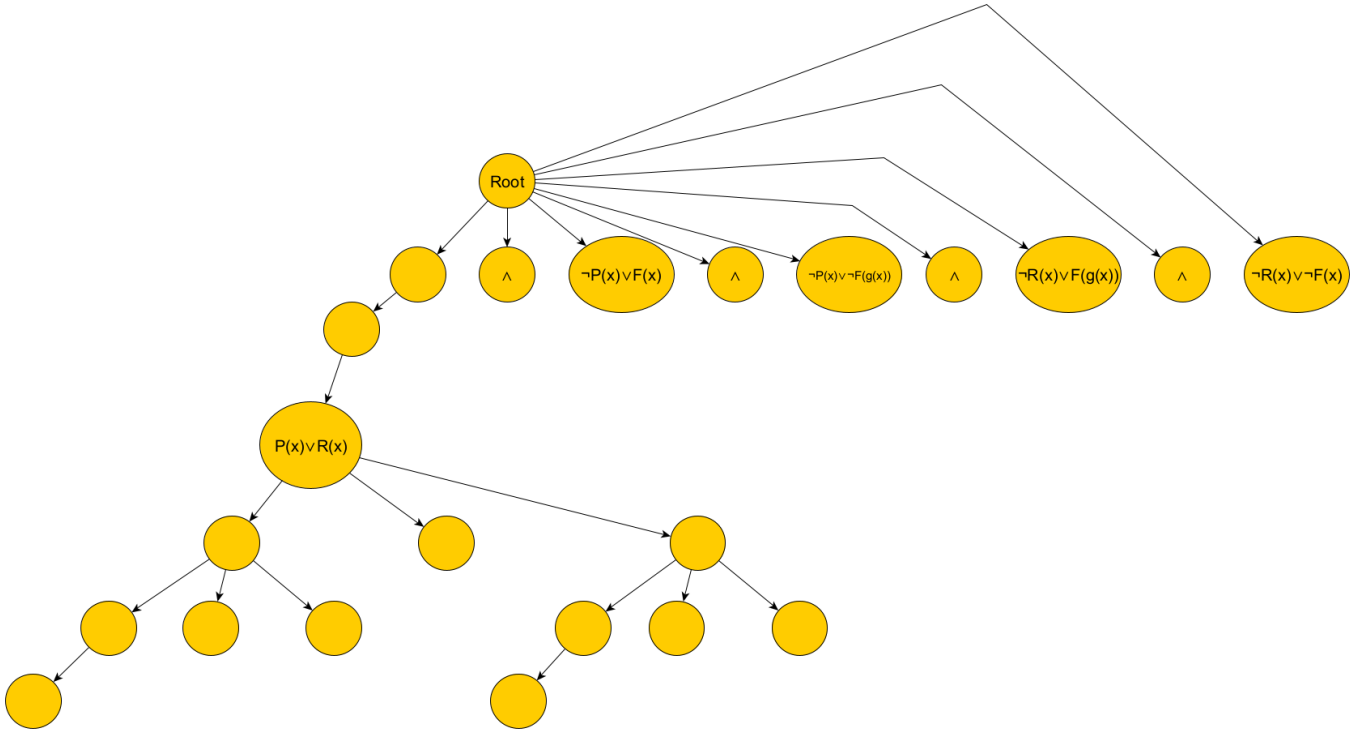

Root
∀x
∃y
∨
∧
¬F(y)
F(x)
F(y)
∧
¬F(x)

