# Verified QBF Solving

Axel Bergström and Tjark Weber

March 8, 2024

**Abstract**

Quantified Boolean logic extends propositional logic with universal and existential quantification over Boolean variables. A Quantified Boolean Formula (QBF) is satisfiable iff there is an assignment of Boolean values to the formula's free variables that makes the formula true, and a QBF solver is a software tool that determines whether a given QBF is satisfiable.

We formalise two simple QBF solvers and prove their correctness. One solver is based on naive quantifier expansion, while the other utilises a search-based algorithm. Additionally, we formalise a parser for the QDIMACS input format and use Isabelle's code generation feature to obtain executable versions of both solvers.

The formalisation is discussed in detail in [1].

# Contents

# 1 Naive Solver Implementation and Verification

**theory** *NaiveSolver*
  **imports** *Main*
**begin**

## 1.1 QBF Datatype, Semantics, and Satisfiability

### 1.1.1 QBF Datatype

QBFs based on [2].

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

### 1.1.2 Formalisation of Semantics and Termination of Semantics

Substitute True or False for a variable:

**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*)

Measures the number of QBF constructors in argument, required to show termination of semantics.

**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*

Substituting for variable does not change the QBF measure.

**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*)
    **by** (*cases b*) *auto*
**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*)

**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**

The measure of an element in a disjunction/conjunction is less than the measure of the disjunction/conjunction.

**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**

Semantics based on [2].

**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*))

$\vee$ (*qbf-semantics I* (*substitute-var x False qbf*)))
| *qbf-semantics I* (*All x qbf*) = ((*qbf-semantics I* (*substitute-var x True qbf*))
$\wedge$ (*qbf-semantics I* (*substitute-var x False qbf*)))
**by** *pat-completeness auto*
**termination**
**apply** (*relation measure* ($\lambda$(*I*,*qbf*). *qbf-measure qbf*))
**by** (*auto simp add*: *qbf-measure-substitute qbf-measure-lt-sum-list*)

Simple tests.

**definition** *test-qbf* = (*All 3* (*Conj* [*Disj* [*Neg* (*Var 2*), *Var 3*, *Var 1*], *Disj* [*Neg* (*Var 1*), *Var 2*]]))

**value** *substitute-var 1 False test-qbf*
**value** *substitute-var 1 True test-qbf*
**value** *substitute-var 2 False test-qbf*
**value** *substitute-var 2 True test-qbf*
**value** *substitute-var 3 False test-qbf*
**value** *substitute-var 3 True test-qbf*

**value** *qbf-semantics* ($\lambda x$. *False*) *test-qbf*
**value** *qbf-semantics* (($\lambda x$. *False*)(*2* := *True*)) *test-qbf*
**value** *qbf-semantics* ((($\lambda x$. *False*)(*2* := *True*))(*1* := *True*)) *test-qbf*

### 1.1.3 Formalisation of Satisfiability

**definition** *satisfiable* :: *QBF* $\Rightarrow$ *bool* **where**
*satisfiable qbf* = ($\exists$ *I*. *qbf-semantics I qbf*)

**definition** *logically-eq* :: *QBF* $\Rightarrow$ *QBF* $\Rightarrow$ *bool* **where**
*logically-eq qbf1 qbf2* = ($\forall$ *I*. *qbf-semantics I qbf1* = *qbf-semantics I qbf2*)

## 1.2 Existential Closure

### 1.2.1 Formalisation of Free Variables

**fun** *free-variables-aux* :: *nat set* $\Rightarrow$ *QBF* $\Rightarrow$ *nat list* **where**
*free-variables-aux bound* (*Var x*) = (*if x* $\in$ *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* $\Rightarrow$ *nat list* **where**
*free-variables qbf* = *sort* (*remdups* (*free-variables-aux* {} *qbf*))

**lemma** *bound-subtract-equiv*:
*set* (*free-variables-aux* (*bound* $\cup$ *new*) *qbf*) = *set* (*free-variables-aux bound qbf*) $-$ *new*
**by** (*induction bound qbf rule*: *free-variables-aux.induct*) *auto*

