A Verified QBF Solver

Axel Bergström
A Verified QBF Solver

Axel Bergström

Abstract

Quantified Boolean Formulas (QBFs) extend propositional logic with universal and existential quantification over Boolean variables. A QBF is satisfiable if and only if there is an assignment of Boolean values to the free variables that makes the formula true, and a QBF solver is a software tool determining if a given QBF is satisfiable.

This thesis introduces formalizations of, and correctness proofs for, two QBF solvers in the Isabelle/HOL interactive theorem prover. One solver is based on naive quantifier expansion, while the other uses a search-based algorithm. While experiments show that the verified solvers have performance far behind the state-of-the-art in QBF solving, they provide unparalleled confidence in correctness. To our knowledge, these are the first QBF solvers to be formally verified using interactive theorem proving.
Acknowledgements

I want to thank my supervisor, Tjark Weber, for all the help I have received during this thesis, for all of our discussions about the Isabelle/HOL formalizations, which saved me from a lot of headaches, for defining the xor problem used in the evaluation, and, last but not least, for making sure that I stay on track while working on this thesis.

I also want to thank all the great people I got to share an office with while working on this thesis. You all made my time working on this thesis into a much more enjoyable experience.
7 Related Work

7.1 State of the Art QBF Solvers

7.1.1 Search-based Solvers

7.1.2 Variable Elimination-based Solvers

7.2 Verified SAT Solving

7.3 Verified Proof Checking

8 Future Work

8.1 More Advanced Algorithms

8.2 A Verified Proof Checker

8.3 Verifying the Parser and Code Export

9 Conclusion

References

A Isabelle/HOL Theory: Naive Solver

A.1 QBF Datatype, Semantics, and Satisfiability

A.1.1 A QBF datatype

A.1.2 Formalization of Semantics & Termination of Semantics

A.1.3 Formalization of Satisfiability

A.2 Existential Closure

A.2.1 Formalisation of Free Variables

A.2.2 Formalisation of Existential Closure

A.2.3 Preservation of Satisfiability under Existential Quantification

A.2.4 Preservation of Satisfiability under Existential Closure

A.2.5 Non-existence of Free Variables in Existential Closure

A.3 Sequence Utility Function

A.4 Naive solver

A.4.1 Expanding Quantifiers

A.4.2 Expanding Formulas

A.4.3 Evaluating Expanded Formulas

A.4.4 Naive Solver

B Isabelle/HOL Theory: PCNF Datatype

B.1 A Prenex Conjugate Normal Form Datatype

B.1.1 PCNF Predicate for Generic QBFs

B.1.2 Bijection with PCNF subset of Generic QBF Datatype

B.1.3 Preservation of Semantics under the Bijection

C Isabelle/HOL Theory: QDIMACS Parser

D Isabelle/HOL Theory: Search Solver

D.1 Formalisation of PCNF Assignment

D.2 Effect of PCNF Assignments on the Set of all Free Variables

D.2.1 Variables, Prefix Variables, and Free Variables

D.2.2 Free Variables is Variables without Prefix Variables

D.2.3 Set of Matrix Variables is Non-increasing under PCNF Assignments

D.2.4 PCNF Assignment Removes Variable from Prefix
D.2.5 Set of Free Variables is Non-increasing under PCNF Assignments ........................................ 138
D.3 PCNF Existential Closure ........................................................................................................ 138
  D.3.1 Formalization of PCNF Existential Closure ......................................................................... 138
  D.3.2 PCNF Existential Closure Preserves Satisfiability ................................................................. 138
  D.3.3 No Free Variables in PCNF Existential Closure .................................................................. 138
D.4 Search Solver (Part 1: Preliminaries) ....................................................................................... 138
  D.4.1 Conditions for True and False PCNF Formulas ................................................................. 138
  D.4.2 Satisfiability Equivalences for First Variable in Prefix ....................................................... 139
D.5 Cleansed PCNF Formulas ......................................................................................................... 149
  D.5.1 Predicate for Cleansed Formulas ....................................................................................... 149
  D.5.2 The Cleansed Predicate is Invariant under PCNF Assignment ........................................... 151
  D.5.3 Cleansing PCNF Formulas ................................................................................................ 154
  D.5.4 Cleansing Yields a Cleansed Formula ................................................................................... 154
  D.5.5 Cleansing Preserves Semantics ........................................................................................... 156
D.6 Search Solver (Part 2: The Solver) ............................................................................................ 158
  D.6.1 Correctness of the Search Function ..................................................................................... 160
  D.6.2 Correctness of the Search Solver .......................................................................................... 165

E Isabelle/HOL Theory: Solver Export 165
1 Introduction

Quantified Boolean formulas (QBFs) extend propositional formulas with quantifiers over Boolean variables. In addition to Boolean variables and the usual connectives of propositional logic (e.g., $\land$, $\lor$, and $\neg$), QBFs may contain existential quantifiers ($\exists$) and universal quantifiers ($\forall$) that range over Booleans. A QBF is said to be satisfiable if and only if there is an assignment of Boolean values to the free variables making the formula true. For example, this is a QBF:

$$\forall y \exists x((y \lor z) \land x)$$

This particular formula has a universally quantified variable ($y$), an existentially quantified variable ($x$), and a free variable ($z$); and is satisfiable since there is an assignment of Boolean values to all free variables, in this case just $z$, making the formula true. If we assign $z$ as true and choose $x$ to be true, then the formula evaluates to true both when $y$ is true and when $y$ is false.

A software tool determining if a given QBF is satisfiable or not is known as a QBF solver, while formal verification using interactive theorem proving can be used to create machine-checkable proofs, providing very high confidence that the proofs are correct. The main contributions of this thesis are the formalization of two simple QBF solvers and correctness proofs for both solvers in the Isabelle/HOL [1] interactive theorem prover.\(^1\)

1.1 Motivation for Studying QBF Solving

Many problems have been encoded as QBFs, including problems from planning, two-player games, verification, and synthesis. This means that a QBF solver can be used to solve a wide range of different problems. Some examples of such problems are given here:

- Different planning tasks have been encoded using QBFs, including cases where there is uncertainty about the state or the effect of performing an action [2, 3].
- Two-player games can often be encoded using QBFs by representing the moves of one player with existential variables and the moves of the other player with universal variables [4].
- In verification, QBFs have been used for bounded model checking, yielding a lower memory usage than SAT-based methods [5, 6].
- In synthesis for field programmable gate arrays (FPGAs), QBFs have been used to determine if a function can be implemented using a given programmable circuit [7].

More examples of QBF applications can be found in [8].

The wide range of applications for QBF solving can be explained by QBF satisfiability being a PSPACE-complete problem; this is similar to how SAT is NP-complete, and can be used to solve problems in NP [9]. Work in QBF solving can, therefore, also be motivated by PSPACE-completeness, and also the success of SAT solving [10].

\(^1\)The solvers are available at: https://github.com/iswoqqe/a_verified_qbf_solver/.
1.2 Motivation for using Formal Verification

The high confidence in proof correctness gained from using interactive theorem proving is especially desirable for correctness proofs for QBF solvers. There are a couple of reasons for this:

1. QBF solvers can be used as a foundational building block in many different applications, examples of which are given in the previous subsection. This wide range of applications for QBF solving increases the value gained from proving solver correctness, since we can benefit from the correctness of a solver in many different applications.

2. Erroneous solver output can be difficult to identify, since it is, in general, difficult to check if a satisfiable or unsatisfiable result for a given formula is correct. A simple boolean output does not allow the result to be checked in any meaningful way. Some solvers can output proof certificates, but different solvers use different formats for this, and the proof length will be exponential in at least some cases [11].

2 Background

Quantified Boolean formulas, basic methods for solving them, and interactive theorem proving are all introduced here. It is assumed that the reader is familiar with propositional logic and basic functional programming. Familiarity with SAT solving is also beneficial.

2.1 Quantified Boolean Formulas

Quantified Boolean Formulas (QBFs) extend propositional logic with quantification over Boolean variables. In addition to propositional variables and connectives, QBFs may contain universal ($\forall$) and existential ($\exists$) quantifiers over Boolean variables. Informally, $\forall y \Phi$ is true if and only if $\Phi$ is true both when $y$ is true and when $y$ is false, and $\exists x \Phi$ is true if and only if $\Phi$ is true when $x$ is true or when $x$ is false.

In this subsection, quantified Boolean formulas are introduced based on definitions in chapter 29 of the Handbook of Satisfiability, “Theory of Quantified Boolean Formulas”, by Büning and Bubeck [12]; and chapter 30, “Reasoning with Quantified Boolean Formulas” by Giunchiglia, Marin, and Narizzano [13]. For more in-depth information, see these chapters in the handbook.

2.1.1 QBF Syntax, QBF Semantics, and Basic QBF Terminology

This thesis uses the QBF syntax and semantics defined in [13], which are introduced here. Some other definitions, mainly from [12] and [13], are also introduced here.

Local Conventions: In this thesis, the following convention is used for formulas and variables: Quantified Boolean formulas are denoted by uppercase Greek letters, while propositional formulas are denoted by lowercase ones, and variables are denoted by Latin letters. Existentially quantified variables are usually denoted by $x$ and universal ones usually by $y$. 

11
Syntax. The syntax of QBFs is defined inductively using: variables, the universal quantifier (\(\forall\)), the existential quantifier (\(\exists\)), conjunction (\(\land\)), disjunction (\(\lor\)), negation (\(\neg\)), and parentheses; as the minimal set where:

1. If \(z\) is a variable, then \(z\) is a QBF.
2. If \(\Phi\) is a QBF, then \(\neg\Phi\) is a QBF.
3. If \(\Phi_1, \ldots, \Phi_n\) are QBFs where \(n \geq 0\), then \((\Phi_1 \land \cdots \land \Phi_n)\) is a QBF.
4. If \(\Phi_1, \ldots, \Phi_n\) are QBFs where \(n \geq 0\), then \((\Phi_1 \lor \cdots \lor \Phi_n)\) is a QBF.
5. If \(\Phi\) is a QBF, then \(\exists x\Phi\) is a QBF.
6. If \(\Phi\) is a QBF, then \(\forall y\Phi\) is a QBF.

The empty conjunction is considered to be true, and is abbreviated by 1; while the empty disjunction is considered to be false, and is abbreviated by 0.

QBF Terminology. The following is defined for QBFs:

- A literal is a variable or a negated variable.
- A literal is positive if and only if it is a variable, and negative if and only if it is a negated variable.
- The length of a quantified Boolean formula \(\Phi\) is the number of occurrences of variables in \(\Phi\).
- A quantified Boolean formula \(\Psi\) is a subformula of a quantified Boolean formula \(\Phi\) if and only if \(\Psi\) is a substring of \(\Phi\).
- In \(Qz\Phi\), where \(Q \in \{\forall, \exists\}\), the variable \(z\) is a quantified variable, and the occurrence of \(z\) is a quantified occurrence where \(\Phi\) is the scope of the quantified occurrence of \(z\). Moreover, if \(Q = \forall\), then \(z\) is universally quantified, and if \(Q = \exists\), then \(z\) is existentially quantified.
- An occurrence of a variable \(z\) is called bound if and only if it is in the scope of a quantified occurrence of \(z\), otherwise it is called free.
- A variable \(z\) is said to be bound in \(\Phi\) if and only if \(\Phi\) is a subformula of the scope of a quantified occurrence of \(z\).
- A variable \(z\) is said to be free in \(\Phi\) if and only if there is a free occurrence of \(z\) in \(\Phi\).
- A formula is closed if and only if it does not have any free variables.
- The existential closure of a quantified Boolean formula \(\Phi\) is \(\exists z_1 \ldots \exists z_n \Phi\) where \(z_1, \ldots, z_n\) are all of the free variables in \(\Phi\).
Semantics. A truth assignment or an interpretation is a mapping $I$ from the set of variables to $\{0, 1\}$. Any such assignment can be used to evaluate any QBF, under the assignment $I$, by extending the domain of $I$ to include all QBFs. Let $\Phi$ be a QBF, let $\Phi[a/z]$ denote substitution of all free occurrences of $z$ in $\Phi$ by $a \in \{0, 1\}$, and let $I(\Phi)$ denote the valuation of $\Phi$ under the interpretation $I$. Then, the evaluation is done by extending the domain while satisfying the following conditions:

- If $\Phi = z$ is a variable, then $I(\Phi) = I(z)$.
- If $\Phi = \neg \Phi'$, then $I(\Phi) = 1$ iff $I(\Phi') = 0$.
- If $\Phi = (\Phi_1 \land \cdots \land \Phi_n)$, then $I(\Phi) = 1$ iff $\forall i : 1 \leq i \leq n : I(\Phi_i) = 1$.
- If $\Phi = (\Phi_1 \lor \cdots \lor \Phi_n)$, then $I(\Phi) = 1$ iff $\exists i : 1 \leq i \leq n : I(\Phi_i) = 1$.
- If $\Phi = \exists x \Phi'$, then $I(\exists x \Phi') = 1$ iff $I(\Phi'[0/x]) = 1$ or $I(\Phi'[1/x]) = 1$.
- If $\Phi = \forall y \Phi'$, then $I(\forall y \Phi') = 1$ iff $I(\Phi'[0/y]) = 1$ and $I(\Phi'[1/y]) = 1$.

More QBF Terminology. Using the semantics, the following can be defined:

- A quantified Boolean formula $\Phi$ is satisfiable if and only if there is a truth assignment $I$ such that $I(\Phi) = 1$. Otherwise, $\Phi$ is unsatisfiable.
- Two quantified Boolean formulas $\Phi_1$ and $\Phi_2$ are logically equivalent if and only if $\forall I. I(\Phi_1) \leftrightarrow I(\Phi_2)$. We use $\Phi_1 \approx \Phi_2$ to denote this.
- Finally, two formulas $\Phi_1$ and $\Phi_2$ are satisfiability equivalent if and only if ($\Phi_1$ is satisfiable $\iff$ $\Phi_2$ is satisfiable). We use $\Phi_1 \approx_{sat} \Phi_2$ to denote this.

QBF Satisfiability. We also note the following about QBF satisfiability:

- For any formulas $\Phi_1$ and $\Phi_2$: if $\Phi_1 \approx \Phi_2$, then $\Phi_1 \approx_{sat} \Phi_2$.
- Any quantified Boolean formula $\Phi$ is satisfiable if and only if the existential closure of $\Phi$ is. That is, $\Phi \approx_{sat} \exists z_1 \ldots \exists z_n \Phi$ where $z_1, \ldots, z_n$ are all of the free variables in $\Phi$.

2.1.2 QBF Normal Forms

There are several common normal forms for quantified Boolean formulas, some of which are introduced here, based on descriptions in [12].

Cleansed Formulas. A QBF $\Phi$ is cleansed if and only if the following holds:

1. Two distinct occurrences of quantifiers have distinct quantified variables.
2. For all occurrences of $Q z \Phi'$ in $\Phi$, where $Q \in \{\forall, \exists\}$, it holds that $z$ is free in $\Phi'$ and not free in $\Phi$.  


Any quantified Boolean formula can be transformed into a cleansed formula by renaming variables. Because of this, we will assume that all formulas in the rest of this section are cleansed.

As an example, consider the QBF
\[ \Phi = (\forall z \Phi_1) \land (\exists z \Phi_2) \land z, \]
where the variable \( z \) occurs three times; once as a universally quantified variable, once as an existentially quantified variable, and once as a free variable. To avoid this, we can rename the variables of \( \Phi \) in a way that preserves logical equivalence; yielding, for example, the formula \( \Phi' = (\forall y \Phi_1) \land (\exists x \Phi_2) \land z \).

Negation Normal Form. A quantified Boolean formula is said to be in negation normal form if and only if all negation symbols (\( \neg \)) in the formula occur in front of a variable. Every QBF can be written in negation normal form by using De Morgan's law, the double negation law, and quantifier switching. That is, by using the following rules:

- \( \neg(\Phi_1 \lor \Phi_2) \approx (\neg \Phi_1 \land \neg \Phi_2) \)
- \( \neg(\Phi_1 \land \Phi_2) \approx (\neg \Phi_1 \lor \neg \Phi_2) \)
- \( \neg \neg \Phi \approx \Phi \)
- \( \neg(\exists x \Phi) \approx \forall x \neg \Phi \)
- \( \neg(\forall y \Phi) \approx \exists y \neg \Phi \)

Prenex Normal Form. A quantified Boolean formula is said to be in prenex normal form, or just prenex form, if and only if the formula consists of a sequence of quantifiers, called the prefix, followed by a quantifier-free formula, called the matrix. That is, a quantified Boolean formula is in prenex form if and only if it is written \( Q_1 z_1 \ldots Q_n z_n \phi \) where \( Q_i \in \{ \forall, \exists \}, z_i \) are the quantified variables, and \( \phi \) is the matrix. A simple way to transform a quantified Boolean formula \( \Phi \) to prenex form is to first transform \( \Phi \) to negation normal form and then apply the following rules:

- \( (\Phi_1 \lor \forall y \Phi_2) \approx \forall y(\Phi_1 \lor \Phi_2) \)
- \( (\Phi_1 \land \forall y \Phi_2) \approx \forall y(\Phi_1 \land \Phi_2) \)
- \( (\Phi_1 \land \exists x \Phi_2) \approx \exists x(\Phi_1 \land \Phi_2) \)
- \( (\Phi_1 \lor \exists x \Phi_2) \approx \exists x(\Phi_1 \lor \Phi_2) \).

Conjunctive Normal Form. A quantified Boolean formula is said to be in prenex conjunctive normal form, or just conjunctive normal form, if and only if it is in prenex normal form and its matrix is a propositional formula in conjunctive normal form. A propositional formula \( \phi \) is said to be in conjunctive normal form if and only if it is a conjunction of clauses, where a clause is a disjunction of literals.

Any propositional formula can be transformed to conjunctive normal form by using the distributive law \( ((\alpha \land \beta) \lor \sigma) \approx ((\alpha \lor \sigma) \land (\beta \lor \sigma)) \). This method works, but the length of the formula may grow exponentially. To avoid this, we can use the equivalence \( (\alpha \lor (\beta \land \gamma)) \approx_{\text{sat}} ((\alpha \lor \neg x) \land (\beta \lor x) \land (\gamma \lor x)) \), where \( x \) is
a new variable. Existentially binding the variable \( x \) yields a logical equivalence:

\[
(\alpha \lor (\beta \land \gamma)) \approx \exists x((\alpha \lor \neg x) \land (\beta \lor x) \land (\gamma \lor x)).
\]

This is using a so-called Tseitin encoding [14], to create a satisfiability equivalent propositional formula, before existentially binding the new variable. More details about how to encode propositional formulas in this way are given, for example, by Prestwich in [15].

This method can be used to convert any quantified Boolean formula in prenex normal form to conjunctive normal form. Let \( \Phi = Q\phi \) be a QBF in prenex normal form, where \( Q \) is the prefix and \( \phi \) is the matrix. The procedure described above can then be applied to the matrix \( \phi \), yielding a formula \( \exists x_1 \ldots \exists x_n \psi \), where \( x_1, \ldots, x_n \) are the introduced variables, and \( \psi \) is a propositional formula in conjunctive normal form. This new formula is logically equivalent to the original matrix: \( \phi \approx \exists x_1 \ldots \exists x_n \psi \). Therefore, we have: \( \Phi \approx Q\exists x_1 \ldots \exists x_n \psi \).

This new formula will have a length that is linear in the length of \( \Phi \), since every subformula of the form \( (\alpha \lor (\beta \land \gamma)) \) in \( \Phi \) will cause three new literals to be added.

2.1.3 Variable Levels and Definitions for Literals

Now that we have introduced the conjunctive normal form, we can introduce some definitions from [13] for any QBF in this form: The \textit{level} of a variable \( z_i \) is one plus the number of expressions \( Q_j z_j Q_{j+1} z_{j+1} \), where \( Q_j \neq Q_{j+1} \) and \( j \geq i \), occurring after \( \forall z_i \) or \( \exists z_i \) in the prefix. This means that the variable with the rightmost quantified occurrence has a level of 1, and that when moving left through the prefix until the quantifier type changes, all variables encountered will have level 1. After this, all variables will have level 2 until the quantifier type changes, and then level 3, and so on. We also introduce the following notation and definitions for any literal \( l \):

- \( |l| \) is the variable occurring in \( l \).
- \( \neg l \) is \( \neg l \) if \( l \) is a variable, and \( |\neg l| \) if \( l \) is the negation of a variable.
- We say that \( l \) is \textit{existential} if and only if \( \exists |l| \) occurs in the prefix.
- We say that \( l \) is \textit{universal} if and only if \( \forall |l| \) occurs in the prefix.
- For any QBF \( \Phi \) in conjunctive normal form, we define the assignment of the literal \( l \) in \( \Phi \), denoted by \( \Phi_l \), as the QBF where:

1. All clauses containing \( l \) have been removed from the matrix, since those clauses are true when \( l \) is.
2. All occurrences of \( \neg l \) have been removed from all clauses, since those literals are false when \( l \) is true.
3. The occurrence of \( \forall |l| \) or \( \exists |l| \) has been removed from the prefix, since no non-quantified occurrences of \( |l| \) remain.

This effectively assigns \( l \) as true and performs some basic simplifications.

2.2 QBF Solving

A \textit{QBF solver} is a software tool that determines whether a given QBF is satisfiable or not. Three approaches to QBF solving are introduced here: naive
expansion-based solving, search-based solving, and variable elimination-based solving.

2.2.1 A Naive Solver

A straightforward way to determine whether a quantified Boolean formula is satisfiable is to expand all of the quantifiers. We assume that the formula \( \Phi \) that we solve is closed. If it is not, then we can solve the existential closure of \( \Phi \) instead, since any non-closed formula is satisfiability equivalent to its existential closure.

Let \( \Phi = Qz\Phi' \) be a closed quantified Boolean formula where \( Q \in \{\forall, \exists\} \), \( z \) is a variable, and \( \Phi' \) is a subformula. Additionally, let \( \Phi'[1/z] \) and \( \Phi'[0/z] \) be the subformula \( \Phi' \) with all free occurrences of \( z \) substituted by 1 and 0, respectively. Then, we have \( \Phi = Qz\Phi' \approx (\Phi'[1/z] \land \Phi'[0/z]) \) if \( Q = \forall \) and \( \Phi = Qz\Phi' \approx (\Phi'[1/z] \lor \Phi'[0/z]) \) if \( Q = \exists \). This fact can be used to recursively rewrite \( \Phi \) until no quantifiers remain, obtaining \( \Phi_{\text{exp}} \), where \( \Phi_{\text{exp}} \approx \Phi \) and \( \Phi_{\text{exp}} \) is a propositional formula without any variables.

Evaluating \( \Phi_{\text{exp}} \) will, therefore, show \( \Phi \approx 1 \) or \( \Phi \approx 0 \), which implies \( \Phi \approx_{\text{sat}} 1 \) or \( \Phi \approx_{\text{sat}} 0 \). This decides if \( \Phi \) is satisfiable or not, since 1 is trivially satisfiable and 0 is trivially unsatisfiable. The main drawback of this approach is that the size of the formula will grow exponentially with the number of quantifiers in the original one.

Example. To clarify how the solver works, we will solve Formula 1:

\[
\forall y \exists x_1 \exists x_2 ((\neg y \lor x_1) \land (y \lor \neg x_1) \land (\neg y \lor x_2) \land (y \lor \neg x_2) \land (\neg x_1 \lor \neg x_2 \lor y)) \tag{1}
\]

Expanding all quantifiers yields a propositional formula without any variables:

\[
(((\neg 1 \lor 1) \land (1 \lor \neg 1) \land (\neg 1 \lor 1) \land (1 \lor \neg 1) \land (\neg 1 \lor \neg 1)) \\
(\neg 1 \lor 1) \land (1 \lor \neg 0) \land (1 \lor \neg 0) \land (\neg 1 \lor \neg 0) \land (\neg 1 \lor \neg 1) \\
(\neg 1 \lor 0) \land (1 \lor \neg 0) \land (\neg 1 \lor 0) \land (1 \lor \neg 0) \land (\neg 0 \lor \neg 1) \\
(\neg 1 \lor 0) \land (1 \lor \neg 0) \land (\neg 1 \lor 0) \land (1 \lor \neg 0) \land (\neg 0 \lor \neg 1) \\
(\neg 0 \lor 1) \land (0 \lor \neg 1) \land (\neg 0 \lor 1) \land (0 \lor \neg 1) \land (\neg 1 \lor \neg 1) \\
(\neg 0 \lor 0) \land (0 \lor \neg 0) \land (\neg 0 \lor 0) \land (0 \lor \neg 0) \land (\neg 1 \lor \neg 0) \\
(\neg 0 \lor 0) \land (0 \lor \neg 0) \land (\neg 0 \lor 0) \land (0 \lor \neg 0) \land (\neg 0 \lor 0))
\]

This formula evaluates to true, showing that Formula 1 is satisfiable.

2.2.2 Assumptions About Input

For the sake of simplicity, we will, for the rest of Section 2.2, assume that all quantified Boolean formulas are closed, cleansed, and in conjunctive normal form. This assumption is motivated by:

1. The satisfiability equivalence between formulas and their existential closures, which is mentioned at the end of Section 2.1.1.

2. The availability of efficient methods for cleansing and normalizing any formula, which are briefly described in Section 2.1.2.
2.2.3 Simplifying QBFs

When solving a quantified Boolean formula, it is beneficial to be able to simplify it. Some methods from [12] and [13] for simplifying a formula $\Phi$ are presented here:

1. All tautological clauses can be deleted from $\Phi$, yielding a logically equivalent formula.

2. If $\Phi$ contains a non-tautological clause containing only universal variables, then $\Phi$ is unsatisfiable.

3. Universal Reduction: If a non-tautological clause $\phi$ in $\Phi$ contains a universal literal $l$ and every existential literal in $\phi$ has a level higher than the level of $l$, then $l$ can be deleted from the clause, yielding a logically equivalent formula.

4. Unit Propagation: If there is a clause $\phi$ in $\Phi$ containing a single existential literal $l$, and all other literals in $l$ are universal and have a level lower than $l$, then $\Phi \approx \Phi[l]$. In this case $l$ is called a unit literal.

5. Monotone (or Pure) Literal Detection: If (i) there is a positive or negative existential literal $l$ such that $l$ occurs in $\Phi$ but $\neg l$ does not, or (ii) there is a positive or negative universal literal $l$ such that $\neg l$ occurs in $\Phi$ but $l$ does not, then $\Phi \approx \Phi[l]$. In this case $l$ is called a monotone literal.

2.2.4 Search-based Solving

By using a search-based approach to solving QBFs, it is possible to avoid the exponential growth seen in the naive solver. Instead of expanding a formula $\Phi = Qz\Phi'$ where $Q \in \{\forall, \exists\}$, we can use two recursive calls on $\Phi[z]$ and $\Phi[\neg z]$ to try both the case when $z$ is true and the case when $z$ is false; effectively using the equivalences $\forall y \Phi' \approx (\Phi'_{\forall y} \land \Phi'_{\neg y})$ and $\exists x \Phi' \approx (\Phi'_{\forall x} \lor \Phi'_{\neg x})$ to determine the satisfiability of $\Phi$.

A naive search-based solver can do this for each variable in the prefix, starting with the first one, until the matrix is reached. In each call, if the matrix is empty, then we return 1; and if the matrix contains an empty clause, then we return 0. This approach is used in the search-based algorithm in Listing 1.
∀y∃x1∃x2((¬y ∨ x1) ∧ (y ∨ ¬x1) ∧ (¬y ∨ x2) ∧ (y ∨ ¬x2) ∧ (¬x1 ∨ ¬x2 ∨ y))

A SEARCH Example. Figure 1 contains a tree showing all recursive calls made by the SEARCH algorithm when solving Formula 1. Each node is labeled with the QBF argument to the corresponding call; non-root nodes are also labeled with the assignment made for the corresponding call. The notation $z \mapsto b$ is used for the assignment of $b$ to $z$. The base cases of the recursion either contain an empty clause, denoted by 0, or have an empty matrix, denoted by 1. It is assumed that the or and and operators are short-circuiting.

The Q-DLL Algorithm. It is possible to improve the search procedure using unit propagation and monotone literal detection. Additionally, it is not necessary to always select the first variable, any literal at the top level can be selected, making it possible to use heuristics for the search order. Adding these improvements will yield the Q-DLL search algorithm in Listing 2, which is not formalized in this thesis, but is still included here to give a more complete introduction to QBF solving.

This algorithm is based on the one in [13], the main difference is that we are not storing the sequence of assignments to variables done during the search. This is not needed here, since we are not discussing how satisfiability and unsatisfiability proofs can be generated. Methods for generating satisfiability and unsatisfiability proofs, methods for learning clauses, disjunctions of literals that are in conjunction with the matrix, and methods for learning cubes, conjunctions of literals that are in disjunction with matrix, are all presented in [13].

Readers familiar with SAT-solving might recognize a similarity between the algorithm in Listing 2 and the well-known Davis-Putnam-Logemann-Loveland (DPLL) algorithm; originally introduced by Davis, Logemann, and Loveland in [16]. In fact, if there are only existential quantifiers in the formula $\Phi$, then the function Q-DLL effectively behaves like the DPLL algorithm. A more modern explanation of this algorithm, by Darwiche and Pipatsrisawat, can be found in [17].

A Q-DLL Example. Figure 2 contains a tree showing all recursive calls made by the Q-DLL algorithm when solving Formula 1. In addition to the notation used
function Q-DLL(Φ)
if (the matrix of Φ contains an empty clause) return 0;
if (the matrix of Φ is empty) return 1;
if (there is a unit literal l in Φ) return Q-DLL(Φ_l);
if (there is a monotone literal l in Φ) return Q-DLL(Φ_l);
let l := a literal at the highest level in Φ;
if (l is existential)
   return Q-DLL(Φ_l) or Q-DLL(Φ_{¬l});
else
   return Q-DLL(Φ_l) and Q-DLL(Φ_{¬l});

Listing 2: Q-DLL algorithm based on the one in [13].

∀y∃x_1∃x_2((¬y ∨ x_1) ∧ (y ∨ ¬x_1) ∧ (¬y ∨ x_2) ∧ (y ∨ ¬x_2) ∧ (¬x_1 ∨ ¬x_2 ∨ y))

Figure 2: Tree showing the recursive calls made by the Q-DLL algorithm (Listing 2) when solving Formula 1.

in Figure 1, recursive calls are labeled using “unit: z” when unit propagation is applied. As can be seen from the figure, the Q-DLL algorithm can, in this case, avoid some search by using unit propagation.

2.2.5 Variable Elimination-based Solving
Another way to approach solving, is to expand quantifiers like in the naive solver, but to do so in a more sophisticated way. A method for this from [13] are presented here. This approach is not formalized in this thesis, but is still included here to give a more complete introduction to QBF solving.

Instead of substituting 0 or 1 for variables, it is possible to use the equivalences ∀yΦ ≈ (Φ_y ∧ Φ_{¬y}) and ∃xΦ ≈ (Φ_x ∨ Φ_{¬x}) to expand a formula. It is also possible to use unit propagation and monotone literal detection to simplify the formula during solving. This will yield an algorithm that can be slightly more efficient than the naive solver. To improve it even further, it is possible to use more advanced strategies for expanding quantifiers.

Expanding Universal Variables. When expanding universal literals, it is possible to avoid duplicating clauses not in the minimal scope of the variable. The minimal scope of a universal variable y is the set of all clauses that contain
y or are \(y\)-connected to clauses in the minimal scope of \(y\). Two clauses \(C_1\) and \(C_2\) are \(y\)-connected if and only if there is an existential variable \(x\) occurring in both \(C_1\) and \(C_2\) where \(x\) has a lower level than \(y\).

To expand a universal variable \(y\), while avoiding duplication of clauses not in the minimal scope, the following procedure can be used:

1. For each variable in the minimal scope of \(y\) with a lower level than \(y\), add a new variable \(z'\).
2. For each new variable \(z'\), quantify \(z'\) in the same way and at the same level as the original variable \(z\).
3. For each clause \(C\) in the minimal scope of \(y\), substitute \(z\) with \(z'\) in \(C\) yielding a new clause \(C'\).
4. For each new clause \(C'\), delete \(C'\) if it contains \(\overline{y}\), otherwise delete \(y\) from \(C'\).
5. For each original clause \(C\) in the minimal scope of \(y\), delete \(C\) if it contains \(y\), otherwise delete \(\overline{y}\) from \(C\).

It is possible to compute how much the matrix will shrink or grow when expanding a universal variable. For this, we define:

- \(o(l)\) as the number of clauses containing \(l\).
- \(s(l)\) as the sum of the number of literals in all clauses containing \(l\).
- \(\text{minscope}(l)\) as the sum of the number of literals in all clauses in the minimal scope of \(l\).

When a universal variable \(y\) is expanded as described here, the length of the matrix will grow or shrink by the following amount:

\[
\text{minscope}(y) - (s(y) + s(\overline{y}) + o(y) + o(\overline{y}))
\]

**Eliminating Existential Variables.** In addition to expanding universally quantified variables, it is also possible to eliminate innermost existential variables. An existential variable is called an innermost variable if and only if it only occurs in clauses where all other variables have a level at least equal to the level of \(x\). To eliminate an innermost variable \(x\), every clause containing \(x\) is resolved with every clause containing \(\neg x\).

Treating clauses as sets of literals, this operation can be described using set terminology. Let \(S_x\) be the set of all clauses containing \(x\) and let \(S_\overline{x}\) be the set of all clauses containing \(\overline{x}\). The existential variable \(x\) is then eliminated by deleting all clauses in \(S_x\) and \(S_\overline{x}\) and then adding all clauses in:

\[
\{C \cup C' : C \cup \{x\} \in S_x, x \notin C, C' \cup \{\overline{x}\} \in S_\overline{x}, \overline{x} \notin C'\}
\]

When an existential variable \(y\) is eliminated as described here, the length change of the matrix is upper-bounded by:

\[
o(x) \times (s(\overline{x}) - o(\overline{x})) + o(\overline{x}) \times (s(x) - o(x)) - (s(x) + s(\overline{x}))
\]
function \text{Q-DP}(\Phi)
\begin{align*}
\text{if (the matrix of } \Phi \text{ contains an empty clause)} & \quad \text{return } 0; \\
\text{if (the matrix of } \Phi \text{ is empty) } & \quad \text{return } 1; \\
\text{if (there is a unit literal } l \text{ in } \Phi) & \quad \text{return } \text{Q-DP}(\Phi_l); \\
\text{if (there is a monotone literal } l \text{ in } \Phi) & \quad \text{return } \text{Q-DP}(\Phi_l); \\
\text{let } z := \text{a universal variable or an innermost existential} & \quad \text{variable in } \Phi \text{ with level } \leq 2; \\
\text{if (} z \text{ is existential)} & \quad \text{return } \text{Q-DP(eliminate-existential}(\Phi, z)); \\
\text{else} & \quad \text{return } \text{Q-DP(expand-universal}(\Phi, z));
\end{align*}

Listing 3: Q-DP algorithm based on the one in [13].

The \text{Q-DP} Algorithm. Using the expansion and elimination methods described above, it is possible to implement the QBF solving algorithm in Listing 3. Like for \text{Q-DLL}, the algorithm checks if the matrix is empty or if it contains an empty clause, and uses unit propagation and pure literal detection to simplify the formula.

In the algorithm, the requirement that the variable \( z \) should have a level of at most 2 is not necessary. However, based on the definitions of minimal scopes for universal variables and innermost existential variables, it is straightforward to see why preferring variables at a lower level is a reasonable heuristic. The \( y \)-connectivity of universal variables can only be propagated by variables at a lower level; and existential variables can only be prevented from being innermost by variables at a lower level.

Again, readers familiar with SAT-solving might recognize a similarity between the algorithm in Listing 3 and the resolution-based Davis-Putnam algorithm; originally introduced by Davis and Putnam in [18]. Analogously to \text{Q-DLL} and \text{DPPL}, if there only are existential quantifiers in \( \Phi \), then the function \text{Q-DP} will effectively behave like the Davis-Putnam algorithm. A more modern explanation of this algorithm can, again, be found in [17].

Example. To give an example of variable elimination-based solving, we will solve Formula 1 using the \text{Q-DP} algorithm. Assume that the variable \( y \) is selected to be eliminated. Expansion of the variable will then yield:
\[
\exists x_1 \exists x_2 \exists x'_1 \exists x'_2 ((x_1) \land (x_2) \land (\lnot x'_1) \land (\lnot x'_2) \land (\lnot x'_1 \lor \lnot x'_2))
\]
Which will be reduced to 1 by unit propagation on all remaining variables, showing that the formula is satisfiable.

2.3 Interactive Theorem Proving in Isabelle/HOL

Interactive Theorem Provers are computer programs that are used to write and formally verify mathematical theorems. They are used both to check that a proof is correct and to assist the user in the proof-writing process. There are many theorem provers, examples include the Lean Theorem Prover [19], Coq [20], and Isabelle/HOL [1]. This thesis uses the Isabelle/HOL interactive theorem prover.
2.3.1 Isabelle

*Isabelle* is a generic framework for implementing different logical formalisms, originally developed by Paulson [21], while *Isabelle/HOL* is a specialization of *Isabelle* implementing Higher-Order Logic, which is abbreviated as HOL. To ensure the correctness of proved theorems, Isabelle uses the approach introduced by Milner for the Edinburgh LCF theorem prover [22]. This approach uses an abstract data type \texttt{thm} to represent theorems. The constructors of the \texttt{thm} data type correspond to inference rules mapping theorems to other theorems. Constructing a theorem, represented as an instance of the \texttt{thm} type, will therefore perform all steps of a proof of the theorem, using the inference rules of the proof system. We will refer to the implementation of this \texttt{thm} type as the *core* of the theorem prover.

Altogether, this approach ensures the correctness of all proved theorems, up to the correctness of the implementation of the core. This provides a very high confidence in the correctness of the proofs. Moreover, this approach can enable different extensions to be built on top of the core, while preserving soundness. Even if there is an error in an extension built on top of the core, it will not be possible to construct an instance of \texttt{thm} corresponding to an incorrect proof.

2.3.2 Basic Isabelle/HOL concepts

Higher-Order Logic can, among other things, express many common functional programming and logic concepts. Some basic concepts from [1, ch. 1–2] are introduced here.

**Theories.** In Isabelle/HOL, a theory is a collection of definitions for types, functions, theorems, and more. A theory can extend other theories to include all definitions from the extended theories. The theory \texttt{Main} includes many commonly used definitions, including definitions for lists, arithmetic, recursive functions, algebraic datatypes, and more. In this thesis, it is always assumed that \texttt{Main} has been imported.

**Types.** The type system in Isabelle/HOL is similar to that of many functional programming languages and includes:

- **Base types** like Booleans which are represented by the type \texttt{bool}, and natural numbers which are represented by the type \texttt{nat}.
- **Type constructors** like \texttt{list} and \texttt{set}, which are used to construct other types. Postfix notations is used for this, e.g. \texttt{bool list} is used to represent a list of booleans.
- **Function types** which represent total functions and are denoted by expressions like $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ where $\tau_i$ are other types. Function arrows associate to the right, meaning that $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ is the same as $(\tau_2 \Rightarrow \tau_3) \Rightarrow \tau_1$.
- **Type variables** denoted by $'a$, $'b$, etc., which allow for polymorphic types like $'a \Rightarrow 'a$ or $'a \Rightarrow 'a list \Rightarrow bool$. 

22
Terms. Terms are built from constants, variables, function applications, and lambda-abstractions. Applying a function $f$ of type $\tau_1 \Rightarrow \tau_2$ to a term $x$ of type $\tau_1$ will yield a term $f \, x$ of type $\tau_2$. Infix functions, like the function $+$, and some functional programming constructs, like if ... then ... else ..., are also supported. A lambda-abstraction, denoted by $\lambda x. t$, is a function taking an argument $x$ and returning the term $t$, which may depend on $x$.

Formulas. Formulas are terms of type $\text{bool}$. The constants True and False exist, as do the common logical connectives like $\rightarrow$, $\land$, $\lor$, etc., and the quantifiers $\forall$ and $\exists$.

Variables. Isabelle differentiates between free and bound variables, as is usually done in logic. In addition to free and bound variables, Isabelle also has schematic variables. Schematic variables or unknown variables are free variables that may be instantiated by a term during the proof process. Schematic variables are prefixed with a question mark, e.g. $?x$ is a schematic variable, and therefore also a free variable, and $?x = $?x is a theorem using that schematic variable.

Inner and Outer Syntax. In Isabelle/HOL sources, some types and formulas are enclosed in quotation marks ("..."). An example of this can be seen in Listing 4. This is done to differentiate between the generic Isabelle syntax, often referred to as the outer syntax, and the HOL syntax, often referred to as the inner syntax. Everything that is a part of the inner syntax is enclosed in quotation marks, except individual identifiers.

2.3.3 Isabelle/HOL Datatypes and Functions

In Isabelle/HOL, algebraic datatypes can be defined using the datatype keyword, and recursive function can be defined using the fun keyword [23]. Listing 4 contains a definition for an algebraic datatype used to represent binary trees, and a definition for a recursive function that will reverse the order of the nodes in the tree. The datatype 'a tree is polymorphic with respect to the type variable 'a and has two constructors, namely Leaf and Node. The function tree_rev uses pattern matching to differentiate between leaves and nodes. If the argument is a node, then the function calls itself recursively two times, otherwise, if the argument is a leaf, there is no recursive call. The contents of Listing 4 should look familiar to readers who know some functional programming.

Function Termination. If a recursive function is defined using fun, then Isabelle will prove that it has the following two properties: 1. It is total. That is, it is defined for all possible values of the arguments. 2. It is terminating. That is, it will terminate after some finite amount of recursive calls for all possible inputs. If Isabelle does not manage to prove these properties, then the function definition will be considered erroneous and will be rejected. More complicated functions might need to be defined using the keyword function instead, which provides the user with the ability to manually specify how these properties should be proved [24].
datatype 'a tree = Leaf | Node 'a tree 'a 'a tree

fun tree_rev :: 'a tree ⇒ 'a tree where
"tree_rev Leaf = Leaf" |
"tree_rev (Node l x r) = Node (tree_rev r) x (tree_rev l)"

Listing 4: Isabelle/HOL definition of a tree datatype and a recursive function reversing the order of the nodes.

code_print_tree

theorem rev_rev_tree: "tree_rev (tree_rev t) = t"
apply (induction t)
apply simp
apply simp
done

Listing 5: Isabelle/HOL proof that reversing a tree twice will yield the original tree.

2.3.4 Basic Isabelle/HOL Proofs

Basic Isabelle/HOL proofs are introduced here, including Isar proofs, which are introduced in Section 2.3.5. For more in-depth information about Isabelle/HOL proof-writing, see, for example, Tobias Nipkow’s “Programming and Proving in Isabelle/HOL” [23].

Basic Proofs. Consider the definitions in Listing 4. The function tree_rev should have the property that, for all t, it holds that tree_rev (tree_rev t) = t. In Isabelle/HOL, this can be shown using induction on the structure of the tree datatype. A proof of this property, can be found in Listing 5.

The theorem keyword is used to specify a theorem, which is then followed by the (optional) name of the theorem, and then the theorem itself. After this we have the proof, which starts out by using apply (induction t) to specify that the induction proof method should be used on the variable t. After this apply simp is used twice to apply the simplification proof method twice. Finally, the proof is done, which is marked by the keyword done. To understand the details of this proof, we can inspect the proof state. Before introducing proof states, we first have to introduce meta-implication and meta-introduction.

Meta-implication and Meta-quantification. Isabelle proof states and theorems will often be stated using meta-implication, denoted by ⇒, and meta-quantification, denoted by ∃. They are part of the generic Isabelle framework rather than HOL in particular and have typical implication and universal quantification semantics. Meta-implication states that if the left hand side holds, then right hand side also holds; while meta-quantification states that something holds for all values of the correct type.

Proof States. A proof state is a list of subgoals that needs to be proven, in order to prove a theorem. The state after each step in the proof in Listing 5, as displayed by the Isabelle/jEdit IDE [25], is shown and explained here.
At the start of the proof, the proof state is simply the theorem itself:

\begin{verbatim}
proof (prove)
goal (1 subgoal):
  1. tree_rev (tree_rev t) = t
\end{verbatim}

After applying induction the original subgoal is replaced by two new subgoals, one for the base case and one for the inductive case:

\begin{verbatim}
proof (prove)
goal (2 subgoals):
  1. tree_rev (tree_rev Leaf) = Leaf
  2. \(\forall t1 x2 t2.\) tree_rev (tree_rev t1) = t1 \(\implies\) tree_rev (tree_rev t2) = t2 \(\implies\) tree_rev (tree_rev (Node t1 x2 t2)) = Node t1 x2 t2
\end{verbatim}

Applying the simplifier will then show that the first subgoal is true, removing it from the proof state:

\begin{verbatim}
proof (prove)
goal (1 subgoal):
  1. \(\forall t1 x2 t2.\) tree_rev (tree_rev t1) = t1 \(\implies\) tree_rev (tree_rev t2) = t2 \(\implies\) tree_rev (tree_rev (Node t1 x2 t2)) = Node t1 x2 t2
\end{verbatim}

The same can also be done for the remaining subgoal:

\begin{verbatim}
proof (prove)
goal: No subgoals!
\end{verbatim}

Since all subgoals were shown to be true, the theorem has been proved:

\begin{verbatim}
\end{verbatim}

As can be seen above, the theorem has a schematic variable \(?t\), which can be instantiated with any expression of the correct type. By instantiating this variable, the theorem can be used to show that reversing any tree twice will yield the original tree.

**A Shorter Proof.** There are some ways to make the proof in Listing 5 shorter. The following two ways of doing this are shown in Listing 6:

1. Instead of using the simp method twice, it is possible to use the auto method, which will use a collection of different strategies to try an automatically prove as many subgoals as possible. This is done in the proof for rev_rev_tree2. For this simple proof, the method is able to prove both subgoals after applying induction.