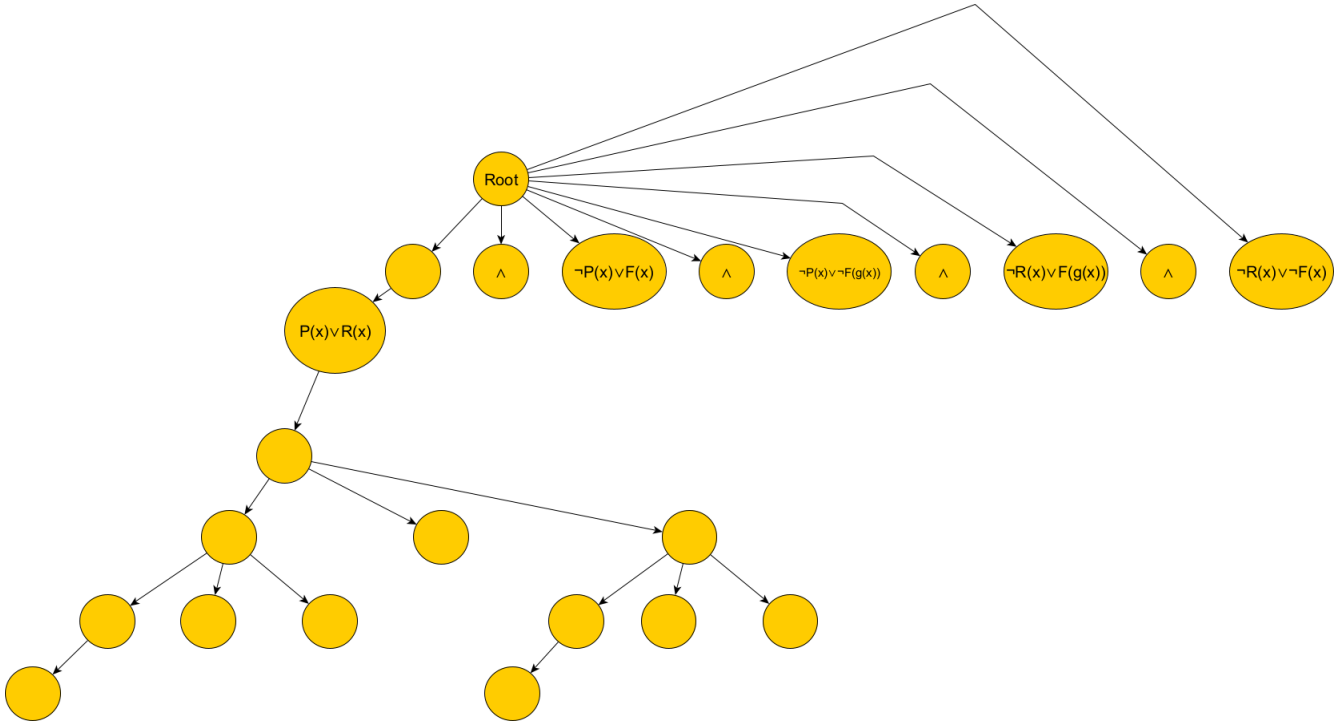

Root
∀x
∨
∧
¬F(g(x))
F(x)
F(g(x))
∧
¬F(x)