### 1.2.2 Formalisation of Existential Closure

```
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)
```

### 1.2.3 Preservation of Satisfiability under Existential Quantification

```
lemma swap-substitute-var-order:
  assumes x1 ≠ x2 ∨ b1 = b2
  shows substitute-var x1 b1 (substitute-var x2 b2 qbf)
      = substitute-var x2 b2 (substitute-var x1 b1 qbf)
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
qed simp-all

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
qed simp-all

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** (*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** *auto*

**lemma** *sat-iff-ex-sat*: *satisfiable qbf* $\longleftrightarrow$ *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** $I(x := I\ x) = I(x := \mathit{True}) \vee I(x := I\ x) = I(x := \mathit{False})$ **by** (*cases I x*) *auto*
**hence** $I = I(x := \mathit{True}) \vee I = I(x := \mathit{False})$ **by** *simp*
**hence** *qbf-semantics* ($I(x := \mathit{True})$) *qbf* $\vee$ *qbf-semantics* ($I(x := \mathit{False})$) *qbf*
**using** *I-def* **by** *fastforce*
**moreover have** *satisfiable* (*Ex x qbf*)
= ($\exists I.$ *qbf-semantics* ($I(x := \mathit{True})$) *qbf*
$\vee$ *qbf-semantics* ($I(x := \mathit{False})$) *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**

### 1.2.4 Preservation of Satisfiability under Existential Closure

**lemma** *sat-iff-ex-close-aux-sat*: *satisfiable qbf* $\longleftrightarrow$ *satisfiable* (*existential-closure-aux qbf vars*)
**using** *sat-iff-ex-sat* **by** (*induction vars*) *auto*

**theorem** *sat-iff-ex-close-sat*: *satisfiable qbf* $\longleftrightarrow$ *satisfiable* (*existential-closure qbf*)
**using** *sat-iff-ex-close-aux-sat* **by** *simp*

### 1.2.5 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** (*Cons x xs*)
**thus** *?case* **using** *bound-subtract-equiv*[*of* {} {*x*}] **by** *auto*
**qed** *auto*

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

## 1.3 Sequence Utility Function

Like sequence in Haskell specialised for option types.

**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* (λ*x*. *x* ≠ *None*) *list*
**shows** ∃ *xs*. *map Some xs* = *list* **using** *assms*
**proof** (*induction list*)
**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** *auto*

**lemma** *sequence-aux-content*: *sequence-aux* (*map Some xs*) *list* = *Some* (*rev xs* @ *list*)
**by** (*induction xs arbitrary*: *list*) *auto*

**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**

## 1.4 Naive Solver

### 1.4.1 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** (*Conj xs*)
  **thus** *?case* **by** (*induction xs*) *auto*
**next**
  **case** (*Disj xs*)
  **thus** *?case* **by** (*induction xs*) *auto*
**qed** *auto*

**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** (*Cons x xs*)
  **thus** *qbf-quantifier-depth z* = *list-max* (*map qbf-quantifier-depth* (*x* # *xs*))
    **by** (*cases x* = *z*) *auto*
**qed** *auto*

**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*)

Property 1: no quantifiers after expansion.

**lemma** *no-quants-after-expand-quants*: *qbf-quantifier-depth* (*expand-quantifiers qbf*) = *0*
**proof** (*induction qbf*)
**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** *auto*

Property 2: semantics invariant under expansion (logical equivalence).

**lemma** *semantics-inv-under-expand*:
*qbf-semantics I qbf* = *qbf-semantics I* (*expand-quantifiers qbf*)
**proof** (*induction qbf arbitrary*: *I*)
**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** *auto*

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

Property 3: free variables invariant under expansion.

**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** (*6 z b x qbf*)
**then show** *?case*
**proof** (*cases x* = *z*)
**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** *simp*
**next**
**case** (*7 z b y qbf*)
**thus** *?case*
**proof** (*cases y* = *z*)
**case** *False*
**thus** *?thesis* **using** *7 bound-subtract-equiv*[**where** *?new* = {*y*}] **by** *simp*
**qed** *simp*
**qed** *auto*

**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** (*6 z b x qbf*)
**then show** *?case*
**proof** (*cases x* = *z*)
**case** *False*
**thus** *?thesis* **using** *6 bound-subtract-equiv*[**where** *?new* = {*x*}] **by** *simp*
**qed** *auto*
**next**
**case** (*7 z b y qbf*)
**thus** *?case*
**proof** (*cases y* = *z*)
**case** *False*
**thus** *?thesis* **using** *7 bound-subtract-equiv*[**where** *?new* = {*y*}] **by** *simp*
**qed** *auto*
**qed** *auto*

**lemma** *free-vars-inv-under-expand-quants*:
*set* (*free-variables* (*expand-quantifiers qbf*)) = *set* (*free-variables qbf*)
**proof** (*induction qbf*)
**case** (*Ex x1a qbf*)
**have** *set* (*free-variables* (*expand-quantifiers* (*QBF.Ex x1a qbf*)))

$= set\ (free\text{-}variables\text{-}aux\ \{x1a\}\ (expand\text{-}quantifiers\ qbf))$
**using** *set-free-vars-subst-ex-eq* **by** *simp*
**also have** ... $= set\ (free\text{-}variables\ (expand\text{-}quantifiers\ qbf)) - \{x1a\}$
**using** *bound-subtract-equiv*[**where** $?new = \{x1a\}$] **by** *simp*
**also have** ... $= set\ (free\text{-}variables\ qbf) - \{x1a\}$ **using** *Ex.IH* **by** *simp*
**also have** ... $= set\ (free\text{-}variables\text{-}aux\ \{x1a\}\ qbf)$
**using** *bound-subtract-equiv*[**where** $?new = \{x1a\}$] **by** *simp*
**also have** ... $= set\ (free\text{-}variables\ (QBF.Ex\ x1a\ qbf))$ **by** *simp*
**finally show** *?case* **.**
**next**
**case** (*All x1a qbf*)
**thus** *?case* **using** *bound-subtract-equiv*[**where** $?new = \{x1a\}$] *set-free-vars-subst-all-eq*
**by** *simp*
**qed** *auto*

### 1.4.2 Expanding Formulas

**fun** *expand-qbf* :: $QBF \Rightarrow QBF$ **where**
*expand-qbf qbf* = *expand-quantifiers* (*existential-closure qbf*)

The important properties from the existential closure and quantifier expansion are preserved.

**lemma** *sat-iff-expand-qbf-sat*: *satisfiable* (*expand-qbf qbf*) $\longleftrightarrow$ *satisfiable qbf*
**using** *sat-iff-ex-close-sat sat-iff-expand-quants-sat* **by** *simp*

**lemma** *expand-qbf-no-free*: $set\ (free\text{-}variables\ (expand\text{-}qbf\ qbf)) = \{\}$
**proof** $-$
**have** $set\ (free\text{-}variables\ (expand\text{-}qbf\ qbf)) = set\ (free\text{-}variables\ (existential\text{-}closure\ qbf))$
**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*

### 1.4.3 Evaluating Expanded Formulas

**fun** *eval-qbf* :: $QBF \Rightarrow bool\ option$ **where**
*eval-qbf* (*Var x*) = *None* |
*eval-qbf* (*Neg qbf*) = *map-option* ($\lambda x.\ \neg x$) (*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** *pred-map-ex*: *list-ex Q* (*map f x*) = *list-ex* ($Q \circ f$) *x*
**by** (*induction x*) *auto*

The evaluation implements the semantics.

**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*
**proof** (*induction qbf*)
**case** (*Conj x*)
**hence** ∀ *q* ∈ *set x*. *eval-qbf q* = *Some* (*qbf-semantics I q*) **by** (*induction x*) *auto*
**thus** *eval-qbf* (*Conj x*) = *Some* (*qbf-semantics I* (*Conj x*))
**proof** (*induction x*)
**case** *Nil*
**show** *eval-qbf* (*Conj* []) = *Some* (*qbf-semantics I* (*Conj* [])) **by** *simp*
**next**
**case** (*Cons y ys*)
**have** *map eval-qbf ys* = *map Some* (*map* (*qbf-semantics I*) *ys*) **using** *Cons* **by** *simp*
**moreover have** *eval-qbf y* = *Some* (*qbf-semantics I y*) **using** *Cons.prems* **by** *simp*
**ultimately have** *map eval-qbf* (*y* # *ys*) = *map Some* (*map* (*qbf-semantics I*) (*y* # *ys*)) **by** *simp*
**hence** *sequence* (*map eval-qbf* (*y* # *ys*)) = *Some* (*map* (*qbf-semantics I*) (*y* # *ys*))
**using** *sequence-content* **by** *metis*
**hence** *eval-qbf* (*Conj* (*y* # *ys*))
= *Some* (*list-all id* (*map* (*qbf-semantics I*) (*y* # *ys*)))
**by** *simp*
**hence** *eval-qbf* (*Conj* (*y* # *ys*)) = *Some* (*list-all* (*qbf-semantics I*) (*y* # *ys*))
**by** (*simp add*: *fun.map-ident list.pred-map*)
**thus** *eval-qbf* (*Conj* (*y* # *ys*)) = *Some* (*qbf-semantics I* (*Conj* (*y* # *ys*))) **by** *simp*
**qed**
**next**
**case** (*Disj x*)
**hence** ∀ *q* ∈ *set x*. *eval-qbf q* = *Some* (*qbf-semantics I q*) **by** (*induction x*) *auto*
**thus** *eval-qbf* (*Disj x*) = *Some* (*qbf-semantics I* (*Disj x*))
**proof** (*induction x*)
**case** *Nil*
**show** *eval-qbf* (*Disj* []) = *Some* (*qbf-semantics I* (*Disj* [])) **by** *simp*
**next**
**case** (*Cons y ys*)
**have** *map eval-qbf ys* = *map Some* (*map* (*qbf-semantics I*) *ys*) **using** *Cons* **by** *simp*
**moreover have** *eval-qbf y* = *Some* (*qbf-semantics I y*) **using** *Cons.prems* **by** *simp*
**ultimately have** *map eval-qbf* (*y* # *ys*) = *map Some* (*map* (*qbf-semantics I*) (*y* # *ys*)) **by** *simp*
**hence** *sequence* (*map eval-qbf* (*y* # *ys*)) = *Some* (*map* (*qbf-semantics I*) (*y* # *ys*))
**using** *sequence-content* **by** *metis*
**hence** *eval-qbf* (*Disj* (*y* # *ys*))
= *Some* (*list-ex id* (*map* (*qbf-semantics I*) (*y* # *ys*)))

**by** *simp*
**hence** *eval-qbf* (*Disj* (*y* # *ys*)) = *Some* (*list-ex* (*qbf-semantics I*) (*y* # *ys*))
**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**
**qed** *auto*

### 1.4.4 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-iff-expand-qbf-sat* **by** *simp*
**qed**

Simple tests.

**value** *test-qbf*
**value** *existential-closure test-qbf*
**value** *expand-qbf test-qbf*
**value** *naive-solver test-qbf*

**end**

# 2 Prenex Conjunctive Normal Form Datatype

**theory** *PCNF*
**imports** *NaiveSolver*
**begin**

## 2.1 Prenex Conjunctive 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*

### 2.1.1 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*

### 2.1.2 Bijection with PCNF Subset of Generic QBF Datatype

Conversion functions, left-inverses thereof, and proofs of the left-inverseness.

**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* :: *matrix* ⇒ *QBF* **where**
*convert-matrix matrix* = *Conj* (*map convert-clause matrix*)

**lemma** *convert-cnf-p*: *cnf-p* (*convert-matrix mat*)
**using** *convert-clause-p* **by** (*induction mat*) *auto*

**fun** *convert-matrix-inv* :: *QBF* ⇒ *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 mat*) = *Some mat*
**proof** −
**let** *?list* = *map convert-clause-inv* (*map convert-clause mat*)
**have** ∀ *x* ∈ *set mat*. *convert-clause-inv* (*convert-clause x*) = *Some x* **using** *clause-inv* **by** *simp*
**hence** *map Some mat* = *?list* **using** *list-no-None-ex-list-map-Some* **by** *fastforce*
**hence** *sequence ?list* = *Some mat* **using** *sequence-content* **by** *metis*
**thus** *convert-matrix-inv* (*convert-matrix mat*) = *Some mat* **by** *simp*
**qed**

**fun** *q-length* :: *'a* × *'a list* ⇒ *nat* **where**
*q-length* (*x*, *xs*) = *1* + *length xs*

**fun** *measure-prefix-length* :: *pcnf* ⇒ *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* :: *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** *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* (λ*p*. *add-universal-to-front x p*) (*convert-inv qbf*)
| *convert-inv* (*Ex x qbf*) = *map-option* (λ*p*. *add-existential-to-front x p*) (*convert-inv qbf*)
| *convert-inv qbf* = *map-option* (λ*m*. (*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* ∘ *convert-inv*])
  **by** (*simp add*: *convert-inv*)

There is a PCNF formula yielding any *pcnf-p* QBF formula:

**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** ∃ *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

lemma convert-cnf-p-ex:
  assumes cnf-p mat
  shows ∃ m. convert-matrix m = mat
proof −
  from assms obtain xs where 0: Conj xs = mat by (metis cnf-p.elims(2))
  hence list-all clause-p xs using assms by fastforce
  hence ∃ ys. map convert-clause ys = xs using convert-clause-p-ex
  proof (induction xs)
    case Nil
    show ∃ ys. map convert-clause ys = [] by simp
  next
    case (Cons x xs)
    from this obtain ys where map convert-clause ys = xs by fastforce
    moreover from Cons obtain y where convert-clause y = x by fastforce
    ultimately have map convert-clause (y # ys) = x # xs by simp
    thus ∃ ys. map convert-clause ys = x # xs by (rule exI)
  qed
  thus ∃ m. convert-matrix m = mat using 0 by fastforce
qed

theorem convert-pcnf-p-ex:
  assumes pcnf-p qbf
  shows ∃ pcnf. convert pcnf = qbf using assms
proof (induction qbf)
  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
  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 con-
vert-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 auto


theorem convert-range: range convert = {p. pcnf-p p}
```