2. The keyword by can be used instead of the keywords apply and done. This is done in the proof for rev_rev_tree3. If by is used, then it is required that proof methods listed can finish the proof.
Listing 6: Two more Isabelle/HOL proofs that reversing a tree twice will yield the original tree.

2.3.5 Isar Proofs

Isar. As an alternative to apply-based proofs, it is also possible to write Isabelle/HOL proofs in the Isar language introduced by Wenzel in [26]. Isar, which abbreviates Intelligible semi-automated reasoning, is a language for writing formal proofs that is designed to be human-readable and machine-checkable, allowing the writer to explicitly express the interesting parts of the proof, while leaving other parts of the proof to the machine.

An Isar proof consists of a proof...qed-block containing the proof steps. The two main keywords are have, which is used for intermediate results; and show, which is used when proving a subgoal. After a subgoal has been shown, the keyword next can be used to start the proof for the next one. When starting an Isar proof, the keyword proof can be followed by an initial proof method which replaces the theorem with one or more subgoals. If no initial proof method is specified, an attempt will be made to select one automatically.

Isar Example. An example Isar proof can be found in Listing 7. The proof shows, again, that reversing a tree twice will yield the original tree. It starts by using induction on t as the initial proof method, after which the Leaf case is shown first, which is done using simplification. Next, the Node case is shown. The variables l, x, and r representing the left subtree, the value stored in the node, and the right subtree, respectively, are introduced. Simplification and the induction hypothesis Node.IH is used to show that reversing the left subtree twice yields the original one. The same is also shown for the right subtree. These intermediate results are then used to show that reversing the node twice will yield the original one, concluding the proof.

The keywords from and using are both used to specify some theorems that should be used as assumptions when proving some fact. They fill the same function, the main difference is simply that from is used before the fact to be proven while using is used after. Both of them are used in the example. Intermediate results can also be given labels, which also can be seen in the example where the intermediate results are labeled 1 and 2.

More Advanced Isar Features. Isar also includes many more advanced features that can be used in proofs. Some examples, all of which are used in Listing 11, are briefly explained here:

- Definitions introduced using definition are similar to functions, but are not expanded automatically in proofs, making them more suitable for
theorem rev_rev_tree_isar: "tree_rev (tree_rev t) = t"
proof (induction t)
  case Leaf
  show "tree_rev (tree_rev Leaf) = Leaf" by simp
next
  case (Node l x r)
  from Node.IH have 1: "tree_rev (tree_rev l) = l" by simp
  from Node.IH have 2: "tree_rev (tree_rev r) = r" by simp
  show "tree_rev (tree_rev (Node l x r)) = Node l x r"
    using 1 2 by simp
qed

Listing 7: Isar proof that reversing a tree twice will yield the original tree.

representing abstract concepts. The keyword unfolding can be used for this.
• The identifier this can be used to refer to the fact or theorem immediately preceding the current one.
• By using obtain, it is sometimes possible to obtain some variable satisfying a specified property.
• The keyword hence abbreviates from this have while thus abbreviates from this show.
• If more than one intermediate result is required for some proof step, then the keyword moreover can be used to gather multiple results, and then the keyword ultimately can be used to use all the gathered results. In the example, two results are gathered before they are used with ultimately.
• The special variable ?thesis represents the expression that is currently being proved.

3 Naive Solver Implementation and Verification

This section introduces the Isabelle/HOL formalization of the naive solver introduced in Section 2.2.1 and the correctness proof for the solver. A brief and informal overview of the main definitions, lemmas, and theorems can be found in Section 3.1. The remaining subsections provide more details about the formalization, including formal statements of some of the more important definitions, theorems, and proofs; explanations for some of the proofs and some design decisions are also provided. Section 3.2 is about the basic QBF datatype and the semantics, Section 3.3 is about the existential closure, while Section 3.4 is about the naive solver itself. For the full Isabelle/HOL formalization, see Appendix A.

3.1 Informal Overview of Main Isabelle/HOL Definitions and Theorems

A list of informal summaries of some of the main definitions and theorems, together with references to the corresponding formalizations, can be found here:
Definition 3.1: QBF Datatype. An Isabelle/HOL algebraic datatype based on the QBF Syntax is used to represent QBFs (QBF, Listing 8, Section 3.2.1).

Definition 3.2: Semantics and Satisfiability. The semantics are formalized as an Isabelle/HOL function taking two arguments: an interpretation mapping variables to booleans, and a QBF (qbf_semantics, Listing 8, Section 3.2.2); satisfiability is formalized as the existence of an interpretation making the semantics true (satisfiable, Listing 8, Section 3.2.4).

Definition 3.3: Free Variables. Free variables are formalized by an Isabelle/HOL function traversing a QBF formula to find all variables with a free occurrence (free_variables, Listing 9, Section 3.3.1).

Definition 3.4: Existential Closure. The existential closure of a QBF is defined using an Isabelle/HOL function adding one existential quantifier to the front of a QBF for each free variable (existential_closure, Listing 10, Section 3.3.2).

Theorem 3.1: Existential Closure Properties. The existential closure of a formula has the following properties:

1. It is satisfiable if and only if the original formula is (sat_iff_ex_close_sat, Listing 12, Section 3.3.4).
2. It does not have any free variables (ex_closure_no_free, Listing 13, Section 3.3.5).

Definition 3.5: Quantifier Expansion. Quantifier expansion is defined as an Isabelle/HOL function expanding all quantifiers using the equivalences ∀yΦ ≈ (Φ[0/y] ∧ Φ[1/y]) and ∃xΦ ≈ (Φ[0/x] ∨ Φ[1/x]) (expand_quantifiers, Listing 15, Section 3.4.1).

Definition 3.6: Formula Expansion. Formula expansion is defined as an Isabelle/HOL function expanding a QBF using the existential closure and quantifier expansion (expand_qbf, Listing 18, Section 3.4.2).

Lemma 3.1: Formula Expansion Properties. The expansion of a formula has the following properties (Listing 18, Section 3.4.2):

1. It is satisfiable if and only if the formula is satisfiable before expansion (sat_iff_expand_qbf_sat).
2. It does not have any free variables (expand_qbf_no_free).
3. It does not contain any quantifiers (expand_qbf_no_quants).

Definition 3.7: Formula Evaluation. The valuation of a formula without any free variables or quantifiers is defined using an Isabelle/HOL function (eval_qbf, Listing 19, Section 3.4.3).

Lemma 3.2: Formula Evaluation Correctness. The formula evaluation is correct, meaning that it yields the value true if and only if the formula is true (eval_qbf_implements_semantics, Listing 19, Section 3.4.3).

Definition 3.8: Naive Solver. The naive solver described in Section 2.2.1 is formalized as an Isabelle/HOL function using formula expansion and evaluation (naive_solver, Listing 20, Section 3.4.4).
Theorem 3.2: Naive Solver Correctness. The naive solver is correct, yielding the value true if and only if the formula is satisfiable (naive_solver_correct, Listing 20, Section 3.4.4).

3.2 QBF Datatype, Semantics, and Satisfiability

This subsection formalizes a couple of basic QBF concepts from Section 2.1.1 in Isabelle/HOL. An algebraic datatype based on the syntax is defined, the semantics are formalized as a function, and satisfiability is formalized using a definition. All three formalizations can be seen in Listing 8.

3.2.1 A QBF datatype

In the QBF datatype, there is one constructor for each case in the syntax definition. Variables are represented using the Var constructor, which represents variables using a natural number. The Neg constructor makes it possible to represent the negation of any QBF. A conjunction of zero or more QBFs is represented by the Conj constructor, which has a QBF list parameter representing all of the conjuncts. Disjunctions are represented in the same way by the Disj constructor. The Ex constructor represents existential quantification; the first parameter represents the quantified variable, while the second one is the scope of the quantified occurrence of the variable. Finally, the All constructor represents universal quantification in the same way. This is intended to make it easy to represent any syntactically valid QBF using the QBF datatype.

3.2.2 Formalization of Semantics

The semantics are formalized using a function taking an interpretation and a QBF as parameters, where interpretations are formalized as functions from nat to bool. With this definition, a partial application qbf_semantics I can be seen as an extension of domain of the interpretation I to include all QBFs; which is how the semantics were defined in Section 2.1.1.

For variables, the semantics will simply look up the value of the variable in I, which is always possible since Isabelle/HOL functions are total. The semantics of a negation is simply the negation of the semantics of the negated QBF. Conjunctions are a little more complicated; in this case list_all is used to check if the domain-extended interpretation qbf_semantics I is true for all conjuncts. Disjunctions are formalized in a very similar way; the difference is that list_ex is used to check if qbf_semantics I is true at least once. From this it follows that the empty conjunction (abbreviated by 1) is true, while the empty disjunction (abbreviated by 0) is false; as they should be according to Section 2.1.1.

Having a way of representing 0 and 1 as QBFs enables the implementation of substitute_var, which substitutes all free occurrences of a variables by 0 or 1. This can then be used to formalize the semantics for the quantifiers; which, according to Section 2.1.1, are defined using substitution of all free occurrences of the quantified variables by 0 or 1. This completes the formalization of the semantics, but we still have to show that the function terminates, since no termination proof is found automatically.
datatype QBF = Var nat | Neg QBF | Conj "QBF list" | Disj "QBF list" | Ex nat QBF | All nat QBF

fun substitute_var :: " nat ⇒ bool ⇒ QBF ⇒ QBF " where
  "substitute_var z True (Var z') =
    (if z = z' then Conj [] else Var z')" |
  "substitute_var z False (Var z') =
    (if z = z' then Disj [] else Var z')" |
  "substitute_var z b (Neg qbf) =
    Neg (substitute_var z b qbf)" |
  "substitute_var z b (Conj qbf_list) =
    Conj (map (substitute_var z b) qbf_list)" |
  "substitute_var z b (Disj qbf_list) =
    Disj (map (substitute_var z b) qbf_list)" |
  "substitute_var z b (Ex x qbf) =
    Ex x (if x = z then qbf else substitute_var z b qbf)" |
  "substitute_var z b (All y qbf) =
    All y (if z = y then qbf else substitute_var z b qbf)"

function qbf_semantics :: "(nat ⇒ bool) ⇒ QBF ⇒ bool " where
  "qbf_semantics I (Var z) = I z" |
  "qbf_semantics I (Neg qbf) = (¬(qbf_semantics I qbf))" |
  "qbf_semantics I (Conj qbf_list) =
    list_all (qbf_semantics I) qbf_list" |
  "qbf_semantics I (Disj qbf_list) =
    list_ex (qbf_semantics I) qbf_list" |
  "qbf_semantics I (Ex x qbf) =
    ((qbf_semantics I (substitute_var x True qbf))
     ∨ (qbf_semantics I (substitute_var x False qbf)))" |
  "qbf_semantics I (All x qbf) =
    ((qbf_semantics I (substitute_var x True qbf))
     ∧ (qbf_semantics I (substitute_var x False qbf)))"
by pat_completeness auto
termination
  apply (relation "measure (λ(I,qbf). qbf_measure qbf)")
  by (auto simp add: qbf_measure_substitute
       qbf_measure_lt_sum_list)
definition satisfiable :: "QBF ⇒ bool" where
  "satisfiable qbf = (∃I. qbf_semantics I qbf)"

Listing 8: Isabelle/HOL definitions for a generic QBF datatype (Definition 3.1), substitution of true or false for free variables, the semantics for this datatype, and satisfiability (Definition 3.2). All of these definitions are based on Section 2.1.1. The QBF datatype is designed to make it easy to represent any syntactically valid QBF, while the qbf_semantics function is designed to match the informal description of the semantics as closely as possible.
3.2.3 Termination of Semantics

Termination is shown by showing that the number of constructors of the QBF datatype in the second parameter decreases each recursive call. Crucially, list constructors are ignored. This is done using the relation proof method, that constructs subgoals for showing that a relation is well-founded and that arguments to recursive calls are decreasing with respect to the relation; the measure function, which constructs such a relation from a measure function mapping arguments to natural numbers; and the qbf_measure function, which works by recursively counting the number of constructors in a QBF. For more details about termination proofs, see [24].

To show that the arguments to recursive calls decrease, two lemmas are needed. The lemma qbf_measure_substitute shows that qbf_measure is invariant under variables substitution. Formally, the lemma states the following: qbf_measure (substitute_var z b qbf) = qbf_measure qbf. The second lemma, qbf_measure_lt_sum_list, states that qbf_measure qbf < Suc (sum_list (map qbf_measure qbf_list)) holds under the assumption qbf ∈ set qbf_list. That is, the value of qbf_measure of a QBF in a list, say qbf_list, is less than the successor of the sum of the value of qbf_measure for all QBFs in qbf_list. The successor function Suc is the function yielding the next natural number; the function adds one to a natural number. Using the first lemma, it can be shown that the measure decreases for Ex and All; and using the second lemma, it can be shown that the measure decreases for Conj and Disj.

3.2.4 Formalization of Satisfiability

In Section 2.1.1, we defined a QBF as satisfiable if and only if it is true under some truth assignment. With the formalization of the semantics, the definition of satisfiability can be expressed in Isabelle/HOL as ∃I. qbf_semantics I qbf. This is used in the Isabelle/HOL definition satisfiable.

Logical Equivalence and Satisfiability Equivalence: With the definitions of qbf_semantics and satisfiable, it is possible to express logical equivalences and satisfiability equivalences in Isabelle. A logical equivalence, say Φ₁ ≈ Φ₂, can be expressed as qbf_semantics I phi₁ = qbf_semantics I phi₂; while the corresponding satisfiability equivalence, which is Φ₁ ≈ sat Φ₂, can be expressed as satisfiable phi₁ = satisfiable phi₂.

3.3 Existential Closure

There are three main steps in the formalization of the existential closure. First, free variables are formalized. After this, the existential closure is formalized. Finally, the satisfiability equivalence between any QBF and its existential closure is shown to hold.
fun free_variables_aux :: "nat set ⇒ QBF ⇒ nat list" where
"free_variables_aux bound (Var x) = (if x ∈ bound then [] else [x])" |
"free_variables_aux bound (Neg qbf) = free_variables_aux bound qbf" |
"free_variables_aux bound (Conj list) = concat (map (free_variables_aux bound) list)" |
"free_variables_aux bound (Disj list) = concat (map (free_variables_aux bound) list)" |
"free_variables_aux bound (Ex x qbf) = free_variables_aux (insert x bound) qbf" |
"free_variables_aux bound (All x qbf) = free_variables_aux (insert x bound) qbf"

fun free_variables :: "QBF ⇒ nat list" where
"free_variables qbf = sort (remdups (free_variables_aux {} qbf))"

lemma bound_subtract_equiv:
"set (free_variables_aux (bound ∪ new) qbf) = set (free_variables_aux bound qbf) - new"
by (induction bound qbf rule: free_variables_aux.induct) auto

Listing 9: Formalization of the free variables of a QBF formula (Definition 3.3). The function free_variables_aux traverses the formula, creating a list with one entry for each occurrence of a free variable in the QBF; the function free_variables removes duplicates and sorts the result to make it easier to use. The lemma bound_subtract_equiv shows that treating variables as bound when computing the set of free variables is equivalent to subtracting them from the set.

3.3.1 Formalisation of Free Variables
Free variables have been formalized using the functions free_variables_aux and free_variables. The definitions of both of these functions can be found in Listing 9. The function free_variables_aux has two parameters, a set of variables that are considered to be bound in the QBF, and the QBF whose free variables should be returned; and returns a list containing one entry for each occurrence of a variables in the QBF. To make the formalization easier to work with, we also have the function free_variables; which assumes that no variables are considered to be bound at the top level, and returns a sorted list without any duplicates.

Representation of Free Variables: The function free_variables returns a list instead of a set, despite sets being a more natural choice for representing the free variables of a formula. There is not, for example, any obvious ordering for the free variables, while a list requires the elements to be ordered in some way; we also want to avoid duplicate elements, which sets do by definition.

This decision, to return lists, rather than sets, is motivated by lists being defined as algebraic data types, unlike sets; which makes it easy to define recur-
fun existential_closure_aux :: "QBF ⇒ nat list ⇒ QBF" where
  "existential_closure_aux qbf Nil = qbf" |
  "existential_closure_aux qbf (Cons x xs) = Ex x (existential_closure_aux qbf xs)"

fun existential_closure :: "QBF ⇒ QBF" where
  "existential_closure qbf = existential_closure_aux qbf (free_variables qbf)"

Listing 10: Formalization of the existential closure for QBF formulas (Definition 3.4). The function existential_closure_aux adds a quantifier for each variable in a list, while the function existential_closure passes the list of free variables as the argument to the former.

The Bound Subtract Equivalence: With the formalization of free variables, we can show an important fact about the set of all variables returned by free_variables_aux. If any new variables are added to the set of bound variables, this has the same effect, on the set of returned variables, as subtracting the same new variables from the set of returned variables. This is formalized as the lemma bound_subtract_equiv in Listing 9, and will be useful for several proofs involving free variables.

3.3.2 Formalisation of Existential Closure

The existential closure of a QBF is defined by the functions existential_closure and existential_closure_aux, both which can be found in Listing 10. To compute the existential closure of a QBF, the function existential_closure uses the function existential_closure_aux, and the formalization of free variables described above.

Since free_variables yields a list with one entry for each free variables, it is straightforward to compute the existential closure using a recursive function. This is done by existential_closure_aux, which adds existential quantifiers, one for each entry in the second argument, to the front of a QBF. Using this function, it is easy to compute the existential closure by passing the list of free variables as an argument, which is done by existential_closure.

3.3.3 Preservation of Satisfiability under Existential Quantification

The lemma sat_iff_ex_sat, shown in Listing 11, shows that satisfiability is preserved when adding an existential quantifier to the front of a QBF. That is, it shows that $\Phi \approx_{sat} \exists x \Phi$ for all $x$ and $\Phi$. Formally, the lemma states that the following equivalence holds: satisfiable $\Phi$ ↔ satisfiable ($\exists x \Phi$). Proving this is the first step towards proving that satisfiability is preserved when taking the existential closure of a QBF.

The proof of this uses another lemma, qbf_semantics_substitute_eq_assign, which is also shown in Listing 11. This lemma shows that, as far as the semantics are concerned, substituting true (respectively false) for all free occurrences
lemma qbf_semantics_substitute_eq_assign: 
"qbf_semantics I (substitute_var x b qbf) 
↔ qbf_semantics (I(x := b)) qbf"
proof (induction "I(x := b)" qbf rule: qbf_semantics.induct)

... 
qed

lemma sat_iff_ex_sat: 
"satisfiable qbf ↔ satisfiable (Ex x qbf)"
proof (cases "satisfiable qbf")
case True
from this obtain I where I_def : "qbf_semantics I qbf"
  unfolding satisfiable_def by blast
have "I(x := I x) = I(x := True) 
  ∨ I(x := I x) = I(x := False)"
  by (cases "I x") auto
  hence "I = I(x := True) ∨ I = I(x := False)" by simp
  hence "qbf_semantics (I(x := True)) qbf 
  ∨ qbf_semantics (I(x := False)) qbf"
  using I_def by fastforce
  moreover have "satisfiable (Ex x qbf) 
  = (∃I. qbf_semantics (I(x := True)) qbf 
  ∨ qbf_semantics (I(x := False)) qbf)"
    by (simp add: satisfiable_def 
    qbf_semantics_substitute_eq_assign)
  ultimately have "satisfiable (QBF.Ex x qbf)" by blast
  thus ?thesis using True by simp
next
case False
  thus ?thesis unfolding satisfiable_def 
    using qbf_semantics_substitute_eq_assign by simp
qed

Listing 11: Proof showing that a QBF is satisfiable if and only if it is satisfiable after adding an existential quantifier to the front. The lemma qbf_semantics_substitute_eq_assign shows that substituting all free occurrences of a variable for true or false has the same effect as assigning that variable in the interpretation, while the lemma sat_iff_ex_sat uses the former to show that arbitrary existential quantifiers can be added to the front of a QBF.
lemma sat_iff_ex_close_aux_sat:
"satisfiable qbf
leftrightarrow satisfiable (existential_closure_aux qbf vars)"
using sat_iff_ex_sat by (induction vars) auto

theorem sat_iff_ex_close_sat:
"satisfiable qbf \leftrightarrow satisfiable (existential_closure qbf)"
using sat_iff_ex_close_aux_sat by simp

Listing 12: Formalization of Theorem 3.1.1: a QBF is satisfiable if and only if the existential closure is satisfiable. The lemma sat_iff_ex_close_aux_sat shows that satisfiability is preserved by the function existential_closure_aux, while the theorem sat_iff_ex_close_sat shows that satisfiability is preserved by the function existential_closure.

of some variable has the same effect as updating the value of that variable in the interpretation to be true (respectively false). With this, we can show sat_iff_ex_sat using case analysis.

If a QBF is satisfiable, then there is, by definition, some interpretation I, where the semantics are true. Additionally, for any variable x, the semantics are true either when I x is true, when I x is false, or in both cases. Together with the definition of satisfiable and the lemma qbf_semantics_substitute_eq_assign, it follows that satisfiable qbf ↔ satisfiable (Ex x qbf) by simplification.

If a QBF is not satisfiable, then the semantics are false for all possible interpretations, which is not explicitly stated in the proof, but can be shown by simplification. Using this, the equality follows from the definitions of satisfiable, and qbf_semantics_substitute_eq_assign.

3.3.4 Preservation of Satisfiability under Existential Closure

With the lemma sat_iff_ex_sat, it is possible to show that satisfiability is preserved when taking the existential closure of a QBF. That is, if x₁, . . . , xₙ are all of the free variables in a QBF Φ, then Φ ≈ sat ∃x₁ . . . ∃xₙ Φ. This is formalized as the theorem sat_iff_ex_close_sat, which is shown in Listing 12.

The theorem is a special case of the lemma sat_iff_ex_close_aux_sat, which shows that satisfiability is preserved when adding any number of existential quantifiers to the front of a QBF using existential_closure_aux. This can be shown using the lemma sat_iff_ex_sat by induction on the list of variables to add quantifiers for. Then, the theorem can be shown by simplification.

3.3.5 Non-existence of Free Variables in Existential Closure

Finally, it is also important to show that the existential closure of a QBF does not have any free variables. This is done in Listing 13, using the lemma ex_closure_aux_vars_not_free, which shows that the set of free variables does not contain any variables with quantified occurrences added by the function existential_closure_aux; and the theorem ex_closure_no_free, which shows that the existential closure of a formula does not have any free variables.

The proof of the lemma uses induction on the list of variables to add quantifiers for. To show the inductive case, we use the bound subtract equivalence
lemma ex_closure_aux_vars_not_free:
"set (free_variables (existential_closure_aux qbf vars)) = set (free_variables qbf) - set vars"
proof (induction vars)
case Nil
  then show ?case by simp
next
case (Cons x xs)
  thus ?case using bound_subtract_equiv[of "{}" "{x}""]
    by auto
qed

theorem ex_closure_no_free:
"set (free_variables (existential_closure qbf)) = {}"
using ex_closure_aux_vars_not_free by simp

Listing 13: Formalization of Theorem 3.1.2: the existential closure of a QBF does not have any free variables. The bound_subtract_equiv lemma can be used to show how existential_closure_aux affects the set of free variables, which is formalized as the lemma ex_closure_aux_vars_not_free; which, then, can be used to show that no free variables remain after applying existential_closure, which is formalized as the theorem ex_closure_no_free.

from Section 3.3.1, instantiated with appropriate terms. After this, the theorem follows from the lemma.

3.4 Naive Solver

With the formalization of the existential closure, it is possible to solve any QBF using a naive expansion-based method. Such a method is used by the naive solver introduced in Section 2.2.1, which is formalized in this subsection.

3.4.1 Expanding Quantifiers

Here, quantifier expansion refers to the substitution of \((\Phi[0/y] \land \Phi[1/y])\) for \(\forall y \Phi\) and \((\Phi[0/x] \lor \Phi[1/x])\) for \(\exists x \Phi\). and is formalized by the expand_quantifiers function shown in Listing 15. The formalization uses a straightforward traversal of a QBF, replacing any occurrences of quantifiers when they are encountered, reusing the substitute_var function from Section 3.2.

Termination of Quantifier Expansion: A termination proof is required, since no such proof is discovered automatically. The proof is similar to the one for the semantics in Section 3.2, the main difference is that, here, it is shown that recursive calls decrease with respect to a lexicographic tuple, rather than a natural numbers. This tuple is constructed using the function qbf_quantifier_depth, which is described below; and the function qbf_measure from the termination proof of the semantics.

The quantifier depth of a QBF, defined by the function qbf_quantifier_depth in Listing 14, is used to show termination of expand_quantifiers. Viewing the QBF as a tree-shaped graph, the function measures the maximum number of
fun qbf_quantifier_depth :: "QBF ⇒ nat" where
"qbf_quantifier_depth (Var x) = 0" |
"qbf_quantifier_depth (Neg qbf) =
  qbf_quantifier_depth qbf" |
"qbf_quantifier_depth (Conj list) =
  list_max (map qbf_quantifier_depth list)" |
"qbf_quantifier_depth (Disj list) =
  list_max (map qbf_quantifier_depth list)" |
"qbf_quantifier_depth (Ex x qbf) =
  1 + (qbf_quantifier_depth qbf)" |
"qbf_quantifier_depth (All x qbf) =
  1 + (qbf_quantifier_depth qbf)"

Listing 14: Function measuring the maximum quantifier depth in a QBF formula. The quantifier depth is defined as the maximum number of quantifiers on any path from the root to a leaf in a (tree-shaped) QBF-formula.

function expand_quantifiers :: "QBF ⇒ QBF" where
"expand_quantifiers (Var x) = (Var x)" |
"expand_quantifiers (Neg qbf) =
  Neg (expand_quantifiers qbf)" |
"expand_quantifiers (Conj list) =
  Conj (map expand_quantifiers list)" |
"expand_quantifiers (Disj list) =
  Disj (map expand_quantifiers list)" |
"expand_quantifiers (Ex x qbf) =
  (Disj [substitute_var x True (expand_quantifiers qbf),
         substitute_var x False (expand_quantifiers qbf)])" |
"expand_quantifiers (All x qbf) =
  (Conj [substitute_var x True (expand_quantifiers qbf),
         substitute_var x False (expand_quantifiers qbf)])"

by pat_completeness auto
termination
apply (relation "measures [qbf_quantifier_depth,
               qbf_measure]")
by (auto simp add: qbf_quantifier_depth_substitute
              qbf_quantifier_depth_eq_max)

(auto simp add: qbf_measure_lt_sum_list)

lemma no_quants_after_expand_quants:
  "qbf_quantifier_depth (expand_quantifiers qbf) = 0"
proof (induction qbf)
  ... 
qed

Listing 15: Formalization of quantifier expansion (Definition 3.5) and a lemma showing that no quantifiers remain after expansion. The function expand_quantifiers recursively expands all quantifiers using the equivalences $\forall y \Phi \approx (\Phi[0/y] \land \Phi[1/y])$ and $\exists x \Phi \approx (\Phi[0/x] \lor \Phi[1/x])$. The lemma no_quants_after_expand_quants shows that the quantifier depth is zero after applying the function, which is equivalent to showing that no quantifiers remain.
Lemma semantics_inv_under_expand:
    "qbf_semantics I qbf = qbf_semantics I (expand_quantifiers qbf)"
proof (induction qbf arbitrary: I)
    ...
qed

Lemma sat_iff_expand_quants_sat:
    "satisfiable qbf ↔ satisfiable (expand_quantifiers qbf)"
unfolding satisfiable_def
using semantics_inv_under_expand by simp

Listing 16: Lemmas showing that semantics and satisfiability are preserved when quantifiers are expanded. The lemma semantics_inv_under_expand shows that the semantics are preserved, which implies the lemma sat_iff_expand_quants_sat showing that satisfiability also is preserved.

quantifiers on any path from the root to a leaf. Equivalently, the function measures the maximum number variables bound in any subformula of the argument.

Two new lemmas are also used in the termination proof. The first new lemma is qbf_quantifier_depth_substitute, which shows that the quantifier depth is invariant under substitute var. The second one is qbf_quantifier_depth_eq_max, which shows that, if the quantifier depth of a formula in a QBF list is not less than the maximum quantifier depth in the list, then the quantifier depth of the formula is equal to the maximum.

Non-existence of Quantifiers after Expansion: One important property of expand_quantifiers is that no quantifiers remain after expansion. This is formalized as the lemma no_quants_after_expand_quants in Listing 15. The proof is not shown because of its length, but it is a simple induction proof using qbf_quantifier_depth_substitute.

Preservation of Satisfiability under Quantifier Expansion: Another important property is that the semantics is preserved under quantifier expansion. That is, there is a logical equivalence between the original formula and the expanded one; this this can be shown by induction since \(\forall y \Phi \approx (\Phi[0/y] \land \Phi[1/y])\) and \(\exists x \Phi \approx (\Phi[0/x] \lor \Phi[1/x])\). Because of this, there should also be a satisfiability equivalence.

This is formalized as the lemma semantics_inv_under_expand for the logical equivalence and the lemma sat_iff_expand_quants_sat for the satisfiability equivalence. The former is shown by induction on the QBF using the lemma qbf_semantics_substitute_eq_assign, which is needed to make it possible to apply the induction hypothesis; while the latter is shown using the former and the definition of satisfiable by simplification. Both lemmas are shown in Listing 16.

Preservation of Free Variables under Quantifier Expansion: The final important property shown here is that the set of free variables are preserved under quantifier expansion. Two lemmas showing that substituting
lemma set_free_vars_subst_all_eq:
  "set (free_variables (substitute_var x b qbf))
  = set (free_variables (All x qbf))"
proof (induction x b qbf rule: substitute_var.induct)
  ...
qed

lemma set_free_vars_subst_ex_eq:
  "set (free_variables (substitute_var x b qbf))
  = set (free_variables (Ex x qbf))"
proof (induction x b qbf rule: substitute_var.induct)
  ...
qed

lemma free_vars_inv_under_expand_quants:
  "set (free_variables (expand_quantifiers qbf))
  = set (free_variables qbf)"
proof (induction qbf)
  ...
qed

Listing 17: Lemmas showing that the set of free variables is preserved under quantifier expansion. The lemmas set_free_vars_subst_ex_eq and set_free_vars_subst_all_eq show that the set of free variables is affected in the same way when adding a quantifier to the front as when substituting all free occurrences of a variables for true or false. These can be used to show the lemma free_vars_inv_under_expand_quants, which states that the set of free variables is invariant under quantifier expansion.
fun expand_qbf :: "QBF ⇒ QBF" where
"expand_qbf qbf =
  expand_quantifiers (existential_closure qbf)"

lemma sat_iff_expand_qbf_sat:
  "satisfiable (expand_qbf qbf) ↔ satisfiable qbf"
  using sat_iff_ex_close_sat sat_iff_expand_quants_sat
  by simp

lemma expand_qbf_no_free:
  "set (free_variables (expand_qbf qbf)) = {}"
proof -
  have "set (free_variables (expand_qbf qbf)) = set (free_variables (existential_closure qbf))"
    using free_vars_inv_under_expand_quants
    by simp
  thus ?thesis using ex_closure_no_free by metis
qed

lemma expand_qbf_no_quants:
  "qbf_quantifier_depth (expand_qbf qbf) = 0"
  using no_quants_after_expand_quants by simp

Listing 18: Formalization of formula expansion (Definition 3.6) as a function expanding any QBF together with the formalization of Lemma 3.1 as lemmas showing that satisfiability is preserved under expansion and that the expansion does not have any quantifiers or free variables.

all free occurrences of a variable affects the set of free variables in the same way as binding the variable with a quantifier are used in the proof, namely set_free_vars_subst_all_eq and set_free_vars_subst_ex_eq. Both of these can be found in Listing 17, and are shown using the bound subtract equivalence from Section 3.3.1, by induction. Using these two results together with, again, the bound subtract equivalence, it is possible to use induction to show the main lemma free_vars_inv_under_expand_quants.

3.4.2 Expanding Formulas

By combining the existential closure function from Section 3.3.2 and the quantifier expansion function from Section 3.4.1, any QBF can be expanded into a propositional formula without any variables, which we will call the expanded QBF. This is done by the expand_qbf function in Listing 18; which also contains three lemmas showing that the important properties of the existential closure and quantifier expansion are preserved under QBF expansion.

The lemma sat_iff_expand_qbf_sat shows that the expanded QBF is satisfiable if and only if the original QBF is; the lemma expand_qbf_no_free shows that the expanded QBF does not have any free variables; and finally, the lemma expand_qbf_no_quants shows that the expanded QBF does not have any quantifiers. Together, the latter two lemmas show that the expanded QBF is a propositional formula without any variables. Evaluating this formula will, because of the first lemma, show if the original QBF is satisfiable or not.
fun eval_qbf :: "QBF ⇒ bool option" where
  "eval_qbf (Var x) = None" |
  "eval_qbf (Neg qbf) = map_option (λx. ¬x) (eval_qbf qbf)" |
  "eval_qbf (Conj list) = map_option (list_all id) (sequence (map eval_qbf list))" |
  "eval_qbf (Disj list) = map_option (list_ex id) (sequence (map eval_qbf list))" |
  "eval_qbf (Ex x qbf) = None" |
  "eval_qbf (All x qbf) = None"

lemma eval_qbf_implments_semantics :
  assumes "set (free_variables qbf) = {}"
  and "qbf_quantifier_depth qbf = 0"
  shows "eval_qbf qbf = Some (qbf_semantics I qbf)"
  using assms
proof (induction qbf)
  ... qed

Listing 19: Formalization of formula evaluation (Definition 3.7) as a function
evaluating any QBF without any quantifiers or free variables together with
the formalization of Lemma 3.2 as the lemma eval_qbf_implments_semantics
showing that the evaluation implements the semantics.

3.4.3 Evaluating Expanded Formulas

Propositional formulas without any variables, represented as a QBF, can be eval-
uated by the function eval_qbf in Listing 19. The function returns an op-
tional Boolean value if the argument is a propositional formula without any
variables; this Boolean is true if and only if the formula evaluates to true ac-
cording to the qbf_semantics function. This fact is formalized by the lemma
eval_qbf_implments_semantics, which also is in Listing 19.

If eval_qbf encounters a variable or a quantifier, then it evaluates to None,
as these should not occur in variable-free propositional formulas. The case for
negation is straightforward, the map_option function is used to negate the result,
if and only if it is not None. For conjunctions, the sequence function is used.
This function has the type 'a option list ⇒ 'a list option, and returns None
if and only if there is a None in the list argument, otherwise it returns a list
containing the same elements as the original one, in the same order. After this,
the function list_all id is mapped over the result to check if all elements are
ture. The case for disjunctions is implemented in a similar way.

The proof of eval_qbf_implments_semantics is an inductive proof where all
cases except Conj and Disj are immediately shown by the simplifier. These two
cases require a lot of intermediate steps to guide the simplifier, and induction to
deal with the list of subformulas, but are otherwise relatively straightforward.
With this final lemma, it is possible to implement and show the correctness of
the naive solver.

3.4.4 Naive Solver

The naive solver is implemented as the function naive_solver in Listing 20.
fun naive_solver :: "QBF ⇒ bool" where
"naive_solver qbf = the (eval_qbf (expand_qbf qbf))"

theorem naive_solver_correct:
"naive_solver qbf ↔ satisfiable qbf"
proof -
  have "∀I. naive_solver qbf = the (Some (qbf_semantics I (expand_qbf qbf)))"
    using expand_qbf_no_free expand_qbf_no_quants eval_qbf_implements_semantics by simp
  hence "naive_solver qbf = satisfiable (expand_qbf qbf)"
  unfolding satisfiable_def by simp
  thus "naive_solver qbf = satisfiable qbf"
    using sat_iff_expand_qbf_sat by simp
qed

Listing 20: Formalization of the naive solver (Definition 3.8) as a function,
together with the formalization of Theorem 3.2, showing correctness.

4 Prenex Conjunctive Normal Form Datatype and QDIMACS Parser

QBF encodings of problems often use the QDIMACS format. The ability to read input in this format will, therefore, make it possible to use the verified solvers developed in this thesis to solve existing QBF encodings of problems.

Section 4.2 introduces the QDIMACS format. A prenex conjunctive normal form (PCNF) datatype based on the format is then introduced in Section 4.3. After this, a parser for the QDIMACS format, outputting the new datatype, is introduced in Section 4.4. A brief and informal overview of the main definitions and theorems can be found in Section 4.1. For the full Isabelle/HOL formalization of the datatype, see Appendix B; for the parser, see Appendix C.
4.1 Informal Overview of Main Isabelle/HOL Definitions and Theorems

A list of informal statements of some of the main definitions and theorems, together with references to the corresponding formalizations, can be found here:

Definition 4.1: PCNF Datatype. An Isabelle/HOL type for quantified Boolean formulas in prenex conjugate normal form has been defined based on the QDIMACS format (pcnf, Listing 22, Section 4.3).

Definition 4.2: Conversion Functions. Isabelle/HOL functions for converting between the generic QBF datatype and the PCNF type have been defined (convert and convert_inv, Listing 26, Section 4.3.2).

Theorem 4.1: Bijectivity of Conversion. The conversion function is a bijection between the PCNF type and the subset of the QBF datatype in prenex conjugate normal form (convert_bijective_on, Listing 32, Section 4.3.2).

Definition 4.3: PCNF Semantics. Semantics have been defined for the PCNF type (pcnf_semantics, Listing 33, Section 4.3.3).

Theorem 4.2: Conversion Preserves the Semantics. The conversion between the types preserves the semantics. Under a fixed interpretation, a quantified Boolean formula represented using the PCNF type is true, according to the PCNF semantics, if and only if it is true, according to the semantics for the generic QBF datatype, after conversion (qbf_semantics_eq_pcnf_semantics, Listing 34, Section 4.3.3).

Definition 4.4: QDIMACS Parser. An unverified parser for the QDIMACS input format has been defined using Isabelle/HOL functions (Section 4.4).
datatype literal = P nat | N nat
type_synonym clause = "literal list"
type_synonym matrix = "clause list"

type_synonym quant_set = "nat × nat list"
type_synonym quant_sets = "quant_set list"
datatype prefix = UniversalFirst quant_set quant_sets | ExistentialFirst quant_set quant_sets | Empty
type_synonym pcnf = "prefix × matrix"

Listing 22: Type definitions for representing QBFs in conjunctive normal form (Definition 4.1). Variables are represented by nat. Literals are represented as variables tagged as positive or negative, clauses as lists of literals, and matrices as lists of clauses. The prefix is represented as a non-empty sequence of non-empty sequences of variables, where the quantifier type is alternating.

4.2 QDIMACS Format

The QDIMACS format\(^2\) is a suggested standard for input to and output from QBF solvers. The grammar for the input format is shown in Listing 21; the output format is not used here. In the grammar, the terminal symbols EOF and EOL are end-of-file respectively end-of-line symbols; variables are represented by the <pnum> symbol; and literals are represented by the <num> symbol, where positive number represent positive literals and negative number represent negative literals.

Additional Constraints: In addition to the grammar, there are some additional constraints for valid QDIMACS instances, which are summarized here:

- In the <problem_line>, the first <pnum> corresponds to the number of variables in the instance, while the second one corresponds to the number of clauses.
- Variables in the prefix must occur in the matrix.
- Variables not in the prefix are to be considered existentially quantified.
- Contiguous quantified sets may not have the same quantifier type.
- A variable can only occur in the prefix once.
- The innermost quantified set is existential.

4.3 A Prenex Conjugate Normal Form Datatype

Listing 22 contains the definition of the Isabelle/HOL datatype pcnf for prenex conjunctive normal form formulas, which is based on the grammar in Listing 21. This datatype is based on the QDIMACS input format, but is designed to be

\(^{2}\)https://www.qbflib.org/qdimacs.html
able to represent a superset of valid QDIMACS input instances. This design choice is made for two main reasons:

1. To make it possible to define a bijection between the pcnf type and the subset of the QBF type that is in prenex conjunctive normal form.

2. To make it possible to use the datatype as the internal representation of formulas in the pcnf-based solvers described in Section 2.2.4 and 2.2.5.

In particular, the following relaxations were made to the QDIMACS format when designing the pcnf datatype:

- The preamble is ignored.
- Variables are represented by natural numbers, rather than 32-bit unsigned integers different from zero.
- Literals are represented by natural numbers tagged as positive or negative, rather than 32-bit signed integers different from zero.
- The only additional constraint enforced by the datatype is that contiguous quantified sets may not have the same quantifier type.
- Clauses are allowed to be empty, and the matrix is allowed to be empty; which is needed for the pcnf-based solvers.

Description of Datatype: For completeness, here is a short description of all definitions in Listing 22:

- A pcnf is a tuple of a prefix and a matrix.
- A prefix represents a sequence of quant_sets with alternating quantifiers. The UniversalFirst constructor represents a non-empty sequence of quantifier sets where the first one, represented by the quant_set parameter, is universally quantified, and the rest of the sets are represented by the quant_sets parameter; the ExistentialFirst constructor represents one where the first set is existentially quantified; while the Empty constructor represents an empty prefix.
- A quant_sets is a, possibly empty, sequence of quant sets represented as a quant_set list.
- A quant_set is a non-empty sequence of natural number, represented as a tuple of a nat and a nat list. Here, the natural numbers represent variables.
- A matrix is a list of clauses, where a matrix is considered to be true if and only if all clauses in the list are true.
- A clause is a list of literals, where a clause is considered to be true if and only if one of the literals in the list is true.
- A literal is a nat tagged as positive or negative, where the nat represent a variable, a positive literal is considered to be true if and only if the corresponding variable is true, and a negative literal is considered to be true if and only if the corresponding variable is false.
fun literal_p :: "QBF ⇒ bool" where
"literal_p (Var _) = True" |
"literal_p (Neg (Var _)) = True" |
"literal_p _ = False"

fun clause_p :: "QBF ⇒ bool" where
"clause_p (Disj list) = list_all literal_p list" |
"clause_p _ = False"

fun cnf_p :: "QBF ⇒ bool" where
"cnf_p (Conj list) = list_all clause_p list" |
"cnf_p _ = False"

fun pcnf_p :: "QBF ⇒ bool" where
"pcnf_p (Ex _ qbf) = pcnf_p qbf" |
"pcnf_p (All _ qbf) = pcnf_p qbf" |
"pcnf_p (Conj list) = cnf_p (Conj list)" |
"pcnf_p _ = False"

Listing 23: Predicates checking if a generic QBF formula is a literal, clause, in
conjugate normal form, or in prenex conjugate normal form, respectively.

4.3.1 PCNF Predicate for Generic QBFs

To make it possible to show that there is a bijection between the pcnf type
and the subset of the QBF formulas in prenex conjugate normal form, we must
first formalize the concept of prenex conjugate normal form for QBFs. This
formalization is done using predicates, implemented as Isabelle/HOL functions,
that can be found in Listing 23.

The predicate literal_p is true if and only if the argument is a literal, which is,
by definition, either a variable or the negation of a variable. Similarly, the
predicate clause_p is true if and only if the argument is a clause, which is a
disjunction of literals, and is implemented using list_all and literal_p; while
cnf_p is true if and only if the argument is in conjugate normal form, and is
implemented using list_all and clause_p. Finally, the predicate pcnf_p is true if
and only if the argument either satisfies cnf_p or starts with a quantifier,
where the quantified formula satisfies pcnf_p.

4.3.2 Bijection with PCNF subset of Generic QBF Datatype

This section introduces a bijection between the pcnf type and the subset of the
QBF type that is in prenex conjugate normal form. Conversion functions are
defined for literals, clauses, cnf formulas, and pcnf formulas. It is also shown that
the resulting QBF satisfies the corresponding predicate from Section 4.3.1,
for all conversion functions. A left-inverse is also defined for each conversion
function, showing that the conversion is injective. Then, surjectivity is shown
by proving that there is a literal, clause, matrix, or pcnf representation for
each QBF satisfying the corresponding predicate from Section 4.3.1. Finally, it
follows that the conversion is bijective.
Converting Literals: Literals are converted by the function `convert_literal` in Listing 24. The lemma `convert_literal_p`, showing that the result satisfies `literal_p`, is shown by case analysis. The function `convert_literal_inv` is a left-inverse of `convert_literal`. This is shown by the lemma `literal_inv`, which is also shown by case analysis. Since the inverse function is partial while Isabelle/HOL functions are total, it returns a `literal option`, instead of a simple `literal`.

Converting Clauses: Clauses can be converted using the `convert_clause` function in Listing 25, which is implemented using the `convert_literal` function. Like for literals, there is a lemma `convert_clause_p`, showing that the result satisfies `clause_p`; which is shown by induction using the lemma `convert_literal_p`. Showing that there is a left-inverse, however, is slightly more complicated in this case.

The function `convert_clause_inv` is a left-inverse of `convert_clause`. Again, an optional value is returned, since this is a partial function. In this case, the `sequence` function is used in the definition of the left-inverse. The `sequence` function was also used in Section 3.4.3, and returns `None` if and only if there is a `None` in the argument, otherwise it returns a list containing the same elements as the original one, in the same order.

To show that this function is a left-inverse, two lemmas are used. The lemma `list_no_None_ex_list_map_Some` shows that, if a list of optional values, say `?list`, does not contain `None`, then there is some `xs` where `map Some xs = ?list`; while the lemma `sequence_content` shows that `sequence (map Some xs) = Some xs`. Together with the lemma `literal_inv`, it is then possible to show that `convert_clause_inv` can recover a clause, say `cl`, that has been converted using `convert_clause`. This is done in the proof of the lemma `clause_inv`, which shows that `convert_clause_inv` is a left-inverse of `convert_clause`. 
fun convert_clause :: "clause ⇒ QBF" where
"convert_clause cl = Disj (map convert_literal cl)"

lemma convert_clause_p : "clause_p (convert_clause cl)"
using convert_literal_p by (induction cl) auto

fun convert_clause_inv :: "QBF ⇒ clause option"
where
"convert_clause_inv (Disj list) = sequence (map convert_literal_inv list)"
"convert_clause_inv _ = None"

lemma clause_inv :
"convert_clause_inv (convert_clause cl) = Some cl"
proof -
  let ?list = "map convert_literal_inv (map convert_literal cl)"
  have "∀x ∈ set cl. convert_literal_inv (convert_literal x) = Some x" using literal_inv by simp
  hence "map Some cl = ?list" using list_no_None_ex_list_map_Some by fastforce
  hence "sequence ?list = Some cl" using sequence_content by metis
  hence "convert_clause_inv (convert_clause cl) = Some cl"
  thus "convert_clause_inv (convert_clause cl) = Some cl" by simp
qed

Listing 25: Conversion functions between the clause type and the subset of the QBF type satisfying clause_p, with correctness proofs.