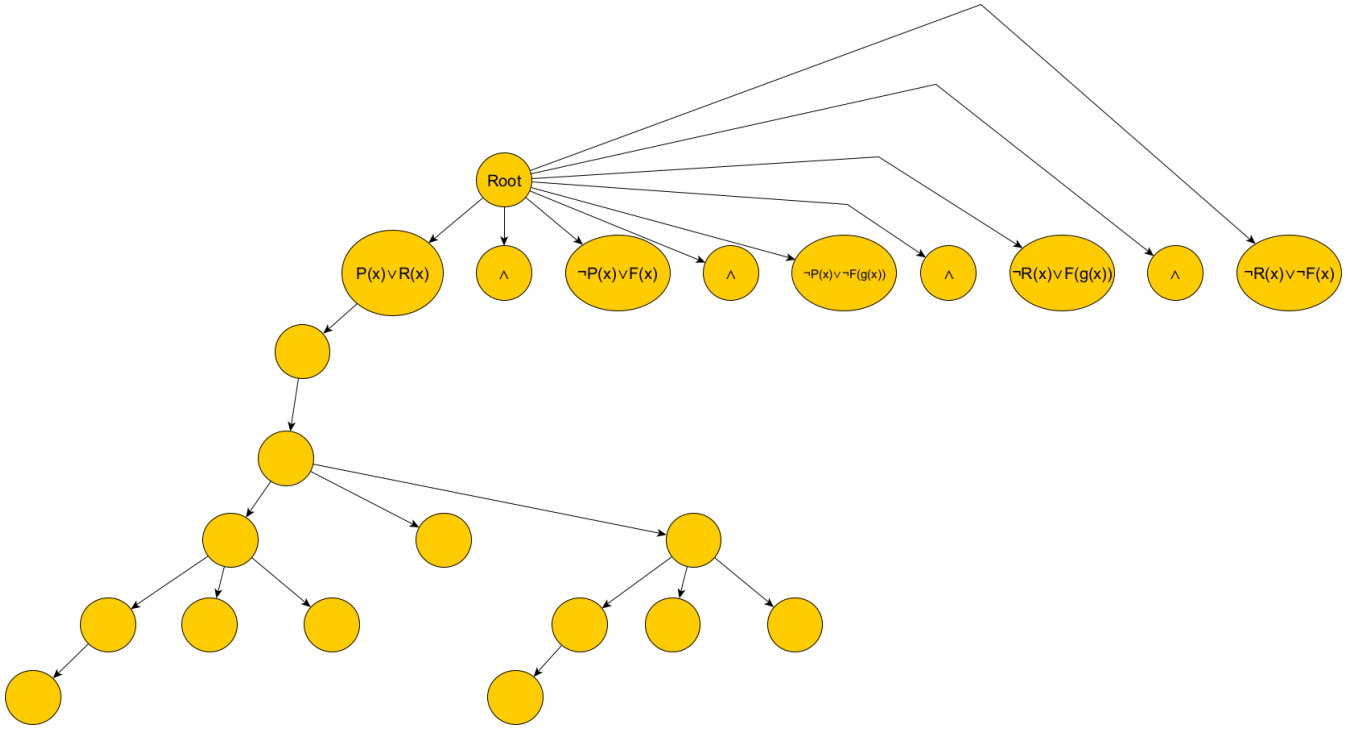

Root
∨
∧
¬F(g(x))
F(x)
F(g(x))
∧
¬F(x)


Root
∧
¬P(x)∨F(x)
∧
¬P(x)∨¬F(g(x))
P(x)
∨
F(g(x))
∧
¬F(x)


Root
∧
¬P(x)∨F(x)
∧
¬P(x)∨¬F(g(x))
∧
¬R(x)∨F(g(x))
∧
¬R(x)∨¬F(x)
P(x)
∨
R(x)


Root
∧
¬P(x)∨F(x)
∧
¬P(x)∨¬F(g(x))
∧
¬R(x)∨F(g(x))
∧
¬R(x)∨¬F(x)
P(x)∨R(x)





References

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

Gore, R., Leitsch, A. and Nipkow, T. (2001) 'Automated Reasoning', Proceedings of the First International Joint Conference, IJCAR, Siena, Italy, 2001, pp.211-225
Wilfrid, H. (1977). *Logic.* 1st edn. Penguin.
Leitsch, A. (1997). *The Resolution Calculus.* 1st edn. Springer.
Joyner Jr., W.H., Fermueller, C.G., Leitsch, A., Hustadt, U. and Tammet, T. (2002). 'Ordered Resolution' [PowerPoint presentation]. Available at: https://people.mpi-inf.mpg.de/~hillen/documents/4_OrderedResolution.pdf (Accessed: 7 April 2021).
CPP Reference (2021). *C++ Utilities Library: std::variant.* Available at: https://en.cppreference.com/w/cpp/utility/variant (Accessed: 7 April 2021)
CPP Reference (2021). *C++ Utilities Library: std::execution.* Available at: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t (Accessed: 7 April 2021).
Debian (2021). *Package: libtbb-dev (2020.3-1 and others).* Available at: https://packages.debian.org/sid/libtbb-dev (Accessed: 7 April 2021).
Googletest (2021). *Googletest.* Available at: https://github.com/google/googletest (Accessed: 7 April 2021).
CPP Reference (2021). *C++ Utilities Library: std::unordered_set.* Available at: https://en.cppreference.com/w/cpp/container/unordered_set (Accessed: 7 April 2021).