**using** *convert-pcnf-p convert-pcnf-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*)

### 2.1.3 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* (*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*, []) (*q* # *qs*), *matrix*) =
(*pcnf-semantics* (*I*(*y* := *True*)) (*ExistentialFirst q qs*, *matrix*)
∧ *pcnf-semantics* (*I*(*y* := *False*)) (*ExistentialFirst q qs*, *matrix*))
| *pcnf-semantics I* (*ExistentialFirst* (*x*, []) (*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*, *yy* # *ys*) *qs*, *matrix*) =
(*pcnf-semantics* (*I*(*y* := *True*)) (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*)
∧ *pcnf-semantics* (*I*(*y* := *False*)) (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*))
| *pcnf-semantics I* (*ExistentialFirst* (*x*, *xx* # *xs*) *qs*, *matrix*) =
(*pcnf-semantics* (*I*(*x* := *True*)) (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*)
∨ *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*

**theorem** *qbf-semantics-eq-pcnf-semantics*:
*pcnf-semantics I pcnf* = *qbf-semantics I* (*convert pcnf*)
**proof** (*induction pcnf arbitrary*: *I rule*: *convert.induct*)
**case** (*1 matrix*)
**then show** *?case*
**proof** (*induction matrix*)

```
    case (Cons cl cls)
    then show ?case
    proof (induction cl)
      case (Cons l ls)
      then show ?case by (induction l) force+
    qed auto
  qed auto
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:
  pcnf-p qbf ⟹ convert (the (convert-inv qbf)) = qbf
  by (metis convert-inv convert-pcnf-p-ex option.sel)

theorem qbf-semantics-eq-pcnf-semantics':
  assumes pcnf-p qbf
  shows qbf-semantics I qbf = pcnf-semantics I (the (convert-inv qbf))
  using qbf-semantics-eq-pcnf-semantics assms convert-inv-inv by simp

end
```

# 3 QDIMACS Parser

```
theory Parser
  imports 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*) $\leq$ *length s*
  **by** (*induction s*) *auto*

**lemma** *non-increasing-trim-ws-lemmas* [*intro*]:
  **shows** *length s* $\leq$ *length s'* $\Longrightarrow$ *length* (*trim-ws s*) $\leq$ *length s'*
    **and** *length s* $<$ *length s'* $\Longrightarrow$ *length* (*trim-ws s*) $<$ *length s'*
    **and** *length s* $\leq$ *length* (*trim-ws s'*) $\Longrightarrow$ *length s* $\leq$ *length s'*
    **and** *length s* $<$ *length* (*trim-ws s'*) $\Longrightarrow$ *length s* $<$ *length s'*
     **apply** (*induction s*)
        **apply** *simp-all*
  **subgoal using** *trim-ws.simps*(*1*) **by** *blast*
  **subgoal using** *non-increasing-trim-ws order-less-le-trans* **by** *blast*
  **done**

**lemma** *whitespace-and-parse-le* [*intro*]:
  **assumes** $\bigwedge$*s s' r. p s* = *Some* (*r*, *s'*) $\Longrightarrow$ *length s'* $\leq$ *length s*
   **shows** $\bigwedge$*s s' r. p* (*trim-ws s*) = *Some* (*r*, *s'*) $\Longrightarrow$ *length s'* $\leq$ *length s* **using** *assms*
  **using** *le-trans non-increasing-trim-ws* **by** *blast*

**lemma** *whitespace-and-parse-unit-le* [*intro*]:
  **assumes** $\bigwedge$*s s'. p s* = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $\leq$ *length s*
  **shows** $\bigwedge$*s s'. p* (*trim-ws s*) = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $\leq$ *length s* **using** *assms*
  **using** *le-trans non-increasing-trim-ws* **by** *blast*

**lemma** *whitespace-and-parse-less* [*intro*]:
  **assumes** $\bigwedge$*s s' r. p s* = *Some* (*r*, *s'*) $\Longrightarrow$ *length s'* $<$ *length s*
   **shows** $\bigwedge$*s s' r. p* (*trim-ws s*) = *Some* (*r*, *s'*) $\Longrightarrow$ *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** $\bigwedge$*s s'. p s* = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $<$ *length s*
  **shows** $\bigwedge$*s s'. p* (*trim-ws s*) = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $<$ *length s* **using** *assms*
  **using** *non-increasing-trim-ws order-less-le-trans* **by** *blast*

**fun** *match* :: *string* ⇒ *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'*) $\Longrightarrow$ *length s'* $\leq$ *length s*
  **apply** (*induction xs s rule*: *match.induct*)
    **apply** *auto*[*2*]