Converting CNF formulas: For the matrix type, the conversion is almost identical to the conversion for clauses; a clause and a matrix are both lists containing a type with a conversion function. The conversion function for matrices is called convert_matrix, while the left-inverse is called convert_matrix_inv; there are also lemmas convert_matrix_p and matrix_inv, showing that the converted matrix satisfies cnf_p and that convert_matrix_inv is a left-inverse, respectively. A listing is not included since the function definitions, lemmas, and proofs are basically identical to the ones for clauses in Listing 25.

Converting PCNF formulas: The conversion function for the pcnf type is convert and can be found in Listing 26. There are a lot of cases in the function, but it is essentially converting the prefix one variable at a time before converting the matrix using convert_matrix. Termination is not shown automatically, but it is straightforward to prove it by showing that the prefix length, measured by measure_prefix_length, decreases in each recursive call. The theorem convert_pcnf_p shows that the result satisfies the pcnf_p predicate. This is shown using a straightforward induction proof.

The function convert_inv is a left-inverse of convert, which is shown by the theorem convert_inv. New quantifiers can be added to the front of a pcnf using the functions add_universal_to_front and add_existential_to_front; both of these functions are used in the implementation of convert_inv, which works by
function convert :: "pcnf ⇒ QBF" where
"convert (Empty, matrix) = convert_matrix matrix" |
"convert (UniversalFirst (x, []) [], matrix) =
  All x (convert (Empty, matrix))" |
"convert (ExistentialFirst (x, []) [], matrix) =
  Ex x (convert (Empty, matrix))" |
"convert (UniversalFirst (x, []) (q # qs), matrix) =
  All x (convert (ExistentialFirst q qs, matrix))" |
"convert (ExistentialFirst (x, []) (q # qs), matrix) =
  Ex x (convert (UniversalFirst q qs, matrix))" |
"convert (UniversalFirst (x, y # ys) qs, matrix) =
  All x (convert (UniversalFirst (y, ys) qs, matrix))" |
"convert (ExistentialFirst (x, y # ys) qs, matrix) =
  Ex x (convert (ExistentialFirst (y, ys) qs, matrix))"
by pat_completeness auto
termination
  by (relation "measure measure_prefix_length") auto

theorem convert_pcnf_p: "pcnf_p (convert pcnf)"
  using convert_cnf_p
  by (induction rule: convert . induct ) auto

fun convert_inv :: "QBF ⇒ pcnf option" where
"convert_inv (All x qbf) =
  map_option (λp. add_universal_to_front x p)
    (convert_inv qbf)" |
"convert_inv (Ex x qbf) =
  map_option (λp. add_existential_to_front x p)
    (convert_inv qbf)" |
"convert_inv qbf =
  map_option (λm. (Empty, m)) (convert_matrix_inv qbf)"

theorem convert_inv:
  "convert_inv (convert pcnf) = Some pcnf"
proof (induction pcnf)
  case (Pair prefix matrix)
  show "convert_inv (convert (prefix, matrix)) = Some (prefix, matrix)"
    using matrix_inv
  by (induction rule: convert . induct ) auto
qed

Listing 26: Formalization of conversion functions between the pcnf type and the subset of the QBF type satisfying pcnf_p (Definition 4.2), with correctness proofs.
theorem convert_injective: "inj convert"
apply (rule inj_on_inverseI [where ?g = "the ◦ convert_inv"])
by (simp add: convert_inv)

Listing 27: Theorem showing that the convert function is injective, which follows from convert_inv being a left-inverse.

lemma convert_literal_p_ex:
assumes "literal_p lit"
shows "∃l. convert_literal l = lit"
proof -
have "∃l. convert_literal l = Var x" for x using convert_literal . simps by blast
moreover have "∃l. convert_literal l = Neg (Var x)" for x using convert_literal . simps by blast
ultimately show "∃l. convert_literal l = lit"
using assms by (induction rule: literal_p.induct) auto
qed

Listing 28: Lemma showing that any QBF satisfying literal_p has a corresponding representation as a literal.

converting the prefix one quantifier at a time, before converting the matrix. The theorem convert_inv shows that convert_inv is a left-inverse of convert, and is proven by induction using the matrix_inv lemma.

Conversion is Injective: Since the convert function has a left-inverse it is also injective. This is formalized as the theorem convert_inj in Listing 27. The proof uses the theorem inj_on_inverseI, which states that a function is injective if it has a left-inverse, in this case the ◦ convert_inv. Here, the composition of the and convert_inj is used, rather than just convert_inj, since a function returning a pcnf is needed, and not a function returning a pcnf option. It is then shown that the ◦ convert_inv is a left-inverse by using simplification with the theorem convert_inv.

Every literal_p QBF has a literal Representation: Listing 28 contains the lemma convert_literal_p_ex, which shows that every QBF satisfying literal_p has literal that, when used as an argument to convert_literal, yields the original QBF. The proof first shows that this holds for both positive and negative literals. After this, induction is used to show that the lemma holds for all possible arguments to literal_p, concluding the proof.

Every clause_p QBF has a clause Representation: For clauses, Listing 29 contains the lemma convert_clause_p_ex, which shows that every QBF satisfying clause_p has a clause that, when used as an argument to the function convert_clause, yields the original QBF. The proof of the lemma is more complicated than the corresponding proof for literals.
lemma convert_clause_p_ex:
  assumes "clause_p cl"
  shows "∃c. convert_clause c = cl"
proof -
  from assms obtain xs where 0: "Disj xs = cl"
    by (metis clause_p . elims (2))
  hence "list_all literal_p xs" using assms by fastforce
  hence "∃ys. map convert_literal ys = xs"
    using convert_literal_p_ex
  proof (induction xs)
    case Nil
    show "∃ys. map convert_literal ys = []" by simp
    next
    case (Cons x xs)
    from this obtain ys where
      "map convert_literal ys = xs" by fastforce
    moreover from Cons obtain y where
      "convert_literal y = x" by fastforce
    ultimately have "map convert_literal (y # ys) = x # xs"
      by simp
    thus "∃ys. map convert_literal ys = x # xs"
      by (rule exI)
  qed
  thus "∃c. convert_clause c = cl" using 0 by fastforce
qed
Listing 29: Lemma showing that any QBF satisfying clause_p has a corresponding
representation as a clause.

First of all, since cl satisfies clause_p, it is possible to obtain a list, in this
case called xs, where cl = Disj xs and xs is a list of QBFs satisfying literal_p.
After this, it is possible to use induction and the lemma convert_literal_p_ex
to show that there is another list, in this case called ys, that contains the literal
representation of every literal in xs. From the definition of the type clause and
the function convert_clause, it is then clear that ys is a clause that yields cl
when used as an argument to convert_clause; which concludes the proof.

Every cnf_p QBF has a matrix Representation: There is also a lemma
convert_cnf_p_ex, showing that every QBF satisfying cnf_p has a matrix that,
when used as an argument to the function convert_matrix, yields the original
QBF. No listing is included for it since it, and the proof, are almost identical to
the ones for clauses shown in Listing 29.

Every pcnf_p QBF has a pcnf Representation: Listing 30 contains the
theorem convert_pcnf_p_ex, which shows that for every QBF satisfying pcnf_p,
there is a corresponding pcnf that will yield the original QBF when used as an
argument to convert. The proof uses induction on the QBF; only the non-trivial
cases are shown in the listing.
If the QBF is a conjunction, then it is straightforward to show that there
is a pcnf yielding the conjunction when used as an argument to convert by
theorem convert_pcnf_p_ex: 
  assumes "pcnf_p qbf"
  shows "∃pcnf. convert pcnf = qbf" using assms
proof (induction qbf)
  ...
next 
case (Conj x)
  hence "cnf_p (Conj x)" by simp
  from this obtain m where "convert_matrix m = Conj x"
  using convert_cnf_p_ex by blast
  hence "convert (Empty, m) = Conj x" by simp
  thus "∃pcnf. convert pcnf = Conj x" by (rule exI)
next
  ...
next 
case (Ex x1a qbf)
  from this obtain pcnf where "convert pcnf = qbf"
  by fastforce
  hence "convert (add_existential_to_front x1a pcnf) = Ex x1a qbf" using convert_add_ex by simp
  thus "∃pcnf. convert pcnf = QBF.Ex x1a qbf" by (rule exI)
next 
case (All x1a qbf)
  from this obtain pcnf where "convert pcnf = qbf"
  by fastforce
  hence "convert (add_universal_to_front x1a pcnf) = All x1a qbf" using convert_add_all by simp
  thus "∃pcnf. convert pcnf = QBF.All x1a qbf" by (rule exI)
qed

Listing 30: Lemma showing that any QBF satisfying pcnf_p has a corresponding representation as a pcnf.

using the lemma convert_cnf_p_ex. If the QBF is an existential quantifier, then the induction hypothesis and the lemma convert_add_ex, stating that the equality convert (add_existential_to_front x pcnf) = Ex x (convert pcnf), can be used to construct the desired pcnf. The proof for the case where the QBF is a universal quantifier is similar. It is trivial to show convert_add_ex and convert_add_all using induction. Since a pcnf with the desired property exists in all cases, this concludes the proof.

Conversion is Bijective: From the theorem convert_pcnf_p_ex, it follows that the range of the convert function is the subset of the QBF type satisfying pcnf_p. This is formalized as the theorem convert_range in Listing 31. Also, since convert is injective, it also follows that it is bijective between the type pcnf and the subset of the type QBF satisfying pcnf_p. This bijectivity is formalized as the theorem convert_bijective_on in Listing 32.

4.3.3 Preservation of Semantics under the Bijection
In addition to showing that the convert function is bijective, we will also define
\textbf{Theorem} convert\_range: "range convert = \{p. pcnf\_p p\}"
\begin{verbatim}
  using convert\_pcnf\_p convert\_pcnf\_p\_ex by blast
\end{verbatim}

Listing 31: Theorem showing that the range of the convert function is exactly the subset of the QBF type satisfying pcnf\_p.

\textbf{Theorem} convert\_bijective\_on:
"bij_betw convert UNIV \{p. pcnf\_p p\}"
\begin{verbatim}
  by (simp add: bij_betw_def convert\_injective convert\_range)
\end{verbatim}

Listing 32: Formalization of Theorem 4.1 showing that the convert function is a bijection between the pcnf type and the subset of the QBF type satisfying pcnf\_p.

semantics for the pcnf type and show that convert preserves the semantics of the two types. That is, we will do the following:

1. Define semantics for the pcnf type.

2. Show that the new semantics, for the pcnf type, are true if and only if the original semantics, for the QBF type, are true after converting from a pcnf to a QBF using convert.

\textbf{Formalization of PCNF semantics:} Listing 33 contains a formalization of semantics for the pcnf type. The semantics are formalized as the functions literal\_semantics, clause\_semantics, matrix\_semantics, and pcnf\_semantics. All of these take an interpretation as the first argument and either a literal, a clause, a matrix, or a pcnf, respectively, as the second argument; this is similar to the semantics for the QBF type introduced in Section 3.2, which take an interpretation and a QBF. The new semantics are defined with two main goals in mind:

1. The semantics are defined to yield the Boolean that is usually expected for the given pcnf formula and interpretation; that is, the semantics should match our intuitions for when a quantified Boolean formula is true.

2. At the same time, the semantics for the pcnf type are defined to make it easy to show equivalence with the semantics for the QBF type composed with convert.

To achieve these goals, the semantics for the pcnf type are defined to be similar to the semantics for the QBF type. For literals, the value of the variables is looked up in the interpretation, like the variable case in qbf\_semantics. For clauses, which are interpreted as a disjunction of literals, the function list\_ex is used together with the literal\_semantics function to check if at least one literal in the clause is true; this is similar to how the list\_ex function is used in the case for disjunctions in the qbf\_semantics function to check if at least one subformula is true. Matrices are similar, but use list\_all instead of list\_ex, like the case for conjunctions in qbf\_semantics.

Finally, we have the function pcnf\_semantics. If the prefix is empty, then the function will use matrix\_semantics to check if the matrix is true. Otherwise,
fun literal_semantics :: "(nat ⇒ bool) ⇒ literal ⇒ bool"
  where
  "literal_semantics I (P x) = I x" |
  "literal_semantics I (N x) = (¬I x)"

fun clause_semantics :: "(nat ⇒ bool) ⇒ clause ⇒ bool"
  where
  "clause_semantics I clause =
    list_ex (literal_semantics I) clause"

fun matrix_semantics :: "(nat ⇒ bool) ⇒ matrix ⇒ bool"
  where
  "matrix_semantics I matrix =
    list_all (clause_semantics I) matrix"

function pcnf_semantics :: "(nat ⇒ bool) ⇒ pcnf ⇒ bool"
  where
  "pcnf_semantics I (Empty, matrix) = matrix_semantics I matrix" |
  "pcnf_semantics I (UniversalFirst (y,[]) [] , matrix) =
    (pcnf_semantics (I(y := True)) (Empty, matrix)
    ∧ pcnf_semantics (I(y := False)) (Empty, matrix))" |
  "pcnf_semantics I (ExistentialFirst (x,[]) [] , matrix) =
    (pcnf_semantics (I(x := True)) (Empty, matrix)
    ∨ pcnf_semantics (I(x := False)) (Empty, matrix))" |
  "pcnf_semantics I (UniversalFirst (y,yy#ys) qs , matrix) =
    (pcnf_semantics (I(y := True))
    ∧ pcnf_semantics (I(y := False))
    (UniversalFirst (yy , ys) qs , matrix))" |
  "pcnf_semantics I (ExistentialFirst (x,xx#xs) qs , matrix) =
    (pcnf_semantics (I(x := True))
    ∨ pcnf_semantics (I(x := False))
    (ExistentialFirst (xx , xs) qs , matrix))"
  by pat_completeness auto
termination
by (relation "measure (λ(I,p). measure_prefix_length p)") auto

Listing 33: Semantics for the literal, clause, matrix type, and pcnf types (Definition 4.3). The semantics are defined to yield the formula valuations and to make it easy to show that they agree with the semantics for the QBF type.
**Listing 34:** Formalization of Theorem 4.2 showing that the semantics of the `pcnf` type agrees with the semantics of the `QBF` type, after applying `convert`, with a corollary showing that that semantics also are preserved under `convert_inv`.

if the prefix starts with a universally quantified variable, then the semantics are defined as the conjunction of two recursive calls on the `pcnf` argument with the first variable removed; in one of the recursive calls, the variable is updated to be true in the interpretation, while it is updated to be false in the other call. If the first variable is existential, then the semantics are defined as the disjunction of the recursive calls, rather than the conjunction. Termination is shown by showing that the prefix length, defined as the number of variables in the prefix, decreases in each recursive call.

Together, the recursive cases of `pcnf_semantics` are similar to the quantifier cases of `qbf_semantics`. The similarity is that, for both functions, the semantics are equal to either a conjunction or a disjunction of the case when the first variable is true and when it is false, depending on if the first variable in the prefix is universal or existential.

An alternative way of defining the semantics for the `pcnf` type would have been to combine `convert` and `qbf_semantics`. There is nothing preventing a definition like this, but there are two main advantages to defining `pcnf_semantics` separately:

1. It simplifies a lot of proofs for the search-based solver.
2. It ensures that the PCNF semantics behave as expected, by preventing the conversion from, for example, switching universal and existential quantifiers.

**Preservation of Semantics:** The preservation of the semantics under the function `convert` is formalized by the theorem `qbf_semantics_eq_pcnf_semantics` in Listing 34; the formalization uses `pcnf_semantics` as the semantics for the `pcnf` type and `qbf_semantics` as the semantics for the `QBF` type. It is straightforward to show this using induction and the lemma `qbf_semantics_substitute_eq_assign`, since the semantics for the `pcnf` type were defined with this proof in mind. Listing 34 also contains the corollary `qbf_semantics_eq_pcnf_semantics'`, stating the equality using `convert_inv`, rather than `convert`.

Since we have now shown that `convert` is a semantics-preserving bijection, it is possible to treat a `pcnf`, say `p`, and the corresponding `QBF`, which is `convert p`,
type_synonym 'a parser = "string ⇒ ('a × string) option"

fun clause_list :: "Solver.matrix parser" where
"clause_list str = (case clause str of
None ⇒ None |
Some (cl, str') ⇒
(case clause_list str' of
None ⇒ Some (Cons cl Nil, str') | 
Some (cls, str'') ⇒ Some (Cons cl cls, str'')))"

Listing 35: Definition of a 'a parser type together with implementation of a clause_list parser returning a matrix. Similar functions exist for all parts of the QDIMACS grammar; together, they formalize Definition 4.4.

as two representations of the same formula. That is, any quantified Boolean formula that is in prenex conjugate normal form can be represented in Isabelle/HOL as either a pcnf or a QBF. It is also possible to freely convert between the pcnf type and the subset of the QBF type satisfying pcnf_p in theorems, proofs, and function definitions.

4.4 QDIMACS parser

A simple top-down parser for QDIMACS input files has been implemented in Isabelle/HOL. Termination has been shown, but there is currently no correctness proof, due to time limitations. A function for parsing a list of clauses as a matrix is shown in Listing 35, most other functions in the parser follow the same pattern; they call other parsing functions, and then use pattern matching on the result. Together, these functions formalize Definition 4.4.

The clause_list function takes a string as an argument and returns either a pair of a matrix and a str or None. To do this, the function consumes some number of characters from the front of the string argument and tries to parse the consumed characters as a matrix. If the consumed characters are successfully parsed as a matrix, then the function returns a tuple of the parsed matrix and the non-consumed part of the string. Similar functions exist for all other parts of the QDIMACS grammar, allowing any valid QDIMACS file to be parsed into a pcnf.

To show that the function terminates, it is sufficient to show that the string returned by the clause function, on success, is strictly shorter than the string argument to clause; after this, termination of clause_list can be shown automatically. The termination proof for all other parsing functions work in a similar way.

5 Search Solver Implementation and Verification

This section introduces the Isabelle/HOL formalization of the search solver introduced in Section 2.2.4 and the correctness proof for the solver. Unlike the naive solver, the search solver uses the prenex normal form datatype from Section 4.3 to represent formulas during solving. This is mainly to conform to standard solver descriptions, like the ones introduced in Section 2.2. Another
benefit of this is that the solver can utilize the simplifications done as a part
of literal assignments, that were introduced as the notation $\Phi_l$ in Section 2.1.3.
No normalization procedure converting arbitrary QBFs to prenex normal form
has been implemented due to time limitations.

A brief and informal overview of the main definitions, lemmas, and theorems
can be found in Section 5.1. The remaining subsections provide more details
about the formalization, including formal statements of some of the more im-
portant definitions, theorems, and proofs; explanations for some of the proofs
and some design decisions are also provided. Section 5.2 is about PCNF as-
signments, Section 5.3 is about how PCNF assignments change the set of free
variables, Section 5.4 is about the existential closure, Section 5.5 is about cleans-
ing PCNF formulas, while Section 5.6 is about the search solver itself. For the
full Isabelle/HOL formalization, see Appendix D.

5.1 Informal Overview of Main Isabelle/HOL Definitions
and Theorems

A list of informal summaries of some of the main definitions and theorems,
together with references to the corresponding formalizations, can be found here:

**Definition 5.1: PCNF Assignment.** PCNF assignments, the notation $\Phi_l$
for some formula $\Phi$ and literal $l$, have been formalized (pcnf_assign, Listing 38,
Section 5.2).

**Definition 5.2: PCNF Free Variables.** The free variables of a PCNF for-
mula has been formalized by reusing the definition for generic QBFs together
with the conversion function (pcnf_free_variables, Listing 40, Section 5.3.1).

**Theorem 5.1: Free Variables after PCNF Assignment.** The set of free
variables after a PCNF assignment, say $\Phi_l$, is a subset of the free variables of $\Phi$
minus the variable $|l|$ (pcnf_assign_free_subseteq_free_minus_lit, Listing 46,
Section 5.3.5).

**Definition 5.3: PCNF Existential Closure.** The existential closure of a
PCNF formula has been formalized by reusing the existential closure for generic
QBFs together with the conversion function (pcnf_existential_closure, List-
ing 47, Section 5.4).

**Theorem 5.2: PCNF Existential Closure Properties.** The existential
closure of a PCNF formula has the following properties:

1. It is satisfiable if and only if the original PCNF formula is satisfiable
   (pcnf_sat_iff_ex_close_sat, Listing 48, Section 5.4.2).

2. It does not have any free variables (pcnf_ex_closure_no_free, Listing 49,
   Section 5.4.3).

**Definition 5.4: Cleansed PCNF Formulas.** The concept of cleansed for-
mulas has been formalized as a predicate (cleansed_p, Listing 50, Section 5.5.1).

**Theorem 5.3: Cleansed PCNF is Cleansed after PCNF Assignment.** A
cleansed prenex normal form formula remains cleansed after a PCNF assignment
(pcnf_assign_cleansed_inv, Listing 51, Section 5.5.2).

**Definition 5.5: PCNF Formula Cleansing.** A function for cleansing PCNF
formulas has been defined (pcnf_cleanse, Listing 52, Section 5.5.3).
Theorem 5.4: Properties of PCNF Formula Cleansing. The function for cleansing PCNF formulas has the following properties:

1. It always yields a cleansed formula (\texttt{pcnf\_cleanse\_cleanses}, Listing 54, Section 5.5.4).
2. It preserves the set of free variables (\texttt{cleanse\_free\_vars\_inv}, Listing 56, Section 5.5.5).
3. It preserves the semantics (\texttt{pcnf\_cleanse\_preserves\_semantics}, Listing 59, Section 5.5.6).

Definition 5.6: Search Function. The search solver introduced in Section 2.2.4 has been formalized as an Isabelle/HOL function (\texttt{search}, Listing 60, Section 5.6).

Definition 5.7: Search Solver. A search solver is defined based on the existential closure, formula cleansing, and the search function (\texttt{search\_solver}, Listing 60, Section 5.6).

Lemma 5.1: Conditions for True and False PCNF Formulas. The following holds for any PCNF formula:

1. A PCNF formula is true if the matrix is empty and false if the matrix contains an empty clause (\texttt{false\_if\_empty\_clause\_in\_matrix} respectively \texttt{true\_if\_matrix\_empty}, Listing 61, Section 5.6.1).
2. A PCNF formula with an empty prefix and without any non-quantified variable occurrences has an empty matrix or an empty clause in the matrix (\texttt{empty\_clause\_or\_matrix\_if\_no\_variables}, Listing 62, Section 5.6.1).

Lemma 5.2: Satisfiability Equivalences for First Variable in Prefix. For a PCNF formula $\Phi = \forall y \Phi'$ where $y$ is not in the prefix of $\Phi'$, it holds that $\Phi \equiv_{\text{sat}} (\Phi_y \land \Phi_{\neg y})$ (\texttt{sat\_all\_first\_iff\_assign\_conj\_sat}, Listing 66, Section 5.6.2); also, for a PCNF formula $\Phi = \exists x \Phi'$ where $x$ is not in the prefix of $\Phi'$, it holds that $\Phi \equiv_{\text{sat}} (\Phi_x \lor \Phi_{\neg x})$.

Theorem 5.5: Search Function Correctness. The search function is correct, yielding true if and only if the argument is satisfiable, assuming that the argument is closed and cleansed (\texttt{search\_cleansed\_closed\_correct}, Listing 68, Section 5.6.3).

Theorem 5.6: Search Solver Correctness. The search solver is correct, yielding true if and only if the argument is satisfiable (\texttt{search\_solver\_correct}, Listing 69, Section 5.6.4).

5.2 Formalization of PCNF Assignment

To implement the SEARCH solver introduced in Section 2.2.4, we must first formalize the notation $\Phi_1$, which we will refer to as a PCNF assignment. Such an assignment will update the matrix by removing true clauses and false literals, which we will refer to as a matrix assignment; and will update the prefix by removing quantified occurrences of $|t|$, which we will refer to as a prefix assignment. The formalizations of these two types of updates, to the matrix and the prefix, will be formalized individually before being combined.
fun lit_neg :: "literal ⇒ literal" where  
  "lit_neg (P l) = N l" |  
  "lit_neg (N l) = P l"  

fun remove_lit_neg :: "literal ⇒ clause ⇒ clause" where  
  "remove_lit_neg lit clause =  
    filter (λl. l ≠ lit_neg lit) clause"  

fun remove_lit_clauses :: "literal ⇒ matrix ⇒ matrix" where  
  "remove_lit_clauses lit matrix =  
    filter (λcl. ¬(list_ex (λl. l = lit) cl)) matrix"  

fun matrix_assign :: "literal ⇒ matrix ⇒ matrix" where  
  "matrix_assign lit matrix =  
    remove_lit_clauses lit (map (remove_lit_neg lit) matrix)"

Listing 36: Formalization of matrix assignments; when given a literal $l$, a matrix assignment removes all clauses containing $l$ and removes all occurrences of $\overline{l}$ from all other clauses.

**Formalization of Matrix Assignment:** Listing 36 contains the formalization of matrix assignments. The formalization uses the utility function $\text{lit\_neg}$, which computes the negation of a literal; that is, it computes $\overline{l}$ for any literal $l$. With this function, it is easy to implement the $\text{remove\_lit\_neg}$ function, which filters away all occurrences of the negation of a specific literal from a clause; the function $\text{remove\_lit\_clauses}$ instead filters away all clauses containing a specific literal from a matrix.

Together, these can be used to formalize matrix assignments, which is done in the $\text{matrix\_assign}$ function. It maps the $\text{remove\_lit\_neg}$ function over the matrix to remove the negation of the literal argument from all clauses; after this, it uses $\text{remove\_lit\_clauses}$ function to remove all clauses containing the literal argument.

**Formalization of Prefix Assignment** Prefix assignments are formalized by the function $\text{remove\_var\_prefix}$ in Listing 37 which removes all occurrences of a specific variable from the prefix. Some new utility functions are used for the implementation, namely the functions $\text{prefix\_pop}$, $\text{add\_universal\_to\_prefix}$, and $\text{add\_existential\_to\_prefix}$; the functions $\text{pop}$ the first variable from the prefix, adds a universally quantified variable to the front of the prefix, and adds an existentially quantified variable to the front of the prefix, respectively.

Termination is shown using the $\text{prefix\_measure}$ function, which measures the number of variables occurring in the prefix.

**Formalization of PCNF Assignment** Listing 38 contains the formalization of PCNF assignments, as the function $\text{pcnf\_assign}$, which is implemented using the functions $\text{remove\_lit\_neg}$ and $\text{remove\_var\_prefix}$ together with the new function $\text{lit\_var}$. The function $\text{lit\_var}$ yields the variable in a literal, it computes $|l|$ for a literal $l$; this is then used with the $\text{remove\_var\_prefix}$ function to remove all occurrences of $|l|$ from the prefix. Together with a call to $\text{matrix\_assign}$, this completes the formalization of PCNF assignments.
function remove_var_prefix :: "nat ⇒ prefix ⇒ prefix" where
"remove_var_prefix x Empty = Empty" |
"remove_var_prefix x (UniversalFirst (y, ys) qs) =
  (if x = y
   then remove_var_prefix x (prefix_pop (UniversalFirst (y, ys) qs))
   else add_universal_to_prefix y (remove_var_prefix x
      (prefix_pop (UniversalFirst (y, ys) qs))))" |
"remove_var_prefix x (ExistentialFirst (y, ys) qs) =
  (if x = y
   then remove_var_prefix x (prefix_pop (ExistentialFirst (y, ys) qs))
   else add_existential_to_prefix y (remove_var_prefix x
      (prefix_pop (ExistentialFirst (y, ys) qs))))"

by pat_completeness auto
termination
by (relation "measure (λ(x, pre). prefix_measure pre)"
  (auto simp add: prefix_pop_decreases_measure
    simp del: prefix_measure.simps)
Listing 37: Formalization of prefix assignments; when given a variable z, a prefix assignment removes all occurrences of z from the prefix.

definition lit_var :: "literal ⇒ nat" where
"lit_var (P l) = l" |
"lit_var (N l) = l"
definition pcnf_assign :: "literal ⇒ pcnf ⇒ pcnf" where
"pcnf_assign lit (prefix, matrix) =
  (remove_var_prefix (lit_var lit) prefix,
    matrix_assign lit matrix)"

Listing 38: Formalization of PCNF assignments (Definition 5.1). When given a literal l, a PCNF assignment removes all clauses containing l from the matrix, removes all occurrences of ¬l from all other clauses in the matrix, and removes all quantified occurrences of |l|; the lit_var function computes |l|. 
5.3 Effect of PCNF Assignments on the Set of all Free Variables

To make it possible to show that a closed formula remains closed after a PCNF assignment, we will describe the effect a PCNF assignment has on the set of free variables. In order to give a brief explanation of how, we introduce three new definitions. They are formalized in Section 5.3.1, while informal definitions can be found here:

- \( \text{vars}(\Phi) \) is the set of all variables with non-quantified occurrences in \( \Phi \).
- \( \text{prefix}_\text{vars}(\Phi) \) is the set of all variables in the prefix of \( \Phi \).
- \( \text{free}_\text{vars}(\Phi) \) is the set of all free variables in \( \Phi \).

With these definitions, the following can be show:

- \( \text{free}_\text{vars}(\Phi) = \text{vars}(\Phi) - \text{prefix}_\text{vars}(\Phi) \), which is shown in Section 5.3.2.
- \( \text{vars}(\Phi_l) \subseteq \text{vars}(\Phi) - \{|l|\} \), which is shown in Section 5.3.3.
- \( \text{prefix}_\text{vars}(\Phi_l) = \text{prefix}_\text{vars}(\Phi) - \{|l|\} \), which is shown in Section 5.3.4.
- \( \text{vars}(\Phi_l) \subseteq \text{vars}(\Phi) - \{|l|\} \), which follows from the previous three statements and is shown in Section 5.3.5.

5.3.1 Variables, Prefix Variables, and Free Variables

Listing 39 contains the functions \text{variables} and \text{prefix_variables} computing the variables with non-quantified occurrences and quantified occurrences in the prefix, respectively. Here, the prefix is defined as the set of all quantifiers, with the corresponding variables, occurring before any other part of the formula. The implementations use a method similar to the one for \text{free_variables} in Listing 9 in Section 3.3.1; a QBF is traversed by the function \text{variables} or \text{prefix_variables}, before the result is normalized by removing duplicates and sorting the result.

Using the functions \text{variables}, \text{prefix_variables}, and \text{free_variables} together with the \text{convert} function, it is possible to define corresponding functions for the \text{pcnf} type. The corresponding \text{pcnf} functions are \text{pcnf_variables}, \text{pcnf_prefix_variables}, and \text{pcnf_free_variables}, respectively, and can all be found in Listing 40.

5.3.2 Free Variables is Variables without Prefix Variables

A couple of intermediate lemmas are used to show that \( \text{free}_\text{vars}(\Phi) = \text{vars}(\Phi) - \text{prefix}_\text{vars}(\Phi) \) for any PCNF formula represented using the \text{pcnf} type. First, it is shown that the equality holds for any literal, clause, matrix, or PCNF formula, all represented using the \text{QBF} type. Then, it is shown that the equality holds for any PCNF formula represented using the \text{pcnf} type.
fun variables_aux :: "QBF ⇒ nat list" where
"variables_aux (Var x) = [x]" |
"variables_aux (Neg qbf) = variables_aux qbf" |
"variables_aux (Conj list) =
  concat (map variables_aux list)" |
"variables_aux (Disj list) =
  concat (map variables_aux list)" |
"variables_aux (Ex x qbf) = variables_aux qbf" |
"variables_aux (All x qbf) = variables_aux qbf"

fun variables :: "QBF ⇒ nat list" where
"variables qbf = sort (remdups (variables_aux qbf))"

fun prefix_variables_aux :: "QBF ⇒ nat list" where
"prefix_variables_aux (All y qbf) =
  Cons y (prefix_variables_aux qbf)" |
"prefix_variables_aux (Ex x qbf) =
  Cons x (prefix_variables_aux qbf)" |
"prefix_variables_aux _ = Nil"

fun prefix_variables :: "QBF ⇒ nat list" where
"prefix_variables qbf =
  sort (remdups (prefix_variables_aux qbf))"

Listing 39: Functions computing a list of all variables with a non-quantified occurrence respectively a quantified occurrence in the prefix for any QBF.

fun pcnf_variables :: "pcnf ⇒ nat list" where
"pcnf_variables pcnf = variables (convert pcnf)"

fun pcnf_prefix_variables :: "pcnf ⇒ nat list" where
"pcnf_prefix_variables pcnf =
  prefix_variables (convert pcnf)"

fun pcnf_free_variables :: "pcnf ⇒ nat list" where
"pcnf_free_variables pcnf = free_variables (convert pcnf)"

Listing 40: Functions computing, for any pcnf, all variables with a non-quantified occurrence, a quantified occurrence in the prefix, and a list of all free variables (Definition 5.2), respectively.
lemma lit_p_free_eq_vars:
    "literal_p qbf ⇒
    set (free_variables qbf) = set (variables qbf)"
by (induction qbf rule: literal_p.induct) auto

lemma cl_p_free_eq_vars:
    assumes "clause_p qbf"
    shows "set (free_variables qbf) = set (variables qbf)"
proof -
    obtain qbf_list where list_def: "qbf = Disj qbf_list"
        using assms
        by (induction qbf rule: clause_p.induct) auto
    moreover from this have "list_all literal_p qbf_list"
        using assms by simp
    ultimately show ?thesis
        using lit_p_free_eq_vars
        by (induction qbf_list arbitrary: qbf) auto
qed

Listing 41: Lemmas showing that the set of free variables is equal to the set of variables with a non-quantified occurrence, for any QBF satisfying literal_p or clause_p.

Literals, Clauses, and Matrices: The lemma lit_p_free_eq_vars and the lemma cl_p_free_eq_vars in Listing 41 show that for any literal, respectively clause, represented as a QBF, it holds that the set of free variables is equal to the set of variables with a non-quantified occurrence. There is also a lemma cnf_p_free_eq_vars, showing the corresponding fact for matrices; this lemma is not included in any listing since it, including its proof, is very similar to cl_p_free_eq_vars. In the case of literals, the proof is shown by induction; for clauses, the proof obtains the list of all literals in the clause, and use the lit_p_free_eq_vars lemma to show cl_p_free_eq_vars by induction.

PCNF Formulas: Then, it is possible to show that, for any formula in prenex normal form represented as a QBF, the set of free variables is equal to the set of variables with a non-quantified occurrence minus the set of variables with quantified occurrences in the prefix. This is shown by the lemma pcnf_p_free_eq_vars_minus_prefix in Listing 42.

Another lemma is used to show pcnf_p_free_eq_vars_minus_prefix, namely pcnf_p_free_eq_vars_minus_prefix_aux, which is also in Listing 42; this lemma is similar, but uses prefix_variables_aux instead of prefix_variables, enabling a simple inductive proof. The non-trivial cases of the proof are shown in the listing, and are shown using either the bound subtract equivalence for free variables or the cnf_p_free_eq_vars lemma.

PCNF Type: Since the equality between the sets holds for any QBF satisfying pcnf_p, it should also hold for any pcnf. This fact is formalized by the lemma pcnf_free_eq_vars_minus_prefix in Listing 43, which is shown using the lemma pcnf_p_free_eq_vars_minus_prefix together with the lemma convert_pcnf_p, which shows that convert yields a QBF satisfying pcnf_p.
lemma pcnf_p_free_eq_vars_minus_prefix_aux:
  "pcnf_p qbf \implies
   set (free_variables qbf)
 = set (variables qbf) - set (prefix_variables_aux qbf)"
proof (induction qbf rule: prefix_variables_aux.induct)
  case (1 y qbf)
  thus \(?case
   using bound_subtract_equiv[of "\{\}" "\{y\}" qbf] by force
next
  case (2 x qbf)
  thus \(?case
   using bound_subtract_equiv[of "\{\}" "\{x\}" qbf] by force
next
  ...
next
  case ("3_3" v)
  hence "cnf_p (Conj v)"
  by (induction "Conj v" rule: pcnf_p.induct) auto
  thus \(?case using cnf_p_free_eq_vars by fastforce
next
  ...
qed

lemma pcnf_p_free_eq_vars_minus_prefix:
  "pcnf_p qbf \implies
   set (free_variables qbf)
 = set (variables qbf) - set (prefix_variables qbf)"
using pcnf_p_free_eq_vars_minus_prefix_aux by simp

Listing 42: Lemmas showing that the set of free variables is equal to the set of variables with a non-quantified occurrence minus the set of variables with a quantified occurrence in the prefix, for any QBF satisfying pcnf_p.

lemma pcnf_free_eq_vars_minus_prefix:
  "set (pcnf_free_variables pcnf)
 = set (pcnf_variables pcnf)
 - set (pcnf_prefix_variables pcnf)"
using pcnf_p_free_eq_vars_minus_prefix convert_pcnf_p
by simp

Listing 43: Lemma showing that the set of free variables is equal to the set of variables with a non-quantified occurrence minus the set of variables with a quantified occurrence in the prefix, for any pcnf.
lemma lit_not_in_matrix_assign_variables:
  "lit_var lit \notin set (variables (convert_matrix (matrix_assign lit matrix)))"
proof (induction matrix)
...
qed

lemma matrix_assign_vars_subseteq_matrix_vars_minus_lit:
  "set (variables (convert_matrix (matrix_assign lit matrix))) \subseteq set (variables (convert_matrix matrix)) - {lit_var lit}"
using lit_not_in_matrix_assign_variables by force

lemma pcnf_vars_eq_matrix_vars:
  "set (pcnf_variables (prefix, matrix)) = set (variables (convert_matrix matrix))"
by (induction "(prefix, matrix)" arbitrary: prefix rule: convert.induct) auto

lemma pcnf_assign_vars_subseteq_vars_minus_lit:
  "set (pcnf_variables (pcnf_assign x pcnf)) \subseteq set (pcnf_variables pcnf) - {lit_var x}"
using matrix_assign_vars_subseteq_matrix_vars_minus_lit pcnf_vars_eq_matrix_vars
by (induction pcnf) simp

Listing 44: Lemmas showing that the set of variables with a non-quantified occurrence after a PCNF assignment is a subset of the set before the assignment minus the variable in the assigned literal.

5.3.3 Set of Matrix Variables is Non-increasing under PCNF Assignments

The set of variables with a non-quantified occurrence is non-increasing\(^3\) under PCNF assignments; moreover, it holds that \(\text{vars}(\Phi_1) \subseteq \text{vars}(\Phi) - \{|l|\}\) for any PCNF formula \(\Phi\). To show this for any PCNF formula, we first show that \(|l|\) is not in the matrix of \(\Phi_1\), and then show the subset relation; Listing 44 contains a formalization of the proof.

The lemma lit_not_in_matrix_assign_variables shows that, for any literal \(l\), it holds that \(|l|\) is not in the matrix of a PCNF formula after assigning \(l\) in the matrix. This can be shown in a straightforward way by nesting inductions for the matrix, clauses, and literals. Using this lemma, it is possible to show matrix_assign_vars_subseteq_matrix_vars_minus_lit, which shows that the subset relation holds for any matrix of a PCNF formula. Moreover, the lemma pcnf_vars_eq_matrix_vars shows that the set of variables with non-quantified occurrences is the same for a PCNF formula as for the matrix of that formula. Together, this can be used to show the desired subset relation, which is formalized as the lemma pcnf_assign_vars_subseteq_vars_minus_lit.

\(^3\)With respect to the order defined by the subset relation (\(\subseteq\)).
lemma prefix_assign_vars_eq_prefix_vars_minus_lit:
  "set (pcnf_prefix_variables
    (remove_var_prefix x prefix, matrix))
  = set (pcnf_prefix_variables (prefix, matrix)) - {x}"
proof (induction "(prefix, matrix)" arbitrary: prefix
  rule: convert.induct)
  ...
qed

lemma prefix_vars_matrix_inv:
  "set (pcnf_prefix_variables (prefix, matrix1))
  = set (pcnf_prefix_variables (prefix, matrix2))"
by (induction "(prefix, matrix1)" arbitrary: prefix
  rule: convert.induct) auto

lemma pcnf_prefix_vars_eq_prefix_minus_lit:
  "set (pcnf_prefix_variables (pcnf_assign x pcnf))
  = set (pcnf_prefix_variables pcnf) - {lit_var x}"
using prefix_assign_vars_eq_prefix_vars_minus_lit
  prefix_vars_matrix_inv
by (induction pcnf) fastforce

Listing 45: Lemmas showing that the set of variables with a quantified
occurrence in the prefix after a PCNF assignment is equal to the set before
the assignment minus the variable in the assigned literal.

5.3.4 PCNF Assignment Removes Variable from Prefix
The lemma pcnf_prefix_vars_eq_prefix_minus_lit in Listing 45 shows that
prefix_vars(Φ_l) = prefix_vars(Φ) − {l} for any literal l. Two lemmas are used
in the proof. The first one is prefix_assign_vars_eq_prefix_vars_minus_lit,
which shows that the equality holds for the prefix of any PCNF formula, and can
be shown by using induction together with some lemmas showing that adding
a variable to the front of the prefix adds that variable to the set of prefix vari-
ables. The second one is prefix_vars_matrix_inv, which shows that the set of
prefix variables does not depend on the matrix, and is shown using induction.
Together, these two lemmas can be used to show the equality.

5.3.5 Set of Free Variables is Non-increasing under PCNF Assign-
ments
Listing 46 shows the theorem pcnf_assign_free_subseteq_free_minus_lit, for-
malizing the fact that free_vars(Φ_l) ⊆ free_vars(Φ) − {l}. The proof uses the
more abstract lemma free_assign_proof_skeleton, which is instantiated with
other lemmas shown in this section, all of which are themselves instantiated
with appropriate terms. This theorem can be used to show that a closed PCNF
formula remains closed after a PCNF assignment.
lemma free_assgn_proof_skeleton:
  "free = var - pre ⇒ free_assgn = var_assgn - pre_assgn
  ⇒ var_assgn ⊆ var - lit
  ⇒ pre_assgn = pre - lit
  ⇒ free_assgn ⊆ free - lit"
by auto

theorem pcnf_assign_free_subseteq_free_minus_lit:
  "set (pcnf_free_variables (pcnf_assign x pcnf))
   ⊆ set (pcnf_free_variables pcnf) - {lit_var x}"
using free_assgn_proof_skeleton[OF
  pcnf_free_eq_vars_minus_prefix[of pcnf]
  pcnf_free_eq_vars_minus_prefix[of "pcnf_assign x pcnf"]
  pcnf_assign_vars_subseteq_vars_minus_lit[of x pcnf]
  pcnf_prefix_vars_eq_prefix_minus_lit[of x pcnf]] .

Listing 46: Formalization of Theorem 5.1 together with a lemma used in the proof. The theorem shows that the set of free variables after a PCNF assignment is a subset of the set of free variables before the assignment minus the variable in the assigned literal.

fun pcnf_existential_closure :: "pcnf ⇒ pcnf"
where
  "pcnf_existential_closure pcnf =
  the (convert_inv (existential_closure (convert pcnf)))"

Listing 47: Formalization of existential closure for the pcnf type (Definition 5.3), re-using the formalization of the existential closure for the QBF type.

5.4 PCNF Existential Closure

To ensure that the search solver terminates successfully, the argument must be closed. From the results in the previous subsection, it follows that the arguments to all recursive calls will be closed if the argument to the initial call is closed. Motivated by this, a function for getting the existential closure of a pcnf is introduced in this subsection.

5.4.1 Formalization of PCNF Existential Closure

It is possible to reuse the existential closure function from the generic QBF case, since we have a bijective semantics-preserving function between the pcnf type and the pcnf_p subset of the QBF type. This is done by the function pcnf_existential_closure in Listing 47, which convert the pcnf argument to a QBF, applies the existential_closure function, and then converts the result back to a pcnf.

5.4.2 PCNF Existential Closure Preserves Satisfiability

It is straightforward to show that pcnf_existential_closure preserves satisfiability, since we already have a proof that existential_closure preserves satis-
Lemma \text{convert\_inv\_inv}:
\begin{quote}
"pcnf\_p qbf \Rightarrow convert (the (convert\_inv qbf)) = qbf"
\end{quote}
by (metis convert\_inv convert\_pcnf\_p\_ex option\_sel)

Lemma \text{ex\_closure\_aux\_pcnf\_p\_inv}:
\begin{quote}
"pcnf\_p qbf \Rightarrow pcnf\_p (existential\_closure\_aux qbf vars)"
\end{quote}
by (induction qbf vars rule:
existent\_closure\_aux\_induct) auto

Lemma \text{ex\_closure\_pcnf\_p\_inv}:
\begin{quote}
"pcnf\_p qbf \Rightarrow pcnf\_p (existential\_closure qbf)"
\end{quote}
using \text{ex\_closure\_aux\_pcnf\_p\_inv} by simp

Theorem \text{pcnf\_sat\_iff\_ex\_close\_sat}:
\begin{quote}
"satisfiable (convert pcnf) = satisfiable (convert (pcnf\_existential\_closure pcnf))"
\end{quote}
using convert\_inv\_inv convert\_pcnf\_p
\begin{quote}
ex\_closure\_pcnf\_p\_inv sat\_iff\_ex\_close\_sat by auto
\end{quote}

Listing 48: Formalization of Theorem 5.2.1, showing that a pcnf is satisfiable if and only if its existential closure is satisfiable, together with lemmas used in the proof of the theorem.