```
  by (metis le-imp-less-Suc length-Cons match.simps(3) option.simps(3) order-less-imp-le)

lemma decreasing-match [simp]:
  xs ≠ [] ⟹ match xs s = Some ((), s') ⟹ length s' < length s
proof (induction xs s rule: match.induct)
  case (3 x xs y ys)
  hence x = y by (cases x = y) auto
  hence match (Cons x xs) (Cons y ys) = match xs ys by simp
  hence match xs ys = Some ((), s') using 3 by simp
  hence length s' ≤ length ys by simp
  thus length s' < length (Cons y ys) by simp
qed auto

fun digit-to-nat :: char ⇒ nat option where
  digit-to-nat c = (
    if c = CHR ''0'' then Some 0 else
    if c = CHR ''1'' then Some 1 else
    if c = CHR ''2'' then Some 2 else
    if c = CHR ''3'' then Some 3 else
    if c = CHR ''4'' then Some 4 else
    if c = CHR ''5'' then Some 5 else
    if c = CHR ''6'' then Some 6 else
    if c = CHR ''7'' then Some 7 else
    if c = CHR ''8'' then Some 8 else
    if c = CHR ''9'' then Some 9 else
    None)

fun num-aux :: nat ⇒ nat parser where
  num-aux n Nil = Some (n, Nil)
| num-aux n (Cons x xs) =
     (if List.member ''0123456789'' x
        then num-aux (10 * n + the (digit-to-nat x)) xs
        else Some (n, Cons x xs))

lemma non-increasing-num-aux [simp]: num-aux n s = Some (m, s') ⟹ length
s' ≤ length s
  apply (induction n s rule: num-aux.induct)
   apply simp
  by (metis (no-types, lifting) le-imp-less-Suc length-Cons nle-le num-aux.simps(2)
option.sel order-less-imp-le prod.sel(2))

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'))
```

Simple tests.

```
value pnum ''123''
value pnum ''−123''
value pnum ''0123''
value pnum ''0''

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 (-, str') ⇒ (case pnum str' of
      None ⇒ None |
      Some (n, str'') ⇒ Some (N n, str'')))
```

Simple tests.

```
value literal ''123''
value literal ''−123''
value literal ''− 123''
value literal ''0123''
value literal ''0''

lemma decreasing-literal [simp]:
  assumes literal s = Some (l, s')
```

**shows** *length s′ < length s*
**proof** (*cases match ′′−′′ s*)
**case** *None*
**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** *length s′′ ≤ length s* **using** *Some* **by** *simp*
**moreover have** *length s′ < length s′′* **using** *s′′-def assms Some* **by** (*cases pnum s′′*) *auto*
**ultimately show** *length s′ < length s* **by** *simp*
**qed**

**fun** *clause* :: *PCNF.clause parser* **where**
*clause str* = (*case literal* (*trim-ws str*) *of*
*None* ⇒ *None* |
*Some* (*l*, *str′*) ⇒
(*case clause str′ of*
*None* ⇒
(*case match ′′0′′* (*trim-ws str′*) *of*
*None* ⇒ *None* |
*Some* (-, *str′′*) ⇒
(*case match ′′*$\boxed{\hookleftarrow}$*′′* (*trim-ws str′′*) *of*
*None* ⇒ *None* |
*Some* (-, *str′′′*) ⇒ *Some* (*Cons l Nil*, *str′′′*))) |
*Some* (*cl*, *str′′*) ⇒ *Some* (*Cons l cl*, *str′′*)))

Simple tests.

**value** *clause ′′1 2 −3 4 0* $\boxed{\hookleftarrow}$*′′*
**value** *clause ′′1 2 −3   4   0* $\boxed{\hookleftarrow}$*′′*
**value** *clause ′′1 2 −3 40* $\boxed{\hookleftarrow}$*′′*
**value** *clause ′′1 2 −3 4 0*$\boxed{\hookleftarrow}$*′′*
**value** *clause ′′1 2 −3 4 0′′*
**value** *clause ′′ 1 2 −3 4 0* $\boxed{\hookleftarrow}$*′′*

**lemma** *decreasing-clause* [*simp*]:
**assumes** *clause s* = *Some* (*c*, *s′*)
**shows** *length s′ < length s* **using** *assms*
**proof** (*induction s arbitrary*: *c rule*: *clause.induct*)
**case** (*1 s*)
**show** *?case*
**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: length s'' < length s using Some-a by fastforce
    show ?thesis
    proof (cases clause s'')
      case None': None
      show ?thesis
      proof (cases match ''0'' (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: length s''' ≤ 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 u s'''' where c-def: c = (u, s'''') by (meson surj-pair)
          hence le2: length s'''' ≤ length s''' using Some-c 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 less1 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 less1 by simp
    qed
  qed
qed
```

```
fun clause-list :: PCNF.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'')))
```

Simple tests.

**value** *clause-list* $''1\ 2\ -3\ 0\boxed{\hookleftarrow}1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$
**value** *clause-list* $''1\ 2\ -3\ \boxed{\hookleftarrow}1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$
**value** *clause-list* $''1\ 2\ -3\ 0\ \boxed{\hookleftarrow}\ 1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$

**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** *?case*
  **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** *less1* **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**
  **qed**
**qed**

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

Simple tests.

**value** *matrix* $''1\ 2\ -3\ 0\boxed{\hookleftarrow}1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$
**value** *matrix* $''1\ 2\ -3\ \boxed{\hookleftarrow}1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$
**value** *matrix* $''1\ 2\ -3\ 0\ \boxed{\hookleftarrow}\ 1\ -2\ 3\ 0\boxed{\hookleftarrow}-1\ 2\ \ 3\ 0\boxed{\hookleftarrow}''$

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

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

Simple tests.

```
value atom-set ''1 2 3 4''
value atom-set ''1 2 −3 4''
value atom-set ''1 2 3   4   0 ↩''
value atom-set ''1 2 3 40''
value atom-set ''1 2 3 4 0↩''
value atom-set ''1 2 3 4''
value atom-set '' 1   2  3 4 0 ↩  ''

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 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 atom-set s'' = Some ((a', as), s') using Some-c c-def by simp
      hence length s' < length s'' using 1 Some-b b-def by blast
      thus length s' < 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 ⇒ (case match ''a'' str of
      None ⇒ None |
      Some (-, str') ⇒ Some (Universal, str')) |
    Some (-, str') ⇒ Some (Existential, str'))
```

Simple tests.

```
value quantifier ''a 1 2 3''
value quantifier ''e 1 2 3''
value quantifier ''a 1 2 3''
value quantifier '' e 1 2 3''

lemma non-increasing-quant [simp]:
  assumes quantifier s = Some (q, s')
  shows length s' ≤ length s
proof (cases match ''e'' s)
  case None-e: None
  then show ?thesis
  proof (cases match ''a'' s)
    case None
    hence False using None-e assms by simp
    thus ?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 length s' ≤ 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 length s' ≤ length s using Some-a a-def by simp
qed


fun quant-set :: (quant × (nat × nat list)) parser where
  quant-set str = (case quantifier (trim-ws str) of
    None ⇒ None |
```

```
  Some (q, str') ⇒
    (case atom-set (trim-ws str') of
      None ⇒ None |
      Some (as, str'') ⇒
        (case match ''0'' (trim-ws str'') of
          None ⇒ None |
          Some (-, str''') ⇒
            (case match ''↩'' (trim-ws str''') of
              None ⇒ None |
              Some (-, str'''') ⇒ Some ((q, as), str'''')))))
```

Simple tests.

```
value quant-set ''e 1 2  3 0↩''
value quant-set ''a 1 2 3 0↩''
value quant-set ''a 1 2 −3 0↩''

lemma decreasing-quant-set [simp]:
  assumes quant-set s = Some (q-set, s')
  shows length s' < length s
proof (cases quantifier (trim-ws 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'' ≤ length s using Some-a by fastforce
  show ?thesis
  proof (cases atom-set (trim-ws 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-ws 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'''' ≤ length s''' using Some-c by fastforce
      show ?thesis
      proof (cases match ''↩'' (trim-ws 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''''' ≤ 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
qed
```

```
fun quant-sets :: (quant × (nat × nat list)) list parser where
  quant-sets str = (case quant-set str of
    None ⇒ None |
    Some (q-set, str') ⇒
      (case quant-sets str' of
        None ⇒ Some (Cons q-set Nil, str') |
        Some (q-sets, str'') ⇒ Some (Cons q-set q-sets, str'')))
```

Simple tests.

```
value quant-sets ''a 1 2 3 0↩e 4 5 6 0↩a 7 8  9 0 ↩''
value quant-sets ''a 1 2 3 0↩  e  4  5  6  0 ↩e 7 8  9 0 ↩''
```

```
lemma decreasing-quant-sets [simp]:
  assumes quant-sets s = Some (q-sets, s')
  shows length s' < length s using assms
proof (induction s arbitrary: q-sets rule: quant-sets.induct)
  case (1 s)
  show ?case
  proof (cases quant-set s)
    case None
    hence False using 1 by simp
    thus ?thesis by simp
  next
    case Some-a: (Some a)
    obtain q-set s'' where a-def: a = (q-set, s'') by fastforce
    hence less1: length s'' < length s using Some-a by simp
    show ?thesis
    proof (cases quant-sets s'')
      case None
     hence quant-sets s = Some (Cons q-set Nil, s'') using Some-a a-def by simp
      hence s' = s'' using 1 by simp
      thus length s' < length s using less1 by simp
    next
      case Some-b: (Some b)
```

```
      obtain q-sets s''' where b-def: b = (q-sets, s''') by fastforce
     hence quant-sets s = Some (Cons q-set q-sets, s''') using Some-a a-def Some-b
by simp
      hence s' = s''' using 1 by simp
      hence quant-sets s'' = Some (q-sets, s') using Some-b b-def by simp
      hence length s' < length s'' using 1 Some-a a-def by simp
      thus length s' < length s using less1 by simp
    qed
  qed
qed

fun convert-quant-sets :: (quant × (nat × nat list)) list ⇒ PCNF.prefix option
where
  convert-quant-sets Nil = Some Empty
| convert-quant-sets (Cons (Universal, as) qs) =
    (case convert-quant-sets qs of
      None ⇒ None |
      Some Empty ⇒ Some (UniversalFirst as Nil) |
      Some (ExistentialFirst as' qs') ⇒ Some (UniversalFirst as (Cons as' qs')) |
      Some (UniversalFirst - -) ⇒ None)
| convert-quant-sets (Cons (Existential, as) qs) =
    (case convert-quant-sets qs of
      None ⇒ None |
      Some Empty ⇒ Some (ExistentialFirst as Nil) |
      Some (ExistentialFirst - -) ⇒ None |
      Some (UniversalFirst as' qs') ⇒ Some (ExistentialFirst as (Cons as' qs')))


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')))
```

Simple tests.

```
value prefix ''a 1 2 3 0⏎e 4 5 6 0⏎a 7 8  9 0 ⏎''
value prefix ''a 1 2 3 0⏎e 4 5 6 0⏎e 7 8  9 0 ⏎''

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
  hence s' = s'' using Some assms by (cases convert-quant-sets pre) auto
  moreover have 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-ws str) of
    None ⇒ None |
    Some (-, str1) ⇒
      (case match ''cnf'' (trim-ws str1) of
        None ⇒ None |
        Some (-, str2) ⇒
          (case pnum (trim-ws str2) of
            None ⇒ None |
            Some (lits, str3) ⇒
              (case pnum (trim-ws str3) of
                None ⇒ None |
                Some (clauses, str4) ⇒
                  (case match ''↩'' (trim-ws str4) of
                    None ⇒ None |
                    Some (-, str5) ⇒ Some ((lits, clauses), str5))))))
```

Simple tests.

```
value problem-line ''p cnf 123 321↩''
value problem-line ''p   cnf   123 321↩''
value problem-line ''p cnf 123 −321↩''
value problem-line ''  p cnf 123 321↩''

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


fun consume-text :: unit parser where
  consume-text Nil = Some ((), Nil) |
  consume-text (Cons x xs) = (if x = CHR ''↩'' then Some ((), Cons x xs) else
consume-text xs)
```

**lemma** *non-increasing-consume-text* [*simp*]: *consume-text s* = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $\leq$ *length s*
  **apply** (*induction s rule*: *consume-text.induct*)
   **apply** *simp*
  **by** (*metis* (*mono-tags*, *lifting*) *consume-text.simps*(*2*) *le-imp-less-Suc length-Cons nle-le option.sel order-less-imp-le prod.sel*(*2*))

**fun** *comment-line* :: *unit parser* **where**
  *comment-line str* = (*case match* ''*c*'' (*trim-ws str*) *of*
    *None* ⇒ *None* |
    *Some* (-, *str'*) ⇒
      (*case consume-text str'* *of*
        *None* ⇒ *None* |
        *Some* (-, *str''*) ⇒
          (*case match* ''$\boxed{\hookleftarrow}$'' *str''* *of*
            *None* ⇒ *None* |
            *Some* (-, *str'''*) ⇒ *Some* ((), *str'''*))))

Simple tests.

**value** *comment-line* ''*c e 1 2 3*$\boxed{\hookleftarrow}$*e 1 2 3*''
**value** *comment-line* ''*e 1 2 3*$\boxed{\hookleftarrow}$*e 1 2 3*''
**value** *comment-line* '' *c  e  1  2   3*  $\boxed{\hookleftarrow}$*e 1 2 3*''

**lemma** *decreasing-comment-line* [*simp*]:
  **assumes** *comment-line s* = *Some* ((), *s'*)
  **shows** *length s'* < *length s*
**proof** (*cases match* ''*c*'' (*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* < *length s*
    **using** *Some-a decreasing-match*[*of* ''*c*'' *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* $\leq$ *length s1* **using** *Some-b* **by** *simp*
    **show** *?thesis*
    **proof** (*cases match* ''$\boxed{\hookleftarrow}$'' *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 = 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 ′′↩′′ s2 = Some (u, s′) using Some-c c-def by simp
      hence length s′ ≤ length s2 by simp
      thus length s′ < length s using less1 le1 by simp
    qed
  qed
qed
```

```
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′′)))
```

Simple tests.

```
value comment-lines ′′c a comment↩c another comment↩′′
value comment-lines ′′c a comment↩  c another comment↩′′
```

```
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′*)

Simple tests.