Theorem \text{pcnf\_ex\_closure\_no\_free}:
\begin{quote}
"pcnf\_free\_variables (pcnf\_existential\_closure pcnf) = []"
\end{quote}
using convert\_inv\_inv convert\_pcnf\_p
\begin{quote}
ex\_closure\_pcnf\_p\_inv ex\_closure\_no\_free by auto
\end{quote}

Listing 49: Formalization of Theorem 5.2.2, showing that the existential closure function for the pcnf type yields a pcnf without any free variables.

5.4.3 No Free Variables in PCNF Existential Closure

The theorem \text{pcnf\_ex\_closure\_no\_free} in Listing 49 shows that the existential closure of a pcnf does not have any free variables. This can be shown by using mostly the same lemmas as in the proof of \text{pcnf\_sat\_iff\_ex\_close\_sat}, only replacing \text{sat\_iff\_ex\_close\_sat} with \text{ex\_closure\_no\_free}.

5.5 Cleansed PCNF Formulas

For the search solver to be correct, the equivalence \( \Phi \approx_{\text{sat}} (\Phi_y \land \Phi_{\neg y}) \) must
fun cleansed_p :: "pcnf ⇒ bool" where
    "cleansed_p pcnf = 
        distinct (prefix_variables_aux (convert pcnf))"

lemma prefix_pop_cleansed_if_cleansed:
    "cleansed_p (prefix, matrix) ⇒ 
        cleansed_p (prefix_pop prefix, matrix)"
    by (induction prefix rule: prefix_pop.induct) auto

lemma cleansed_prefix_first_ex_unique:
    assumes "cleansed_p (ExistentialFirst (x, xs) qs, matrix)"
    shows "x \not\in \set (pcnf_prefix_variables (prefix_pop 
        (ExistentialFirst (x, xs) qs), matrix))"
    using assms by (induction "ExistentialFirst (x, xs) qs" 
        rule: prefix_pop.induct) auto

Listing 50: A predicate checking if a pcnf is cleansed (Definition 5.4), a lemma showing that a cleansed formula is still cleansed after removing the first quantifier, and a lemma showing that the first variable in the prefix of a cleansed formula does not occur in the rest of the prefix.

theorem pcnf_assign_cleansed_inv:
    "cleansed_p pcnf ⇒ 
        cleansed_p (pcnf_assign lit pcnf)"
proof (induction pcnf rule: convert.induct)
    ...
qed

Listing 51: Formalization of Theorem 5.3, showing that a cleansed pcnf formula will remain cleansed after applying pcnf_assign.

5.5.1 Predicate for Cleansed Formulas

The concept of cleansed formulas is formalized by the predicate cleansed_p in Listing 50, which collects all variable occurrences in the prefix, and then checks if they are distinct. Some lemmas showing important properties of cleansed formulas are also included in the listing. The lemma prefix_pop_cleansed_if_cleansed shows that a formula remains cleansed after removing the first variable from the prefix. Also, for any cleansed formula, the first variable in the prefix does not occur in the prefix after removing the first variable from it. This is formalized, for existential variables, as the lemma cleansed_prefix_first_ex_unique; for universal variables, there is a similar lemma cleansed_prefix_first_all_unique, not shown in the listing.

hold when \( \Phi = \forall y \Phi' \) and the equivalence \( \Phi \approx_{sat} (\Phi_x \lor \Phi_{\neg x}) \) must hold when \( \Phi = \exists x \Phi' \). In Section 5.6.2, we will show that this holds under the additional assumption that \( y \), respectively \( x \), does not occur in the prefix of \( \Phi' \). These assumptions will always hold if \( \Phi \) is cleansed, which motivates the formalization of cleansed PCNF formulas and a cleansing function.
function pcnf_cleanse :: "pcnf ⇒ pcnf"
where
"pcnf_cleanse (Empty, matrix) = (Empty, matrix)" |
"pcnf_cleanse (UniversalFirst (y, ys) qs, matrix) = 
(if y ∈ set (pcnf_prefix_variables (prefix_pop
(UniversalFirst (y, ys) qs), matrix))
  then pcnf_cleanse (prefix_pop
  (UniversalFirst (y, ys) qs), matrix)
  else add_universal_to_front y
  (pcnf_cleanse (prefix_pop
  (UniversalFirst (y, ys) qs), matrix)))" |
"pcnf_cleanse (ExistentialFirst (x, xs) qs, matrix) = 
(if x ∈ set (pcnf_prefix_variables (prefix_pop
(ExistentialFirst (x, xs) qs), matrix))
  then pcnf_cleanse (prefix_pop
  (ExistentialFirst (x, xs) qs), matrix)
  else add_existential_to_front x
  (pcnf_cleanse (prefix_pop
  (ExistentialFirst (x, xs) qs), matrix)))"
by pat_completeness auto
termination
by (relation "measure (λ(pre, mat). prefix_measure pre)"
(auto simp add: prefix_pop_decreases_measure
  simp del: prefix_measure.simps)
Listing 52: Function cleansing any pcnf formula by removing all quantified occurrences of variables in the prefix where there is another quantified occurrence of the same variable at a lower level (Definition 5.5).

5.5.2 The Cleansed Predicate is Invariant under PCNF Assignment
Another important property is that pcnf_assign yields a cleansed formula when applied to a cleansed formula, which is formalized in Listing 51 as the theorem pcnf_assign_cleansed_inv. The proof of this theorem is a long and quite intricate inductive proof reusing a lot of earlier results. Intuitively, the theorem should hold since a formula in prenex normal form is cleansed if and only if all quantified occurrences of variables are distinct, which is also the main idea used in the proof.

5.5.3 Cleansing PCNF Formulas
To make it possible to solve any pcnf formula, regardless of if it is cleansed or not, a cleansing function is introduced. The function pcnf_cleanse in Listing 52 will cleanse any pcnf formula. To ensure that the formula is cleansed, the function will, for each variable in the prefix, check if there is another quantified occurrence of the same variable at a lower level, and remove it if and only if there is. Termination is shown by showing that the prefix length decreases in each recursive call. After showing some properties of this function, it will be possible to create a SEARCH-based solver capable of solving any pcnf.
lemma cleanse_prefix_vars_inv:
"set (pcnf_prefix_variables (prefix, matrix))
= set (pcnf_prefix_variables
  (pcnf_cleanse (prefix, matrix)))"
using add_all_adds_prefix_var
  prefix_pop_all_prefix_vars_set add_ex_adds_prefix_var
  prefix_pop_ex_prefix_vars_set
by (induction "(prefix, matrix)" arbitrary: prefix
  rule: pcnf_cleanse.induct) auto

Listing 53: Lemma showing that the set of variables with a quantified occurrence
in the prefix is invariant under formula cleansing.

theorem pcnf_cleanse_cleanses:
"cleansed_p (pcnf_cleanse pcnf)"
using cleanse_prefix_vars_inv
  cleansed_add_new_all_to_front
  cleansed_add_new_ex_to_front
by (induction pcnf rule: pcnf_cleanse.induct) auto

Listing 54: Formalization of Theorem 5.4.1, showing that cleansing a formula
will yield a cleansed formula.

5.5.4 Cleansing Yields a Cleansed Formula

To show that the pcnf_cleanse function yields a cleansed formula, some other
lemmas are used. The most important one is cleanse_prefix_vars_inv in List-
ing 53, which shows that the set of free variables is not changed by pcnf_cleanse.
This can be shown by induction using some other lemmas describing how adding
and removing quantifiers to the front of the formula affects the set of prefix vari-
ables.

The proof also uses the two lemmas cleansed_add_new_all_to_front and
  cleansed_add_new_ex_to_front, which show that adding variables not in the
prefix to the front of a cleansed formula yields a cleansed formula. Together
with cleanse_prefix_vars_inv, these lemmas can be used to show the theorem
pcnf_cleanse_cleanses in Listing 54, which shows that pcnf_cleanse yields a
formula satisfying cleansed_p.

5.5.5 Cleansing Preserves the Set of Free Variables

The lemma cleanse_vars_inv in Listing 55 shows that the set of all variables
with a non-quantified occurrence is invariant when applying pcnf_cleanse. This
can be shown using lemmas showing that the set does not change when modify-

ing the prefix. It can then be shown that the set of free variables is invariant by
using this lemma, together with cleanse_prefix_vars_inv, showing that the set
of prefix variables is invariant, and pcnf_free_eq_vars_minus_prefix, showing
that the set of all free variables is the set difference of the two other sets. This
is formalized as the theorem cleanse_free_vars_inv in Listing 56.
lemma cleanse_vars_inv:
"set (pcnf_variables (prefix, matrix))
= set (pcnf_variables (pcnf_cleanse (prefix, matrix)))"
using add_all_vars_inv prefix_pop_all_vars_inv
add_ex_vars_inv prefix_pop_ex_vars_inv
by (induction "(prefix, matrix)" arbitrary: prefix rule:
 pcnf_cleanse.induct) auto

Listing 55: Lemma showing that the set of variables with a non-quantified occurrence in a formula remains the same after cleansing the formula.

theorem cleanse_free_vars_inv:
"set (pcnf_free_variables pcnf)
= set (pcnf_free_variables (pcnf_cleanse pcnf))"
using cleanse_prefix_vars_inv cleanse_vars_inv
pcnf_free_eq_vars_minus_prefix
by (induction pcnf) simp_all

Listing 56: Formalization of Theorem 5.4.2, showing that the set of free variables in a formula remains the same after cleansing the formula.

5.5.6 Cleansing Preserves Semantics

To show that the semantics are preserved under formula cleansing, we introduce the term redundant variable occurrence, which refers to a variable occurrence in the prefix where there is another quantified occurrence of the same variable at a lower level. Redundant variable occurrences do not affect the semantics of the formula. The lemma pop_redundant_all_prefix_var_semantics_eq in Listing 57 formalizes this fact. For existential quantifiers, there is a corresponding lemma pop_redundant_ex_prefix_var_semantics_eq.

To show these lemmas, it is first shown that the set of free variables is the same after removing the redundant variable occurrence; it is also shown that the semantics are equal for two interpretations if they agree on all free variables, which is formalized as the lemma semantics_eq_if_free_vars_eq in Listing 57. Together, this implies the lemmas showing that the semantics do not change when removing redundant variables.

When adding a universally quantified variable to the front of a formula, the semantics are true if and only if they are true both when the variable is updated to be true in the interpretation and when it is updated to be false. This is formalized by the lemma pcnf_semantics_conj_eq_add_all in Listing 58. There is also a corresponding lemma for existentially quantified variables, namely pcnf_semantics_disj_eq_add_ex. The proofs are straightforward simplification proofs, re-using some earlier theorems and lemmas.

These lemmas enable the proof of the pcnf_cleanse_preserves_semantics theorem in Listing 59. The proof uses induction and shows the cases where the first variable in the prefix is redundant and where it is not separately. When it is redundant, the lemmas pop_redundant_all_prefix_var_semantics_eq and pop_redundant_ex_prefix_var_semantics_eq are used to show the theorem; when it is not redundant, the lemmas pcnf_semantics_conj_eq_add_all and pcnf_semantics_disj_eq_add_ex are used instead.
Lemma 57: Lemmas showing that the semantics are equal if the interpretation of all free variables are equal, and that a universally quantified redundant variable can be removed from the prefix of a formula without changing the semantics.

```
lemma semantics_eq_if_free_vars_eq:
  assumes "\(\forall x \in \text{set (free_variables qbf)}. I(x) = J(x)\)"
  shows "\(\text{qbf_semantics } I \text{ qbf} = \text{qbf_semantics } J \text{ qbf}\)"
  using assms
  proof (induction I qbf rule: qbf_semantics.induct)
    ...
  qed

lemma pop_redundant_all_prefix_var_semantics_eq:
  assumes "\(y \in \text{set (pcnf_prefix_variables (prefix_pop (UniversalFirst (y, ys) qs), matrix))}\)"
  shows "\(\text{pcnf_semantics } I \text{ (UniversalFirst (y, ys) qs, matrix}) = \text{pcnf_semantics } I \text{ (prefix_pop (UniversalFirst (y, ys) qs), matrix})}\)"
  proof -
    ...
  qed
```

Listing 57: Lemmas showing that the semantics are equal if the interpretation of all free variables are equal, and that a universally quantified redundant variable can be removed from the prefix of a formula without changing the semantics.

```
lemma pcnf_semantics_conj_eq_add_all:
  "\(\text{pcnf_semantics } (I(y := True)) \text{ pcnf} \wedge \text{pcnf_semantics } (I(y := False)) \text{ pcnf}\)"
  \(\leftrightarrow\) \(\text{pcnf_semantics } I \text{ (add_universal_to_front y pcnf})\)"
  using convert_add_all qbf_semantics_eq_pcnf_semantics
  qbf_semantics_substitute_eq_assign by simp

Listing 58: Lemma showing that the semantics of a formula are true both when a variable, say \(y\), is true in the interpretation and when it is false, if and only if the semantics are true after adding \(\forall y\) to the front of the formula.

```
theorem pcnf_cleanse_preserves_semantics:
  "\(\text{pcnf_semantics } I \text{ pcnf} = \text{pcnf_semantics } I \text{ (pcnf_cleanse pcnf)}\)"
  proof (induction pcnf arbitrary: I rule: pcnf_cleanse.induct)
    ...
  qed
```

Listing 59: Formalization of Theorem 5.4.3, showing that cleansing a formula preserves the semantics of that formula.
function search :: "pcnf ⇒ bool option" where
"search (prefix, matrix) =
  (if [] ∈ set matrix then Some False
   else if matrix = [] then Some True
   else Option.bind (first_var prefix) (\z. Option.bind (first_existential prefix) (\e. if e then combine_options (\) (search (pcnf_assign (P z) (prefix, matrix))))
                           (search (pcnf_assign (N z) (prefix, matrix))))
   else combine_options (\) (search (pcnf_assign (P z) (prefix, matrix))))
   (search (pcnf_assign (N z) (prefix, matrix))))))" by pat_completeness auto

termination
apply (relation "measure (\(p, m). prefix_measure p)")
apply (auto simp add: prefix_pop_decreases_measure
        simp del: prefix_measure.simps)
using remove_first_var_decreases_measure
  first_var.simps(3) option.discI option.sel by metis+

fun search_solver :: "pcnf ⇒ bool" where
"search_solver pcnf = the (search (pcnf_cleanse (pcnf_existential_closure pcnf)))"

Listing 60: Formalization of the SEARCH procedure from Section 2.2.4 (Definition 5.6) and a SEARCH-based solver using this procedure (Definition 5.7).

5.6 Search Solver

Finally, we have all of the components to implement and show correctness of the SEARCH solver introduced in Section 2.2.4. Listing 60 contains the function search implementing the SEARCH procedure, for which termination is shown by showing that the prefix length decreases in each recursive call; and the function search_solver, which solves a pcnf by taking the existential closure and cleansing the formula, before using search to determine the satisfiability of the formula.

Two new utility functions are used to implement search, both of which return optional values. The function first_var returns the first variable in the prefix if it exists, and None if the prefix is empty; while the function first_existential returns true if and only if the first variable in the prefix exists and is existential, and None if the prefix is empty. These two functions can be considered as partial functions, returning None when they are undefined since Isabelle/HOL functions must be total.

The search function is also partial, since it cannot decide the satisfiability for all inputs. Again, this partiality is implemented using optional values. To deal with these optional values, the implementation of search uses monadic bind functions to bind the values returned from first_var and first_existential, and uses combine_options to combine the optional Booleans returned from the recursive calls.
lemma false_if_empty_clause_in_matrix:
"[] ∈ set matrix ⇒ pcnf_semantics I (prefix, matrix) = False"
by (induction I "(prefix, matrix)" arbitrary: prefix
rule: pcnf_semantics.induct)
(induction matrix, auto)

lemma true_if_matrix_empty:
"matrix = [] ⇒ pcnf_semantics I (prefix, matrix) = True"
by (induction I "(prefix, matrix)" arbitrary: prefix
rule: pcnf_semantics.induct) auto

Listing 61: Formalization of Lemma 5.1.1 as two lemmas showing that a pcnf is false if there is an empty clause in the matrix and that a pcnf is true if the matrix is empty, respectively.

lemma matrix_shape_if_no_variables:
"pcnf_variables (Empty, matrix) = [] ⇒ (∃n. matrix = replicate n [])"
proof (induction matrix)
...
qed

lemma empty_clause_or_matrix_if_no_variables:
"pcnf_variables (Empty, matrix) = [] ⇒ [] ∈ set matrix ∨ matrix = []"
using matrix_shape_if_no_variables
by fastforce

Listing 62: A lemma describing the shape a pcnf with an empty matrix and no variables, and the formalization of Lemma 5.1.2 showing that such a pcnf must have an empty matrix or an empty clause in the matrix.

5.6.1 Conditions for True and False PCNF Formulas

To show correctness of the SEARCH procedure, it is also necessary to have some sufficiently powerful conditions for when a formula is true or false. In particular, it is necessary to show that a PCNF formula with an empty clause in the matrix is false, and that a formula with an empty matrix is true. This is shown by the lemmas false_if_empty_clause_in_matrix and true_if_matrix_empty in Listing 61; both of which are shown by induction. Moreover, it is necessary to show that a PCNF formula without any variables either has an empty matrix, or has an empty clause in the matrix. This is shown by the lemma empty_clause_or_matrix_if_no_variables in Listing 62, which is a straightforward consequence of the lemma matrix_shape_if_no_variables. The matrix_shape_if_no_variables lemma can be shown using induction by deriving a contradiction if any clause contains a variable. Together, these results are sufficient to show that a closed PCNF formula with an empty prefix is either true or false, and will be used to show that the base case of the recursion in SEARCH is correct.
lemma matrix_semantics_inv_remove_true:
  "matrix_semantics (I(z := True))
   (matrix_assign (P z) matrix)
 = matrix_semantics (I(z := True)) matrix"
proof (induction matrix)
  ...
qed

lemma pcnf_assign_free_eq_matrix_assign:
  assumes "lit_var lit \notin set (pcnf_prefix_variables (prefix, matrix))"
  shows "pcnf_assign lit (prefix, matrix)
    = (prefix, matrix_assign lit matrix)"
  using assms pcnf_assign_free_eq_matrix_assgn by simp

lemma pcnf_semantics_inv_assign_true:
  assumes "z \notin set (pcnf_prefix_variables (prefix, matrix))"
  shows "pcnf_semantics (I(z := True))
    (prefix, matrix_assign (P z) matrix)
 = pcnf_semantics (I(z := True)) (prefix, matrix)"
  using assms
proof (induction I "(prefix, matrix)" arbitrary: I prefix
    matrix rule: pcnf_semantics_induct)
  ...
qed

Listing 63: The lemma pcnf_semantics_inv_matrix_assign_true shows that the semantic
are not affected by matrix assignments using a true positive literal;
the two other lemmas are used in the proof.

5.6.2 Satisfiability Equivalences for First Variable in Prefix

To establish correctness for the recursive cases in SEARCH, it will be shown that
\( \Phi \approx_{sat} (\Phi_1 \lor \Phi_2) \) if \( \Phi = \exists y \Phi' \) for some \( \Phi' \), and that \( \Phi \approx_{sat} (\Phi_1 \land \Phi_2) \) if
\( \Phi = \forall y \Phi' \) for some \( \Phi' \). The proofs of these equivalences are long inductive
proofs, that do not offer much insight into why the equivalences hold, and use
lemmas that themselves have long inductive proofs, which also do not offer a lot
of insight. Because of this, most of the details of the proofs will not be presented
here, in favor of just presenting the main results.

Some Lemmas Used to Show Equivalences:

Listing 63 contains several
lemmas that are important for the proof of the equivalences. The lemma
matrix_semantics_inv_remove_true shows that the matrix semantics are not
changed when a variable, say \( z \), that is true in the interpretation is removed from
the matrix using matrix_assign (P z). A corresponding lemma also exists for
the case when the variable is false, called matrix_semantics_inv_remove_false.

The lemma pcnf_assign_free_eq_matrix_assgn is also used to show the two
equivalences. This lemma states that a PCNF assignment using a literal, say \( l \),
is the same as a matrix assignment, if \( \|l\| \) does not occur in the prefix.

By using the lemma matrix_semantics_inv_remove_true, we can show the
lemma pcnf_semantics_inv_matrix_assign_true; the two lemmas are similar,
Lemma \( \text{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj} \):

\[
\begin{align*}
\text{assumes } & \text{ "} z \notin \text{ set (pcnf\_prefix\_variables (prefix, matrix))} \text{"} \\
\text{shows } & \text{ "} \text{pcnf\_semantics (I(z := True)) (prefix, matrix)} \\
& \quad \land \text{ pcnf\_semantics (I(z := False)) (prefix, matrix)} \\
& \quad \leftrightarrow \\
& \quad \text{pcnf\_semantics (I(z := True)) (prefix, matrix\_assign (P z) matrix)} \\
& \quad \land \text{ pcnf\_semantics (I(z := False)) (prefix, matrix\_assign (N z) matrix)} \text{"}
\end{align*}
\]

using \( \text{assms} \)

proof (induction I "(prefix, matrix\_assign (P z) matrix)"

... 

del

Listing 64: Lemma combining \( \text{pcnf\_semantics\_inv\_matrix\_assign\_true} \) and \( \text{pcnf\_semantics\_inv\_matrix\_assign\_false} \) using a conjunction.

Lemma \( \text{interp\_value\_ignored\_for\_pcnf\_P\_assign} \):

\[
\begin{align*}
\text{"} \text{pcnf\_semantics (I(x := b)) (pcnf\_assign (P x) pcnf)} \\
& = \text{ pcnf\_semantics I (pcnf\_assign (P x) pcnf)} \text{"}
\end{align*}
\]

using \( \text{pcnf\_semantics\_eq\_if\_free\_vars\_eq\_x\_notin\_assign\_P\_x\_pcnf\_free\_eq\_vars\_minus\_prefix} \) by simp

Listing 65: Lemma showing that the value of a variable in the interpretation can be ignored if that variable is assigned using a PCNF assignment.

but the new one uses the PCNF semantics instead of the matrix semantics and requires that the assigned variables does not occur in the prefix. Again, there is a similar lemma for the case when the variable is false, which, unsurprisingly, is called \( \text{pcnf\_semantics\_inv\_matrix\_assign\_false} \).

The lemma \( \text{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj} \) in Listing 64 uses a conjunction to combine the lemmas \( \text{pcnf\_semantics\_inv\_matrix\_assign\_true} \) and \( \text{pcnf\_semantics\_inv\_matrix\_assign\_false} \). There is also a similar lemma for disjunctions, called \( \text{pcnf\_semantics\_disj\_iff\_matrix\_assign\_disj} \).

Finally, we have the lemma \( \text{interp\_value\_ignored\_for\_pcnf\_P\_assign} \) which can be found in Listing 65. Roughly speaking, this lemma shows that if a positive literal, say \( l \), is used in a PCNF assignment, then the value of \(|l|\) in the interpretation does not matter. There is also a similar lemma for negative literals called \( \text{interp\_value\_ignored\_for\_pcnf\_N\_assign} \).

The Satisfiability Equivalences: Listing 66 contains the important lemma \( \text{sat\_all\_first\_iff\_assign\_conj\_sat} \), which shows that if \( \Phi = \forall y \Phi' \) and \( y \) does not occur in the prefix of \( \Phi' \), then \( \Phi \approx_{sat} (\Phi_y \land \Phi_{\neg y}) \). The proof works by first expanding the formula using \( \text{satisfiable\_def} \), \( \text{qbf\_semantics\_eq\_pcnf\_semantics} \), and \( \text{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj} \); the expanded formula includes updates to the interpretation and calls to \( \text{prefix\_pop} \) and \( \text{matrix\_assign} \). After this, the lemma \( \text{pcnf\_assign\_free\_eq\_matrix\_assign} \) is used to replace the \( \text{prefix\_pop} \) and \( \text{matrix\_assign} \) calls with a \( \text{pcnf\_assign} \) call. Then, the lemma \( \text{interp\_value\_ignored\_for\_pcnf\_P\_assign} \) is used together with the sim-
Theorem \texttt{sat\_all\_first\_iff\_assign\_conj\_sat}:

Assumes
\[ \forall y \not\in \text{set} (\text{pcnf\_prefix\_variables} (\text{prefix\_pop} (\text{UniversalFirst} (y, ys) qs), \text{matrix})) \]

Shows
\[ \text{satisfiable} (\text{convert} (\text{UniversalFirst} (y, ys) qs, \text{matrix})) \]
\[ \iff \text{satisfiable} (\text{Conj} \]
\[ \text{convert} (\text{pcnf\_assign} (P y) (\text{UniversalFirst} (y, ys) qs, \text{matrix})), \]
\[ \text{convert} (\text{pcnf\_assign} (N y) (\text{UniversalFirst} (y, ys) qs, \text{matrix}))) \]

Proof:

Let \( ?\text{pre} = "\text{UniversalFirst} (y, ys) qs" \)

Have
\[ \text{satisfiable} (\text{convert} (?\text{pre}, \text{matrix})) \]
\[ = (\exists I. \text{pcnf\_semantics} I (?\text{pre}, \text{matrix})) \]

Using \texttt{satisfiable\_def} \texttt{qbf\_semantics\_eq\_pcnf\_semantics} by simp

Also have \( \ldots = \)
\[ (\exists I. \text{pcnf\_semantics} (I(y := \text{True}))(\text{prefix\_pop} ?\text{pre}, \text{matrix}) \land \]
\[ \text{pcnf\_semantics} (I(y := \text{False}))(\text{prefix\_pop} ?\text{pre}, \text{matrix})) \]

By (induction "?\text{pre}" rule: \texttt{prefix\_pop\_induct}) auto

Also have \( \ldots = \)
\[ (\exists I. \text{pcnf\_semantics} (I(y := \text{True}))(\text{pcnf\_assign}(P y) (?\text{pre}, \text{matrix})) \land \]
\[ \text{pcnf\_semantics} (I(y := \text{False}))(\text{pcnf\_assign}(N y) (?\text{pre}, \text{matrix}))) \]

Using \texttt{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj} \texttt{assms} by blast

Also have \( \ldots = \)
\[ (\exists I. \text{pcnf\_semantics} I (\text{pcnf\_assign}(P y) (?\text{pre}, \text{matrix})) \land \]
\[ \text{pcnf\_semantics} I (\text{pcnf\_assign}(N y) (?\text{pre}, \text{matrix}))) \]

Using \texttt{interp\_value\_ignored\_for\_pcnf\_N\_assign} \texttt{interp\_value\_ignored\_for\_pcnf\_P\_assign} by blast

Also have \( \ldots = \)
\[ (\exists I. \text{qbf\_semantics} I (\text{convert}(\text{pcnf\_assign}(P y) (?\text{pre}, \text{matrix})))) \land \]
\[ \text{qbf\_semantics} I (\text{convert}(\text{pcnf\_assign}(N y) (?\text{pre}, \text{matrix})))) \]

Using \texttt{qbf\_semantics\_eq\_pcnf\_semantics} by blast

Also have \( \ldots = \)
\[ \text{satisfiable} (\text{Conj} \]
\[ \text{convert} (\text{pcnf\_assign}(P y) (?\text{pre}, \text{matrix})), \]
\[ \text{convert} (\text{pcnf\_assign}(N y) (?\text{pre}, \text{matrix})))) \]

Unfolding \texttt{satisfiable\_def} by simp

Finally Show \( ?\text{thesis} \).

QED

Listing 66: Formalization of Lemma 5.2 for universal variables, showing that if \( \Phi = \forall y \Phi' \) for some \( \Phi' \) without \( y \) in the prefix, then \( \Phi \approx (\Phi_y \land \Phi_{\neg y}) \). The proof of this uses, among other results, \texttt{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj} from Listing 64, \texttt{pcnf\_assign\_free\_eq\_matrix\_assign[of \"P y"]} from Listing 63, and \texttt{interp\_value\_ignored\_for\_pcnf\_N\_assign} from Listing 65.
lemma search_cleansed_closed_yields_Some:  
  assumes "cleansed_p pcnf"  
  and "pcnf_free_variables pcnf = []"  
  shows "(∃b. search pcnf = Some b)" using assms  
proof (induction pcnf rule: search.induct)  
  ...  
qed

Listing 67: Lemma showing that \texttt{search} will return a Boolean for all cleansed and closed arguments.

After proving the lemma, similar lemma \texttt{interp\_value\_ignored\_for\_pcnf\_N\_assign} to remove the update to the interpretation. Finally, the proof can be finished using \texttt{satisfiable_def} and \texttt{qbf\_semantics_eq\_pcnf\_semantics} again.

There is also another lemma \texttt{sat\_ex\_first\_iff\_assign\_disj\_sat}, showing that if $\Phi = \exists x \Phi'$ and $x$ does not occur in the prefix of $\Phi'$, then $\Phi \approx_{sat} (\Phi_z \lor \Phi_{\neg z})$. This lemma is not in any listing, but can be shown by a proof with structure similar to the one for \texttt{sat\_all\_first\_iff\_assign\_conj\_sat}; the main difference being that the lemma \texttt{pcnf\_semantics\_disj\_iff\_matrix\_assign\_disj} has to be substituted for \texttt{pcnf\_semantics\_conj\_iff\_matrix\_assign\_conj}.

5.6.3 Correctness of the Search Function

To show that the \texttt{search} function is correct, we first show that the function yields some Boolean, and not \texttt{None}, when the argument is cleansed and closed. This is formalized as the lemma \texttt{search\_cleansed\_closed\_yields\_Some} in Listing 67. The proof is a long inductive proof that does not fit in a listing. Because of this, we will, rather than including the full proof in a listing, summarize the structure of the proof here, letting the $\Phi$ represent the argument, and letting $z$ represent the first variable in the prefix when it exists:

1. If $\Phi$ has an empty matrix or an empty clause in a matrix, then the function yields a Boolean.
2. Otherwise, the prefix is not empty, since a contradiction can be derived if it is. If the prefix is empty, then $\Phi$ has no variable occurrences since it is closed. Therefore, the matrix is empty or contains an empty clause, according to \texttt{empty\_clause\_or\_matrix\_if\_no\_variables}, which is a contradiction.
3. Two recursive calls are made with the argument $\Phi_z$ respectively $\Phi_{\neg z}$, which are obtained by applying \texttt{pcnf\_assign} to $\Phi$, regardless of if $z$ is universally quantified or existentially quantified. Both $\Phi_z$ and $\Phi_{\neg z}$ are closed according to the theorem \texttt{pcnf\_assign\_free\_subseq\_free\_minus\_lit}, and are cleansed according to the theorem \texttt{pcnf\_assign\_cleansed\_inv}. All recursive calls, therefore, yield some Boolean, which follows from the induction hypothesis since $\Phi_z$ and $\Phi_{\neg z}$ are cleansed and closed, which consequently concludes the proof.

The lemma \texttt{search\_cleansed\_closed\_correct} in Listing 68 shows that the \texttt{search} function yields true if and only if the argument is satisfiable. Again, the proof
is a long inductive proof that does not fit in a listing, the structure of which will be summarized here, letting $\Phi$ represent the argument, and letting $z$ represent the first variable in the prefix when it exists:

1. If $\Phi$ has an empty clause in the matrix, then it is not satisfiable according to `false_if_empty_clause_in_matrix`, and the function correctly yields false.

2. If $\Phi$ has an empty matrix, then it is satisfiable, which follows from the lemma `true_if_matrix_empty`, and the function correctly yields true.

3. Otherwise, the prefix is not empty, for the same reason as in the proof of `search_cleansed_closed_yields_Some`.

4. The formulas $\Phi_z$ and $\Phi_{\neg z}$ are cleansed and closed; again, for the same reason as in the proof of `search_cleansed_closed_yields_Some`. All recursive calls, therefore, yield true or false depending on the satisfiability of the argument, which follows from the induction hypothesis and `search_cleansed_closed_yields_Some`. It follows that the `search` function return either the conjunction or disjunction of the satisfiability of $\Phi_z$ and $\Phi_{\neg z}$, depending on if $z$ is universally quantified or existentially quantified.

5. Since $\Phi_z$ and $\Phi_{\neg z}$ are closed, it follows from `semantics_eq_if_free_vars_eq` that the conjunction, respectively disjunction, of the satisfiability of $\Phi_z$ and $\Phi_{\neg z}$ is equal to the satisfiability of ($\Phi_z \land \Phi_{\neg z}$), respectively ($\Phi_z \lor \Phi_{\neg z}$).

6. We also have $\Phi \approx_{sat} (\Phi_z \land \Phi_{\neg z})$ and $\Phi \approx_{sat} (\Phi_z \lor \Phi_{\neg z})$, which follows from the lemma `sat_ex_first_iff_assign_disj_sat` and the lemma `sat_all_first_iff_assign_conj_sat`, respectively, since $\Phi$ is cleansed.

7. All cases have now been covered, and it follows that the function yields a Boolean that is equal to the satisfiability of $\Phi$.

### 5.6.4 Correctness of the Search Solver

With the correctness proof for the `search` function, it is now possible to show that `search_solver` is correct. The theorem `search_solver_correct` in Listing 69 shows this. There are four main steps to the proof:
Theorem 5.6 showing that `search_solver` is correct.

1. The proof first shows that `pcnf_cleanse (pcnf_existential_closure pcnf)` is satisfiability equivalent to `pcnf`, which follows from the satisfiability equivalence between a `pcnf` formula and its existential closure, the logical equivalence between a `pcnf` formula before and after cleansing, the equivalence between the semantics for the QBF type and the `pcnf` type, and the definition of satisfiability.

2. Next, it is shown that `pcnf_cleanse (pcnf_existential_closure pcnf)` is closed, which follows from the existential closure being closed, and formula cleansing preserving the set of free variables.

3. Moreover, it holds that `pcnf_cleanse (pcnf_existential_closure pcnf)` is cleansed, which follows from the cleansing function yielding a cleansed formula.

4. All of the above, together with `search_cleaned_closed_correct`, shows that the `search_solver` function is correct, returning true if and only if the argument is satisfiable. This concludes the correctness proof for the solver.

6 Performance Evaluation and Discussion

The performance of the two verified solvers and a state-of-the-art solver, CAQE, have been measured on a number of instances. These measurements are, then, used to compare the performance of the naive solver and the search solver with each other, and also with the CAQE solver. To run the verified solvers, they are first exported to an executable programming language using the code export facilities in Isabelle/HOL, which are described below.
structure Example : sig
  type 'a tree
  val tree_rev : 'a tree -> 'a tree
end = struct

datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree ;

fun tree_rev Leaf = Leaf
  | tree_rev (Node (l, x, r)) = Node ( tree_rev r, x, tree_rev l);
end; (*struct Example*)

Listing 70: Standard ML export of the tree_rev function in Listing 4. The tree
  type is included automatically since tree_rev depends on it.

Isabelle/HOL Code Export

Isabelle/HOL has code generation facilities that can convert HOL specifications
to executable programs in the programming languages SML, OCaml, Haskell,
and Scala [24]. The code generation makes it possible to export the QBF solvers
defined in Section 3 and Section 5 as executable programs. In turn, this makes
it possible to empirically evaluate the performance of the solvers.

The translation from Isabelle/HOL definitions to the target language uses
a shallow embedding; logical entities like types and constants are translated to
corresponding entities in the target language. This translation is straightforward
for HOL definitions using the datatype, fun, or function keywords since all
the target languages support both algebraic datatypes and recursive functions.
Being able to export these constructs is sufficient to be able to export the two
QBF solvers created for this thesis.

As an example of a code export, consider Listing 70, which contains a SML
export of the tree_rev function from Listing 4. The tree_rev function has been
exported as an SML function. The tree datatype has also been, automatically,
included in the export, as an SML datatype, since it is used by the tree_rev
function.

Evaluation Setup

The evaluation setup is described here, explaining how the data presented in
the next subsection was gathered. Section 6.2.1 describes the evaluated solvers,
including details about the code exports of the two verified solvers; Section 6.2.2
describes the problems used to evaluate the solvers; while Section 6.2.3 describes
the hardware used for the evaluation.

6.2.1 Evaluated Solvers

The evaluation uses the following solvers:

- The naïve solver from Section 3 exported to SML, OCaml, Haskell, and
  Scala.
The rest of this subsection provides more details about the solvers.

**Naive Solver and Search Solver Exports:** Two new Isabelle/HOL functions have been implemented: a function combining the parser and conversion function from Section 4 with the naive solver from Section 3, and a function combining the parser with the search solver from Section 5. Using these functions, the naive solver and the search solver can both be used to solve any valid QDIMACS instance, represented as an Isabelle/HOL string. Both new functions have been exported to all four possible target languages.

Small wrappers reading input and calling the exported function have been created in each target language. These wrappers can then be used to run the solver, after the program has been compiled.

Table 1 lists the compiler and optimization flags used for each target language. These compilers and flags were chosen with high performance as the main goal. To improve the performance further, we also include some Isabelle theories modifying the embedding used for the code exports.

Before the code is exported, the theories HOL-Library.Code_Abstract_Char, HOL-Library.Code_Target_Numeral, and HOL-Library.RBT_Set are imported, all of which are part of the default Isabelle/HOL distribution. Together, the first two theories ensure that characters and numbers are implemented using target language integers, avoiding implementations based on lists of Booleans or successor-function applications, respectively. The third theory ensures that sets are implemented as red-black-trees instead of as lists. Appendix E contains the Isabelle theory defining the code export.

**CAQE Solver:** The CAQE solver is a state-of-the-art QBF solver based on counterexample guided abstraction and refinement [27, 28], and performed the best out of all the solvers in the 2022 QBF solver evaluation (QBFEVAL 2022).\(^4\)

Because of this state-of-the-art performance, the solver is a suitable reference point to compare the performance of the verified solvers to. This comparison can clarify how big the performance gap between the verified solvers and the state-of-the-art in QBF solving is. Version 4.0.2 of CAQE is used for the comparison,\(^4\)

---

\(^4\)https://www.qbflib.org/qbfeval2022_results.php
compiled with the default release configuration using version 1.73.0 of rustc. No command-line options are used when running the solver.

6.2.2 Problems Used

The instances from QBFEVAL 2022 are used to evaluate the solvers. Since these instances are too difficult to solve to give much information about the performance of the two verified solvers, new instances have also been created.

The new instances are QBF encodings of two different problems, called the domino game and the xor problem. Both of these problems can scale to different sizes depending on an integer parameter, and can therefore reveal how the solvers scale when the length of the input grows. These problems are described in more detail in the following paragraphs.

**Domino Game:** In the domino game, two players take turns placing $1 \times 2$ domino bricks on a $1 \times n$ board, for some $n \geq 2$; when a player is not able to place another brick, they lose. QBFEVAL 2022 included instances that are encodings of the domino game, where an instance is either satisfiable or not depending on which player wins the game. A generator for encoding the game on any board size as a QBF instance has been made available[^5], and is used for the evaluation here.

However, the length of the encodings does not scale linearly with $n$. This motivates using the xor problem for the evaluation, in addition to the domino game.

**Xor Problem:** The core of the xor problem is the following, satisfiable, formula, where the symbol $\oplus$ denotes the exclusive or operation and $n \geq 1$:

$$\forall y_1 \exists x_1 \forall y_2 \exists x_2 \ldots \forall y_n \exists x_n ((y_1 \oplus x_1) \land (x_1 \oplus y_2 \oplus x_2) \land \cdots \land (x_{n-1} \oplus y_n \oplus x_n))$$

The exclusive or operations can be encoded in conjugate normal form using the following equivalences:

$$((z_1 \oplus z_2) \equiv ((z_1 \lor z_2) \land (\neg z_1 \lor \neg z_2)))$$

$$(z_1 \oplus z_2 \oplus z_3) \equiv ((z_1 \lor z_2 \lor z_3) \land (z_1 \lor \neg z_2 \lor \neg z_3)$$

$$\land (\neg z_1 \lor z_2 \lor \neg z_3) \land (\neg z_1 \lor \neg z_2 \lor z_3))$$

This ensures that, for each assignment of values to the universally quantified variables, there is exactly one assignment to the existentially quantified ones making the formula true; meaning that it is satisfiable, but not trivially so.

It is possible to make this formula unsatisfiable by adding a clause that is a disjunction of all existentially quantified variables. If this is done, there is exactly one assignment to the universal variables for which there is no assignment to the existential ones making the formula true.

The length of these formulas, both the satisfiable and the unsatisfiable ones, scale linearly when $n$ is increased. With this more graceful scaling, it is easier to see how the solvers scale with the input length.

[^5]: [https://www.qbflib.org/index_eval.php](https://www.qbflib.org/index_eval.php)
6.2.3 Hardware Used

The evaluation is done using the StarExec [29] cluster, on nodes with an Intel E5-2609 CPU running at 2.40GHz, with a 10240KB cache. For all solvers and instances, the memory usage is limited to a maximum of 32GB. The CPU time is also limited to 900 seconds. This is the same environment and configuration as the one used in QBFEVAL 2022.

6.3 Evaluation Results

Table 2 and Table 3 contain the results from using the evaluated solvers to attempt to solve a number of instances of the domino game and the xor problem, respectively. The result for each pair of solver and problem instance is either the CPU time to solve the instance, a timeout, an out-of-memory error, or a stack overflow. Out of all the QBFEVAL 2022 instances, the naive solver solved none, the search solver solved 1, and CAQE solved 329.

The set of instances in the tables were selected to ensure that the verified solvers time out for the largest couple instances while avoiding instance sizes that are too small to yield any informative results. For the domino game, the board size ranges from $1 \times 2$ up to $1 \times 10$; while, for the xor problem, the number of quantifier pairs ranges from 5 up to 100. This is sufficient to give a rough overview of the performance of the solvers, and how they compare to the state-of-the-art.

6.4 Discussion

Some aspects of the results are discussed here. Section 6.4.1 discusses the performance of the different solvers, while Section 6.4.2 discusses the performance of the different target languages. Section 6.4.3 discusses the scaling of the solver on the xor problem. Finally, Section 6.4.4 discusses the stack overflows encountered during the evaluation.

6.4.1 Solver Performance Comparison

The relative performance of the solvers is clear from the results in Tables 2 and 3. The search solver performs much better than the naive solver, and CAQE performs much better than both of the verified solvers. It is expected that CAQE outperforms the two verified solvers, since it uses a much more powerful algorithm. But, it is less obvious why the search solver outperforms the naive solver, since, based on the implementations, both of them appear to have time complexities that are exponential in the number of quantifiers. Some possible reasons are discussed in the following paragraphs.

When the search solver assigns a literal, it deletes all clauses containing that literal and all occurrences of the negation of that literal. This simplification can, in some cases, prune the search space by allowing empty clauses to be derived. An empty matrix can also be derived after only exploring a small fraction of the search space, causing the search to stop early. As an example of how the

---

6. For the Scala exports of the verified solvers, the maximum heap size has been manually tuned to yield a total memory consumption close to 32GB.
<table>
<thead>
<tr>
<th>Solver:</th>
<th>Naive Solver</th>
<th>Search Solver</th>
<th>CAQE</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>SML</td>
<td>OCaml</td>
<td>Haskell</td>
</tr>
<tr>
<td>domino2_sat</td>
<td>0.004096</td>
<td>0.004599</td>
<td>0.00555</td>
</tr>
<tr>
<td>domino3_sat</td>
<td>0.007834</td>
<td>0.008394</td>
<td>0.01106</td>
</tr>
<tr>
<td>domino4_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino5i_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino6_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino7_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino8_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino9i_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino10_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino2i_unsat</td>
<td>0.004216</td>
<td>0.004594</td>
<td>0.005572</td>
</tr>
<tr>
<td>domino3i_unsat</td>
<td>0.007228</td>
<td>0.00729</td>
<td>0.010985</td>
</tr>
<tr>
<td>domino4i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino5i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino6i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino7i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino8i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino9i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>domino10i_unsat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
</tbody>
</table>

Table 2: Results from running each solver on instances of the domino game. If an instance was solved, then the CPU time used by the solver is reported; if the solver ran for more than 900 CPU seconds, then a timeout (‘t/o’) is reported; if the solver used more than 32GB of memory, then an out-of-memory error (‘oom’) is reported; otherwise, if a stack overflow is encountered during execution, then a stack overflow (‘so’) is reported. The integer in the instance name indicates the size of the $1 \times n$ board; instances with a ‘sat’ suffix are satisfiable, while instances with an ‘unsat’ suffix have an extra clause added to make them unsatisfiable; instances without an ‘i’ after the integer $n$ are satisfiable if and only if the first player wins, while instances with an ‘i’ after the integer $n$ are inverse instances that are satisfiable if and only if the second player wins.
<table>
<thead>
<tr>
<th>Solver:</th>
<th>Naive Solver</th>
<th>Search Solver</th>
<th>CAQE</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>SML</td>
<td>OCaml</td>
<td>Haskell</td>
</tr>
<tr>
<td>xor5_sat</td>
<td>0.052822</td>
<td>0.070217</td>
<td>1.155837</td>
</tr>
<tr>
<td>xor6_sat</td>
<td>0.264197</td>
<td>0.541426</td>
<td>0.981361</td>
</tr>
<tr>
<td>xor7_sat</td>
<td>1.56001</td>
<td>3.54155</td>
<td>5.66529</td>
</tr>
<tr>
<td>xor8_sat</td>
<td>8.72671</td>
<td>20.3925</td>
<td>29.2366</td>
</tr>
<tr>
<td>xor9_sat</td>
<td>44.92</td>
<td>108.92</td>
<td>146.857</td>
</tr>
<tr>
<td>xor10_sat</td>
<td>oom</td>
<td>563.585</td>
<td>728.619</td>
</tr>
<tr>
<td>xor15_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>xor20_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>xor25_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>xor30_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>xor60_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>xor100_sat</td>
<td>oom</td>
<td>t/o</td>
<td>t/o</td>
</tr>
</tbody>
</table>

Table 3: Results from running each solver on instances of the xor problem. If an instance was solved, then the CPU time used by the solver is reported; if the solver ran for more than 900 CPU seconds, then a timeout ('t/o') is reported; if the solver used more than 32GB of memory, then an out-of-memory error ('oom') is reported; otherwise, if a stack overflow is encountered during execution, then a stack overflow ('so') is reported. The integer in the instance name indicates the number of pairs of universal and existential quantifiers in the prefix; instances with a 'sat' suffix are satisfiable, while instances with an 'unsat' suffix have an extra clause added to make them unsatisfiable.
derivation of empty clauses can be beneficial, consider the following instance of
the xor problem:

\[ \forall y_1 \exists x_1 \forall y_2 \exists x_2((y_1 \lor x_1) \land (\neg y_1 \lor \neg x_1) \land (x_1 \lor y_2 \lor x_2) \land (x_1 \lor \neg y_2 \lor \neg x_2) \land (\neg x_1 \lor y_2 \lor \neg x_2) \land (\neg x_1 \lor \neg y_2 \lor x_2)) \]

After assigning \( y_1 \) as true, the formula is simplified to:

\[ \exists x_1 \forall y_2 \exists x_2((\neg x_1) \land (x_1 \lor y_2 \lor x_2) \land (x_1 \lor \neg y_2 \lor \neg x_2) \land (\neg x_1 \lor y_2 \lor \neg x_2) \land (\neg x_1 \lor \neg y_2 \lor x_2)) \]

After also assigning \( x_1 \) as true, the formula is simplified to:

\[ \forall y_2 \exists x_2() \land (y_2 \lor \neg x_2) \land (\neg y_2 \lor x_2) \]

Since the formula now contains an empty clause, it is false, and this recursive call
to the search function will return false. This means that, after \( y_1 \) and \( x_1 \)
both have been assigned as true, no search is done for \( y_2 \) or \( x_2 \). For some instances,
this simplification, leading to search space pruning, might have a large effect.
Unlike the search solver, the naive solver always expands the formula fully, and
does not benefit from any similar simplifications.

Another potential contributing factor is the memory usage of the solvers.
Based on the implementations, the solvers have different memory complexities.
The naive solver creates a formula that has a size that is exponential in the size
of the input formula, while the search solver avoids this exponential memory
requirement. This could affect the runtime negatively, since higher memory usage
often causes worse performance. That being said, the evaluation done here
is not sufficient to determine if this contributes to the naive solvers lower performance,
compared to the search solver. Some further investigation is required
to determine if this is likely to be the case.

The performance of the naive solver could also be negatively impacted if the
compilers are not able to perform certain optimizations. Consider, for example,
the function expand_quantifiers in Listing 15; if no common subexpression
is applied, then this could have a large negative impact on the performance. Again,
the evaluation here is insufficient to determine if this is likely to be the cause
of the lower performance, some further investigation is required to determine if it is.

6.4.2 Target Language Performance Comparison

From Table 2 and Table 3, it is clear that the SML exports of the solvers are faster than
the ones using other target languages. The OCaml, Haskell, and Scala exports all perform better than the other two in at some cases. Out of
these three, no target language is consistently faster than the other two.
Based on this, SML seems like the preferred target language. There is,
however, one potential issue with using SML: It causes more out of memory
errors than any other target language. This could be because of a higher memory
usage, but it could also be the case that the other exports are simple too slow
to run out of memory before the 900 second timeout. A more thorough analysis
in needed to determine exactly what causes the high amount of out of memory
errors.
6.4.3 Solver Scaling for the Xor Problem

Based on Table 2 and Table 3, the CPU time scales rapidly with increasing instance sizes for all solver and both problems, with one notable exception: The CPU time to solve the satisfiable instances of the xor problem using CAQE scales basically linearly, which can be seen in Table 3. The time to solve the unsatisfiable ones also scales well. In both cases, CAQE scales better than for the domino game instances in Table 2.

This might be related to the fact that polynomial-time algorithms exist for quantified Boolean formulas in prenex normal form where the matrix is a conjunction of exclusive or constraints [12, p. 1139]. It seems like CAQE is able to utilize this structure in the matrix to solve the instances more efficiently, which is not something either of the verified solvers are able to do.

It is desirable for a solver to be able to, efficiently, solve QBF instances for which polynomial-time algorithms are known. This evaluation provides some preliminary empirical evidence for CAQE having this property, for conjunctions of xor constraints, and for the verified solvers not having this property. But, a more thorough investigation is needed if one wishes to determine if this is actually the case.

6.4.4 Stack Overflows

Several of the larger instances in Table 2 and Table 3 cause a stack overflow for the Scala exports of the verified solvers. By inspecting the stack traces, it becomes clear that the stack overflows are caused by some basic Isabelle/HOL functions used in the parser, like map, not being tail recursive. This is not desirable, since it puts a low limit on the input size.

There are several ways of addressing this issue. One option is to use a modified definition for map, and other problematic functions, in Isabelle. Another option is to modify the shallow embedding used for the code export to use tail-recursive implementations of the problematic functions. These are just two out of, probably, many possible approaches that could be used to address the stack overflow issue.

7 Related Work

There are three main categories of related work for this project: work related to QBF solving, discussed in Section 7.1; work related to verified SAT solving, discussed in Section 7.2; and work related to verified proof checking, discussed in Section 7.3.

7.1 State of the Art QBF Solvers

There are two main types of state-of-the-art QBF solvers: solvers based on search, and solvers based on variable elimination.\textsuperscript{7} Both types of solvers are discussed here.

\textsuperscript{7}Based on the prenex normal form solvers in QBFEVAL 2020 and 2022. Other solver types with state-of-the-art performance also exist.
7.1.1 Search-based Solvers

QuBE [30] is an early search-based QBF solver, implementing a basic search and propagation procedure similar to the one in the q-DLL algorithm described in Section 2.2.4. The main difference between QuBE and the q-DLL algorithm is that QuBE allows branching on literals not at the highest level, in certain cases. This relaxation of the branching order is taken even further by state-of-the-art search-based QBF solvers, which often incorporate dependency learning [31, 32] to relax the branching order.

Search-based state-of-the-art QBF solvers include miniQU [33], depQBF [34, 35], and Qute [32], all of which use dependency learning. Other techniques used includes watched data structures for detecting unit literals, monotone literals, and quantifiers without any bound variables [36]; and conflict learning [37, 38], which allows new clauses and cubes (conjunctions of literals), that are in conjunction with the matrix, to be learned during search. These techniques are inspired by corresponding techniques in SAT solving, namely the 2-watched literals scheme [39] and conflict clause learning [40, 41]. More techniques are also used in state-of-the-art solvers, examples include heuristics for selecting decision variables and search restarts [33, 34, 32].

7.1.2 Variable Elimination-based Solvers

An early example of a variable elimination-based solver is Quantor [42], which uses resolution to eliminate existential variables and expansion to eliminate universal variables, like the q-DP algorithm described in Section 2.2.5. The main difference is that Quantor uses more advanced heuristics to select variables. More advanced solvers based on variable elimination often use counterexample guided abstraction refinement [43] (CEGAR) to solve QBFs.

The state-of-the-art solver RAReQS [44] utilizes CEGAR and partial expansion of formulas to solve QBFs. This works since partial expansions can be sufficient to determine the satisfiability of a QBF; the solver will gradually expand more parts of the formula until the expansion is sufficient to determine satisfiability.

The state-of-the-art solver CAQE originally used CEGAR based on clausal abstractions [27], which are described in more detail in, for example, [45]. Later versions of the CAQE solver have, however, been extended to also use partial expansions [28], improving the solvers performance.

7.2 Verified SAT Solving

An Isabelle/HOL formalization and total correctness proof of a modern SAT solver, based on MiniSAT [46], is introduced by Marić in [47]. The total correctness proof covers both the high-level algorithms used in the solver, like conflict-driven clause learning, and low-level details, like the two-watched literals scheme. Isabelle’s code generator can be used to extract executable code from the solver formalization.

A formally verified framework for SAT solving using conflict-driven clause learning, based on [48], is introduced by Blanchette et al. in [49]. This framework is then used to implement a formally verified solver, called IsaSAT, through a chain of refinements. An abstract calculus for conflict-driven clause learning is,
with some intermediate steps, connected to a formally verified imperative SAT solver, with total correctness guarantees. This imperative solver uses several optimizations found in modern SAT solvers, including the two-watched literals scheme.

7.3 Verified Proof Checking

An alternative to using formally verified solvers is to use a formally verified checker for proofs generated by solvers. A proof checker for propositional satisfiability has been created by, for example, Tan et al. in [50]. Such a checker can verify the correctness of a proof generated by a solver, with the high confidence gained by using formal verification, but without requiring the solver itself to be formally verified. This is beneficial since unverified SAT solvers currently have better performance than verified ones; comparisons between IsaSAT and unverified solvers can be found in [51] and [52].

Some similar work has also been done for QBFs. A formally verified proof checker for Q-resolution [53] unsatisfiability proofs generated by the QBF solver Squolem [54] has been implemented in the HOL4 interactive theorem prover [55] by Weber [56]. Similar proof checkers for Squolem satisfiability proofs, based on Skolem functions [57], have also been implemented in HOL4 in [58] and HOL Light [59] in [60].

8 Future Work

Two simple QBF solvers have been formalized in this thesis, one based on naive expansion and one based on search. Some different areas of future work in verified QBF solving are briefly discussed in this section.

8.1 More Advanced Algorithms

One relatively straightforward option for future work is to extend the search-based solver with unit propagation and monotone literal detection, yielding a basic Q-DLL solver. This is likely to give a solver with better performance, that could eventually be used to implement search-based state-of-the-art solvers. If more work is done based on the search-based solver created for this thesis, then it might be beneficial to change to a different formalization of the prefix in the prenex normal form datatype.

The current formalization of the prefix, based on an algebraic datatype, is difficult to work with for two main reasons: 1. Many basic operations, like removing the first quantified variable in the prefix, require a lot of different cases to be analysed. 2. Inductive proofs about the prefix have a surprising number of cases that need to be shown, often 7 different cases. Based on my experience using Isabelle/HOL for this project, I think it is likely that this could be avoided by formalizing the prefix as a list of variables tagged as existential or universal. But, it is difficult to say if this assumption is correct or not, without implementing the change.

Another option is to implement solvers based on variable elimination. It is possible to start with either a Q-DP solver or, immediately, a CEGAR solver based on RAReQS, CAQE, or some similar solver. A CEGAR solver might be
able to yield relatively high performance for relatively little work, since they often are implemented as small programs calling SAT solvers, and there are several Isabelle/HOL formalizations of state-of-the-art SAT solvers [27, 28, 44].

8.2 A Verified Proof Checker
As an alternative to implementing more advanced algorithms, a verified checker for proof certificates could be implemented instead. This approach could be beneficial since the performance gap between verified and unverified QBF solvers is big (see Section 6.3), especially compared to the gap between verified and unverified SAT solvers [51, 52].

A potential obstacle is that different QBF solvers produce proof certificates of different types. However, some progress towards a unified certification format has also been made [11]. If a proof checker is created, then a proof certificate format has to be selected, for which there is no, single, obvious choice. Multiple checkers for different certificate formats could also be created. As an alternative to proofs in some syntactic proof calculus, a checker could also be based on Skolem functions and Herbrand functions [57].

8.3 Verifying the Parser and Code Export
The parser the solvers use for reading input is unverified. Work could be done to, for example, prove that all valid inputs are accepted. Formal verification of the parser could be beneficial since bugs also can occur in the parser, not just in the solver itself.

Isabelle’s code export facilities are, also, not formally verified. Future work could use formally verified code generation. There is, for example, a verified compiler from Isabelle/HOL to a subset of SML created by Hupel and Nikpow [61]. Formally verified code generation could be beneficial since the code export also is a potential source of bugs.

9 Conclusion
This thesis has formalized quantified Boolean formulas and their semantics, formalized two simple solvers deciding if a given quantified Boolean formula is satisfiable, and shown their correctness, all in the Isabelle/HOL interactive theorem prover. The first solver naively expands all quantifiers in a formula before evaluating it, while the second solver implements a simple search-based algorithm. To our knowledge, these are the first formally verified solvers for quantified Boolean formulas.

Executable code has been extracted from both the solvers using Isabelle’s code generation facilities. This extracted code is used to measure the performance of the solvers, revealing that the search-based solver outperforms the naive expansion-based one and confirming that both the verified solvers are far behind the state-of-the-art.

In exchange for the lower performance, the solvers implemented here benefit from the strong correctness guarantees gained from interactive theorem proving. These guarantees give unparalleled confidence that the solvers can correctly determine whether a given quantified Boolean formula is satisfiable. Future
work could, for example, implement more advanced algorithms, create a verified proof checker for unverified solvers, or formally verify the parser and use formally verified code generation for the solvers.

References


A.1 QBF Datatype, Semantics, and Satisfiability

A.1.1 A QBF datatype