**value** *preamble* ″*c an example*$\boxed{\hookleftarrow}$*p cnf 4 5*$\boxed{\hookleftarrow}$″
**value** *preamble* ″ *c an example*$\boxed{\hookleftarrow}$ *p cnf 4 5*$\boxed{\hookleftarrow}$″

**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* (-, *str*′′′′) ⇒ *Some* ((*pre*, *mat*), *str*′′′′)))))

Simple tests.

**value** *input*
*′′c an example from the QDIMACS specification*
*c multiple*
*c lines*
*cwith*
*c comments*
*p cnf 4 2*
*e 1 2 3 4 0*
*−1  2 0*
*2 −3 −4 0*
*′′*

**value** *input*
*′′c an extension of the example from the QDIMACS specification*
*c multiple*
*c lines*
*cwith*
*c comments*
*p cnf 40 4*
*e 1 2 3 4 0*
*a 11 12 13 14 0*
*e 21 22 23 24 0*
*−1  2 0*
*2 −3 −4 0*
*40 −13 −24 0*
*12 −23 −24 0*
*′′*

**lemma** *input-nil* [*simp*]:
  **assumes** *input s* = *Some* (*p*, *s*′)
  **shows** *s*′ = *Nil* **using** *assms*
**proof** (*cases preamble s*)
  **case** *None*
  **hence** *False* **using** *assms* **by** *simp*
  **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)
      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))
```

Simple tests.

```
value parse (String.implode
''c an example from the QDIMACS specification
c multiple
c lines
cwith
c comments
p cnf 4 2
e 1 2 3 4 0
−1  2 0
```

*2 −3 −4 0*
*′′*)

**value** *parse* (*String.implode*
*′′c an extension of the example from the QDIMACS specification*
*c multiple*
*c lines*
*cwith*
*c comments*
*p cnf 40 4*
*e 1 2 3 4 0*
*a 11 12 13 14 0*
*e 21 22 23 24 0*
*−1  2 0*
*2 −3 −4 0*
*40 −13 −24 0*
*12 −23 −24 0*
*′′*)

**end**

# 4 Search-Based Solver Implementation and Verification

**theory** *SearchSolver*
  **imports** *PCNF*
**begin**

## 4.1 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* (λ*l*. *l* ≠ *lit-neg lit*) *clause*

**fun** *remove-lit-clauses* :: *literal* ⇒ *matrix* ⇒ *matrix* **where**
  *remove-lit-clauses lit matrix* = *filter* (λ*cl*. ¬(*list-ex* (λ*l*. *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*) *qs*
| *add-universal-to-prefix x* (*ExistentialFirst* (*y*, *ys*) *qs*) = *UniversalFirst* (*x*, []) ((*y*, *ys*) # *qs*)

**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*) *qs*
| *add-existential-to-prefix x* (*UniversalFirst* (*y*, *ys*) *qs*) = *ExistentialFirst* (*x*, []) ((*y*, *ys*) # *qs*)

**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* = *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* ($\lambda$(*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*)

Simple tests.

**value** *the* (*convert-inv test-qbf*)
**value** *pcnf-assign* (*P 1*) (*the* (*convert-inv test-qbf*))
**value** *pcnf-assign* (*P 3*) (*the* (*convert-inv test-qbf*))

## 4.2 Effect of PCNF Assignments on the Set of all Free Variables

### 4.2.1 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*

$\Longrightarrow$ *free-assgn* $\subseteq$ *free* $-$ *lit*
**by** *auto*

### 4.2.2 Free Variables is Variables without Prefix Variables

**lemma** *lit-p-free-eq-vars*:
*literal-p qbf* $\Longrightarrow$ *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* $\Longrightarrow$ *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** (*3-3 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*
**qed** *auto*

**lemma** *pcnf-p-free-eq-vars-minus-prefix*:
*pcnf-p qbf* $\Longrightarrow$ *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
```

### 4.2.3 Set of Matrix Variables is Non-increasing under PCNF Assignments

```
lemma lit-not-in-matrix-assign-variables:
  lit-var lit ∉ set (variables (convert-matrix (matrix-assign lit matrix)))
proof (induction matrix)
  case (Cons cl cls)
  then show ?case
  proof (induction cl)
    case (Cons l ls)
    then show ?case
    proof (induction l)
      case (P 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
    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
  qed auto
qed auto

lemma matrix-assign-vars-subseteq-matrix-vars-minus-lit:
  set (variables (convert-matrix (matrix-assign lit matrix)))
  ⊆ 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*))
⊆ *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*

### 4.2.4 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** (*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** *auto*

**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*

### 4.2.5 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*)) ⊆ *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*]] **.**

## 4.3 PCNF Existential Closure

### 4.3.1 Formalization of PCNF Existential Closure

**fun** *pcnf-existential-closure* :: *pcnf* ⇒ *pcnf* **where**
*pcnf-existential-closure pcnf* = *the* (*convert-inv* (*existential-closure* (*convert pcnf*)))

### 4.3.2 PCNF Existential Closure Preserves Satisfiability

**lemma** *ex-closure-aux-pcnf-p-inv*:
*pcnf-p qbf* ⟹ *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* ⟹ *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*

### 4.3.3 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*

## 4.4 Search Solver (Part 1: Preliminaries)

### 4.4.1 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** (*Cons l ls*)
  **then show** *?case* **by** (*induction l*) *auto*
**qed** *auto*

**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** (*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** *auto*

**lemma** *empty-clause-or-matrix-if-no-variables*:
*pcnf-variables* (*Empty*, *matrix*) = [] $\Longrightarrow$ [] $\in$ *set matrix* $\vee$ *matrix* = []
**using** *matrix-shape-if-no-variables* **by** *fastforce*

### 4.4.2 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** (*Cons cl cls*)
**then show** *?case*
**proof** (*cases* $P$ $z$ $\in$ *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*))
$\longleftrightarrow$ *clause-semantics* ($I$($z$ := *True*)) (*remove-lit-neg* ($P$ $z$) *cl*)
$\wedge$ *matrix-semantics* ($I$($z$ := *True*)) (*matrix-assign* ($P$ $z$) *cls*) **by** *simp*
**moreover have** *matrix-semantics* ($I$($z$ := *True*)) (*cl* # *cls*)
$\longleftrightarrow$ *clause-semantics* ($I$($z$ := *True*)) *cl*
$\wedge$ *matrix-semantics* ($I$($z$ := *True*)) *cls* **by** *simp*
**ultimately show** *?thesis* **using** *Cons clause-semantics-inv-remove-false* **by** *blast*
**qed**
**qed** *auto*

**lemma** *matrix-semantics-inv-remove-true'*:
  **assumes** $y \neq z$
  **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** (*Cons cl cls*)
  **then show** *?case*
  **proof** (*cases* $N\ z \in$ *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-semantics* ($I(z := False)$) (*matrix-assign* ($N\ z$) (*cl* # *cls*))
        $\longleftrightarrow$ *clause-semantics* ($I(z := False)$) (*remove-lit-neg* ($N\ z$) *cl*)
          $\wedge$ *matrix-semantics* ($I(z := False)$) (*matrix-assign* ($N\ z$) *cls*) **by** *simp*
    **moreover have** *matrix-semantics* ($I(z := False)$) (*cl* # *cls*)
                $\longleftrightarrow$ *clause-semantics* ($I(z := False)$) *cl*
                  $\wedge$ *matrix-semantics* ($I(z := False)$) *cls* **by** *simp*
    **ultimately show** *?thesis* **using** *Cons clause-semantics-inv-remove-true* **by** *blast*
  **qed**
**qed** *auto*

**lemma** *matrix-semantics-inv-remove-false'*:
  **assumes** $y \neq z$
  **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-false fun-upd-twist* **by** *metis*

**lemma** *matrix-semantics-disj-iff-true-assgn*:
  ($\exists\, b.$ *matrix-semantics* ($I(z := b)$) *matrix*)
  $\longleftrightarrow$ *matrix-semantics* ($I(z := True)$) (*matrix-assign* ($P\ z$) *matrix*)

$\vee$ *matrix-semantics* $(I(z := \mathit{False}))$ (*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-assgn*:
$(\forall b.$ *matrix-semantics* $(I(z := b))$ *matrix*)
$\longleftrightarrow$ *matrix-semantics* $(I(z := \mathit{True}))$ (*matrix-assign* $(P\ z)$ *matrix*)
$\wedge$ *matrix-semantics* $(I(z := \mathit{False}))$ (*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-assgn'*:
**assumes** *lit-var lit* $\notin$ *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-assgn*:
**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** *neq-first-if-notin-all-prefix*:
$z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst* $(y, ys)$ *qs*, *matrix*)) $\Longrightarrow z \neq y$
**by** (*induction* (*UniversalFirst* $(y, ys)$ *qs*, *matrix*) *rule*: *convert.induct*) *auto*

**lemma** *neq-first-if-notin-ex-prefix*:
$z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* $(x, xs)$ *qs*, *matrix*)) $\Longrightarrow z \neq x$
**by** (*induction* (*ExistentialFirst* $(x, xs)$ *qs*, *matrix*) *rule*: *convert.induct*) *auto*

**lemma** *notin-pop-prefix-if-notin-prefix*:
**assumes** $z \notin$ *set* (*pcnf-prefix-variables* (*prefix*, *matrix*))
**shows** $z \notin$ *set* (*pcnf-prefix-variables* (*prefix-pop prefix*, *matrix*))
**using** *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
qed auto


lemma pcnf-semantics-inv-matrix-assign-true:
  assumes z ∉ 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)
  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 neq: z ≠ 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 ≠ 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-pop-prefix-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 ys qs matrix)
  hence neq: z ≠ y using neq-first-if-notin-all-prefix by blast
  have z ∉ set (pcnf-prefix-variables (UniversalFirst (yy, ys) qs, matrix))
    using 6(3) notin-pop-prefix-if-notin-prefix by fastforce
  hence pcnf-semantics (I(z := True)) (UniversalFirst (yy, ys) qs, matrix) =
```