```isabelle
datatype QBF = Var nat | Neg QBF | Conj QBF list | Disj QBF list | Ex nat QBF | All nat QBF
```

A.1.2 Formalization of Semantics & Termination of Semantics

```isabelle
fun substitute-var :: nat ⇒ bool ⇒ QBF ⇒ QBF where
substitute-var z True (Var z′) = (if z = z′ then Conj [] else Var z′) |
substitute-var z False (Var z′) = (if z = z′ then Disj [] else Var z′) |
substitute-var z b (Neg qbf) = Neg (substitute-var z b qbf) |
substitute-var z b (Conj qbf-list) = Conj (map (substitute-var z b) qbf-list) |
substitute-var z b (Disj qbf-list) = Disj (map (substitute-var z b) qbf-list) |
substitute-var z b (Ex x qbf) = Ex x (if x = z then qbf else substitute-var z b qbf) |
substitute-var z b (All y qbf) = All y (if z = y then qbf else substitute-var z b qbf)

fun qbf-measure :: QBF ⇒ nat where
qbf-measure (Var ·) = 1 |
qbf-measure (Neg qbf) = 1 + qbf-measure qbf |
qbf-measure (Conj qbf-list) = 1 + sum-list (map qbf-measure qbf-list) |
qbf-measure (Disj qbf-list) = 1 + sum-list (map qbf-measure qbf-list) |
qbf-measure (Ex - qbf) = 1 + qbf-measure qbf |
qbf-measure (All - qbf) = 1 + qbf-measure qbf