*pcnf-semantics* (*I*(*z* := *True*)) (*UniversalFirst* (*yy*, *ys*) *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*: $z \neq x$ **using** *neq-first-if-notin-ex-prefix* **by** *blast*
**have** $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*))
**using** *7*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**hence** *pcnf-semantics* (*I*(*z* := *True*)) (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*) =
*pcnf-semantics* (*I*(*z* := *True*)) (*ExistentialFirst* (*xx*, *xs*) *qs*, *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** $z \notin$ *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** *?case* **using** *matrix-semantics-inv-remove-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* # *qs*)) = *ExistentialFirst q qs*
**by** (*induction q*) *auto*
**hence** $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst q qs*, *matrix*))
**using** *4*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *metis*
**hence** *pcnf-semantics* (*I*(*z* := *False*)) (*ExistentialFirst q qs*, *matrix*) =
*pcnf-semantics* (*I*(*z* := *False*)) (*ExistentialFirst q qs*, *matrix-assign* (*N 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*: $z \neq x$ **using** *neq-first-if-notin-ex-prefix* **by** *blast*
**have** *prefix-pop* (*ExistentialFirst* (*x*, []) (*q* # *qs*)) = *UniversalFirst q qs*
**by** (*induction q*) *auto*
**hence** $z \notin$ *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** *?case* **using** *neq* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*6 I y yy ys qs matrix*)
**hence** *neq*: $z \neq y$ **using** *neq-first-if-notin-all-prefix* **by** *blast*
**have** $z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*))
**using** *6*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**hence** *pcnf-semantics* (*I*(*z* := *False*)) (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*) =
*pcnf-semantics* (*I*(*z* := *False*)) (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix-assign* (*N 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*: $z \neq x$ **using** *neq-first-if-notin-ex-prefix* **by** *blast*
**have** $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*))
**using** *7*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**hence** *pcnf-semantics* (*I*(*z* := *False*)) (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*) =
*pcnf-semantics* (*I*(*z* := *False*)) (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix-assign* (*N*
*z*) *matrix*)
**for** *I* **using** *7* **by** *blast*
**then show** *?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*)
$\vee$ *pcnf-semantics* (*I*(*z* := *False*)) (*prefix*, *matrix*)
$\longleftrightarrow$
*pcnf-semantics* (*I*(*z* := *True*)) (*prefix*, *matrix-assign* (*P z*) *matrix*)
$\vee$ *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** (*1 I*)
**then show** *?case* **using** *ex-bool-eq matrix-semantics-disj-iff-true-assgn* **by** *simp*
**next**
**case** (*2 I y*)
**hence** *neq*: $y \neq z$ **by** *simp*
**show** *?case* **using** *ex-bool-eq matrix-semantics-inv-remove-true′*
*matrix-semantics-inv-remove-false′ neq* **by** *simp*
**next**
**case** (*3 I x*)
**hence** *neq*: $x \neq z$ **by** *simp*
**show** *?case* **using** *ex-bool-eq matrix-semantics-inv-remove-true′*
*matrix-semantics-inv-remove-false′ neq* **by** *simp*
**next**

**case** (*4 I y q qs*)
**hence** *neq*: $y \neq z$ **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*: $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst q qs*, *matrix*))
**using** *4*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *metis*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*5 I x q qs*)
**hence** *neq*: $x \neq z$ **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*: $z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst q qs*, *matrix*))
**using** *5*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *metis*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*6 I y yy ys qs*)
**hence** *neq*: $y \neq z$ **using** *neq-first-if-notin-all-prefix* **by** *blast*
**have** *nin*: $z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*))
**using** *6*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*7 I x xx xs qs*)
**hence** *neq*: $x \neq z$ **using** *neq-first-if-notin-ex-prefix* **by** *blast*
**have** *nin*: $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*))
**using** *7*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**qed**

**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*)
$\wedge$ *pcnf-semantics* (*I*(*z* := *False*)) (*prefix*, *matrix*)
$\longleftrightarrow$
*pcnf-semantics* (*I*(*z* := *True*)) (*prefix*, *matrix-assign* (*P z*) *matrix*)
$\wedge$ *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** (*1 I*)
**then show** *?case* **using** *all-bool-eq matrix-semantics-conj-iff-true-assgn* **by** *simp*
**next**
**case** (*2 I y*)
**hence** *neq*: $y \neq z$ **by** *simp*