lemma qbf-measure-substitute: qbf-measure (substitute-var z b qbf) = qbf-measure qbf
proof (induction qbf)
  case (Var x)
  show qbf-measure (substitute-var z b (Var x)) = qbf-measure (Var x)
  proof (cases b)
    case True
    thus ?thesis by simp
  next
    case False
    thus ?thesis by simp
  qed
next
  case (Neg qbf)
  thus qbf-measure (substitute-var z b (Neg qbf)) = qbf-measure (Neg qbf) by simp
next
  case (Conj qbf-list)
  thus qbf-measure (substitute-var z b (Conj qbf-list)) = qbf-measure (Conj qbf-list)
  proof (induction qbf-list)
```

99
case Nil
thus qbf-measure (substitute-var z b (Conj [])) = qbf-measure (Conj []) by simp
next
case (Cons x xs)
thus qbf-measure (substitute-var z b (Conj (x # xs))) = qbf-measure (Conj (x # xs)) by simp
qed
next
case (Disj qbf-list)
thus qbf-measure (substitute-var z b (Disj qbf-list)) = qbf-measure (Disj qbf-list)
proof (induction qbf-list)
case Nil
thus qbf-measure (substitute-var z b (Disj [])) = qbf-measure (Disj []) by simp
next
case (Cons x xs)
thus qbf-measure (substitute-var z b (Disj (x # xs))) = qbf-measure (Disj (x # xs)) by simp
qed
next
case (Ex x qbf)
thus qbf-measure (substitute-var z b (QBF. Ex x qbf)) = qbf-measure (QBF. Ex x qbf)
by simp
next
case (All y qbf)
thus qbf-measure (substitute-var z b (QBF. All y qbf)) = qbf-measure (QBF. All y qbf)
by simp
qed

lemma qbf-measure-lt-sum-list:
assumes qbf ∈ set qbf-list
shows qbf-measure qbf < Suc (sum-list (map qbf-measure qbf-list))
proof –
obtain left right where left @ qbf # right = qbf-list by (metis assms split-list)
hence sum-list (map qbf-measure qbf-list) = sum-list (map qbf-measure left) + qbf-measure qbf + sum-list (map qbf-measure right)
by fastforce
thus qbf-measure qbf < Suc (sum-list (map qbf-measure qbf-list)) by simp
qed

function qbf-semantics :: (nat ⇒ bool) ⇒ QBF ⇒ bool where
qbf-semantics I (Var z) = I z |
qbf-semantics I (Neg qbf) = (~ (qbf-semantics I qbf)) |
qbf-semantics I (Conj qbf-list) = list-all (qbf-semantics I) qbf-list |
qbf-semantics I (Disj qbf-list) = list-ex (qbf-semantics I) qbf-list |
qbf-semantics I (Ex x qbf) = ((qbf-semantics I (substitute-var x True qbf)) ∨ (qbf-semantics I (substitute-var x False qbf))) |
qbf-semantics I (All x qbf) = ((qbf-semantics I (substitute-var x True qbf)) ∧ (qbf-semantics I (substitute-var x False qbf)))
by pat-completeness auto
termination
apply (relation measure (λ(I,qbf). qbf-measure qbf))
by (auto simp add: qbf-measure-substitute qbf-measure-lt-sum-list)
A.1.3 Formalization of Satisfiability

**Definition**  satisfiable :: $QBF \Rightarrow \mathbb{Bool}$ where

$satisfiable \ qbf \ = \ (\exists I. \ qbf\text{-semantics} \ I \ qbf)$

**Definition** logically-eq :: $QBF \Rightarrow QBF \Rightarrow \mathbb{Bool}$ where

$logically-eq \ qbf1 \ qbf2 \ = \ (\forall I. \ qbf\text{-semantics} \ I \ qbf1 = qbf\text{-semantics} \ I \ qbf2)$

A.2 Existential Closure

A.2.1 Formalisation of Free Variables

**Function** free-variables-aux :: $\text{nat set} \Rightarrow QBF \Rightarrow \text{nat list}$ where

$free-variables-aux \ bound \ (\text{Var} \ x) = (\text{if} \ x \in bound \ \text{then} \ [] \ \text{else} \ [x]) \ |$

$free-variables-aux \ bound \ (\text{Neg} \ qbf) = free-variables-aux \ bound \ qbf |$

$free-variables-aux \ bound \ (\text{Conj} \ list) = \text{concat} \ (\text{map} \ (\text{free-variables-aux} \ bound) \ list) |$

$free-variables-aux \ bound \ (\text{Disj} \ list) = \text{concat} \ (\text{map} \ (\text{free-variables-aux} \ bound) \ list) |$

$free-variables-aux \ bound \ (\text{Ex} \ x \ qbf) = free-variables-aux \ (\text{insert} \ x \ bound) \ qbf |$

$free-variables-aux \ bound \ (\text{All} \ x \ qbf) = free-variables-aux \ (\text{insert} \ x \ bound) \ qbf$

**Function** free-variables :: $QBF \Rightarrow \text{nat list}$ where

$free-variables \ qbf = \text{sort} \ (\text{remdups} \ (\text{free-variables-aux} \ \emptyset \ qbf))$

**Lemma** bound-subtract-eqv:

$\text{set} \ (\text{free-variables-aux} \ (\text{bound} \cup \text{new}) \ qbf) = \text{set} \ (\text{free-variables-aux} \ \text{bound} \ qbf) - \text{new}$

_by (induction $\text{bound} \ qbf \ \text{rule: free-variables-aux.induct}) \ auto$

A.2.2 Formalisation of Existential Closure

**Function** existential-closure-aux :: $QBF \Rightarrow \text{nat list} \Rightarrow QBF$ where

$existential-closure-aux \ qbf \ Nil = qbf \ |
existential-closure-aux \ qbf \ (\text{Cons} \ x \ xs) = \text{Ex} \ x \ (\text{existential-closure-aux} \ qbf \ xs)$

**Function** existential-closure :: $QBF \Rightarrow QBF$ where

$existential-closure \ qbf = existential-closure-aux \ qbf \ (\text{free-variables} \ qbf)$

A.2.3 Preservation of Satisfiability under Existential Quantification

**Lemma** swap-substitute-var-order:

_**Assumes** $x1 \neq x2 \lor b1 = b2$

_**Shows** $\text{substitute-var} \ x1 \ b1 \ (\text{substitute-var} \ x2 \ b2 \ qbf)$

$= \text{substitute-var} \ x2 \ b2 \ (\text{substitute-var} \ x1 \ b1 \ qbf)$

**Proof** (induction $qbf$)

**Case** $\text{(Var} \ x)$

**Show** ?case

**Proof** (cases $b2$)

**Case** True

then show ?thesis using assms by (cases $b1$) auto

**Next**

**Case** False

then show ?thesis using assms by (cases $b1$) auto

qed
next
  case (Neg qbf)
  then show ?case by simp
next
  case (Conj x)
  then show ?case by simp
next
  case (Disj x)
  then show ?case by simp
next
  case (Ex x1a qbf)
  then show ?case by simp
next
  case (All x1a qbf)
  then show ?case by simp
qed

lemma remove-outer-substitute-var:
  assumes x1 = x2
  shows substitute-var x1 b1 (substitute-var x2 b2 qbf) = (substitute-var x2 b2 qbf)
using assms
proof (induction qbf)
  case (Var x)
  show ?case
  proof (cases b2)
    case True
    then show ?thesis using assms by (cases b1) auto
  next
    case False
    then show ?thesis using assms by (cases b1) auto
  qed
next
  case (Neg qbf)
  thus ?case by simp
next
  case (Conj x)
  thus ?case by simp
next
  case (Disj x)
  thus ?case by simp
next
  case (Ex x1a qbf)
  thus ?case by simp
next
  case (All x1a qbf)
  thus ?case by simp
qed

lemma qbf-semantics-substitute-eq-assign:
  qbf-semantics I (substitute-var x b qbf) ↔ qbf-semantics (I(x := b)) qbf
proof (induction I(x := b) qbf rule: qbf-semantics.induct)
  case (1 z)
  then show ?case by (cases b) auto
next
case (2 qbf)
then show ?case by simp
next
case (3 qbf-list)
then show ?case by (induction qbf-list) auto
next
case (4 qbf-list)
then show ?case by (induction qbf-list) auto
next
case (5 x' qbf)
thus ?case by (cases x' = x)
(auto simp add: swap-substitute-var-order remove-outer-substitute-var)
next
case (6 x' qbf)
thus ?case by (cases x' = x)
(auto simp add: swap-substitute-var-order remove-outer-substitute-var)
qed

lemma sat-iff-ex-sat: satisfiable qbf ⟷ satisfiable (Ex x qbf)
proof (cases satisfiable qbf)
case True
from this obtain I where I-def: qbf-semantics I qbf unfolding satisfiable-def by blast
have I(x := I x) = I(x := True) ∨ I(x := I x) = I(x := False) by (cases I x) auto
hence I(x := True) ∨ I = I(x := False) by simp
hence qbf-semantics (I(x := True)) qbf ∨ qbf-semantics (I(x := False)) qbf
using I-def by fastforce
moreover have satisfiable (Ex x qbf)
  = (∃ I. qbf-semantics (I(x := True)) qbf
  ∨ qbf-semantics (I(x := False)) qbf)
  by (simp add: satisfiable-def qbf-semantics-substitute-eq-assign)
ultimately have satisfiable (QBF. Ex x qbf) by blast
thus ?thesis using True by simp
next
case False
thus ?thesis unfolding satisfiable-def using qbf-semantics-substitute-eq-assign by simp
qed

A.2.4 Preservation of Satisfiability under Existential Closure

lemma sat-iff-ex-close-aux-sat: satisfiable qbf ⟷ satisfiable (existential-closure-aux qbf vars)
using sat-iff-ex-sat by (induction vars) auto

theorem sat-iff-ex-close-sat: satisfiable qbf ⟷ satisfiable (existential-closure qbf)
using sat-iff-ex-close-aux-sat by simp

A.2.5 Non-existence of Free Variables in Existential Closure

lemma ex-closure-aux-vars-not-free:
set (free-variables (existential-closure-aux qbf vars)) = set (free-variables qbf) − set vars
proof (induction vars)
case Nil
then show ?case by simp
next
  case (Cons x xs)
  thus ?case using bound-subtract-equ[of \{\} \{x\}] by auto
qed

theorem ex-closure-no-free: set (free-variables (existential-closure qbf)) = {}
  using ex-closure-aux-vars-not-free by simp

A.3 Sequence Utility Function

fun sequence-aux :: ‘a option list ⇒ ‘a list ⇒ ‘a list option
where
sequence-aux [] list = Some list |
sequence-aux (Some x # xs) list = sequence-aux xs (x # list) |
sequence-aux (None # xs) list = None

fun sequence :: ‘a option list ⇒ ‘a list option
where
sequence list = map-option rev (sequence-aux list [])

lemma list-no-None-ex-list-map-Some:
  assumes list-all (λx. x ≠ None) list
  shows ∃xs. map Some xs = list
  using assms
proof (induction list)
  case Nil
  show ∃xs. map Some xs = [] by simp
next
  case (Cons a list)
  from this obtain xs where map Some xs = list by fastforce
  moreover from Cons obtain x where Some x = a by fastforce
  ultimately have map Some (x # xs) = a # list by simp
  thus ∃xs. map Some xs = a # list by (rule exI)
qed

lemma sequence-aux-content: sequence-aux (map Some xs) list = Some (rev xs @ list)
proof (induction xs arbitrary: list)
  case Nil
  show sequence-aux (map Some []) list = Some (rev [] @ list) by simp
next
  case (Cons y ys)
  thus sequence-aux (map Some (y # ys)) list = Some (rev (y # ys) @ list) by simp
qed

lemma sequence-content: sequence (map Some xs) = Some xs
proof
  have sequence (map Some xs) = map-option rev (sequence-aux (map Some xs) [])
    by simp
  moreover have sequence-aux (map Some xs) [] = Some (rev xs @ [])
    using sequence-aux-content by fastforce
  ultimately show sequence (map Some xs) = Some xs by simp
qed
A.4 Naive solver

A.4.1 Expanding Quantifiers

fun list-max :: nat list ⇒ nat
where
  list-max Nil = 0 |
  list-max (Cons x xs) = max x (list-max xs)

fun qbf-quantifier-depth :: QBF ⇒ nat
where
  qbf-quantifier-depth (Var x) = 0 |
  qbf-quantifier-depth (Neg qbf) = qbf-quantifier-depth qbf |
  qbf-quantifier-depth (Conj list) = list-max (map qbf-quantifier-depth list) |
  qbf-quantifier-depth (Disj list) = list-max (map qbf-quantifier-depth list) |
  qbf-quantifier-depth (Ex x qbf) = 1 + (qbf-quantifier-depth qbf) |
  qbf-quantifier-depth (All x qbf) = 1 + (qbf-quantifier-depth qbf)

lemma qbf-quantifier-depth-substitute:
  qbf-quantifier-depth (substitute-var z b qbf) = qbf-quantifier-depth qbf

proof (induction qbf)
  case (Var x)
  show ?case by (cases b) auto
next
  case (Neg qbf)
  thus ?case by simp
next
  case (Conj xs)
  thus ?case by (induction xs) auto
next
  case (Disj xs)
  thus ?case by (induction xs) auto
next
  case (Ex x1a qbf)
  thus ?case by simp
next
  case (All x1a qbf)
  thus ?case by simp
qed

lemma qbf-quantifier-depth-eq-max:
  assumes ¬qbf-quantifier-depth z < list-max (map qbf-quantifier-depth qbf-list)
  and z ∈ set qbf-list
  shows qbf-quantifier-depth z = list-max (map qbf-quantifier-depth qbf-list) using assms

proof (induction qbf-list)
  case Nil
  hence False by simp
  thus ?case by simp
next
  case (Cons x xs)
  thus qbf-quantifier-depth z = list-max (map qbf-quantifier-depth (x # xs))
    by (cases x = z) auto
qed

function expand-quantifiers :: QBF ⇒ QBF
where
  expand-quantifiers (Var x) = (Var x) |
expand-quantifiers (Neg qbf) = Neg (expand-quantifiers qbf) |
expand-quantifiers (Conj list) = Conj (map expand-quantifiers list) |
expand-quantifiers (Disj list) = Disj (map expand-quantifiers list) |
expand-quantifiers (Ex x qbf) = (Disj [substitute-var x True (expand-quantifiers qbf),
    substitute-var x False (expand-quantifiers qbf)]) |
expand-quantifiers (All x qbf) = (Conj [substitute-var x True (expand-quantifiers qbf),
    substitute-var x False (expand-quantifiers qbf)])

by pat-completeness auto

termination
apply (relation measures [qbf-quantifier-depth, qbf-measure])
by (auto simp add: qbf-quantifier-depth-substitute qbf-quantifier-depth-eq-max)
(auto simp add: qbf-measure-lt-sum-list)

lemma no-quants-after-expand-quants: qbf-quantifier-depth (expand-quantifiers qbf) = 0
proof (induction qbf)
case (Var x)
  show ?case by simp
next
case (Neg qbf)
  thus ?case by simp
next
case (Conj x)
  thus ?case by (induction x) auto
next
case (Disj x)
  thus ?case by (induction x) auto
next
case (Ex x1a qbf)
  thus ?case using qbf-quantifier-depth-substitute Ex.IH by simp
next
case (All x1a qbf)
  thus ?case using qbf-quantifier-depth-substitute All.IH by simp
qed

lemma semantics-inv-under-expand:
  qbf-semantics I qbf = qbf-semantics I (expand-quantifiers qbf)
proof (induction qbf arbitrary: I)
case (Var x)
  show ?case by force
next
case (Neg qbf)
  thus ?case by simp
next
case (Conj x)
  thus ?case by (induction x) auto
next
case (Disj x)
  thus ?case by (induction x) auto
next
case (Ex x1a qbf)
  thus qbf-semantics I (QBF.Ex x1a qbf) = qbf-semantics I (expand-quantifiers
(QBF.Ex x1a qbf))
    using qbf-semantics-substitute-eq-assign by fastforce
next
  case (All x1a qbf)
    thus qbf-semantics I (QBF.All x1a qbf) = qbf-semantics I (expand-quantifiers (QBF.All x1a qbf))
      using qbf-semantics-substitute-eq-assign by fastforce
qed

lemma sat-iff-expand-quants-sat: satisfiable qbf ⟷ satisfiable (expand-quantifiers qbf)
  unfolding satisfiable-def using semantics-inv-under-expand by simp

lemma set-free-vars-subst-all-eq:
  set (free-variables (substitute-var x b qbf)) = set (free-variables (All x qbf))
proof (induction x b qbf rule: substitute-var.induct)
  case (1 z z')
    then show ?case by simp
next
  case (2 z z')
    then show ?case by simp
next
  case (3 z b qbf)
    then show ?case by simp
next
  case (4 z b qbf-list)
    then show ?case by simp
next
  case (5 z b qbf-list)
    then show ?case by simp
next
  case (6 z b x qbf)
    then show ?thesis by simp
next
proof (cases x = z)
  case True
    thus ?thesis by simp
next
  case False
    hence set (free-variables (substitute-var z b (QBF.Ex x qbf)))
      = set (free-variables (substitute-var z b qbf)) − {x}
        using bound-subtract-equiv[where ?new = {x}] by simp
also have "... = set (free-variables (QBF.All z qbf)) − {x}" using 6 False by simp
also have "... = set (free-variables-aux {x, z} qbf)"
  using bound-subtract-equiv[where ?new = {x}] by simp
also have "... = set (free-variables (QBF.All z (QBF.Ex x qbf)))" by simp
finally show ?thesis.
qed

next
  case (7 z b y qbf)
    thus ?case
    proof (cases y = z)
      case True
        thus ?thesis by simp
next
  case False
    thus ?thesis using 7 bound-subtract-equiv[where ?new = {y}] by simp
lemma set-free-vars-subst-ex-eq:
set (free-variables (substitute-var x b qbf)) = set (free-variables (Ex x qbf))
proof (induction x b qbf rule: substitute-var.induct)
case (1 z z')
then show ?case by simp
next
case (2 z z')
then show ?case by simp
next
case (3 z b qbf)
then show ?case by simp
next
case (4 z b qbf-list)
then show ?case by simp
next
case (5 z b qbf-list)
then show ?case by simp
next
case (6 z b x qbf)
then show ?case
proof (cases x = z)
case True
thus ?thesis by simp
next
case False
thus ?thesis using 6 bound-subtract-equiv[where ?new = {x}] by simp
qed
next
case (7 z b y qbf)
thus ?case
proof (cases y = z)
case True
thus ?thesis by simp
next
case False
thus ?thesis using 7 bound-subtract-equiv[where ?new = {y}] by simp
qed
next

lemma free-vars-inv-under-expand-quant:
set (free-variables (expand-quantifiers qbf)) = set (free-variables qbf)
proof (induction qbf)
case (Var x)
then show ?case by simp
next
case (Neg qbf)
then show ?case by simp
next
case (Conj x)
then show ?case by fastforce
next
case (Dis x)
then show ?case by fastforce
next
case (Ex x a qbf)
have set (free-variables (expand-quantifiers (QBF. Ex x a qbf)))
  = set (free-variables-aux {x a} (expand-quantifiers qbf))
  using set-free-vars-subst-ex-eq by simp
also have ... = set (free-variables (expand-quantifiers qbf)) – {x a}
  using bound-subtract-equiv [where ?new = {x a}] by simp
also have ... = set (free-variables qbf) – {x a} using Ex.IH by simp
also have ... = set (free-variables-aux {x a} qbf)
  using bound-subtract-equiv [where ?new = {x a}] by simp
also have ... = set (free-variables (QBF. Ex x a qbf)) by simp
finally show ?case .
next
case (All x a qbf)
thus ?case using bound-subtract-equiv [where ?new = {x a}]
set-free-vars-subst-all-eq by simp
qed

A.4.2 Expanding Formulas

fun expand-qbf :: QBF ⇒ QBF where
expand-qbf qbf = expand-quantifiers (existential-closure qbf)

lemma sat-iff-expand-qbf-sat:
satisfiable (expand-qbf qbf) ←→ satisfiable qbf
using sat-iff-ex-close-sat sat-iff-expand-quants-sat by simp

lemma expand-qbf-no-free:
set (free-variables (expand-qbf qbf)) = {}
pseudo
have set (free-variables (expand-qbf qbf)) = set (free-variables (existential-closure qbf))
  using free-vars-inv-under-expand-quants by simp
thus ?thesis using ex-closure-no-free by metis
qed

lemma expand-qbf-no-quants:
qbf-quantifier-depth (expand-qbf qbf) = 0
using no-quants-after-expand-quants by simp

A.4.3 Evaluating Expanded Formulas

fun eval-qbf :: QBF ⇒ bool option where
eval-qbf (Var x) = None |
eval-qbf (Neg qbf) = map-option (λ x. ¬ x) (eval-qbf qbf) |
eval-qbf (Conj list) = map-option (list-all id) (sequence (map eval-qbf list)) |
eval-qbf (Dis list) = map-option (list-ex id) (sequence (map eval-qbf list)) |
eval-qbf (Ex x qbf) = None |
eval-qbf (All x qbf) = None

lemma pred-map-ex: list-ex Q (map f x) = list-ex (Q o f) x
  by (induction x) auto

lemma eval-qbf-implements-semantics:
  assumes set (free-variables qbf) = {} and qbf-quantifier-depth qbf = 0
  shows eval-qbf qbf = Some (qbf-semantics I qbf) using assms

109
proof (induction qbf)
  case (Var x)
  then show ?case by simp
next
  case (Neg qbf)
  then show ?case by simp
next
  case (Conj x)
  hence \( \forall q \in \text{set } x. \text{eval-qbf } q = \text{Some } (qbf-semantics I q) \) by (induction x) auto
  thus \( \text{eval-qbf } (\text{Conj } x) = \text{Some } (qbf-semantics I (\text{Conj } x)) \)
proof (induction x)
  case Nil
  show \( \text{eval-qbf } (\text{Conj } []) = \text{Some } (qbf-semantics I (\text{Conj } [])) \) by simp
next
  case (Cons y ys)
  have \( \text{map eval-qbf } ys = \text{map Some } (\text{map } (qbf-semantics I) ys) \) using Cons by simp
moreover have \( \text{eval-qbf } y = \text{Some } (qbf-semantics I y) \) using Cons.prems by simp
ultimately have \( \text{map eval-qbf } (y \# ys) = \text{map Some } (\text{map } (qbf-semantics I) (y \# ys)) \) by simp
  hence \( \text{sequence } (\text{map eval-qbf } (y \# ys)) = \text{Some } (\text{map } (qbf-semantics I) (y \# ys)) \)
    using sequence-content by metis
  hence \( \text{eval-qbf } (\text{Conj } (y \# ys)) = \text{Some } (\text{list-all id } (\text{map } (qbf-semantics I) (y \# ys))) \)
    by simp
  hence \( \text{eval-qbf } (\text{Conj } (y \# ys)) = \text{Some } (\text{list-all } (qbf-semantics I) (y \# ys)) \)
    by (simp add: fun.map-ident list.pred-map)
  thus \( \text{eval-qbf } (\text{Conj } (y \# ys)) = \text{Some } (qbf-semantics I (\text{Conj } (y \# ys))) \) by simp
qed

next
  case (Disj x)
  hence \( \forall q \in \text{set } x. \text{eval-qbf } q = \text{Some } (qbf-semantics I q) \) by (induction x) auto
  thus \( \text{eval-qbf } (\text{Disj } x) = \text{Some } (qbf-semantics I (\text{Disj } x)) \)
proof (induction x)
  case Nil
  show \( \text{eval-qbf } (\text{Disj } []) = \text{Some } (qbf-semantics I (\text{Disj } [])) \) by simp
next
  case (Cons y ys)
  have \( \text{map eval-qbf } ys = \text{map Some } (\text{map } (qbf-semantics I) ys) \) using Cons by simp
moreover have \( \text{eval-qbf } y = \text{Some } (qbf-semantics I y) \) using Cons.prems by simp
ultimately have \( \text{map eval-qbf } (y \# ys) = \text{map Some } (\text{map } (qbf-semantics I) (y \# ys)) \) by simp
  hence \( \text{sequence } (\text{map eval-qbf } (y \# ys)) = \text{Some } (\text{map } (qbf-semantics I) (y \# ys)) \)
    using sequence-content by metis
  hence \( \text{eval-qbf } (\text{Disj } (y \# ys)) = \text{Some } (\text{list-ex id } (\text{map } (qbf-semantics I) (y \# ys))) \)
    by simp
  hence \( \text{eval-qbf } (\text{Disj } (y \# ys)) = \text{Some } (\text{list-ex } (qbf-semantics I) (y \# ys)) \)

110
by (simp add: fun.map_ident pred_map_ex)
thus eval-qbf (Disj (y ≠ ys)) = Some (qbf-semantics I (Disj (y ≠ ys))) by simp
qed

next
case (Ex x1a qbf)
  hence False by simp
thus ?case by simp
next
case (All x1a qbf)
  hence False by simp
thus ?case by simp
qed

A.4.4 Naive Solver

fun naive-solver :: QBF ⇒ bool where
  naive-solver qbf = the (eval-qbf (expand-qbf qbf))

theorem naive-solver-correct: naive-solver qbf ←→ satisfiable qbf
proof −
  have ∀ I. naive-solver qbf = the (Some (qbf-semantics I (expand-qbf qbf)))
    using expand-qbf-no-free expand-qbf-no-quants eval-qbf-implements-semantics by simp
  hence naive-solver qbf = satisfiable (expand-qbf qbf) unfolding satisfiable_def by simp
  thus naive-solver qbf = satisfiable qbf using sat_iffexpand-qbf-sat by simp
qed

B Isabelle/HOL Theory: PCNF Datatype

theory PCNF
imports Main NaiveSolver
begin

B.1 A Prenex Conjugate Normal Form Datatype

datatype literal = P nat | N nat

type-synonym clause = literal list

type-synonym matrix = clause list

type-synonym quant-set = nat × nat list

type-synonym quant-sets = quant-set list

datatype prefix =
    UniversalFirst quant-set quant-sets |
    ExistentialFirst quant-set quant-sets |
    Empty

type-synonym pcnf = prefix × matrix

111
B.1.1 PCNF Predicate for Generic QBFs

fun literal-p :: QBF ⇒ bool where
literal-p (Var -) = True |
literal-p (Neg (Var -)) = True |
literal-p - = False

fun clause-p :: QBF ⇒ bool where
clause-p (Disj list) = list-all literal-p list |
clause-p - = False

fun cnf-p :: QBF ⇒ bool where
cnf-p (Conj list) = list-all clause-p list |
cnf-p - = False

fun pcnf-p :: QBF ⇒ bool where
pcnf-p (Ex - qbf) = pcnf-p qbf |
pcnf-p (All - qbf) = pcnf-p qbf |
pcnf-p (Conj list) = cnf-p (Conj list) |
pcnf-p - = False

B.1.2 Bijection with PCNF subset of Generic QBF Datatype

fun convert-literal :: literal ⇒ QBF where
convert-literal (P z) = Var z |
convert-literal (N z) = Neg (Var z)

lemma convert-literal-p: literal-p (convert-literal lit)
by (cases lit) auto

fun convert-literal-inv :: QBF ⇒ literal option where
convert-literal-inv (Var z) = Some (P z) |
convert-literal-inv (Neg (Var z)) = Some (N z) |
convert-literal-inv - = None

lemma literal-inv: convert-literal-inv (convert-literal lit) = Some lit
by (cases lit) auto

fun convert-clause :: clause ⇒ QBF where
convert-clause cl = Disj (map convert-literal cl)

lemma convert-clause-p: clause-p (convert-clause cl)
using convert-literal-p by (induction cl) auto

fun convert-clause-inv :: QBF ⇒ clause option where
convert-clause-inv (Disj list) = sequence (map convert-literal-inv list) |
convert-clause-inv - = None

lemma clause-inv: convert-clause-inv (convert-clause cl) = Some cl
proof –
let ?list = map convert-literal-inv (map convert-literal cl)
have ∀ x ∈ set cl. convert-literal-inv (convert-literal x) = Some x using literal-inv
by simp
hence map Some $cl = ?list$ using list-no-None-ex-list-map-Some by fastforce
hence sequence ?list = Some $cl$ using sequence-content by metis
thus convert-clause-inv (convert-clause $cl$) = Some $cl$ by simp
qed

fun convert-matrix :: $\text{matrix} \Rightarrow QBF$ where
convert-matrix matrix = Conj (map convert-clause matrix)

lemma convert-cnf-p: $\text{cnf-p} (convert-matrix \text{mat})$
using convert-clause-p by (induction mat) auto

fun convert-matrix-inv :: $QBF \Rightarrow \text{matrix option}$ where
convert-matrix-inv (Conj list) = sequence (map convert-clause-inv list) |
convert-matrix-inv = None

lemma matrix-inv: convert-matrix-inv (convert-matrix $\text{mat}$) = Some $\text{mat}$
proof
  let ?list = map convert-clause-inv (map convert-clause $\text{mat}$)
  have $\forall x \in \text{set mat}. \text{convert-clause-inv} (convert-clause x) = Some x$
  using clause-inv by simp
  hence map Some $\text{mat}$ = ?list using list-no-None-ex-list-map-Some by fastforce
  hence sequence ?list = Some $\text{mat}$ using sequence-content by metis
  thus convert-matrix-inv (convert-matrix $\text{mat}$) = Some $\text{mat}$ by simp
qed

fun $q$-length :: '$a \times 'a list $\Rightarrow$ nat where
$q$-length $(x, xs) = 1 + \text{length} xs$

fun measure-prefix-length :: $\text{pcnf} \Rightarrow \text{nat}$ where
measure-prefix-length (Empty, -) = 0 |
measure-prefix-length (UniversalFirst $q$ qs, -) = $q$-length $q$ + sum-list (map $q$-length qs) |
measure-prefix-length (ExistentialFirst $q$ qs, -) = $q$-length $q$ + sum-list (map $q$-length qs)

function convert :: $\text{pcnf} \Rightarrow QBF$ where
convert (Empty, matrix) = convert-matrix matrix |
convert (UniversalFirst $(x, []) []$, matrix) = All $x$ (convert (Empty, matrix)) |
convert (ExistentialFirst $(x, []) []$, matrix) = Ex $x$ (convert (Empty, matrix)) |
convert (UniversalFirst $(x, []) (q \# q's)$, matrix) = $x$ (convert (UniversalFirst $q$ $q's$, matrix)) |
convert (ExistentialFirst $(x, []) (q \# q's)$, matrix) = Ex $x$ (convert (ExistentialFirst $y$ $y's$ $q's$, matrix)) |
convert (UniversalFirst $(x, y \# y's)$ $q's$, matrix) = $x$ (convert (UniversalFirst $(y, y's)$ $q's$, matrix)) |
convert (ExistentialFirst $(x, y \# y's)$ $q's$, matrix) = Ex $x$ (convert (ExistentialFirst $(y, y's)$ $q's$, matrix))
by pat-completeness auto
termination
  by (relation measure measure-prefix-length) auto

theorem convert-pcnf-p: $\text{pcnf-p} (\text{convert pcnf})$
using convert-cnf-p by (induction rule: convert.induct) auto

113
fun add-universal-to-front :: nat ⇒ pcnf ⇒ pcnf where
add-universal-to-front x (Empty, matrix) = (UniversalFirst (x, []) [], matrix) |
add-universal-to-front x (UniversalFirst (y, ys) qs, matrix) = (UniversalFirst (x, y # ys) qs, matrix) |
add-universal-to-front x (ExistentialFirst (y, ys) qs, matrix) = (UniversalFirst (x, []) ((y, ys) # qs), matrix)

fun add-existential-to-front :: nat ⇒ pcnf ⇒ pcnf where
add-existential-to-front x (Empty, matrix) = (ExistentialFirst (x, []) [], matrix) |
add-existential-to-front x (ExistentialFirst (y, ys) qs, matrix) = (ExistentialFirst (x, y # ys) qs, matrix) |
add-existential-to-front x (UniversalFirst (y, ys) qs, matrix) = (ExistentialFirst (x, []) ((y, ys) # qs), matrix)

fun convert-inv :: QBF ⇒ pcnf option where
convert-inv (All x qbf) = map-option (λp. add-universal-to-front x p) (convert-inv qbf) |
convert-inv (Ex x qbf) = map-option (λp. add-existential-to-front x p) (convert-inv qbf) |
convert-inv qbf = map-option (λm. (Empty, m)) (convert-matrix-inv qbf)

lemma convert-add-all: convert (add-universal-to-front x pcnf) = All x (convert pcnf)
  by (induction rule: add-universal-to-front.induct) auto

lemma convert-add-ex: convert (add-existential-to-front x pcnf) = Ex x (convert pcnf)
  by (induction rule: add-existential-to-front.induct) auto

theorem convert-inv: convert-inv (convert pcnf) = Some pcnf
proof (induction pcnf)
  case (Pair prefix matrix)
  show convert-inv (convert (prefix, matrix)) = Some (prefix, matrix)
    using matrix-inv by (induction rule: convert.induct) auto
qed

theorem convert-injective: inj convert
  apply (rule inj-on-inverseI[where ?g = the o convert-inv])
  by (simp add: convert-inv)

lemma convert-literal-p-ex:
  assumes literal-p lit
  shows ∃l. convert-literal l = lit
proof —
  have ∃l. convert-literal l = Var x for x using convert-literal.simps by blast
  moreover have ∃l. convert-literal l = Neg (Var x) for x using convert-literal.simps
    by blast
  ultimately show ∃l. convert-literal l = lit
    using assms by (induction rule: literal-p.induct) auto
qed

lemma convert-clause-p-ex:
  assumes clause-p cl
shows $\exists c. \text{convert-clause } c = cl$

proof –
from assms obtain $xs$ where $\emptyset: \text{Disj } xs = cl$ by (metis clause-p.elims(2))
hence $\text{list-all literal-p } xs$ using assms by fastforce
hence $\exists ys. \text{map convert-literal } ys = xs$ using convert-literal-p-ex
proof (induction $xs$)
  case Nil
  show $\exists ys. \text{map convert-literal } ys = []$ by simp
next
  case (Cons $x$ $xs$)
  from this obtain $ys$ where $\text{map convert-literal } ys = xs$ by fastforce
  moreover from Cons obtain $y$ where $\text{convert-literal } y = x$ by fastforce
  ultimately have $\text{map convert-literal } (y \# ys) = x \# xs$ by simp
  thus $\exists ys. \text{map convert-literal } ys = x \# xs$ by (rule exI)
qed
thus $\exists c. \text{convert-clause } c = cl$ using $\emptyset$ by fastforce
qed

lemma convert-cnf-p-ex:
  assumes cnf-p $mat$
  shows $\exists m. \text{convert-matrix } m = mat$
proof –
from assms obtain $xs$ where $\emptyset: \text{Conj } xs = mat$ by (metis cnf-p.elims(2))
hence $\text{list-all clause-p } xs$ using assms by fastforce
hence $\exists ys. \text{map convert-clause } ys = xs$ using convert-clause-p-ex
proof (induction $xs$)
  case Nil
  show $\exists ys. \text{map convert-clause } ys = []$ by simp
next
  case (Cons $x$ $xs$)
  from this obtain $ys$ where $\text{map convert-clause } ys = xs$ by fastforce
  moreover from Cons obtain $y$ where $\text{convert-clause } y = x$ by fastforce
  ultimately have $\text{map convert-clause } (y \# ys) = x \# xs$ by simp
  thus $\exists ys. \text{map convert-clause } ys = x \# xs$ by (rule exI)
qed
thus $\exists m. \text{convert-matrix } m = mat$ using $\emptyset$ by fastforce
qed

theorem convert-pcnf-p-ex:
  assumes pcnf-p $qbf$
  shows $\exists \text{pcnf}. \text{convert pcnf } = qbf$ using assms
proof (induction $qbf$)
  case (Var $x$)
  hence False by simp
  thus $\exists case$ by simp
next
  case (Neg $qbf$)
  hence False by simp
  thus $\exists case$ by simp
next
  case (Conj $x$)
  hence $\text{cnf-p } (\text{Conj } x) \text{ by simp}$
  from this obtain $m$ where $\text{convert-matrix } m = \text{Conj } x$ using convert-cnf-p-ex by blast
hence convert (Empty, m) = Conj x by simp
thus ∃pcnf. convert pcnf = Conj x by (rule exI)
next
case (Disj x)
hence False by simp
thus ?case by simp
next
case (Ex x1a qbf)
from this obtain pcnf where convert pcnf = qbf by fastforce
hence convert (add-existential-to-front x1a pcnf) = Ex x1a qbf using convert-add-ex
by simp
thus ∃pcnf. convert pcnf = QBF.Ex x1a qbf by (rule exI)
next
case (All x1a qbf)
from this obtain pcnf where convert pcnf = qbf by fastforce
hence convert (add-universal-to-front x1a pcnf) = All x1a qbf using convert-add-all
by simp
thus ∃pcnf. convert pcnf = QBF.All x1a qbf by (rule exI)
qed

theorem convert-range: range convert = {p. pcnf-p p}
using convert-convert-p convert-convert-p-ex by blast

theorem convert-bijective-on: bij-betw convert UNIV {p. pcnf-p p}
by (simp add: bij-betw-def convert-injective convert-range)

B.1.3 Preservation of Semantics under the Bijection

fun literal-semantics :: (nat ⇒ bool) ⇒ literal ⇒ bool where
literal-semantics I (P x) = I x |
literal-semantics I (N x) = (¬I x)

fun clause-semantics :: (nat ⇒ bool) ⇒ clause ⇒ bool where
clause-semantics I clause = list-ex (literal-semantics I) clause

fun matrix-semantics :: (nat ⇒ bool) ⇒ matrix ⇒ bool where
matrix-semantics I matrix = list-all (clause-semantics I) matrix

function pcnf-semantics :: (nat ⇒ bool) ⇒ pcnf ⇒ bool where
pcnf-semantics I (UniversalFirst (y, []) [], matrix) =

(matrix-semantics I matrix |
pcnf-semantics I (UniversalFirst y, []) [], matrix) =

(pcnf-semantics I (y := True)) (Empty, matrix) |
∧ pcnf-semantics I (y := False)) (Empty, matrix)) |

(pcnf-semantics I (x := True)) (Empty, matrix) |
∨ pcnf-semantics I (x := False)) (Empty, matrix)) |

(pcnf-semantics I (UniversalFirst (y, q) # qs, matrix) =

(pcnf-semantics I (y := True)) (ExistentialFirst q qs, matrix) |
∧ pcnf-semantics I (y := False)) (ExistentialFirst q qs, matrix)) |

(pcnf-semantics I (x := True)) (UniversalFirst q qs, matrix) |
∨ pcnf-semantics I (x := False)) (UniversalFirst q qs, matrix)) |

(pcnf-semantics I (UniversalFirst (y, y # ys) qs, matrix) =

\[(\text{pcnf-semantics} (I (y := True))) (\text{UniversalFirst} (yy, ys) qs, matrix) \land \text{pcnf-semantics} (I (y := False)) (\text{UniversalFirst} (yy, ys) qs, matrix)) \mid \text{pcnf-semantics} I (\text{ExistentialFirst} (x, xx \neq xs) qs, matrix) = (\text{pcnf-semantics} (I (x := True)) (\text{ExistentialFirst} (xx, xs) qs, matrix) \lor \text{pcnf-semantics} (I (x := False)) (\text{ExistentialFirst} (xx, xs) qs, matrix))\]

by pat-completeness auto

termination
by (relation measure (\lambda(I,p). measure-prefix-length p)) auto

theorem qbf-semantics-eq-pcnf-semantics:
\[\text{pcnf-semantics} I \text{pcnf} = \text{qbf-semantics} I (\text{convert pcnf})\]

proof (induction pcnf arbitrary: I rule: convert.induct)
  case (1 matrix)
  then show ?case
  proof (induction matrix)
    case Nil
    then show ?case by simp
  next
    case (Cons cl cls)
    then show ?case
    proof (induction cl)
      case Nil
      then show ?case by simp
    next
      case (Cons l ls)
      then show ?case by (induction l) force+
      qed
      qed
  next
    case (2 x matrix)
    then show ?case using convert.simps(2) pcnf-semantics.simps(2) qbf-semantics.simps(6) qbf-semantics-substitute-eq-assign by presburger
  next
    case (3 x matrix)
    then show ?case using convert.simps(3) pcnf-semantics.simps(3) qbf-semantics.simps(5) qbf-semantics-substitute-eq-assign by presburger
  next
    case (4 x q qs matrix)
    then show ?case using qbf-semantics-substitute-eq-assign by fastforce
  next
    case (5 x q qs matrix)
    then show ?case using qbf-semantics-substitute-eq-assign by fastforce
  next
    case (6 x y ys qs matrix)
    then show ?case using qbf-semantics-substitute-eq-assign by fastforce
  next
    case (7 x y ys qs matrix)
    then show ?case using qbf-semantics-substitute-eq-assign by fastforce
    qed

lemma convert-inv-inv:
\[\text{pcnf-p qbf} \Rightarrow \text{convert (the (convert-inv qbf))} = \text{qbf}\]
by (metis convert-inv convert-pcnf-p-ex option.sel)
C  Isabelle/HOL Theory: QDIMACS Parser

theory Parser
  imports Main PCNF
begin

  type_synonym 'a parser = string ⇒ ('a × string) option

  fun trim-ws :: string ⇒ string where
    trim-ws Nil = Nil |
    trim-ws (Cons x xs) =
      (if x = CHR '"' '"' then trim-ws xs else Cons x xs)

  lemma non-increasing-trim-ws [simp]: length (trim-ws s) ≤ length s
    by (induction s) auto

  lemma [intro]: length s ≤ length s' ⇒ length (trim-ws s) ≤ length s'
    by (induction s) auto

  lemma [intro]: length s < length s' ⇒ length (trim-ws s) < length s'
    apply (induction s')
    apply simp
    using le-trans non-increasing-trim-ws by blast

  lemma [intro]: length s ≤ length (trim-ws s') ⇒ length s ≤ length s'
    apply (induction s')
    apply simp
    using non-increasing-trim-ws order.strict-trans2 by blast

  lemma whitespace-and-parse-le [intro]:
    assumes ∀ s' r. p s = Some (r, s') ⇒ length s' ≤ length s
    shows ∀ s' r. p (trim-ws s) = Some (r, s') ⇒ length s' ≤ length s using assms
    using le-trans non-increasing-trim-ws by blast

  lemma whitespace-and-parse-unit-le [intro]:
    assumes ∀ s' r. p s = Some ((), s') ⇒ length s' ≤ length s
    shows ∀ s' r. (trim-ws s) = Some ((), s') ⇒ length s' ≤ length s using assms
    using le-trans non-increasing-trim-ws by blast

  lemma whitespace-and-parse-less [intro]:
    assumes ∀ s' r. p s = Some (r, s') ⇒ length s' < length s
    shows ∀ s' r. p (trim-ws s) = Some (r, s') ⇒ length s' < length s using assms
    using non-increasing-trim-ws order-less-le-trans by blast
lemma whitespace-and-parse-unit-less [intro]:
assumes \( \forall s, p \ s = \text{Some} ((), s') \implies \text{length} s' < \text{length} s \)
shows \( \forall s, p \ (\text{trim-ws} s) = \text{Some} ((), s') \implies \text{length} s' < \text{length} s \) using assms
using non-increasing-trim-\(s\) order-less-le-trans by blast

fun match :: string \(\Rightarrow\) unit parser where
match Nil str = Some ((), str) |
match (Cons x xs) Nil = None |
match (Cons x xs) (Cons y ys) = (if \( x \neq y \) then None else match xs ys)

lemma non-increasing-match [simp]: match xs s = Some ((), s') \(\implies\) \(\text{length} s' \leq\) \(\text{length} s\)
by (induction xs s rule: match.induct) ((case-tac \( x = y \).auto)+)

lemma decreasing-match [simp]:
\( xs \neq [] \implies\) match \( xs \ s = \text{Some} ((), s') \implies \text{length} s' < \text{length} s\)

proof (induction \( xs \ s \) rule: match.induct)
  case (1 \( x \ \text{str} \))
  hence False by simp
  thus ?case by simp
next
  case (2 \( x \ \text{xs} \))
  hence False by simp
  thus ?case by simp
next
  case (3 \( x \ \text{xs} \ \text{ys} \))
  hence \( x = y \) by (cases \( x = y \).auto)
  hence match (Cons x xs) (Cons y ys) = match \( xs \ \text{ys} \) by simp
  hence match \( xs \ \text{ys} = \text{Some} ((), s') \) using 3 by simp
  hence \( \text{length} s' \leq\) \(\text{length} \text{ys} \) by simp
  thus \( \text{length} s' <\) \(\text{length} \ (\text{Cons} \ \text{ys} \ \text{ys}) \) by simp
qed

fun digit-to-nat :: char \(\Rightarrow\) nat option where
digit-to-nat \( c \) =
  if \( c = \text{CHR} "0" \) then Some 0 else
  if \( c = \text{CHR} "1" \) then Some 1 else
  if \( c = \text{CHR} "2" \) then Some 2 else
  if \( c = \text{CHR} "3" \) then Some 3 else
  if \( c = \text{CHR} "4" \) then Some 4 else
  if \( c = \text{CHR} "5" \) then Some 5 else
  if \( c = \text{CHR} "6" \) then Some 6 else
  if \( c = \text{CHR} "7" \) then Some 7 else
  if \( c = \text{CHR} "8" \) then Some 8 else
  if \( c = \text{CHR} "9" \) then Some 9 else
  None

fun num-aux :: nat \(\Rightarrow\) nat parser where
num-aux \( n \Nil \) = Some \( (n, \Nil) \) |
num-aux \( n \) (Cons \( x \) \( \text{xs} \)) =
  (if List.member "0123456789" \( x \)
    then num-aux \((10 * n + \text{the (digit-to-nat}) \) \( \text{xs} \)
    else Some \( (n, \text{Cons} \ \text{xs}) \))
lemma non-increasing-num-aux [simp]: num-aux n s = Some (m, s') → length s' ≤ length s
by (induction n s rule: num-aux.induct) ((case-tac List.member "0123456789" x.auto)+)

fun pnum-raw :: nat parser where
pnum-raw Nil = None |
pnum-raw (Cons x xs) = (if List.member "0123456789" x then num-aux 0 (Cons x xs) else None)

lemma decreasing-pnum-raw [simp]: pnum-raw s = Some (n, s') → length s' < length s
apply (cases s)
apply simp
apply (metis impossible-Cons nat-less-le non-increasing-num-aux num-aux.simps(2) option.simps(3)
     pnum-raw.simps(2))
done

fun pnum :: nat parser where
pnum str = (case pnum-raw str of
  None ⇒ None |
  Some (n, str') ⇒ if n = 0 then None else Some (n, str'))

lemma decreasing-pnum [simp]:
  assumes pnum s = Some (n, s')
  shows length s' < length s
proof (cases pnum-raw s)
case None
hence False using assms by simp
thus ?thesis by simp
next
case (Some a)
  obtain n' s'' where a = (n', s'') by fastforce
  then show ?thesis using Some assms by (cases n' = 0) auto
qed

fun literal :: PCNF.literal parser where
literal str = (case match "--" str of
  None ⇒ (case pnum str of
    None ⇒ None |
    Some (n, str') ⇒ Some (P n, str')) |
  Some (l, str') ⇒ (case pnum str' of
    None ⇒ None |
    Some (n, str'') ⇒ Some (N n, str'')))
thus \(?\)thesis using assms by (cases pnum s) auto

next
case (Some a)
from this obtain \(s''\) where \(s''\)-def: \(a = ((), s'')\) by (cases match \("\)\) s) auto

hence \(\text{length } s'' \leq \text{length } s\) using Some by simp

moreover have \(\text{length } s' < \text{length } s''\) using \(s''\)-def assms Some by (cases pnum s'') auto

ultimately show \(\text{length } s' < \text{length } s\) by simp

qed

fun clause :: PCNF.clause parser where
clause str = (case literal (trim-ws str) of
  None \Rightarrow None |
  Some (l, str') \Rightarrow (case clause str' of
    None \Rightarrow None |
    Some (cl, str'') \Rightarrow Some (Cons l Nil, str'')) |
  Some (cl, str'') \Rightarrow Some (Cons l cl, str''))

lemma decreasing-clause [simp]:
  assumes clause s = Some (c, s')
  shows \(\text{length } s' < \text{length } s\) using assms
proof (induction s arbitrary: c rule: clause.induct)
case (1 s)
  show \(?\)thesis
proof (cases literal (trim-ws s))
  case None
  hence False using 1 by simp
  thus \(?\)thesis by simp
next
case Some-a: (Some a)
  obtain l s'' where a-def: \(a = (l, s'')\) by fastforce
  hence less1: \(\text{length } s'' < \text{length } s\) using Some-a by fastforce
  show \(?\)thesis
proof (cases clause s'')
  case None'
  show \(?\)thesis
proof (cases match \("\) (trim-ws s''))
    case None
    hence False using 1 Some-a a-def None' by simp
    thus \(?\)thesis by simp
next
case Some-b: (Some b)
  obtain u s''' where b-def: \(b = (u, s''')\) by (meson surj-pair)
  hence le1: \(\text{length } s''' \leq \text{length } s''\) using Some-b by fastforce
  show \(?\)thesis
proof (cases match \("\) (trim-ws s'''))
  case None
hence False using 1 Some-a a-def None' Some-b b-def by simp
thus ?thesis by simp
next
case Some-c: (Some c)
obtain a s''' where c-def: c = (u, s''') by (meson surj-pair)
hence le2: length s''' \leq length s by fastforce
have clause s = Some (Cons l Nil, s''')
  using Some-a a-def None' Some-b b-def Some-c c-def by simp
hence s''' = s' using 1(2) by simp
thus length s' < length s using less le1 le2 by simp
qed
qed
next
case Some-b: (Some b)
obtain c' s''' where b-def: b = (c', s''') by fastforce
hence clause s = Some (Cons l c', s'''') using Some-a Some-b a-def by simp
hence s''' = s' using 1(2) by simp
hence clause s'' = Some (c', s'') using Some-b b-def by simp
hence length s' < length s'' using 1(1) Some-a a-def by blast
thus length s' < length s using less le1 by simp
qed
qed
qed

fun clause-list :: PCNF.matrix parser where
clause-list str = (case clause str of
  None \Rightarrow None |
  Some (cl, str') \Rightarrow
case clause-list str' of
    None \Rightarrow Some (Cons cl Nil, str') |
    Some (cls, str'') \Rightarrow Some (Cons cl cls, str'')))

lemma decreasing-clause-list [simp]:
  assumes clause-list s = Some (cls, s')
  shows length s' < length s using assms
proof (induction s arbitrary: cls rule: clause-list.induct)
case (1 s)
show ?thesis
proof (cases clause s)
case None
  hence False using 1 by simp
thus ?thesis by simp
next
case Some-a: (Some a)
obtain cl s'' where a-def: a = (cl, s'') by fastforce
hence less1: length s'' < length s using Some-a by simp
show ?thesis
proof (cases clause-list s'')
case None
  hence clause-list s = Some (Cons cl Nil, s'') using Some-a a-def by simp
  hence s'' = s'' using 1 by simp
  thus length s' < length s using less le1 by simp
next
case Some-b: (Some b)
  obtain cls s'' where b-def: b = (cls, s'') by fastforce
  hence clause-list s = Some (Cons cl cls, s'') using Some-b Some-a a-def by simp
  hence s' = s'' using 1 by simp
  hence clause-list s'' = Some (cls, s') using Some-b b-def by simp
  hence length s' < length s'' using 1 Some-a a-def by blast
  thus ?thesis using less1 by simp
qed

fun matrix :: PCNF.matrix parser where
matrix s = clause-list s

lemma decreasing-matrix [simp]: matrix s = Some (mat, s') \implies length s' < length s
by simp

fun atom-set :: (nat \times nat list) parser where
atom-set str = (case pnum (trim-ws str) of
  None \Rightarrow None |
  Some (a, str') \Rightarrow
  (case atom-set str' of
    None \Rightarrow Some ((a, Nil), str') |
    Some ((a', as), str'') \Rightarrow Some ((a, Cons a' as), str''))

lemma decreasing-atom-set [simp]:
  assumes atom-set s = Some (as, s')
  shows length s' < length s using assms
proof (induction s arbitrary: as rule: atom-set.induct)
  case (1 s)
  show ?case
  proof (cases pnum (trim-ws s))
    case None
    hence False using 1 by simp
    thus ?thesis by simp
  next
    case Some-b: (Some b)
    obtain a s'' where b-def: b = (a, s'') by fastforce
    hence less1: length s'' < length s using Some-b b-def by fastforce
    show ?thesis
    proof (cases atom-set s'')
      case None
      hence atom-set s = Some ((a, Nil), s'') using Some-b b-def by simp
      hence s' = s'' using 1 by simp
      thus length s' < length s using less1 by simp
    next
      case Some-c: (Some c)
      obtain a-set s''' where c = (a-set, s''') by fastforce
      moreover obtain a' as where a-set = (a', as) by fastforce
      ultimately have c-def: c = ((a', as), s''') by simp
      hence atom-set s = Some ((a, Cons a' as), s''') using Some-c Some-b b-def by
simp
  hence \( s' = s'' \) using \( 1 \) by simp
  hence \( \text{atom-set } s'' = \text{Some } ((a', as), s') \) using \( \text{Some-c } c\)-def by simp
  hence \( \text{length } s' < \text{length } s'' \) using \( 1 \) Some-b b-def by blast
  thus \( \text{length } s' < \text{length } s \) using less1 by simp
qed
qed
qed

datatype quant = Universal | Existential

fun quantifier :: quant parser where
  quantifier str = (case match \( \"e\" \) str of
    None \Rightarrow (case match \( \"a\" \) str of
      None \Rightarrow None |
      Some (-, str') \Rightarrow Some (Universal, str') |
      Some (-, str') \Rightarrow Some (Existential, str')))

lemma non-increasing-quant [simp]:
  assumes quantifier s = Some (q, s')
  shows \( \text{length } s' \leq \text{length } s \)
proof (cases match \( \"e\" \) s)
  case None-e: None
  hence False using None-e assms by simp
  thus \( \text{?thesis} \) by simp
next
  case Some-a: (Some a)
  obtain u s'' where a-def: \( a = (u, s'') \) by (meson surj-pair)
  hence quantifier s = Some (Universal, s'') using None-e Some-a by simp
  hence \( s' = s'' \) using assms by simp
  thus \( \text{length } s' \leq \text{length } s \) using Some-a a-def by simp
qed
next
  case Some-a: (Some a)
  obtain u s'' where a-def: \( a = (u, s'') \) by (meson surj-pair)
  hence quantifier s = Some (Existential, s'') using Some-a by simp
  hence \( s' = s'' \) using assms by simp
  thus \( \text{length } s' \leq \text{length } s \) using Some-a a-def by simp
qed

fun quant-set :: (quant \times (nat \times nat list)) parser where
  quant-set str = (case quantifier (trim-ws str) of
    None \Rightarrow None |
    Some (q, str') \Rightarrow (case atom-set (trim-ws str') of
      None \Rightarrow None |
      Some (as, str'') \Rightarrow (case match \( \"0\" \) (trim-ws str'') of
        None \Rightarrow None |
        Some (-, str''') \Rightarrow (case match \( \[\] \) (trim-ws str''') of
          None \Rightarrow None |
          Some as \Rightarrow

124
lemma decreasing-quant-set [simp]:
  assumes quant-set s = Some \((q\text{-set}, s')\)
  shows length s' < length s
proof (cases quantifier (trim-us s))
  case None
  hence False using assms by simp
thus ?thesis by simp
next
  case Some-a: (Some a)
    obtain \(q s''\) where a-def: \(a = (q, s'')\) by fastforce
    hence le1: length s'' \(\leq\) length s using Some-a by fastforce
    show ?thesis
    proof (cases atom-set (trim-us s''))
      case None
      hence False using Some-a a-def assms by simp
      thus ?thesis by simp
    next
      case Some-b: (Some b)
        obtain as s''' where b-def: \(b = (as, s''')\) by fastforce
        hence less1: length s''' < length s using Some-b by fastforce
        show ?thesis
        proof (cases match '→' '−' '0' (trim-us s''''))
          case None
          hence False using Some-a a-def Some-b b-def assms by simp
          thus ?thesis by simp
        next
          case Some-c: (Some c)
            obtain u s'''' where c-def: \(c = (u, s''')\) by (meson surj-pair)
            hence le2: length s'''' \(\leq\) length s''' using Some-c by fastforce
            show ?thesis
            proof (cases match '→' '−' (trim-us s''''))
              case None
              hence False using Some-a a-def Some-b b-def Some-c c-def assms by simp
              thus ?thesis by simp
            next
              case Some-d: (Some d)
                obtain u s''''' where d-def: \(d = (u, s''')\) by (meson surj-pair)
                hence le3: length s''''' \(\leq\) length s''' using Some-d by fastforce
                have quant-set s = Some \((q, as), s''''\)
                  using Some-a a-def Some-b b-def Some-c c-def Some-d d-def by simp
                hence s' = s'''' using assms by simp
                thus length s' < length s using less1 le1 le2 le3 by simp
                qed
                qed
                qed
fun quant-sets :: (quant \times nat \times nat list) \to parser where
quant-sets str = (case quant-set str of
None \Rightarrow None |
Some \((q\text{-set, } str')\) \Rightarrow
\begin{align*}
& (\text{case quant-sets str' of} \\
& \quad \text{None } \Rightarrow \text{Some (Cons q-set Nil, str')} \mid \\
& \quad \text{Some (q-sets, str'')} \Rightarrow \text{Some (Cons q-set q-sets, str'')})
\end{align*}

\textbf{lemma} decreasing-quant-sets [simp]:
\textbf{assumes} quant-sets \(s = \text{Some (q-sets, s')}\)
\textbf{shows} length \(s' weight<\) length \(s\) \text{using asms}
\textbf{proof} (induction \(s\) arbitrary: quant-sets.induct)
\textbf{case (1 s)}
\textbf{show} ?case
\textbf{proof} (cases quant-set \(s\))
\textbf{case None}
\textbf{hence} False \text{using 1 by simp}
\textbf{thus} ?thesis \text{by simp}
\textbf{next}
\textbf{case Some-a: (Some a)}
\textbf{obtain} q-set \(s''\) \text{where a-def: a = (q-set, s'') by fastforce}
\textbf{hence} less1: length \(s'' weight<\) length \(s\) \text{using Some-a by simp}
\textbf{show} ?thesis
\textbf{proof} (cases quant-sets \(s''\))
\textbf{case None}
\textbf{hence} quant-sets \(s = \text{Some (Cons q-set Nil, s'')}\) \text{using Some-a a-def by simp}
\textbf{hence} \(s'' weight=\) s' \text{using 1 by simp}
\textbf{thus} length \(s'' weight<\) length \(s\) \text{using less1 by simp}
\textbf{next}
\textbf{case Some-b: (Some b)}
\textbf{obtain} q-sets \(s'''\) \text{where b-def: b = (q-sets, s''') by fastforce}
\textbf{hence} quant-sets \(s = \text{Some (Cons q-set q-sets, s''')}\) \text{using Some-a a-def Some-b by simp}
\textbf{hence} \(s'' weight=\) \(s''' weight=\) s' \text{using 1 by simp}
\textbf{hence} quant-sets \(s'' = \text{Some (q-sets, s')}\) \text{using Some-b b-def by simp}
\textbf{hence} length \(s'' weight<\) length \(s'''\) \text{using 1 Some-a a-def by simp}
\textbf{thus} length \(s'' weight<\) length \(s\) \text{using less1 by simp}
\textbf{qed}
\textbf{qed}
\textbf{fun} convert-quant-sets :: \((\text{quant} \times (\text{nat} \times \text{nat list}))\) \(\text{list} \Rightarrow\) PCNF.prefix option \text{where}
\textbf{convert-quant-sets Nil} = Some Empty
\textbf{convert-quant-sets} (Cons (Universal, as) qs) =
\begin{align*}
& (\text{case convert-quant-sets qs of} \\
& \quad \text{None } \Rightarrow \text{None} \mid \\
& \quad \text{Some Empty } \Rightarrow \text{Some (UniversalFirst as Nil)} \mid \\
& \quad \text{Some (ExistentialFirst as' qs')} \Rightarrow \text{Some (UniversalFirst as (Cons as' qs'))} \mid \\
& \quad \text{Some (UniversalFirst - -) } \Rightarrow \text{None} \mid
\end{align*}
\textbf{convert-quant-sets} (Cons (Existential, as) qs) =
\begin{align*}
& (\text{case convert-quant-sets qs of} \\
& \quad \text{None } \Rightarrow \text{None} \mid \\
& \quad \text{Some Empty } \Rightarrow \text{Some (ExistentialFirst as Nil)} \mid \\
& \quad \text{Some (ExistentialFirst - -) } \Rightarrow \text{None} \mid \\
& \quad \text{Some (UniversalFirst as' qs')} \Rightarrow \text{Some (ExistentialFirst as (Cons as' qs'))})
\end{align*}
fun prefix :: PCNF.prefix parser where
prefix str = (case quant-sets str of
  None ⇒ Some (Empty, str) |
  Some (pre, str') ⇒
    (case convert-quant-sets pre of
      None ⇒ None |
      Some converted ⇒ Some (converted, str')))

lemma non-increasing-prefix [simp]:
  assumes prefix s = Some (pre, s')
  shows length s' ≤ length s using assms
proof (cases quant-sets s)
  case None
  hence prefix s = Some (Empty, s) by simp
  hence s' = s using assms by simp
  thus length s' ≤ length s by simp
  next
  case (Some a)
  obtain pre s'' where a-def: a = (pre, s'') by fastforce
  moreover have s' = s'' using Some assms by (cases convert-quant-sets pre) auto
  hence length s' < length s using Some a-def by simp
  ultimately show length s' ≤ length s by simp
qed

fun problem-line :: (nat × nat) parser where
problem-line str = (case match "p" (trim-us str) of
  None ⇒ None |
  Some (a, str1) ⇒
    (case match "cnf" (trim-ws str1) of
      None ⇒ None |
      Some (b, str2) ⇒
        (case pnum (trim-ws str2) of
          None ⇒ None |
          Some (c, str3) ⇒
            (case pnum (trim-ws str3) of
              None ⇒ None |
              Some (clauses, str4) ⇒
                (case match "p" (trim-ws str4) of
                  None ⇒ None |
                  Some (d, str5) ⇒ Some ((lits, clauses), str5)))))

lemma decreasing-problem-line [simp]:
  assumes problem-line s = Some (res, s')
  shows length s' < length s
proof (cases match "p" (trim-us s))
  case None
  hence False using assms by simp
  thus ?thesis by simp
  next
  case Some-a: (Some a)
  obtain u s1 where a-def: a = (u, s1) by (meson surj-pair)
  hence le1: length s1 ≤ length s using Some-a by fastforce
show \(?\text{thesis}\)
proof (cases match \("\text{cnf}\"") (trim-ws s1))
  case None
  hence False using Some-a a-def assms by simp
  thus \(?\text{thesis}\) by simp
next
  case Some-b: (Some b)
  obtain u s2 where b-def: \(b = (u, s2)\) by (meson surj-pair)
  hence le2: length s2 \(\leq\) length s1 using Some-b by fastforce
  show \(?\text{thesis}\)
    proof (cases pnum (trim-ws s2))
      case None
      hence False using Some-a a-def assms by simp
      thus \(?\text{thesis}\) by simp
    next
      case Some-c: (Some c)
      obtain lits s3 where c-def: \(c = (lits, s3)\) by fastforce
      hence less1: length s3 < length s2 using Some-c by fastforce
      show \(?\text{thesis}\)
        proof (cases match \("\text{cnf}\"") (trim-ws s3))
          case None
          hence False using Some-a a-def Some-b b-def assms by simp
          thus \(?\text{thesis}\) by simp
        next
          case Some-d: (Some d)
          obtain clauses s4 where d-def: \(d = (clauses, s4)\) by fastforce
          hence less2: length s4 < length s3 using Some-d by fastforce
          show \(?\text{thesis}\)
            proof (cases match \("\text{cnf}\"") (trim-ws s4))
              case None
              hence False using Some-a a-def Some-b b-def Some-c c-def assms by simp
              thus \(?\text{thesis}\) by simp
            next
              case Some-e: (Some e)
              obtain u s5 where e-def: \(e = (u, s5)\) by (meson surj-pair)
              hence problem-line s = Some ((lits, clauses), s5)
                using Some-a a-def Some-b b-def Some-c c-def assms by simp
              hence s' = s5 using assms by simp
                hence match \("\text{cnf}\"") (trim-ws s4) = Some (u, s') using Some-e e-def by simp
                hence length s' \(\leq\) length s4 by fastforce
                thus length s' < length s using le1 le2 less1 less2 by simp
          qed
          qed
          qed
          qed
        qed
    qed
  qed
  qed
  qed
  qed

fun consume-text :: unit parser where
consume-text Nil = Some ((), Nil) |
consume-text (Cons x xs) = (if x = CHR \("\text{cnf}\"") then Some ((), Cons x xs) else consume-text xs)
lemma non-increasing-consume-text \([\text{simp}]\): consume-text \(s = \text{Some} ((), s') \implies \text{length } s' \leq \text{length } s \)
by (induction \(s\) rule: consume-text.induct) \(((\text{case-tac } x = \text{CHR } '\text{←}' \text{,auto}+)\)

fun comment-line :: unit \(\Rightarrow\) parser where
comment-line str = (case match "c" (trim-ws str) of
None \Rightarrow None |
Some (-, str') \Rightarrow
(case consume-text str' of
None \Rightarrow None |
Some (-, str'') \Rightarrow
(case match '←' str''' of
None \Rightarrow None |
Some (-, str''') \Rightarrow Some (((), str'''))))

lemma decreasing-comment-line \([\text{simp}]\):
assumes comment-line \(s = \text{Some} ((), s')\)
shows length \(s' < \text{length } s\)
proof (cases match "c" \(\leftrightarrow\) \(\text{trim-ws } s\))
case None
hence False using assms by simp
thus ?thesis by simp
next
case Some-a: (Some a)
obtain u s1 where a-def: \(a = (u, s1)\) by (meson surj-pair)
hence less1: length \(s1 < \text{length } s\)
using Some-a decreasing-match[\(\text{of } "c" \text{ trim-ws } s s1\)] by fastforce
show ?thesis
proof (cases consume-text \(s1\))
case None
hence False using Some-a a-def assms by simp
thus ?thesis by simp
next
case Some-b: (Some b)
obtain u s2 where b-def: \(b = (u, s2)\) by (meson surj-pair)
hence le1: length \(s2 \leq \text{length } s1\) using Some-b by simp
show ?thesis
proof (cases match '←'' s2)
case None
hence False using Some-a a-def Some-b b-def assms by simp
thus ?thesis by simp
next
case Some-c: (Some c)
obtain u s3 where c-def: \(c = (u, s3)\) by (meson surj-pair)
hence comment-line \(s = \text{Some} (u, s3)\) using Some-a a-def Some-b b-def Some-c by simp
hence \(s3 = s'\) using assms by simp
hence match '←' s3 = Some (u, s') using Some-c c-def by simp
hence length \(s \leq \text{length } s2\) by simp
thus length \(s' < \text{length } s\) using less1 le1 by simp
qed
qed

129
fun comment-lines :: unit parser where
  comment-lines str = (case comment-line str of
    None ⇒ None | Some (-, str') ⇒ (case comment-lines str' of
      None ⇒ Some ((), str') | Some (-, str'') ⇒ Some ((), str'')))

lemma decreasing-comment-lines [simp]:
  assumes comment-lines s = Some ((), s')
  shows length s' < length s using assms
proof (induction s rule: comment-lines.induct)
  case (1 s)
  show ?case
    proof (cases comment-line s)
      case None
      hence False using 1 by simp
      thus ?thesis by simp
    next
      case Some-a: (Some a)
      obtain u s1 where a-def: a = (u, s1) by (meson surj-pair)
      hence less1: length s1 < length s using Some-a by simp
      show ?thesis
        proof (cases comment-lines s1)
          case None
          hence comment-lines s = Some ((), s1) using Some-a a-def by simp
          hence s1 = s' using 1 by simp
          thus length s' < length s using less1 by simp
        next
          case Some-b: (Some b)
          obtain u s2 where b-def: b = (u, s2) by (meson surj-pair)
          hence comment-lines s = Some ((), s2) using Some-a a-def Some-b by simp
          hence s2 = s' using 1 by simp
          hence comment-lines s1 = Some ((), s') using Some-a a-def Some-b b-def by simp
          hence length s' < length s1 using 1 Some-a a-def by blast
          thus length s' < length s using less1 by simp
          qed
          qed
          qed

fun preamble :: (nat × nat) parser where
  preamble str = (case comment-lines str of
    None ⇒ problem-line str | Some (-, str') ⇒ problem-line str')

lemma decreasing-preamble [simp]:
  assumes preamble s = Some (p, s')
  shows length s' < length s
proof (cases comment-lines s)
case None
hence preamble s = problem-line s by simp
thus ?thesis using assms by simp
next
case (Some a)
  obtain p s'' where a-def: a = (p, s'') by (meson surj-pair)
hence preamble s = problem-line s'' using Some by simp
hence length s' < length s'' using assms by simp
moreover have length s'' < length s using Some a-def by simp
ultimately show length s' < length s by simp
qed

fun eof :: unit parser where
eof Nil = Some (((), Nil)) |
eof (Cons x xs) = None

lemma eof-nil [simp]: eof s = Some (((), s')) ⇔ s' = Nil
  by (cases s) auto

fun input :: PCNF.pcnf parser where
input str = (case preamble str of
  None ⇒ None |
  Some ((lits, clauses), str') ⇒
  (case prefix str' of
    None ⇒ None |
    Some (pre, str'') ⇒
    (case matrix str'' of
      None ⇒ None |
      Some (mat, str''') ⇒
      (case eof str''' of
        None ⇒ None |
        Some (_) ⇒ Some ((pre, mat, str''''))))))

lemma input-nil [simp]:
  assumes input s = Some ((p, s'))
  shows s' = Nil using assms
  proof (cases preamble s)
    case None
    hence False using preamble s
    thus ?thesis by simp
  next
    case Some-a: (Some a)
    obtain p s1 where a-def: a = (p, s1) by (meson surj-pair)
    show ?thesis
    proof (cases prefix s1)
      case None
      hence False using Some-a a-def assms by simp
      thus ?thesis by simp
    next
      case Some-b: (Some b)
      obtain pre s2 where b-def: b = (pre, s2) by fastforce
      show ?thesis
      proof (cases matrix s2)

131
case None
  hence False using Some-a a-def Some-b b-def assms by simp
  thus ?thesis by simp
next
  case Some-c: (Some c)
  obtain mat s3 where c-def: c = (mat, s3) by fastforce
  show ?thesis
  proof (cases eof s3)
    case None
    hence False using Some-a a-def Some-b b-def Some-c c-def assms by simp
    thus ?thesis by simp
  next
    case Some-d: (Some d)
    obtain u s4 where d-def: d = (u, s4) by (meson surj-pair)
    hence input s = Some ((pre, mat), s4)
      using Some-a a-def Some-b b-def Some-c c-def Some-d d-def by simp
    hence s4 = s’ using assms by simp
    moreover have s4 = Nil using Some-d d-def by simp
    ultimately show s’ = Nil by simp
  qed
  qed
  qed
  qed

fun parse :: String.literal ⇒ pcnf option where
parse str = map-option fst (input (String.explode str))

end

D Isabelle/HOL Theory: Search Solver

theory SearchSolver
  imports Main PCNF
begin

D.1 Formalisation of PCNF Assignment

fun lit-neg :: literal ⇒ literal where
  lit-neg (P l) = N l |
  lit-neg (N l) = P l

fun lit-var :: literal ⇒ nat where
  lit-var (P l) = l |
  lit-var (N l) = l

fun remove-lit-neg :: literal ⇒ clause ⇒ clause where
  remove-lit-neg lit clause = filter (λl. l ≠ lit-neg lit) clause

fun remove-lit-clauses :: literal ⇒ matrix ⇒ matrix where
  remove-lit-clauses lit matrix = filter (λcl. ¬(list-ex (λl. l = lit) cl)) matrix

fun matrix-assign :: literal ⇒ matrix ⇒ matrix where
matrix-assign lit matrix = remove-lit-clauses lit (map (remove-lit-neg lit) matrix)

fun prefix-pop :: prefix ⇒ prefix where
prefix-pop Empty = Empty |
prefix-pop (UniversalFirst (x, Nil) Nil) = Empty |
prefix-pop (UniversalFirst (x, Nil) (Cons (y, ys) qs)) = ExistentialFirst (y, ys) qs |
prefix-pop (UniversalFirst (x, (Cons xx xs) qs) = UniversalFirst (xx, xs) qs |
prefix-pop (ExistentialFirst (x, Nil) Nil) = Empty |
prefix-pop (ExistentialFirst (x, Nil) (Cons (y, ys) qs)) = UniversalFirst (y, ys) qs |
prefix-pop (ExistentialFirst (x, (Cons xx xs)) qs) = ExistentialFirst (xx, xs) qs

fun add-universal-to-prefix :: nat ⇒ prefix ⇒ prefix where
add-universal-to-prefix x Empty = UniversalFirst (x, []) [] |
add-universal-to-prefix x (UniversalFirst (y, ys) qs) = UniversalFirst (x, y # ys) q |
add-universal-to-prefix x (ExistentialFirst (y, ys) qs) = ExistentialFirst (x, [] ((y, ys) # q)

fun add-existential-to-prefix :: nat ⇒ prefix ⇒ prefix where
add-existential-to-prefix x Empty = ExistentialFirst (x, []) [] |
add-existential-to-prefix x (ExistentialFirst (y, ys) qs) = ExistentialFirst (x, [] ((y, ys) # q)

fun quant-sets-measure :: quant-sets ⇒ nat where
quant-sets-measure Nil = 0 |
quant-sets-measure (Cons (x, xs) qs) = 1 + length xs + quant-sets-measure qs

fun prefix-measure :: prefix ⇒ nat where
prefix-measure Empty = 0 |
prefix-measure (UniversalFirst q qs) = quant-sets-measure (Cons q qs) |
prefix-measure (ExistentialFirst q qs) = quant-sets-measure (Cons q qs)

lemma prefix-pop-decreases-measure:
  prefix ≠ Empty ==> prefix-measure (prefix-pop prefix) < prefix-measure prefix
  by (induction rule: prefix-pop.induct) auto

function remove-var-prefix :: nat ⇒ prefix ⇒ prefix where
remove-var-prefix x Empty = Empty |
remove-var-prefix x (UniversalFirst (y, ys) qs) = (if x = y then remove-var-prefix x (prefix-pop (UniversalFirst (y, ys) qs)) else add-universal-to-prefix y (remove-var-prefix x (prefix-pop (UniversalFirst (y, ys) qs)))) |
remove-var-prefix x (ExistentialFirst (y, ys) qs) = (if x = then remove-var-prefix x (prefix-pop (ExistentialFirst (y, ys) qs)) else add-existential-to-prefix y (remove-var-prefix x (prefix-pop (ExistentialFirst (y, ys) qs))))
  by pat-completeness auto
termination
  by (relation measure (λ(x, pre), prefix-measure pre))
    (auto simp add: prefix-pop-decreases-measure simp del: prefix-measure.simps)

fun pcnf-assign :: literal ⇒ pcnf ⇒ pcnf where
pcnf-assign lit (prefix, matrix) =
   (remove-var-prefix (lit-var lit) prefix, matrix-assign lit matrix)

D.2 Effect of PCNF Assignments on the Set of all Free
Variables

D.2.1 Variables, Prefix Variables, and Free Variables

fun variables-aux :: QBF ⇒ nat list where
   variables-aux (Var x) = [x] |
   variables-aux (Neg qbf) = variables-aux qbf |
   variables-aux (Conj list) = concat (map variables-aux list) |
   variables-aux (Disj list) = concat (map variables-aux list) |
   variables-aux (Ex x qbf) = variables-aux qbf |
   variables-aux (All x qbf) = variables-aux qbf

fun variables :: QBF ⇒ nat list where
   variables qbf = sort (remdups (variables-aux qbf))

fun prefix-variables-aux :: QBF ⇒ nat list where
   prefix-variables-aux (All y qbf) = Cons y (prefix-variables-aux qbf) |
   prefix-variables-aux (Ex x qbf) = Cons x (prefix-variables-aux qbf) |
   prefix-variables-aux ⊥ = Nil

fun prefix-variables :: QBF ⇒ nat list where
   prefix-variables qbf = sort (remdups (prefix-variables-aux qbf))

fun pcnf-variables :: pcnf ⇒ nat list where
   pcnf-variables pcnf = variables (convert pcnf)

fun pcnf-prefix-variables :: pcnf ⇒ nat list where
   pcnf-prefix-variables pcnf = prefix-variables (convert pcnf)

fun pcnf-free-variables :: pcnf ⇒ nat list where
   pcnf-free-variables pcnf = free-variables (convert pcnf)

lemma free-assgn-proof-skeleton:
   free = var − pre ⇒ free-assgn = var-assgn − pre-assgn
   var-assgn ⊆ var − lit
   ⇒ pre-assgn = pre − lit
   ⇒ free-assgn ⊆ free − lit
   by auto

D.2.2 Free Variables is Variables without Prefix Variables

lemma lit-p-free-eq-vars:
   literal-p qbf ⇒ set (free-variables qbf) = set (variables qbf)
   by (induction qbf rule: literal-p.induct) auto

lemma cl-p-free-eq-vars:
   assumes clause-p qbf
   shows set (free-variables qbf) = set (variables qbf)
   proof −
obtain qbf-list where list-def: qbf = Disj qbf-list
  using assms by (induction qbf rule: clause-p.induct) auto
moreover from this have list-all literal-p qbf-list using assms by simp
ultimately show ?thesis using lit-p-free-eq-vars by (induction qbf-list arbitrary: qbf) auto
qed

lemma cnf-p-free-eq-vars:
  assumes cnf-p qbf
  shows set (free-variables qbf) = set (variables qbf)
proof −
  obtain qbf-list where list-def: qbf = Conj qbf-list
    using assms by (induction qbf rule: cnf-p.induct) auto
moreover from this have list-all clause-p qbf-list using assms by simp
ultimately show ?thesis using cl-p-free-eq-vars by (induction qbf-list arbitrary: qbf) auto
qed

lemma pcnf-p-free-eq-vars-minus-prefix-aux:
  pcnf-p qbf =⇒ set (free-variables qbf) = set (variables qbf) − set (prefix-variables-aux qbf)
proof (induction qbf rule: prefix-variables-aux.induct)
  case (1 y qbf)
    thus ?case using bound-subtract-equiv[of {} {y} qbf] by force
next
  case (2 x qbf)
    thus ?case using bound-subtract-equiv[of {} {x} qbf] by force
next
  case (3-1 v)
    hence False by simp
    thus ?case by simp
next
  case (3-2 v)
    hence False by simp
    thus ?case by simp
next
  case (3-3 v)
    hence cnf-p (Conj v) by (induction Conj v rule: pcnf-p.induct) auto
    thus ?case using cnf-p-free-eq-vars by fastforce
next
  case (3-4 v)
    hence False by simp
    thus ?case by simp
qed

lemma pcnf-p-free-eq-vars-minus-prefix:
  pcnf-p qbf =⇒ set (free-variables qbf) = set (variables qbf) − set (prefix-variables qbf)
  using pcnf-p-free-eq-vars-minus-prefix-aux by simp

lemma pcnf-free-eq-vars-minus-prefix:
  set (pcnf-free-variables pcnf) = set (pcnf-variables pcnf) − set (pcnf-prefix-variables pcnf)
  using pcnf-p-free-eq-vars-minus-prefix convert-pcnf-p by simp
D.2.3 Set of Matrix Variables is Non-increasing under PCNF Assignments

lemma lit-not-in-matrix-assign-variables:
  lit-var lit \notin set (variables (convert-matrix (matrix-assign lit matrix)))
proof (induction matrix)
  case Nil
  then show ?case by simp
next
  case (Cons cl cls)
  then show ?case
  proof (induction cl)
    case Nil
    then show ?case by simp
  next
    case (Cons l ls)
    then show ?case
    proof (induction l)
      case (P x)
      then show ?case by (cases x = x') auto
    next
      case (N x')
      then show ?case by (cases x = x') auto
  qed
next
  case (N x)
  then show ?case
  proof (induction lit)
    case (P x')
    then show ?case by (cases x = x') auto
  next
    case (N x')
    then show ?case by (cases x = x') auto
  qed
qed

lemma matrix-assign-vars-subseteq-matrix-vars-minus-lit:
  set (variables (convert-matrix (matrix-assign lit matrix)))
  \subseteq set (variables (convert-matrix matrix)) \setminus \{lit-var lit\}
using lit-not-in-matrix-assign-variables by force

lemma pcnf-vars-eq-matrix-vars:
  set (pcnf-variables (prefix, matrix))
  = set (variables (convert-matrix matrix))
by (induction (prefix, matrix) arbitrary; prefix rule: convert.induct) auto

lemma pcnf-assign-vars-subseteq-vars-minus-lit:
  set (pcnf-variables (pcnf-assign x pcnf))
  \subseteq set (pcnf-variables pcnf) \setminus \{lit-var x\}
by (induction pcnf) simp

D.2.4 PCNF Assignment Removes Variable from Prefix

lemma add-ex-adds-prefix-var:
  set (pcnf-prefix-variables (add-existential-to-front x pcnf))
  = set (pcnf-prefix-variables pcnf) ∪ {x}
  using convert-add-ex bound-subtract-equiv[of {} {x} convert pcnf] by auto

lemma add-ex-to-prefix-eq-add-to-front:
  (add-existential-to-prefix x prefix, matrix) = add-existential-to-front x (prefix, matrix)
  by (induction prefix) auto

lemma add-all-adds-prefix-var:
  set (pcnf-prefix-variables (add-universal-to-front x pcnf))
  = set (pcnf-prefix-variables pcnf) ∪ {x}
  using convert-add-all bound-subtract-equiv[of {} {x} convert pcnf] by auto

lemma add-all-to-prefix-eq-add-to-front:
  (add-universal-to-prefix x prefix, matrix) = add-universal-to-front x (prefix, matrix)
  by (induction prefix) auto

lemma prefix-assign-vars-eq-prefix-vars-minus-lit:
  set (pcnf-prefix-variables (remove-var-prefix x prefix, matrix))
  = set (pcnf-prefix-variables (prefix, matrix)) − {x}
proof (induction (prefix, matrix) arbitrary: prefix rule: convert.induct)
  case 1
  then show ?case by simp
  next
    case (2 x)
    then show ?case by simp
  next
    case (3 x)
    then show ?case by simp
  next
    case (4 x q qs)
    then show ?case
      using add-all-adds-prefix-var add-all-to-prefix-eq-add-to-front by (induction q) auto
  next
    case (5 x q qs)
    then show ?case using add-ex-adds-prefix-var add-ex-to-prefix-eq-add-to-front by
      (induction q) auto
  next
    case (6 x y ys qs)
    then show ?case using add-all-adds-prefix-var add-all-to-prefix-eq-add-to-front by
      auto
  next
    case (7 x y ys qs)
    then show ?case using add-ex-adds-prefix-var add-ex-to-prefix-eq-add-to-front by
      auto
  qed
lemma prefix-vars-matrix-inv:
set (pcnf-prefix-variables (prefix, matrix1))
= set (pcnf-prefix-variables (prefix, matrix2))
by (induction (prefix, matrix1) arbitrary; prefix rule: convert.induct) auto

lemma pcnf-prefix-vars-eq-prefix-minus-lit:
set (pcnf-prefix-variables (pcnf-assign x pcnf))
= set (pcnf-prefix-variables pcnf) \{lit-var x\}
using prefix-assign-vars-eq-prefix-vars-minus-lit prefix-vars-matrix-inv
by (induction pcnf) fastforce

D.2.5 Set of Free Variables is Non-increasing under PCNF Assignments

theorem pcnf-assign-free-subseteq-free-minus-lit:
set (pcnf-free-variables (pcnf-assign x pcnf)) \subseteq set (pcnf-free-variables pcnf) \{lit-var x\}
using free-assign-proof-skeleton[OF
pcnf-free-eq-vars-minus-prefix[of pcnf]
pcnf-free-eq-vars-minus-prefix[of pcnf-assign x pcnf]
pcnf-assign-vars-subseteq-vars-minus-lit[of x pcnf]
pcnf-prefix-vars-eq-prefix-minus-lit[of x pcnf]].

D.3 PCNF Existential Closure

D.3.1 Formalization of PCNF Existential Closure

fun pcnf-existential-closure :: pcnf \Rightarrow pcnf where
pcnf-existential-closure pcnf = the (convert-inv (existential-closure (convert pcnf)) )

D.3.2 PCNF Existential Closure Preserves Satisfiability

lemma ex-closure-aux-pcnf-p-inv:
pcnf-p qbf \Rightarrow pcnf-p (existential-closure-aux qbf vars)
by (induction qbf vars rule: existential-closure-aux.induct) auto

lemma ex-closure-pcnf-p-inv:
pcnf-p qbf \Rightarrow pcnf-p (existential-closure qbf)
using ex-closure-aux-pcnf-p-inv by simp

theorem pcnf-sat-iff-ex-close-sat:
satisfiable (convert pcnf) = satisfiable (convert (pcnf-existential-closure pcnf))
using convert-inv-inv convert-pcnf-p ex-closure-pcnf-p-inv sat-iff-ex-close-sat by auto

D.3.3 No Free Variables in PCNF Existential Closure

theorem pcnf-ex-closure-no-free:
pcnf-free-variables (pcnf-existential-closure pcnf) = []
using convert-inv-inv convert-pcnf-p ex-closure-pcnf-p-inv ex-closure-no-free by auto

D.4 Search Solver (Part 1: Preliminaries)

D.4.1 Conditions for True and False PCNF Formulas

lemma single-clause-variables:
set (pcnf-variables (Empty, [cl])) = set (map lit-var cl)

proof (induction cl)
  case Nil
  then show ?case by simp
next
  case (Cons l ls)
  then show ?case by (induction l) auto
qed

lemma empty-prefix-cons-matrix-variables:
  set (pcnf-variables (Empty, Cons cl cls))
  = set (pcnf-variables (Empty, cls)) ∪ set (map lit-var cl)
  using single-clause-variables by auto

lemma false-if-empty-clause-in-matrix:
  [] ∈ set matrix ⇒ pcnf-semantics I (prefix, matrix) = False
  by (induction I (prefix, matrix) arbitrary: prefix rule: pcnf-semantics.induct)
  (induction matrix, auto)

lemma true-if-matrix-empty:
  matrix = [] ⇒ pcnf-semantics I (prefix, matrix) = True
  by (induction I (prefix, matrix) arbitrary: prefix rule: pcnf-semantics.induct) auto

lemma matrix-shape-if-no-variables:
  pcnf-variables (Empty, matrix) = [] ⇒ (∃ n. matrix = replicate n [])
proof (induction matrix)
  case Nil
  then show ?case by simp
next
  case (Cons cl cls)
  show ?case
  proof (cases cl = Nil)
    case True
    from this obtain n where cls = replicate n [] using Cons by fastforce
    hence cl ≠ cls = replicate (Suc n) [] using True by simp
    then show ?thesis by (rule exI)
  next
    case False
    hence set (pcnf-variables (Empty, cl ≠ cls)) ≠ {} using empty-prefix-cons-matrix-variables by simp
    hence False using Cons by blast
    then show ?thesis by simp
  qed
  qed

lemma empty-clause-or-matrix-if-no-variables:
  pcnf-variables (Empty, matrix) = [] ⇒ [] ∈ set matrix ∨ matrix = []
  using matrix-shape-if-no-variables by fastforce

D.4.2 Satisfiability Equivalences for First Variable in Prefix

lemma clause-semantics-inv-remove-false:
  clause-semantics (I (z := True)) cl = clause-semantics (I (z := True)) (remove-lit-neg (P z) cl)
by (induction cl) auto

lemma clause-semantics-inv-remove-true:
clause-semantics (I(z := False)) cl = clause-semantics (I(z := False)) (remove-lit-neg (N z) cl)
by (induction cl) auto

lemma matrix-semantics-inv-remove-true:
matrix-semantics (I(z := True)) (matrix-assign (P z) matrix) = matrix-semantics (I(z := True)) matrix
proof (induction matrix)
case Nil
then show ?case by simp
next
case (Cons cl cls)
then show ?case
proof (cases P z ∈ set cl)
case True
hence 0: clause-semantics (I(z := True)) cl by (induction cl) auto
have matrix-semantics (I(z := True)) (matrix-assign (P z) (cl # cls))
= matrix-semantics (I(z := True)) (matrix-assign (P z) cls)
  using 0 clause-semantics-inv-remove-false by simp
moreover have matrix-semantics (I(z := True)) (cl # cls)
= matrix-semantics (I(z := True)) cls
  using 0 by simp
ultimately show ?thesis using Cons by blast
next
case False
hence matrix-assign (P z) (cl # cls) = remove-lit-neg (P z) cl # matrix-assign (P z) cls
  by (induction cl) auto
hence matrix-semantics (I(z := True)) (matrix-assign (P z) (cl # cls))
  ← clause-semantics (I(z := True)) (remove-lit-neg (P z) cl)
  ∧ matrix-semantics (I(z := True)) (matrix-assign (P z) cls) by simp
moreover have matrix-semantics (I(z := True)) (cl # cls)
  ← clause-semantics (I(z := True)) cl
  ∧ matrix-semantics (I(z := True)) cls by simp
ultimately show ?thesis using Cons clause-semantics-inv-remove-false by blast
qed
qed

lemma matrix-semantics-inv-remove-true':
assumes y ≠ z
shows matrix-semantics (I(z := True, y := b)) (matrix-assign (P z) matrix)
  = matrix-semantics (I(z := True, y := b)) matrix
using assms matrix-semantics-inv-remove-true fun-upd-twist by metis

lemma matrix-semantics-inv-remove-false:
matrix-semantics (I(z := False)) (matrix-assign (N z) matrix)
= matrix-semantics (I(z := False)) matrix
proof (induction matrix)
case Nil
then show ?case by simp
next
case (Cons cl cls)
then show ?case
proof (cases N z ∈ set cl)
  case True
  hence 0: clause-semantics (I(z := False)) cl by (induction cl) auto
  have matrix-semantics (I(z := False)) (matrix-assign (N z) (cl # cls))
    = matrix-semantics (I(z := False)) (matrix-assign (N z) cls)
    using 0 clause-semantics-inv-remove-true by simp
  moreover have matrix-semantics (I(z := False)) (cl # cls)
    = matrix-semantics (I(z := False)) cls
    using 0 by simp
  ultimately show ?thesis using Cons by blast
next
  case False
  hence matrix-assign (N z) (cl # cls) = remove-lit-neg (N z) cl # matrix-assign
(N z) cls
    by (induction cl) auto
  hence matrix-assign (I(z := False)) (matrix-assign (N z) (cl # cls))
    = matrix-assign (I(z := False)) (remove-lit-neg (N z) cl)
    ∧ matrix-semantics (I(z := False)) (matrix-assign (N z) cls) by simp
  moreover have matrix-semantics (I(z := False)) (cl # cls)
    = matrix-semantics (I(z := False)) cl
    ∧ matrix-semantics (I(z := False)) cls by simp
  ultimately show ?thesis using Cons clause-semantics-inv-remove-true by blast
qed

lemma matrix-semantics-inv-remove-true':
  assumes y ≠ z
  shows matrix-semantics (I(z := False, y := b)) (matrix-assign (N z) matrix)
    = matrix-semantics (I(z := False, y := b)) matrix
  using assms matrix-semantics-inv-remove-true fun-upd-twist by metis

lemma matrix-semantics-diss-iff-true-assign:
  (∃ b. matrix-semantics (I(z := b)) matrix)
    ↔ matrix-semantics (I(z := True)) (matrix-assign (P z) matrix)
    ∨ matrix-semantics (I(z := False)) (matrix-assign (N z) matrix)
  using matrix-semantics-inv-remove-true matrix-semantics-inv-remove-false by (metis (full-types))

lemma matrix-semantics-conj-iff-true-assign:
  (∀ b. matrix-semantics (I(z := b)) matrix)
    ↔ matrix-semantics (I(z := True)) (matrix-assign (P z) matrix)
    ∧ matrix-semantics (I(z := False)) (matrix-assign (N z) matrix)
  using matrix-semantics-inv-remove-true matrix-semantics-inv-remove-false by (metis (full-types))

lemma pcnf-assign-free-eq-matrix-assign':
  assumes lit-var lit ≠ set (prefix-variables-aux (convert (prefix, matrix))
  shows pcnf-assign lit (prefix, matrix) = (prefix, matrix-assign lit matrix)
  using assms
  by (induction (prefix, matrix) arbitrary; prefix rule: convert.induct) auto

lemma pcnf-assign-free-eq-matrix-assign:
assumes \( \text{lit-var \ lit} \notin \text{set (pcnf-prefix-variables (prefix, matrix))} \)
shows \( \text{pcnf-assign lit (prefix, matrix)} = (\text{prefix, matrix-assign lit matrix}) \)
using \( \text{assms pcnf-assign-free-eq-matrix-assgn'} \) by simp

lemma neq-first-if-notin-all-prefix:
\( \text{z} \notin \text{set (pcnf-prefix-variables (UniversalFirst (y, ys) qs, matrix))} \implies z \neq y \)
by (induction (UniversalFirst (y, ys) qs, matrix) rule: convert.induct) auto

lemma neq-first-if-notin-ex-prefix:
\( \text{z} \notin \text{set (pcnf-prefix-variables (ExistentialFirst (x, xs) qs, matrix))} \implies z \neq x \)
by (induction (ExistentialFirst (x, xs) qs, matrix) rule: convert.induct) auto

lemma notin-pop-prefix-if-notin-prefix:
assumes \( \text{z} \notin \text{set (pcnf-prefix-variables (prefix, matrix))} \)
shows \( \text{z} \notin \text{set (pcnf-prefix-variables (prefix-pop prefix, matrix))} \)
using \( \text{assms} \)
proof (induction prefix)
case (UniversalFirst q qs)
then show \( ?case \)
proof (induction q)
case (Pair y ys)
  then show \( ?case \)
    by (induction (UniversalFirst (y, ys) qs, matrix) rule: convert.induct) auto
qed
next
case (ExistentialFirst q qs)
then show \( ?case \)
proof (induction q)
case (Pair x xs)
  then show \( ?case \)
    by (induction (ExistentialFirst (x, xs) qs, matrix) rule: convert.induct) auto
qed
next
case Empty
then show \( ?case \) by simp
qed

lemma pcnf-semantics-inv-matrix-assign-true:
assumes \( \text{z} \notin \text{set (pcnf-prefix-variables (prefix, matrix))} \)
shows \( \text{pcnf-semantics (I(z := True)) (prefix, matrix-assign (P z) matrix)} = \text{pcnf-semantics (I(z := True)) (prefix, matrix)} \)
using \( \text{assms} \)
proof (induction I (prefix, matrix) arbitrary: I prefix matrix rule: pcnf-semantics.induct)
case (1 I matrix)
then show \( ?case \) using matrix-semantics-inv-remove-true by simp
next
case (2 I y matrix)
then show \( ?case \) using matrix-semantics-inv-remove-true by simp
next
case (3 I x matrix)
then show \( ?case \) using matrix-semantics-inv-remove-true by simp
next
case (4 I y q qs matrix)
hence \( \text{neq: z} \neq y \) using neq-first-if-notin-all-prefix by blast
have prefix-pop (UniversalFirst (y, []) (q # qs)) = ExistentialFirst q qs
by (induction q) auto

hence z /\ set (pcnf-prefix-variables (ExistentialFirst q qs, matrix))
using 4(3) notin-pop-prefix-if-notin-prefix by metis

hence pcnf-semantics (I(z := True)) (ExistentialFirst q qs, matrix) =
pcnf-semantics (I(z := True)) (ExistentialFirst q qs, matrix-assign (P z) matrix)
for I using 4 by blast
then show ?case using neq by (simp add: fun-upd-twist)
next

case (5 I x q qs matrix)
hence neq: z \not= x using neq-first-if-notin-ex-prefix by blast
have prefix-pop (ExistentialFirst (x, []) (q # qs)) = UniversalFirst q qs
by (induction q) auto
hence z /\ set (pcnf-prefix-variables (UniversalFirst q qs, matrix))
using 5(3) notin-prefix-variables-if-notin-prefix by metis
hence pcnf-semantics (I(z := True)) (UniversalFirst q qs, matrix) =
pcnf-semantics (I(z := True)) (UniversalFirst q qs, matrix-assign (P z) matrix)
for I using 5 by blast
then show ?case using neq by (simp add: fun-upd-twist)
next

case (6 I y yy qs matrix)
hence neq: z \not= y using neq-first-if-notin-ex-prefix by blast
have z /\ set (pcnf-prefix-variables (UniversalFirst (yy, qs), matrix))
using 6(3) notin-pop-prefix-if-notin-prefix by fastforce
hence pcnf-semantics (I(z := True)) (UniversalFirst (yy, qs), matrix) =
pcnf-semantics (I(z := True)) (UniversalFirst (yy, qs), matrix-assign (P z) matrix)
for I using 6 by blast
then show ?case using neq by (simp add: fun-upd-twist)
next

case (7 I x xx xs qs matrix)
hence neq: z \not= x using neq-first-if-notin-ex-prefix by blast
have z /\ set (pcnf-prefix-variables (ExistentialFirst (xx, xs), matrix))
using 7(3) notin-prefix-variables-if-notin-prefix by fastforce
hence pcnf-semantics (I(z := True)) (ExistentialFirst (xx, xs), matrix) =
pcnf-semantics (I(z := True)) (ExistentialFirst (xx, xs), matrix-assign (P z) matrix)
for I using 7 by blast
then show ?case using neq by (simp add: fun-upd-twist)
qed

lemma pcnf-semantics-inv-matrix-assign-false:
  assumes z /\ set (pcnf-prefix-variables (prefix, matrix))
  shows pcnf-semantics (I(z := False)) (prefix, matrix-assign (N z) matrix) =
      pcnf-semantics (I(z := False)) (prefix, matrix)
  using assms
proof (induction I (prefix, matrix) arbitrary: I prefix matrix rule: pcnf-semantics.induct)
case (1 I matrix)
then show ?case using matrix-semantics-inv-remove-false by simp
next
case (2 I y matrix)
then show ?case using matrix-semantics-inv-remove-false by simp
next
case (3 I x matrix)
then show \( ?\text{case using matrix-semantics-Inv-rem-False' by simp} \)
next
  case (4 I y q qs matrix)
  hence neq: \( z \neq y \) using neq-first-if-notin-all-prefix by blast
  have prefix-pop (UniversalFirst (y, \([]\)) (q \# q\ s)) = UniversalFirst q qs
    by (induction q) auto
  hence \( z \notin \) set (pcnf-prefix-variables (UniversalFirst q qs, matrix))
    using 4(3) notin-pop-prefix-if-notin-prefix by metis
  hence pcnf-semantics (I(z := False)) (UniversalFirst q qs, matrix) =
    pcnf-semantics (I(z := False)) (UniversalFirst q qs, matrix-assign (N z) matrix)
  for I using 4 by blast
then show \( \text{?case using neq by (simp add: fun-upd-twist)} \)
next
  case (5 I x q qs matrix)
  hence neq: \( z \neq x \) using neq-first-if-notin-ex-prefix by blast
  have prefix-pop (ExistentialFirst (x, \([]\)) (q \# q\ s)) = UniversalFirst q qs
    by (induction q) auto
  hence \( z \notin \) set (pcnf-prefix-variables (UniversalFirst q qs, matrix))
    using 5(3) notin-pop-prefix-if-notin-prefix by metis
  hence pcnf-semantics (I(z := False)) (UniversalFirst q qs, matrix) =
    pcnf-semantics (I(z := False)) (UniversalFirst q qs, matrix-assign (N z) matrix)
  for I using 5 by blast
then show \( \text{?case using neq by (simp add: fun-upd-twist)} \)
next
  case (6 I y q qs matrix)
  hence neq: \( z \neq y \) using neq-first-if-notin-all-prefix by blast
  have z \notin set (pcnf-prefix-variables (UniversalFirst (y, y) q\ s, matrix))
    using 6(3) notin-pop-prefix-if-notin-prefix by fastforce
  hence pcnf-semantics (I(z := False)) (UniversalFirst (y, y) q\ s, matrix) =
    pcnf-semantics (I(z := False)) (UniversalFirst (y, y) q\ s, matrix-assign (N z) matrix)
  for I using 6 by blast
then show \( \text{?case using neq by (simp add: fun-upd-twist)} \)
next
  case (7 I x x x s q\ s matrix)
  hence neq: \( z \neq x \) using neq-first-if-notin-ex-prefix by blast
  have z \notin set (pcnf-prefix-variables (ExistentialFirst (x, x) q\ s, matrix))
    using 7(3) notin-pop-prefix-if-notin-prefix by fastforce
  hence pcnf-semantics (I(z := False)) (ExistentialFirst (x, x) q\ s, matrix) =
    pcnf-semantics (I(z := False)) (ExistentialFirst (x, x) q\ s, matrix-assign (N z) matrix)
  for I using 7 by blast
then show \( \text{?case using neq by (simp add: fun-upd-twist)} \)
qed

lemma pcnf-semantics-disj-iff-matrix-assign-disj:
  assumes z \notin set (pcnf-prefix-variables (prefix, matrix))
  shows pcnf-semantics (I(z := True)) (prefix, matrix)
      \lor pcnf-semantics (I(z := False)) (prefix, matrix)
      \iff pcnf-semantics (I(z := True)) (prefix, matrix-assign (P z) matrix)
      \lor pcnf-semantics (I(z := False)) (prefix, matrix-assign (N z) matrix)
    using assms
proof (induction I (prefix, matrix-assign (P z) matrix)
lemma pcnf-semantics-conj-iff-matrix-assign-conj:
  assumes z :\notin set (pcnf-prefix-variables (prefix, matrix))
  shows pcnf-semantics (I(z := True)) (prefix, matrix)
    \land pcnf-semantics (I(z := False)) (prefix, matrix)
  \iff
    pcnf-semantics (I(z := True)) (prefix, matrix-assign (P z) matrix)
    \land pcnf-semantics (I(z := False)) (prefix, matrix-assign (N z) matrix)
using assms
proof (induction I (prefix, matrix-assign (P z) matrix) arbitrary: I prefix matrix rule: pcnf-semantics.induct)
case (I I)
then show ?case using all-bool-eq matrix-semantics-conj-iff-true-assign by simp
next
case (2 I y)
hence neq: y ≠ z by simp
show ?case using matrix-semantics-inv-remove-true neq by simp
next
case (3 I z)
hence neq: x ≠ z by simp
show ?case using matrix-semantics-inv-remove-false neq by simp
next
case (4 I y q qs)
hence neq: y ≠ z using neq-first-if-notin-all-prefix by blast
have prefix-pop (UniversalFirst (y, []) (q # qs)) = ExistentialFirst q qs by (induction q) auto
hence nin: z /∈ set (pcnf-prefix-variables (ExistentialFirst q qs, matrix))
using 4(3) notin-pop-prefix-if-notin-prefix by metis
next
case (5 I x q qs)
hence neq: x ≠ z using neq-first-if-notin-ex-prefix by blast
have prefix-pop (ExistentialFirst (x, []) (q # qs)) = UniversalFirst q qs by (induction q) auto
hence nin: z /∈ set (pcnf-prefix-variables (UniversalFirst q qs, matrix))
using 5(3) notin-pop-prefix-if-notin-prefix by metis
next
case (6 I y yy ys qs)
hence neq: y ≠ z using neq-first-if-notin-all-prefix by blast
have nin: z /∈ set (pcnf-prefix-variables (UniversalFirst (yy, ys) qs, matrix))
using 6(3) notin-pop-prefix-if-notin-prefix by fastforce
next
case (7 I x xx xs qs)
hence neq: x ≠ z using neq-first-if-notin-ex-prefix by blast
have nin: z /∈ set (pcnf-prefix-variables (ExistentialFirst (xx, xs) qs, matrix))
using 7(3) notin-pop-prefix-if-notin-prefix by fastforce
qed

lemma semantics-eq-if-free-vars-eq:
assumes ∀x ∈ set (free-variables qbf). I(x) = J(x)
shows qbf-semantics I qbf = qbf-semantics J qbf using assms
proof (induction I qbf rule: qbf-semantics.induct)
case (I I)
then show \( \text{?case by \ simp} \)
next
case (2 l qbf)
then show \( \text{?case by \ simp} \)
next
case (3 l qbf-list)
then show \( \text{?case by \ (induction \ qbf-list) auto} \)
next
case (4 l qbf-list)
then show \( \text{?case by \ (induction \ qbf-list) auto} \)
next
case (5 l x qbf)
hence \( \text{qbf-semantics I (substitute-var x b qbf)} = \text{qbf-semantics J (substitute-var x b qbf)} \)
for \( b \) using set-free-vars-subst-ex-eq by \( \text{metis (full-types)} \)
then show \( \text{?case by \ simp} \)
next
case (6 l x qbf)
hence \( \text{qbf-semantics I (substitute-var x b qbf)} = \text{qbf-semantics J (substitute-var x b qbf)} \)
for \( b \) using set-free-vars-subst-all-eq by \( \text{metis (full-types)} \)
then show \( \text{?case by \ simp} \)
qed

lemma pcnf-semantics-eq-if-free-vars-eq:
assumes \( \forall x \in \text{set (pcnf-free-variables pcnf)}, I(x) = J(x) \)
shows \( \text{pcnf-semantics I pcnf = pcnf-semantics J pcnf} \)
using \( \text{assms semantics-eq-if-free-vars-eq qbf-semantics-eq-pcnf-semantics} \) by \( \text{simp} \)

lemma x-notin-assign-P-x:
\( x \notin \text{set (pcnf-variables (pcnf-assign (P x) pcnf))} \)
using \( \text{pcnf-assign-vars-subseteq-vars-minus-lit by \ fastforce} \)

lemma x-notin-assign-N-x:
\( x \notin \text{set (pcnf-variables (pcnf-assign (N x) pcnf))} \)
using \( \text{pcnf-assign-vars-subseteq-vars-minus-lit by \ fastforce} \)

lemma interp-value-ignored-for-pcnf-P-assign:
\( \text{pcnf-semantics (I(x := b)) (pcnf-assign (P x) pcnf)} = \text{pcnf-semantics I (pcnf-assign (P x) pcnf)} \)
using \( \text{pcnf-semantics-eq-if-free-vars-eq x-notin-assign-P-x pcnf-free-eq-vars-minus-prefix by \ simp} \)

lemma interp-value-ignored-for-pcnf-N-assign:
\( \text{pcnf-semantics (I(x := b)) (pcnf-assign (N x) pcnf)} = \text{pcnf-semantics I (pcnf-assign (N x) pcnf)} \)
using \( \text{pcnf-semantics-eq-if-free-vars-eq x-notin-assign-N-x pcnf-free-eq-vars-minus-prefix by \ simp} \)

lemma sat-ex-first-iff-one-assign-sat:
assumes \( x \notin \text{set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix))} \)
shows \( \text{satisfiable (convert (ExistentialFirst (x, xs) qs, matrix))} \)
\( \iff \text{satisfiable (convert (pcnf-assign (P x) (ExistentialFirst (x, xs) qs, matrix))} \)

147
\[
\forall \text{satisfiable } (\text{convert } (\text{pcnf-assign } (N \ x) \ (\text{ExistentialFirst} \ (x, \ xs) \ qs, \ \text{matrix})))
\]

**proof**

let \(?pre = \text{ExistentialFirst} \ (x, \ xs) \ qs\)

have \(\text{satisfiable } (\text{convert } (?pre, \ \text{matrix}))\)

= \(\exists I. \ \text{pcnf-semantics} \ I \ (?pre, \ \text{matrix})\)

using \(\text{satisfiable-def} \ \text{qbf-semantics-eq-pcnf-semantics by simp}\)

also have ...

= \(\exists I. \ \text{pcnf-semantics} \ I (x := \text{True}) \ (\text{prefix-pop} \ ?pre, \ \text{matrix}) \lor \ 
\text{pcnf-semantics} \ I (x := \text{False}) \ (\text{prefix-pop} \ ?pre, \ \text{matrix})\)

by \((\text{induction} ?pre \ \text{rule: prefix-pop.induct}) \ \text{auto}\)

also have ...

= \(\exists I. \ \text{pcnf-semantics} \ I (x := \text{True}) \ (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (P \ x) \ \text{matrix}) \lor 
\text{pcnf-semantics} \ I (x := \text{False}) \ (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (N \ x) \ \text{matrix})\)

using \(\text{pcnf-semantics-disj-iff-matrix-assign-disj-assms by blast}\)

also have ...

\(\exists I. \ \text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (P \ x) \ \text{matrix}) \lor 
\text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ N \ x) \ \text{matrix}\)

using \(\text{pcnf-assign-free-eq-matrix-assign[of P x] pcnf-assign-free-eq-matrix-assign[of N x]} \ \text{assms by auto}\)

also have ...

\(\exists I. \ \text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (P \ x) \ \text{matrix}) \lor 
\text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (N \ x) \ \text{matrix})\)

using \(\text{interp-value-ignored-for-pcnf-N-assign interp-value-ignored-for-pcnf-P-assign}\)

by \(\text{blast}\)

also have ...

\(\exists I. \ \text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (P \ x) \ \text{matrix}) \lor 
\text{pcnf-semantics} \ I (\text{prefix-pop} \ ?pre, \ \text{matrix-assign} \ (N \ x) \ \text{matrix})\)

using \(\text{satisfiable-def} \ \text{qbf-semantics-eq-pcnf-semantics by simp}\)

finally show \(\text{thesis} .\)

qed

**theorem** \(\text{sat-ex-first-iff-assign-disj-sat:}\)

assumes \(x \notin \text{set} \ (\text{pcnf-prefix-variables} \ (\text{prefix-pop} \ (\text{ExistentialFirst} \ (x, \ xs) \ qs), \ \text{matrix}))\)

shows \(\text{satisfiable } (\text{convert } (\text{ExistentialFirst} \ (x, \ xs) \ qs, \ \text{matrix}))\)

\(\iff \text{satisfiable } (\text{Disj})\)

\(\iff \text{satisfiable } (\text{Con})\)

\(\text{convert } (\text{pcnf-assign} \ (P \ x) \ (\text{ExistentialFirst} \ (x, \ xs) \ qs, \ \text{matrix})),\)

\(\text{convert } (\text{pcnf-assign} \ (N \ x) \ (\text{ExistentialFirst} \ (x, \ xs) \ qs, \ \text{matrix})))\)

using \(\text{assms sat-ex-first-iff-one-assign-sat satisfiable-def}\)

\(\text{qbf-semantics-eq-pcnf-semantics by auto}\)

**theorem** \(\text{sat-all-first-iff-assign-conj-sat:}\)

assumes \(y \notin \text{set} \ (\text{pcnf-prefix-variables} \ (\text{prefix-pop} \ (\text{UniversalFirst} \ (y, \ ys) \ qs), \ \text{matrix}))\)

shows \(\text{satisfiable } (\text{convert } (\text{UniversalFirst} \ (y, \ ys) \ qs, \ \text{matrix}))\)

\(\iff \text{satisfiable } (\text{Con})\)

\(\iff \text{satisfiable } (\text{Con})\)

\(\text{convert } (\text{pcnf-assign} \ P \ y \ (\text{UniversalFirst} \ (y, \ ys) \ qs, \ \text{matrix})),\)

\(\text{convert } (\text{pcnf-assign} \ N \ y \ (\text{UniversalFirst} \ (y, \ ys) \ qs, \ \text{matrix})))\)

proof

let \(?pre = \text{UniversalFirst} \ (y, \ ys) \ qs\)

have \(\text{satisfiable } (\text{convert } (?pre, \ \text{matrix}))\)

= \(\exists I. \ \text{pcnf-semantics} \ I \ (?pre, \ \text{matrix})\)
using \text{satisfiable-def} qbf-semantics-eq-pcnf-semantics by simp
also have ... =
(\exists I. \text{pcnf-semantics} (I(y := \text{True})) (\text{prefix-pop ?pre, matrix}) \land
  \text{pcnf-semantics} (I(y := \text{False})) (\text{prefix-pop ?pre, matrix}))
by (induction ?pre rule: prefix-pop.induct) auto
also have ... =
(\exists I. \text{pcnf-semantics} (I(y := \text{True})) (\text{prefix-pop ?pre, matrix-assign (P y) matrix})
  \land
  \text{pcnf-semantics} (I(y := \text{False})) (\text{prefix-pop ?pre, matrix-assign (N y) matrix}))
using \text{pcnf-semantics-conj-iff-matrix-assgn assms} by blast
also have ... =
(\exists I. \text{qbf-semantics} I (\text{convert (pcnf-assign (P y) (?pre, matrix))})
  \land
  \text{qbf-semantics} I (\text{convert (pcnf-assign (N y) (?pre, matrix))}))
using \text{qbf-semantics-eq-pcnf-semantics} by blast
also have ... =
satisfiable (\text{Conj}
  [\text{convert (pcnf-assign (P y) (?pre, matrix))},
   \text{convert (pcnf-assign (N y) (?pre, matrix))}])
unfolding \text{satisfiable-def} by simp
finally show \text{thesis}.
qed

D.5 Cleansed PCNF Formulas
D.5.1 Predicate for Cleansed Formulas
fun \text{cleansed-p} :: \text{pcnf} \Rightarrow \text{bool} where
\text{cleansed-p pcnf} = \text{distinct (prefix-variables-aux (convert pcnf))}

lemma \text{prefix-pop-cleansed-if-cleansed}:
\text{cleansed-p (prefix, matrix)} \Longrightarrow \text{cleansed-p (prefix-pop prefix, matrix)}
by (induction prefix rule: prefix-pop.induct) auto

lemma \text{prefix-variables-aux-matrix-inv}:
\text{prefix-variables-aux (convert (prefix, matrix1)) = prefix-variables-aux (convert (prefix, matrix2))}
by (induction (prefix, matrix1) arbitrary: prefix rule: convert.induct) auto

lemma \text{eq-prefix-cleansed-p-add-all-inv}:
\text{cleansed-p (add-universal-to-front y (prefix, matrix1)) =}
\text{cleansed-p (add-universal-to-front y (prefix, matrix2))}
proof (induction y (prefix, matrix1) arbitrary: prefix rule: add-universal-to-front.induct)
case (1 x)
then show \( ?\text{case by simp} \)
next
case \((2 \ x \ y \ ys \ qs)\)
\begin{align*}
\text{have } \text{prefix-variables-aux } & (\text{convert } (\UniversalFirst \ (y, \ ys) \ qs, \ matrix1)) \\
& = \text{prefix-variables-aux } (\text{convert } (\UniversalFirst \ (y, \ ys) \ qs, \ matrix2)) \\
\text{using } \text{prefix-variables-aux-matrix-inv } & \text{ by simp}
\end{align*}
then show \( ?\text{case by simp} \)
next
case \((3 \ x \ y \ ys \ qs)\)
\begin{align*}
\text{have } \text{prefix-variables-aux } & (\text{convert } (\ExistentialFirst \ (y, \ ys) \ qs, \ matrix1)) \\
& = \text{prefix-variables-aux } (\text{convert } (\ExistentialFirst \ (y, \ ys) \ qs, \ matrix2)) \\
\text{using } \text{prefix-variables-aux-matrix-inv } & \text{ by simp}
\end{align*}
then show \( ?\text{case by simp} \)
qed

\begin{lemma}
\text{eq-prefix-cleansed-p-add-ex-inv:}
\begin{align*}
\text{cleansed-p } & (\text{add-existential-to-front } x \ (\text{prefix}, \ matrix1)) \\
= & \text{cleansed-p } (\text{add-existential-to-front } x \ (\text{prefix}, \ matrix2))
\end{align*}
\end{lemma}
\begin{proof}
(induction \( x \ (\text{prefix}, \ matrix1) \) arbitrary: prefix rule: \text{add-universal-to-front.induct})
\begin{itemize}
\item \text{case } \(1 \ x\)
\begin{align*}
\text{then show } & ?\text{case by simp}
\end{align*}
\item \text{next}
\begin{align*}
\text{case } & (2 \ x \ y \ ys \ qs)
\text{have } \text{prefix-variables-aux } & (\text{convert } (\UniversalFirst \ (y, \ ys) \ qs, \ matrix1)) \\
& = \text{prefix-variables-aux } (\text{convert } (\UniversalFirst \ (y, \ ys) \ qs, \ matrix2)) \\
\text{using } \text{prefix-variables-aux-matrix-inv } & \text{ by simp}
\end{align*}
then show \( ?\text{case by simp} \)
\item \text{next}
\begin{align*}
\text{case } & (3 \ x \ y \ ys \ qs)
\text{have } \text{prefix-variables-aux } & (\text{convert } (\ExistentialFirst \ (y, \ ys) \ qs, \ matrix1)) \\
& = \text{prefix-variables-aux } (\text{convert } (\ExistentialFirst \ (y, \ ys) \ qs, \ matrix2)) \\
\text{using } \text{prefix-variables-aux-matrix-inv } & \text{ by simp}
\end{align*}
then show \( ?\text{case by simp} \)
\item \text{next}
\begin{align*}
\text{case } & (4 \ x \ q \ qs)
\text{have } (\UniversalFirst \ (x, \ []) \ (q \ # \ qs), \ matrix) \\
& = \text{add-universal-to-front } x \ (\ExistentialFirst \ q \ qs, \ matrix) \\
\text{for } \text{matrix by } & (\text{induction } q) \ auto
\end{align*}
then show \( ?\text{case using eq-prefix-cleansed-p-add-all-inv } \text{ by simp} \)
\item \text{next}
\begin{align*}
\text{case } & (5 \ x \ q \ qs)
\end{align*}
\end{itemize}
\end{proof}

\begin{lemma}
\text{cleansed-p-matrix-inv:}
\begin{align*}
\text{cleansed-p } & (\text{prefix}, \ matrix1) \\
= & \text{cleansed-p } (\text{prefix}, \ matrix2)
\end{align*}
\end{lemma}
\begin{proof}
(induction \( \text{prefix}, \ matrix1 \) arbitrary: prefix rule: \text{convert.induct})
\begin{itemize}
\item \text{case } \(1 \)
\begin{align*}
\text{then show } & ?\text{case by simp}
\end{align*}
\item \text{next}
\begin{align*}
\text{case } & (2 \ x)
\text{then show } ?\text{case by simp}
\end{align*}
\item \text{next}
\begin{align*}
\text{case } & (3 \ x)
\text{then show } ?\text{case by simp}
\end{align*}
\item \text{next}
\begin{align*}
\text{case } & (4 \ x \ q \ qs)
\text{have } (\UniversalFirst \ (x, \ []) \ (q \ # \ qs), \ matrix) \\
& = \text{add-universal-to-front } x \ (\ExistentialFirst \ q \ qs, \ matrix) \\
\text{for } \text{matrix by } & (\text{induction } q) \ auto
\end{align*}
then show \( ?\text{case using eq-prefix-cleansed-p-add-all-inv } \text{ by simp} \)
\item \text{next}
\begin{align*}
\text{case } & (5 \ x \ q \ qs)
\end{align*}
\end{itemize}
\end{proof}
D.5.2 The Cleansed Predicate is Invariant under PCNF Assignment

lemma cleansed-prefix-first-ex-unique:
  assumes cleansed-p (ExistentialFirst (x, []) (q ≠ qs), matrix)
  shows x ∉ set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) q) qs)
  using assms by (induction ExistentialFirst (x, x) q rule: prefix-pop.induct) auto

lemma eq-prefix-cleansed-p-add-ex-inv:
  assumes eq-prefix-cleansed-p-add-all-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-ex-inv rule: prefix-pop.induct auto

lemma eq-prefix-cleansed-p-add-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-inv rule: prefix-pop.induct auto

lemma eq-prefix-cleansed-p-add-ex-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-ex-inv rule: prefix-pop.induct auto

lemma eq-prefix-cleansed-p-add-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-inv rule: prefix-pop.induct auto

lemma eq-prefix-cleansed-p-add-ex-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-ex-inv rule: prefix-pop.induct auto

lemma eq-prefix-cleansed-p-add-inv: matrix = matrix
  shows P x
  using eq-prefix-cleansed-p-add-inv rule: prefix-pop.induct auto
using assms by (metis cleansed-prefix-first-all-unique lit-var.simps(1)
  pcnf-assign.simps pcnf-assign-free-eq-matrix-assgn remove-var-prefix.simps(2))

lemma pcnf-assign-n-ex-eq:
assumes cleansed-p (ExistentialFirst (x, xs) qs, matrix)
shows pcnf-assign (N x) (ExistentialFirst (x, xs) qs, matrix)
  = (prefix-pop (ExistentialFirst (x, xs) qs), matrix-assign (N x) matrix)
using assms by (metis cleansed-prefix-first-ex-unique lit-var.simps(2)
  pcnf-assign.simps pcnf-assign-free-eq-matrix-assgn remove-var-prefix.simps(2))

lemma pcnf-assign-n-all-eq:
assumes cleansed-p (UniversalFirst (y, ys) qs, matrix)
shows pcnf-assign (N y) (UniversalFirst (y, ys) qs, matrix)
  = (prefix-pop (UniversalFirst (y, ys) qs), matrix-assign (N y) matrix)
using assms by (metis cleansed-prefix-first-all-unique lit-var.simps(2)
  pcnf-assign.simps pcnf-assign-free-eq-matrix-assgn remove-var-prefix.simps(2))

theorem pcnf-assign-cleansed-inv:
cleansed-p pcnf =⇒ cleansed-p (pcnf-assign lit pcnf)
proof (induction pcnf rule: convert.induct)
case (1 matrix)
then show ?case by simp
next
case (2 x matrix)
then show ?case by simp
next
case (3 x matrix)
then show ?case by simp
next
case (4 x q qs matrix)
let ?z = lit-var lit
show ?case
proof (cases x = ?z)
case True
then show ?thesis using 4 cleansed-p-matrix-inv
  pcnf-assign-n-all-eq[of ?z] pcnf-assign-p-all-eq[of ?z]
  prefix-pop-cleansed-if-cleansed lit-var.elims by metis
next
case False
let ?mat = matrix-assign lit matrix
have cleansed-p (remove-var-prefix ?z (ExistentialFirst q qs), ?mat)
  using 4 by simp
moreover have x ≠ set (pcnf-prefix-variables (remove-var-prefix ?z (ExistentialFirst
  q qs), ?mat))
  by fastforce
ultimately have cleansed-p (add-universal-to-prefix x (remove-var-prefix ?z (ExistentialFirst
  q qs)), ?mat)
  using cleansed-add-new-all-to-front add-all-to-prefix-eq-add-to-front by simp
  then have cleansed-p (remove-var-prefix ?z (UniversalFirst (x, [])) (q # qs)),
    ?mat)
  using False by (induction q) auto
then show ?thesis by simp
qed
next

case \((5 \times x q q s m a t r i x)\)

let \(?z = \text{lit-var} \text{ lit}\)

show \(?case\)

proof (cases \(x = \ ?z\))

\begin{itemize}
  \item case True
  \begin{itemize}
    \item then show \(?thesis\ \text{using} \ 5 \ \text{cleansed-p-matrix-inv}\)
      \begin{itemize}
        \item \text{pcnf-assign-n-ex-eq[of \ ?z]} \ \text{pcnf-assign-p-ex-eq[of \ ?z]}
        \item \text{prefix-pop-cleansed-if-cleansed lit-var.elims by metis}
      \end{itemize}
  \end{itemize}

next

\begin{itemize}
  \item case False
  \begin{itemize}
    \item let \(?mat = \text{matrix-assign} \ \text{lit} \ \text{matrix}\)
      \begin{itemize}
        \item have \text{cleansed-p} (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} q q s s), \ ?mat)
        \begin{itemize}
          \item using \(5 \ \text{by simp}\)
        \end{itemize}
        \begin{itemize}
          \item moreover have \(x \not\in \text{set} (\text{pcnf-prefix-variables} (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} q q s s), \ ?mat))\)
          \begin{itemize}
            \item using \(5 \ \text{False prefix-assign-vars-eq-prefix-vars-minus-lit[of \ ?z]} \ \text{prefix-vars-matrix-inv}\)
            \begin{itemize}
              \item by \text{fastforce}
            \end{itemize}
          \end{itemize}
          \item ultimately have \text{cleansed-p} (\text{add-universal-to-prefix} \ x (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} q q s s), \ ?mat))
          \begin{itemize}
            \item using \text{cleansed-add-new-all-to-front} \text{add-all-to-prefix eq-add-to-front \ simp}
          \end{itemize}
          \begin{itemize}
            \item then have \text{cleansed-p} (\text{remove-var-prefix} \ ?z (\text{ExistentialFirst} (x, []) (q \ # q s s)), \ ?mat)
          \end{itemize}
        \end{itemize}
      \end{itemize}
      \item using \text{False by (induction q) auto}
      \begin{itemize}
        \item then show \(?thesis \ \text{by simp}\)
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

qed

next

\begin{itemize}
  \item case \((6 \times y y s s q q s m a t r i x)\)
  \begin{itemize}
    \item let \(?z = \text{lit-var} \text{ lit}\)
      \begin{itemize}
        \item show \(?case\)
        \begin{itemize}
          \item proof (cases \(x = \ ?z\))
          \begin{itemize}
            \item case True
            \begin{itemize}
              \item then show \(?thesis\ \text{using} \ 6 \ \text{cleansed-p-matrix-inv}\)
                \begin{itemize}
                  \item \text{pcnf-assign-n-all-eq[of \ ?z]} \ \text{pcnf-assign-p-all-eq[of \ ?z]}
                  \item \text{prefix-pop-cleansed-if-cleansed lit-var.elims by metis}
                \end{itemize}
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
  \end{itemize}
\end{itemize}

next

\begin{itemize}
  \item case False
  \begin{itemize}
    \item let \(?mat = \text{matrix-assign} \ \text{lit} \ \text{matrix}\)
      \begin{itemize}
        \item have \text{cleansed-p} (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} (y, y s s) q s s), \ ?mat)
        \begin{itemize}
          \item using \(6 \ \text{by simp}\)
        \end{itemize}
        \begin{itemize}
          \item moreover have \(x \not\in \text{set} (\text{pcnf-prefix-variables} (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} (y, y s s) q s s), \ ?mat))\)
          \begin{itemize}
            \item using \(6(2) \ \text{False prefix-assign-vars-eq-prefix-vars-minus-lit[of \ ?z]} \ \text{prefix-vars-matrix-inv}\)
            \begin{itemize}
              \item by \text{fastforce}
            \end{itemize}
          \end{itemize}
          \item ultimately have \text{cleansed-p} (\text{add-universal-to-prefix} \ x (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} (y, y s s) q s s), \ ?mat))
          \begin{itemize}
            \item using \text{cleansed-add-new-all-to-front} \text{add-all-to-prefix eq-add-to-front \ simp}
          \end{itemize}
          \begin{itemize}
            \item then have \text{cleansed-p} (\text{remove-var-prefix} \ ?z (\text{UniversalFirst} (x, (y \ # y s s)) q s s), \ ?mat)
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

qed

next

\begin{itemize}
  \item case \((7 \times y y s s q q s m a t r i x)\)
\end{itemize}
let \( \text{?z} = \text{lit-var lit} \)
show \( \text{?case} \)
proof (cases \( x = \text{?z} \))
case True
then show \( \text{?thesis} \) using
\begin{align*}
\text{7 cleansed-p-matrix-inv} \\
\text{pcnf-assign-n-ex-eq[of \text{?z}] pcnf-assign-p-ex-eq[of \text{?z}]}
\end{align*}
\text{prefix-pop-cleansed-if-cleansed lit-var.elims by metis}
next
case False
let \( \text{?mat} = \text{matrix-assign lit matrix} \)
have \( \text{cleansed-p (remove-var-prefix \text{?z} (\text{ExistentialFirst (y, ys) qs}), \text{?mat})} \)
using \( 7 \text{ by simp} \)
moreover have \( x \notin \text{set (pcnf-prefix-variables (remove-var-prefix \text{?z} (\text{ExistentialFirst (y, ys) qs}), \text{?mat}))} \)
using \( 7(2) \text{ False prefix-assign-vars-eq-prefix-vars-minus-lit[of \text{?z}] prefix-vars-matrix-inv} \)
by fastforce
ultimately have \( \text{cleansed-p (add-existential-to-prefix x (remove-var-prefix \text{?z} (\text{ExistentialFirst (x, (y \# ys)) qs}), \text{?mat})} \)
using \( \text{False by simp} \)
then show \( \text{?thesis by simp} \)
qed

D.5.3 Cleansing PCNF Formulas

function \( \text{pcnf-cleanse :: pcnf} \Rightarrow \text{pcnf} \text{ where} \)
\begin{align*}
\text{pcnf-cleanse (Empty, matrix)} &= (\text{Empty, matrix}) | \\
\text{pcnf-cleanse (UniversalFirst (y, ys) qs, matrix)} &= \\
(\text{if } y \in \text{set (pcnf-prefix-variables (prefix-pop (UniversalFirst (y, ys) qs), matrix))}) & \text{ then pcnf-cleanse (prefix-pop (UniversalFirst (y, ys) qs), matrix)} \\
& \text{ else add-universal-to-front y} \\
& \text{(pcnf-cleanse (prefix-pop (UniversalFirst (y, ys) qs), matrix)))} | \\
\text{pcnf-cleanse (ExistentialFirst (x, xs) qs, matrix)} &= \\
(\text{if } x \in \text{set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix))}) & \text{ then pcnf-cleanse (prefix-pop (ExistentialFirst (x, xs) qs), matrix)} \\
& \text{ else add-existential-to-front x} \\
& \text{(pcnf-cleanse (prefix-pop (ExistentialFirst (x, xs) qs), matrix)))} \\
\end{align*}
by \( \text{pat-completeness auto} \)
termination
by \( \text{(relation measure (\lambda(pre, mat). prefix-measure pre))} \)
(\text{auto simp add: prefix-pop-decreases-measure simp del: prefix-measure.simps})

D.5.4 Cleansing Yields a Cleansed Formula

lemma \( \text{prefix-pop-all-prefix-vars-set:} \)
set \( (\text{pcnf-prefix-variables (UniversalFirst (y, ys) qs, matrix))} \)
\( = \{y\} \cup \text{set (pcnf-prefix-variables (prefix-pop (UniversalFirst (y, ys) qs), matrix))} \)
by \( \text{(induction (UniversalFirst (y, ys) qs, matrix) rule: convert.induct, induction qs) auto} \)

lemma \( \text{prefix-pop-ex-prefix-vars-set:} \)
set (pcnf-prefix-variables (ExistentialFirst (x, xs) qs, matrix))
= \{x\} \cup set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix))
by (induction (ExistentialFirst (x, xs) qs, matrix) rule: convert.induct, induction qs) auto

lemma cleanse-prefix-vars-inv:
set (pcnf-prefix-variables (prefix, matrix))
= set (pcnf-prefix-variables (pcnf-cleanse (prefix, matrix)))
using add-all-adds-prefix-var prefix-pop-all-prefix-vars-set
add-ex-adds-prefix-var prefix-pop-ex-prefix-vars-set
by (induction (prefix, matrix) arbitrary: prefix rule: pcnf-cleanse.induct) auto

theorem pcnf-cleanse-cleanses:
cleansed-p (pcnf-cleanse pcnf)
using cleanse-prefix-vars-inv cleansed-add-new-all-to-front cleansed-add-new-ex-to-front
by (induction pcnf rule: pcnf-cleanse.induct) auto

lemma prefix-pop-all-vars-inv:
set (pcnf-variables (UniversalFirst (y, ys) qs, matrix))
= set (pcnf-variables (prefix-pop (UniversalFirst (y, ys) qs), matrix))
by (induction (UniversalFirst (y, ys) qs, matrix) rule: convert.induct, induction qs) auto

lemma prefix-pop-ex-vars-inv:
set (pcnf-variables (ExistentialFirst (x, xs) qs, matrix))
= set (pcnf-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix))
by (induction (ExistentialFirst (x, xs) qs, matrix) rule: convert.induct, induction qs) auto

lemma add-all-vars-inv:
set (pcnf-variables (add-universal-to-front y pcnf))
= set (pcnf-variables pcnf)
using convert-add-all by auto

lemma add-ex-vars-inv:
set (pcnf-variables (add-existential-to-front x pcnf))
= set (pcnf-variables pcnf)
using convert-add-ex by auto

lemma cleanse-vars-inv:
set (pcnf-variables (prefix, matrix))
= set (pcnf-variables (pcnf-cleanse (prefix, matrix)))
using add-all-vars-inv prefix-pop-all-vars-inv
add-ex-vars-inv prefix-pop-ex-vars-inv
by (induction (prefix, matrix) arbitrary: prefix rule: pcnf-cleanse.induct) auto

theorem cleanse-free-vars-inv:
set (pcnf-free-variables pcnf)
= set (pcnf-free-variables (pcnf-cleanse pcnf))
using cleanse-prefix-vars-inv cleanse-vars-inv pcnf-free-eq-vars-minus-prefix
by (induction pcnf) simp-all

lemma add-all-vars-inv:
set (pcnf-variables (add-universal-to-front y pcnf))
= set (pcnf-variables pcnf)
using convert-add-all by auto

lemma add-ex-vars-inv:
set (pcnf-variables (add-existential-to-front x pcnf))
= set (pcnf-variables pcnf)
using convert-add-ex by auto

lemma cleanse-vars-inv:
set (pcnf-variables (prefix, matrix))
= set (pcnf-variables (pcnf-cleanse (prefix, matrix)))
using add-all-vars-inv prefix-pop-all-vars-inv
add-ex-vars-inv prefix-pop-ex-vars-inv
by (induction (prefix, matrix) arbitrary: prefix rule: pcnf-cleanse.induct) auto

theorem cleanse-free-vars-inv:
set (pcnf-free-variables pcnf)
= set (pcnf-free-variables (pcnf-cleanse pcnf))
using cleanse-prefix-vars-inv cleanse-vars-inv pcnf-free-eq-vars-minus-prefix
by (induction pcnf) simp-all

lemma add-all-vars-inv:
set (pcnf-variables (add-universal-to-front y pcnf))
= set (pcnf-variables pcnf)
using convert-add-all by auto

lemma add-ex-vars-inv:
set (pcnf-variables (add-existential-to-front x pcnf))
= set (pcnf-variables pcnf)
using convert-add-ex by auto

lemma cleanse-vars-inv:
set (pcnf-variables (prefix, matrix))
= set (pcnf-variables (pcnf-cleanse (prefix, matrix)))
using add-all-vars-inv prefix-pop-all-vars-inv
add-ex-vars-inv prefix-pop-ex-vars-inv
by (induction (prefix, matrix) arbitrary: prefix rule: pcnf-cleanse.induct) auto

theorem cleanse-free-vars-inv:
set (pcnf-free-variables pcnf)
= set (pcnf-free-variables (pcnf-cleanse pcnf))
using cleanse-prefix-vars-inv cleanse-vars-inv pcnf-free-eq-vars-minus-prefix
by (induction pcnf) simp-all
D.5.5 Cleansing Preserves Semantics

Lemma `pop-redundant-ex-prefix-var-semantics-eq`:
- Assumes `x ∈ set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix))`
- Shows `pcnf-semantics I (ExistentialFirst (x, xs) qs, matrix) = pcnf-semantics I (prefix-pop (ExistentialFirst (x, xs) qs), matrix)`

Proof - 
- Let `?pcnf = (ExistentialFirst (x, xs) qs, matrix)`
- Let `?pop = (prefix-pop (ExistentialFirst (x, zs) qs), matrix)`
- Have set `pcnf-prefix-variables ?pcnf = set (pcnf-prefix-variables ?pop)`
- Using `assms pcnf-free-eq-vars-minus-prefix` by simp
- Hence `x ∉ set (pcnf-free-variables ?pop)`
- Using `assms pcnf-free-eq-vars-minus-prefix` by simp
- Hence `0: ∀ z ∈ set (pcnf-free-variables ?pop). (I(x := b)) z = I z`
- For `b` by simp
- Moreover have `pcnf-semantics I ?pcnf`
- `iff pcnf-semantics (I(x := True)) ?pop`
- `∨ pcnf-semantics (I(x := False)) ?pop`
- By (induction ExistentialFirst (x, xs) qs rule: prefix-pop.induct) auto
- Ultimately show `?thesis using pcnf-semantics-eq-if-free-vars-eq` by blast

QED

Lemma `pop-redundant-all-prefix-var-semantics-eq`:
- Assumes `y ∈ set (pcnf-prefix-variables (prefix-pop (UniversalFirst (y, ys) qs), matrix))`
- Shows `pcnf-semantics I (UniversalFirst (y, ys) qs, matrix) = pcnf-semantics I (prefix-pop (UniversalFirst (y, ys) qs), matrix)`

Proof - 
- Let `?pcnf = (UniversalFirst (y, ys) qs, matrix)`
- Let `?pop = (prefix-pop (UniversalFirst (y, ys) qs), matrix)`
- Have set `pcnf-prefix-variables ?pcnf = set (pcnf-prefix-variables ?pop)`
- Using `assms pcnf-free-eq-vars-minus-prefix` by simp
- Hence `y ∉ set (pcnf-free-variables ?pop)`
- Using `assms pcnf-free-eq-vars-minus-prefix` by simp
- Hence `0: ∀ z ∈ set (pcnf-free-variables ?pop). (I(y := b)) z = I z`
- For `b` by simp
- Moreover have `pcnf-semantics I ?pcnf`
- `iff pcnf-semantics (I(y := True)) ?pop`
- `∧ pcnf-semantics (I(y := False)) ?pop`
- By (induction ExistentialFirst (y, ys) qs rule: prefix-pop.induct) auto
- Ultimately show `?thesis using pcnf-semantics-eq-if-free-vars-eq` by blast

QED

Lemma `pcnf-semantics-disj-eq-add-ex`:
- `pcnf-semantics (I(y := True)) pcnf ∨ pcnf-semantics (I(y := False)) pcnf`
- `iff pcnf-semantics I (add-existential-to-front y pcnf)`

QED

Lemma `pcnf-semantics-conj-eq-add-all`:
- `pcnf-semantics (I(y := True)) pcnf ∧ pcnf-semantics (I(y := False)) pcnf`
- `iff pcnf-semantics I (add-universal-to-front y pcnf)`
- Using `convert-add-all qbf-semantics-eq-pcnf-semantics qbf-semantics-substitute-eq-assign`
by simp

theorem pcnf-cleanse-preserved-semantics:
  pcnf-semantics I pcnf = pcnf-semantics I (pcnf-cleanse pcnf)
proof (induction pcnf arbitrary: I rule: pcnf-cleanse.induct)
case (1 matrix)
  then show ?case by simp
next
case (2 y ys qs matrix)
hence 0: pcnf-semantics I (prefix-pop (UniversalFirst (y, ys) qs), matrix) =
  pcnf-semantics I (pcnf-cleanse (prefix-pop (UniversalFirst (y, ys) qs), matrix))
  for I by cases auto
  show ?case
    proof (cases y ∈ set (pcnf-prefix-variables (prefix-pop (UniversalFirst (y, ys) qs), matrix)))
      case True
      then show ?thesis using 0 pop-redundant-all-prefix-var-semantics-eq by simp
    next
case False
  moreover have pcnf-semantics I (UniversalFirst (y, ys) qs, matrix)
    ⌸ pcnf-semantics I (I(y := True)) (prefix-pop (UniversalFirst (y, ys) qs), matrix)
    ∧ pcnf-semantics I (I(y := False)) (prefix-pop (UniversalFirst (y, ys) qs), matrix)
    by (induction UniversalFirst (y, ys) qs rule: prefix-pop.induct) auto
  ultimately show ?thesis using 0 pcnf-semantics-conj-eq-add-all by simp
qed
next
case (3 x xs qs matrix)
hence 0: pcnf-semantics I (prefix-pop (ExistentialFirst (x, xs) qs), matrix) =
  pcnf-semantics I (pcnf-cleanse (prefix-pop (ExistentialFirst (x, xs) qs), matrix))
  for I by cases auto
  show ?case
    proof (cases x ∈ set (pcnf-prefix-variables (prefix-pop (ExistentialFirst (x, xs) qs), matrix)))
      case True
      then show ?thesis using 0 pop-redundant-ex-prefix-var-semantics-eq by simp
    next
case False
  moreover have pcnf-semantics I (ExistentialFirst (x, xs) qs, matrix)
    ⌸ pcnf-semantics I (I(x := True)) (prefix-pop (ExistentialFirst (x, xs) qs), matrix)
    ∨ pcnf-semantics I (I(x := False)) (prefix-pop (ExistentialFirst (x, xs) qs), matrix)
    by (induction ExistentialFirst (x, xs) qs rule: prefix-pop.induct) auto
  ultimately show ?thesis using 0 pcnf-semantics-disj-eq-add-ex by simp
qed
qed

theorem sat-ex-first-iff-assign-disj-sat' :
  assumes cleansed-p (ExistentialFirst (x, xs) qs, matrix)
  shows  satisfiable (convert (ExistentialFirst (x, xs) qs, matrix))
\(\leftrightarrow\) satisfiable \((\text{Disj})\)

\(\left\{\begin{array}{l}
\text{convert (pcnf-assign } (P \, x) \, (\text{ExistentialFirst} \, (x, \, x) \, q, \, \text{matrix})),
\text{convert (pcnf-assign } (N \, x) \, (\text{ExistentialFirst} \, (x, \, x) \, q, \, \text{matrix}))
\end{array}\right.\)

using \(\text{assms cleansed-prefix-first-ex-unique sat-ex-first-iff-assign-disj-sat by auto}\)

**Theorem** sat-all-first-iff-assign-conj-sat:\)

assumes cleansed-p \((\text{UniversalFirst} \, (y, \, y) \, q, \, \text{matrix})\)

shows satisfiable \((\text{convert (UniversalFirst} \, (y, \, y) \, q, \, \text{matrix}))\)

\(\leftrightarrow\) satisfiable \((\text{Con})\)

\(\left\{\begin{array}{l}
\text{convert (pcnf-assign } (P \, y) \, (\text{UniversalFirst} \, (y, \, y) \, q, \, \text{matrix})),
\text{convert (pcnf-assign } (N \, y) \, (\text{UniversalFirst} \, (y, \, y) \, q, \, \text{matrix}))
\end{array}\right.\)

using \(\text{assms cleansed-prefix-first-all-unique sat-all-first-iff-assign-conj-sat by auto}\)

### D.6 Search Solver (Part 2: The Solver)

**Lemma** add-all-inc-prefix-measure:

prefix-measure \((\text{add-universal-to-prefix} \, y \, \text{prefix}) = \text{Suc} \, (\text{prefix-measure} \, \text{prefix})\)

by \((\text{induction } y \, \text{prefix rule: add-universal-to-prefix.induct})\) auto

**Lemma** add-ex-inc-prefix-measure:

prefix-measure \((\text{add-existential-to-prefix} \, x \, \text{prefix}) = \text{Suc} \, (\text{prefix-measure} \, \text{prefix})\)

by \((\text{induction } x \, \text{prefix rule: add-universal-to-prefix.induct})\) auto

**Lemma** remove-var-non-increasing-measure:

prefix-measure \((\text{remove-var-prefix} \, x \, \text{prefix}) \leq \text{prefix-measure} \, \text{prefix}\)

**Proof** \((\text{induction } x \, \text{prefix rule: remove-var-prefix.induct})\)

\(\text{case } (1 \, x)\)

then show \(?\text{case by simp}\)

next

\(\text{case } (2 \, x \, y \, q, \, q)\)

hence \(0: \text{prefix-measure} \, (\text{remove-var-prefix} \, x \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q)))\)

\(\leq \text{prefix-measure} \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q))\)

by \((\text{cases } x = y)\) \((\text{cases prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q) = \text{Empty}, \text{simp-all})+\)

show \(?\text{case}\)

**Proof** \((\text{cases } x = y)\)

\(\text{case True}\)

hence \(\text{prefix-measure} \, (\text{remove-var-prefix} \, x \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q))\)

\(= \text{prefix-measure} \, (\text{remove-var-prefix} \, x \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q)))\)

by simp

also have \(\ldots \leq \text{prefix-measure} \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q))\) using \(0\) by simp

also have \(\ldots \leq \text{prefix-measure} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q)\)

using \(\text{prefix-pop-decreases-measure less-imp-le-nat}\) by blast

finally show \(?\text{thesis }\).

next

\(\text{case False}\)

hence \(\text{prefix-measure} \, (\text{remove-var-prefix} \, x \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q))\)

\(= \text{prefix-measure} \, (\text{add-universal-to-prefix} \, y \, (\text{remove-var-prefix} \, x \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q))))\) by simp

also have \(\ldots \leq \text{Suc} \, (\text{prefix-measure} \, (\text{prefix-pop} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q)))\)

using \(0\) all-inc-prefix-measure by simp

also have \(\ldots \leq \text{prefix-measure} \, (\text{UniversalFirst} \, (y, \, y) \, q, \, q)\)

using Suc-leI \(\text{prefix-pop-decreases-measure}\) by blast

158
finally show \(?thesis\).

qed

next

case (\(3 \ x \ y \ ys \ qs\))

hence 0: prefix-measure (remove-var-prefix \(x\) (prefix-pop (ExistentialFirst \(y, ys\) \(qs\)))) ≤ prefix-measure (prefix-pop (ExistentialFirst \(y, ys\) \(qs\)))

by (cases \(x = y\)) (cases prefix-pop (ExistentialFirst \(y, ys\) \(qs\)) = Empty, simp-all)+

show \(?case\)

proof (cases \(x = y\))

case True

hence prefix-measure (remove-var-prefix \(x\) (ExistentialFirst \(y, ys\) \(qs\))) = prefix-measure (remove-var-prefix \(x\) (prefix-pop (ExistentialFirst \(y, ys\) \(qs\)))) by simp

also have ... ≤ prefix-measure (prefix-pop (ExistentialFirst \(y, ys\) \(qs\))) using 0

by simp

also have ... ≤ prefix-measure (ExistentialFirst \(y, ys\) \(qs\))

using le-eq-less-or-eq prefix-pop-decreases-measure by blast

finally show \(?thesis\).

next

case False

hence prefix-measure (remove-var-prefix \(x\) (ExistentialFirst \(y, ys\) \(qs\))) = prefix-measure (add-existential-to-prefix \(y\) (remove-var-prefix \(x\) (prefix-pop (ExistentialFirst \(y, ys\) \(qs\))))) by simp

also have ... ≤ Suc (prefix-measure (prefix-pop (ExistentialFirst \(y, ys\) \(qs\)))) using 0 add-ex-inc-prefix-measure by simp

also have ... ≤ prefix-measure (ExistentialFirst \(y, ys\) \(qs\)) using Suc-leI prefix-pop-decreases-measure by blast

finally show \(?thesis\).

qed

qed

fun first-var :: prefix ⇒ nat option where

first-var (ExistentialFirst \(x, xs\) \(qs\)) = Some \(x\) |

first-var (UniversalFirst \(y, ys\) \(qs\)) = Some \(y\) |

first-var Empty = None

lemma remove-first-var-decreases-measure:

assumes prefix ≠ Empty

shows prefix-measure (remove-var-prefix (the (first-var prefix)) prefix) < prefix-measure prefix

using assms

proof (induction prefix)

case (UniversalFirst \(q\) \(qs\))

then show \(?case\)

proof (induction \(q\))

case (Pair \(y\) \(ys\))

let \(?pre = UniversalFirst \(y, ys\) \(qs\)\)

let \(?var = the \((first-var \ ?pre)\)\)

have prefix-measure (remove-var-prefix \(?var \ ?pre\)) ≤ prefix-measure (prefix-pop \?pre) using remove-var-non-increasing-measure by simp

also have ... < prefix-measure \(?pre\) using prefix-pop-decreases-measure by blast

also have ... ≤ prefix-measure \(\?pre\) using le-eq-less-or-eq prefix-pop-decreases-measure by blast
finally show \( ?\text{case} \).

\[ \text{qed} \]

next

case (ExistentialFirst q qs)
then show \( ?\text{case} \)
proof
(induction q)

case (Pair x xs)
let \( ?\text{pre} = \text{ExistentialFirst} \ (x, \, xs) \, qs \)
let \( ?\text{var} = \text{the} \ (\text{first-var} \ ?\text{pre}) \)

have prefix-measure (remove-var-prefix ?var ?pre) \( \leq \) prefix-measure (prefix-pop ?pre)
using remove-var-non-increasing-measure by simp
also have \( \ldots < \) prefix-measure ?pre
using prefix-pop-decreases-measure by blast

finally show \( ?\text{case} \).
\[ \text{qed} \]

next

case Empty
then show \( ?\text{case} \) by simp
\[ \text{qed} \]

fun first-existential :: prefix \( \Rightarrow \) bool option where

first-existential (ExistentialFirst q qs) = Some True |
first-existential (UniversalFirst q qs) = Some False |
first-existential Empty = None

function search :: pcnf \( \Rightarrow \) bool option where

search (prefix, matrix) =
(if \( \emptyset \in \text{set matrix} \) then Some False
else if matrix = \( \emptyset \) then Some True.
else Option.bind (first-var prefix) (λz.
Option.bind (first-existential prefix) (λe.
if e then combine-options (\( \lor \))
(search (pcnf-assign (P z) (prefix, matrix)))
(search (pcnf-assign (N z) (prefix, matrix)))
else combine-options (\( \land \))
(search (pcnf-assign (P z) (prefix, matrix)))
(search (pcnf-assign (N z) (prefix, matrix)))))
by pat-completeness auto

termination

apply (relation measure \( \lambda (\text{pre}, \, \text{mat}), \) prefix-measure \( \text{pre} \))
apply (auto simp add: prefix-pop-decreases-measure simp del: prefix-measure.simps)
using remove-first-var-decreases-measure first-var.simps(3) option.disel option.sel
by metis+

fun search-solver :: pcnf \( \Rightarrow \) bool where

search-solver pcnf = the (search (pcnf-cleanse (pcnf-existential-closure pcnf))))

D.6.1 Correctness of the Search Function

lemma no-vars-if-no-free-no-prefix-vars:

\( \text{pcnf-free-variables pcnf} = \emptyset \) \( \Rightarrow \) \( \text{pcnf-prefix-variables pcnf} = \emptyset \) \( \Rightarrow \) \( \text{pcnf-variables} \)
\( \text{pcnf} = [] \)

by (metis Diff-iff list.set-intros(1) neq-Nil-conv pcnf-free-eq-vars-minus-prefix)

lemma no-vars-if-no-free-empty-prefix:
\( \text{pcnf-free-variables} (\text{Empty}, \text{matrix}) = [] \Rightarrow \text{pcnf-variables} (\text{Empty}, \text{matrix}) = [] \)
using no-vars-if-no-free-no-prefix-vars by fastforce

lemma search-cleansed-closed-yields-Some:

assumes cleansed-p pcnf \ and \ pcnf-free-variables pcnf = []

shows (\( \exists b. \text{search} \text{pcnf} = \text{Some} \ b \) \ using \ assms)

proof (induction pcnf rule: search.induct)

\begin{cases} 
\text{case} (1 \ \text{prefix} \ \text{matrix})
\
\text{then show} \ ?\text{thesis by auto}
\end{cases}

next

\begin{cases} 
\text{case} 2: \text{False}
\
\text{then show} \ ?\text{thesis}
\end{cases}

proof (cases \text{matrix} = [])

\begin{cases} 
\text{case} True
\
\text{then show} \ ?\text{thesis by auto}
\end{cases}

next

\begin{cases} 
\text{case} 3: \text{False}
\
\text{then show} \ ?\text{thesis}
\end{cases}

proof (cases \text{first-var} \ \text{prefix})

\begin{cases} 
\text{case} None
\
\text{hence} \ \text{prefix} = \text{Empty} \ \text{by} (\text{induction} \ \text{prefix}) \ \text{auto}
\
\text{hence} False \ \text{using} \ <\text{matrix} \neq []> \ <[] \notin \text{set} \ \text{matrix}>
\
\langle \text{pcnf-free-variables} (\text{prefix}, \text{matrix}) = []\rangle
\
\text{empty-clause-or-matrix-if-no-variables}
\
\text{no-vars-if-no-free-empty-prefix} \ \text{by} \ \text{blast}
\
\text{then show} \ ?\text{thesis by simp}
\end{cases}

next

\begin{cases} 
\text{case} 4: (\text{Some} \ z)
\
\text{then show} \ ?\text{thesis}
\end{cases}

proof (cases \text{first-existential} \ \text{prefix})

\begin{cases} 
\text{case} None
\
\text{hence} \ \text{prefix} = \text{Empty} \ \text{by} (\text{induction} \ \text{prefix}) \ \text{auto}
\
\text{hence} False \ \text{using} \ <\text{matrix} \neq []> \ <[] \notin \text{set} \ \text{matrix}>
\
\langle \text{pcnf-free-variables} (\text{prefix}, \text{matrix}) = []\rangle
\
\text{empty-clause-or-matrix-if-no-variables}
\
\text{no-vars-if-no-free-empty-prefix} \ \text{by} \ \text{blast}
\
\text{then show} \ ?\text{thesis by simp}
\end{cases}

next

\begin{cases} 
\text{case} 5: (\text{Some} \ e)
\
\text{have} 6: \ \text{pcnf-free-variables} (\text{pcnf-assign} \ \text{lit} (\text{prefix}, \text{matrix})) = []
\
\text{for} \ \text{lit} \ \text{using} \ \text{pcnf-assign-free-subseteq-free-minus-lit} 1(6)
\
\text{Diff-empty set-empty subset-Diff-insert subset-empty}
\
\text{by} \ \text{metis}
\
\text{then show} \ ?\text{thesis}
\end{cases}

proof (cases \text{e})

\begin{cases} 
\text{case} 7: True
\
\text{have} \ \text{search} (\text{prefix}, \text{matrix})
\end{cases}

161
= combine-options (∨)
   (search (pcnf-assign (P z) (prefix, matrix)))
   (search (pcnf-assign (N z) (prefix, matrix)))
using 2 3 4 5 7 by simp
moreover have ∃b. search (pcnf-assign (P z) (prefix, matrix)) = Some b
using 2 3 4 5 6 7 1(5,6) pcnf-assign-cleansed-inv 1(1)[af z e] by blast
moreover have ∃b. search (pcnf-assign (N z) (prefix, matrix)) = Some b
using 2 3 4 5 6 7 1(5,6) pcnf-assign-cleansed-inv 1(2)[af z e] by blast
ultimately show □thesis by force
next
case 7: False
have search (prefix, matrix)
   = combine-options (∧)
   (search (pcnf-assign (P z) (prefix, matrix)))
   (search (pcnf-assign (N z) (prefix, matrix)))
using 2 3 4 5 7 by simp
moreover have ∃b. search (pcnf-assign (P z) (prefix, matrix)) = Some b
using 2 3 4 5 6 7 1(5,6) pcnf-assign-cleansed-inv 1(3)[af z e] by blast
moreover have ∃b. search (pcnf-assign (N z) (prefix, matrix)) = Some b
using 2 3 4 5 6 7 1(5,6) pcnf-assign-cleansed-inv 1(4)[af z e] by blast
ultimately show □thesis by force
qed
qed
qed
qed
qed

theorem search-cleansed-closed-correct:
  assumes cleansed-p pcnf and pcnf-free-variables pcnf = []
  shows search pcnf = Some (satisfiable (convert pcnf))
using assms
proof (induction pcnf rule: search.induct)
case (1 prefix matrix)
then show □case
proof (cases [] ∈ set matrix)
case True
then show □thesis
using false-if-empty-clause-in-matrix qbf-semantics-eq-pcnf-semantics satisfiable-def
by simp
next
case 2: False
then show □thesis
proof (cases matrix = [])
case True
then show □thesis
using true-if-matrix-empty qbf-semantics-eq-pcnf-semantics satisfiable-def by simp
next
case 3: False
then show □thesis
proof (cases first-var prefix)
case None
  hence prefix = Empty by (induction prefix) auto
  hence False using ⟨matrix ≠ []⟩, ⟨[] ∉ set matrix⟩
\[\text{pcnf-free-variables} \ (\text{prefix}, \ \text{matrix}) = []\]
empty-clause-or-matrix-if-no-variables
no-vars-if-no-free-empty-prefix then show \(\text{thesis}\) by \(\text{simp}\)

next

\begin{itemize}
\item \textbf{case 4:} (Some \(z\))
  \begin{itemize}
  \item \textbf{then show} \(\text{thesis}\)
  \end{itemize}
\item \textbf{proof} (cases first-existential prefix)
  \begin{itemize}
  \item \textbf{case None}
    \begin{itemize}
    \item \textbf{hence} \(\text{prefix} = \text{Empty}\) by (induction prefix) \(\text{auto}\)
    \item \textbf{hence} \(\text{False}\) using \(\text{matrix} \neq \[]\) \(\langle [\] \notin \text{set} \text{matrix}\)
      \text{(pcnf-free-variables} \ (\text{prefix}, \ \text{matrix}) = []\)
    \end{itemize}
  \item \textbf{then show} \(\text{thesis}\) by \(\text{simp}\)
\end{itemize}
\item \textbf{next}
  \begin{itemize}
  \item \textbf{case 5:} (Some \(e\))
    \begin{itemize}
    \item \textbf{have} \(6\): \(\text{pcnf-free-variables} \ (\text{pcnf-assign} \ \text{lit} \ (\text{prefix}, \ \text{matrix})) = []\)
      \begin{itemize}
      \item \textbf{for} \(\text{lit}\) using \(\text{pcnf-assign-free-subseteq-free-minus-lit} \ (1(6)\)
        \text{Diff-empty set-empty subset-Diff-insert subset-empty by \(\text{metis}\)}
      \end{itemize}
    \item \textbf{hence} \(7\): \(\exists b. \ \text{search} \ (\text{pcnf-assign} \ \text{lit} \ (\text{prefix}, \ \text{matrix})) = \text{Some} b\) \(\text{for} \ \text{lit}\)
      \begin{itemize}
      \item \textbf{using} \(\text{search-cleansed-closed-yields} - \text{Some} \ \text{pcnf-assign-cleansed-inv} \ 1(5,6)\)
      \end{itemize}
    \end{itemize}
  \item \textbf{by} \(\text{blast}\)
\end{itemize}
\item \textbf{then show} \(\text{thesis}\)
\item \textbf{proof} (cases \(e\))
  \begin{itemize}
  \item \textbf{case 8:} \(\text{True}\)
    \begin{itemize}
    \item from this obtain \(x \ xs \ qs\) where \(\text{prefix-def}: \ \text{prefix} = \text{ExistentialFirst} \ (x, \ xs) \ qs\)
      \begin{itemize}
      \item \textbf{using} \(5\) by (induction prefix) \(\text{auto}\)
      \end{itemize}
    \item \textbf{have} \(\text{search} \ (\text{prefix}, \ \text{matrix})\)
      \begin{itemize}
      \item \textbf{= combine-options} \(\lor\)
        \begin{itemize}
        \item \textbf{(search} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix})))\)
        \item \textbf{(search} \ (\text{pcnf-assign} \ (N z) \ (\text{prefix}, \ \text{matrix})))\)
      \end{itemize}
      \end{itemize}
    \item \textbf{using} \(2 \ 3 \ 4 \ 5 \ 8\) \(\text{by} \ \text{simp}\)
    \begin{itemize}
    \item \textbf{hence} \(9\): \(\text{the} \ (\text{search} \ (\text{prefix}, \ \text{matrix}))\)
      \begin{itemize}
      \item \(\leftrightarrow \ \text{the} \ (\text{search} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix})))\)
        \begin{itemize}
        \item \(\lor \ \text{the} \ (\text{search} \ (\text{pcnf-assign} \ (N z) \ (\text{prefix}, \ \text{matrix})))\)
      \end{itemize}
    \item \textbf{using} \(7\) \(\text{combine-options-simps}(3)\) \(\text{option.sel by} \ \text{metis}\)
    \item \textbf{have} \(\text{search} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix}))\)
      \begin{itemize}
      \item \textbf{= Some} \(\text{(satisfiable} \ (\text{convert} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix}))))\)
      \end{itemize}
      \textbf{using} \(2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 1(5,6)\) \(\text{pcnf-assign-cleansed-inv} \ 1(1)[of \ z \ e]\) \(\text{by} \ \text{blast}\)
    \item \textbf{moreover} \textbf{have} \(\text{set} \ (\text{free-variables} \ (\text{convert} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix})))) = \{\}\)
      \textbf{using} \(6[\text{of} \ P z]\) \(\text{by} \ \text{simp}\)
    \item \textbf{ultimately} \textbf{have} \(10: \forall I. \ (\text{the} \ (\text{search} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix})))))\)
      \begin{itemize}
      \item \textbf{= qbf-semantics} \(I \ (\text{(convert} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix}))))\)
      \item \textbf{using} \(\text{semantics-eq-if-free-vars-eq}(\text{of convert} \ (\text{pcnf-assign} \ (P z) \ (\text{prefix}, \ \text{matrix}))))\)
      \end{itemize}
    \item \textbf{by} \(\text{(auto} \ \text{simp add: satisfiable-def}\)
    \item \textbf{have} \(\text{search} \ (\text{pcnf-assign} \ (N z) \ (\text{prefix}, \ \text{matrix}))\)
      \begin{itemize}
      \item \textbf{= Some} \(\text{(satisfiable} \ (\text{convert} \ (\text{pcnf-assign} \ (N z) \ (\text{prefix}, \ \text{matrix})))))\)
      \end{itemize}
      \textbf{using} \(2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 1(5,6)\) \(\text{pcnf-assign-cleansed-inv} \ 1(2)[of \ z \ e]\) \(\text{by} \ \text{blast}\)
    \item \textbf{moreover} \textbf{have} \(\text{set} \ (\text{free-variables} \ (\text{convert} \ (\text{pcnf-assign} \ (N z) \ (\text{prefix}, \ \text{matrix}))))\)
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{itemize}

(163)
matrix))) = {}
using 6[of N z] by simp
ultimately have 11: \forall I. the (search (pcnf-assign (N z) (prefix, matrix)))
= qbf-semantics I (convert (pcnf-assign (N z) (prefix, matrix)))
using semantics-eq-if-free-vars-eq[of convert (pcnf-assign (N z) (prefix, matrix))]
by (auto simp add: satisfiable-def)

have the (search (prefix, matrix))
= satisfiable (Disj
(convert (pcnf-assign (P z) (prefix, matrix))),
convert (pcnf-assign (N z) (prefix, matrix))))
using 9 10 11 satisfiable-def by simp

hence search (prefix, matrix)
= Some (satisfiable (Disj
(convert (pcnf-assign (P z) (prefix, matrix))),
convert (pcnf-assign (N z) (prefix, matrix)))))
using 1(5, 6) search-cleansed-closed-yields-Some by fastforce

moreover have z = x using prefix-def by simp
ultimately show thesis using sat-ex-first-iff-disj-sat' prefix-def
1(5) by simp

next
case 8: False
from this obtain y ys qs where prefix-def: prefix = UniversalFirst (y, ys)
qs

using 5 by (induction prefix) auto

have search (prefix, matrix)
= combine-options (∧
(search (pcnf-assign (P z) (prefix, matrix))))
(search (pcnf-assign (N z) (prefix, matrix))))
using 2 3 4 5 8 by simp

hence 9: the (search (prefix, matrix))
∧ the (search (pcnf-assign (N z) (prefix, matrix))))
using 7 combine-options-simps(3) option.sel by metis

have search (pcnf-assign (P z) (prefix, matrix))
= Some (satisfiable (convert (pcnf-assign (P z) (prefix, matrix)))))
using 2 3 4 5 6 8 1(5, 6) pcnf-assign-cleansed-inv 1(3)(of z e] by blast
moreover have set (free-variables (convert (pcnf-assign (P z) (prefix, matrix)))) = {}
using 6[of P z] by simp
ultimately have 10: \forall I. the (search (pcnf-assign (P z) (prefix, matrix)))
= qbf-semantics I (convert (pcnf-assign (P z) (prefix, matrix)))
using semantics-eq-if-free-vars-eq[of convert (pcnf-assign (P z) (prefix, matrix))]
by (auto simp add: satisfiable-def)

have search (pcnf-assign (N z) (prefix, matrix))
= Some (satisfiable (convert (pcnf-assign (N z) (prefix, matrix))))
using 2 3 4 5 6 8 1(5, 6) pcnf-assign-cleansed-inv 1(4)(of z e] by blast
moreover have set (free-variables (convert (pcnf-assign (N z) (prefix, matrix)))) = {}
using 6[of N z] by simp
ultimately have 11: \forall I. the (search (pcnf-assign (N z) (prefix, matrix)))
= qbf-semantics I (convert (pcnf-assign (N z) (prefix, matrix)))
using semantics-eq-if-free-vars-eq[of convert (pcnf-assign (N z) (prefix,
by (auto simp add: satisfiable_def)
have the (search (prefix, matrix))
  = satisfiable (Conj
    [convert (pcnf-assign (P z) (prefix, matrix)),
     convert (pcnf-assign (N z) (prefix, matrix))])
using 9 10 11 satisfiable_def by simp
hence search (prefix, matrix)
  = Some (satisfiable (Conj
    [convert (pcnf-assign (P z) (prefix, matrix)),
     convert (pcnf-assign (N z) (prefix, matrix))])
using 1(5,6) search-cleansed-closed-yields-Some by fastforce
moreover have z = y using prefix-def by simp
ultimately show ?thesis using sat-all-first-iff-assign-conj-sat
1(5) by simp
qed
qed
qed
qed

D.6.2 Correctness of the Search Solver

theorem search-solver-correct:
solve pcnf ⇔ satisfiable (convert pcnf)
proof
  have satisfiable (convert pcnf)
    = satisfiable (convert (pcnf-cleanse (pcnf-existential-closure pcnf)))
    using pcnf-sat-iff-ex-close-sat pcnf-cleanse-preserves-semantics
    qbf-semantics-eq-pcnf-semantics satisfiable_def by simp
  moreover have pcnf-free-variables (pcnf-cleanse (pcnf-existential-closure pcnf)) = []
    using pcnf-ex-closure-no-free cleanse-free-vars-inv set-empty by metis
  moreover have cleansed-p (pcnf-cleanse (pcnf-existential-closure pcnf))
    using pcnf-cleanse-cleanses by blast
  ultimately show ?thesis using search-cleansed-closed-correct by simp
qed
end

E Isabelle/HOL Theory: Solver Export

theory SolverExport
imports NaiveSolver PCNF SearchSolver Parser HOL-Library.Code-Abstract-Char
  HOL-Library.Code-Target-Numeral HOL-Library.RBT-Set
begin

fun run-naive-solver :: String.literal ⇒ bool where
  run-naive-solver qdimacs-str = naive-solver (convert (the (parse qdimacs-str)))

fun run-search-solver :: String.literal ⇒ bool where
  run-search-solver qdimacs-str = search-solver (the (parse qdimacs-str))

end
code-printing — COMMENT: This fixes a off-by-one error in the OCaml export.

```ml
module Str-Literal = struct

let implode f xs =
  let rec length xs =
    match xs with
    | [] -> 0
    | x :: xs -> 1 + length xs
    let rec nth xs n =
      match xs with
      | x :: xs -> if n <= 0 then x else nth xs (n - 1)
      in String.init (length xs) (fun n -> f (nth xs n));;

let explode f s =
  let rec map-range f lo hi =
    if lo >= hi then [] else f lo :: map-range f (lo + 1) hi
  in map-range (fun n -> f (String.get s n)) 0 (String.length s);;

let z-128 = Z.of-int 128;;

let check-ascii (k : Z.t) =
  if Z.leq Z.zero k && Z.lt k z-128 then k
  else failwith Non-ASCII character in literal;;

let char-of-ascii k = Char.chr (Z.to-int (check-ascii k));;

let ascii-of-char c = check-ascii (Z.of-int (Char.code c));;

let literal-of-asciis ks = implode char-of-ascii ks;;

let asciis-of-literal s = explode ascii-of-char s;;

end;;

for constant String.literal-of-asciis String.ascii-of-literal

export-code run-naive-solver
  in SML file-prefix run-naive-solver

export-code run-naive-solver
  in OCaml file-prefix run-naive-solver

export-code run-naive-solver
  in Scala file-prefix run-naive-solver

export-code run-naive-solver
  in Haskell file-prefix run-naive-solver

export-code run-search-solver
  in run-search-solver
```

166
in SML file-prefix run-search-solver

export-code
run-search-solver
in OCaml file-prefix run-search-solver

export-code
run-search-solver
in Scala file-prefix run-search-solver

export-code
run-search-solver
in Haskell file-prefix run-search-solver

end