**show** *?case* **using** *matrix-semantics-inv-remove-true′*
*matrix-semantics-inv-remove-false′ neq* **by** *simp*
**next**
**case** (*3 I x*)
**hence** *neq*: $x \neq z$ **by** *simp*
**show** *?case* **using** *matrix-semantics-inv-remove-true′*
*matrix-semantics-inv-remove-false′ neq* **by** *simp*
**next**
**case** (*4 I y q qs*)
**hence** *neq*: $y \neq z$ **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*: $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst q qs*, *matrix*))
**using** *4*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *metis*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*5 I x q qs*)
**hence** *neq*: $x \neq z$ **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*: $z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst q qs*, *matrix*))
**using** *5*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *metis*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*6 I y yy ys qs*)
**hence** *neq*: $y \neq z$ **using** *neq-first-if-notin-all-prefix* **by** *blast*
**have** *nin*: $z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst* (*yy*, *ys*) *qs*, *matrix*))
**using** *6*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**next**
**case** (*7 I x xx xs qs*)
**hence** *neq*: $x \neq z$ **using** *neq-first-if-notin-ex-prefix* **by** *blast*
**have** *nin*: $z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*xx*, *xs*) *qs*, *matrix*))
**using** *7*(*3*) *notin-pop-prefix-if-notin-prefix* **by** *fastforce*
**show** *?case* **using** *nin neq pcnf-semantics-inv-matrix-assign-true*
*pcnf-semantics-inv-matrix-assign-false* **by** (*simp add*: *fun-upd-twist*)
**qed**

**lemma** *semantics-eq-if-free-vars-eq*:
**assumes** $\forall x \in$ *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** (*3 I qbf-list*)
**then show** *?case* **by** (*induction qbf-list*) *auto*
**next**

**case** (*4 I qbf-list*)
**then show** *?case* **by** (*induction qbf-list*) *auto*
**next**
**case** (*5 I x qbf*)
**hence** *qbf-semantics I* (*substitute-var x b qbf*)
= *qbf-semantics J* (*substitute-var x b qbf*)
**for** *b* **using** *set-free-vars-subst-ex-eq* **by** (*metis* (*full-types*))
**then show** *?case* **by** *simp*
**next**
**case** (*6 I x qbf*)
**hence** *qbf-semantics I* (*substitute-var x b qbf*)
= *qbf-semantics J* (*substitute-var x b qbf*)
**for** *b* **using** *set-free-vars-subst-all-eq* **by** (*metis* (*full-types*))
**then show** *?case* **by** *simp*
**qed** *auto*

**lemma** *pcnf-semantics-eq-if-free-vars-eq*:
**assumes** $\forall x \in$ *set* (*pcnf-free-variables pcnf*). $I(x) = J(x)$
**shows** *pcnf-semantics I pcnf* = *pcnf-semantics J pcnf*
**using** *assms semantics-eq-if-free-vars-eq qbf-semantics-eq-pcnf-semantics* **by** *simp*

**lemma** *x-notin-assign-P-x*:
$x \notin$ *set* (*pcnf-variables* (*pcnf-assign* (*P x*) *pcnf*))
**using** *pcnf-assign-vars-subseteq-vars-minus-lit* **by** *fastforce*

**lemma** *x-notin-assign-N-x*:
$x \notin$ *set* (*pcnf-variables* (*pcnf-assign* (*N x*) *pcnf*))
**using** *pcnf-assign-vars-subseteq-vars-minus-lit* **by** *fastforce*

**lemma** *interp-value-ignored-for-pcnf-P-assign*:
*pcnf-semantics* ($I(x := b)$) (*pcnf-assign* (*P x*) *pcnf*)
= *pcnf-semantics I* (*pcnf-assign* (*P x*) *pcnf*)
**using** *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*:
*pcnf-semantics* ($I(x := b)$) (*pcnf-assign* (*N x*) *pcnf*)
= *pcnf-semantics I* (*pcnf-assign* (*N x*) *pcnf*)
**using** *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** $x \notin$ *set* (*pcnf-prefix-variables* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*))
**shows** *satisfiable* (*convert* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))
$\longleftrightarrow$ *satisfiable* (*convert* (*pcnf-assign* (*P x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)))
$\vee$ *satisfiable* (*convert* (*pcnf-assign* (*N x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)))

**proof** −
**let** *?pre* = *ExistentialFirst* (*x*, *xs*) *qs*
**have** *satisfiable* (*convert* (*?pre*, *matrix*))
= (∃ *I*. *pcnf-semantics I* (*?pre*, *matrix*))
**using** *satisfiable-def qbf-semantics-eq-pcnf-semantics* **by** *simp*
**also have** ... =
(∃ *I*. *pcnf-semantics* (*I*(*x* := *True*)) (*prefix-pop ?pre*, *matrix*) ∨
*pcnf-semantics* (*I*(*x* := *False*)) (*prefix-pop ?pre*, *matrix*))
**by** (*induction ?pre rule*: *prefix-pop.induct*) *auto*
**also have** ... =
(∃ *I*. *pcnf-semantics* (*I*(*x* := *True*)) (*prefix-pop ?pre*, *matrix-assign* (*P x*) *matrix*)
∨
*pcnf-semantics* (*I*(*x* := *False*)) (*prefix-pop ?pre*, *matrix-assign* (*N x*) *matrix*))
**using** *pcnf-semantics-disj-iff-matrix-assign-disj assms* **by** *blast*
**also have** ... ⟷
(∃ *I*. *pcnf-semantics* (*I*(*x* := *True*)) (*pcnf-assign* (*P x*) (*?pre*, *matrix*))) ∨
(∃ *I*. *pcnf-semantics* (*I*(*x* := *False*)) (*pcnf-assign* (*N x*) (*?pre*, *matrix*)))
**using** *pcnf-assign-free-eq-matrix-assgn*[*of P x*] *pcnf-assign-free-eq-matrix-assgn*[*of*
*N x*]
*assms* **by** *auto*
**also have** ... ⟷
(∃ *I*. *pcnf-semantics I* (*pcnf-assign* (*P x*) (*?pre*, *matrix*))) ∨
(∃ *I*. *pcnf-semantics I* (*pcnf-assign* (*N x*) (*?pre*, *matrix*)))
**using** *interp-value-ignored-for-pcnf-N-assign interp-value-ignored-for-pcnf-P-assign*
**by** *blast*
**also have** ... ⟷
*satisfiable* (*convert* (*pcnf-assign* (*P x*) (*?pre*, *matrix*))) ∨
*satisfiable* (*convert* (*pcnf-assign* (*N x*) (*?pre*, *matrix*)))
**using** *satisfiable-def qbf-semantics-eq-pcnf-semantics* **by** *simp*
**finally show** *?thesis* **.**
**qed**

**theorem** *sat-ex-first-iff-assign-disj-sat*:
**assumes** *x* ∉ *set* (*pcnf-prefix-variables* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*),
*matrix*))
**shows** *satisfiable* (*convert* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))
⟷ *satisfiable* (*Disj*
[*convert* (*pcnf-assign* (*P x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)),
*convert* (*pcnf-assign* (*N x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))])
**using** *assms sat-ex-first-iff-one-assign-sat satisfiable-def*
*qbf-semantics-eq-pcnf-semantics* **by** *auto*

**theorem** *sat-all-first-iff-assign-conj-sat*:
**assumes** *y* ∉ *set* (*pcnf-prefix-variables* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*),
*matrix*))
**shows** *satisfiable* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))
⟷ *satisfiable* (*Conj*

[*convert* (*pcnf-assign* (*P y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)),
*convert* (*pcnf-assign* (*N y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))])
**proof** −
**let** *?pre* = *UniversalFirst* (*y*, *ys*) *qs*
**have** *satisfiable* (*convert* (*?pre*, *matrix*))
= (∃ *I*. *pcnf-semantics I* (*?pre*, *matrix*))
**using** *satisfiable-def qbf-semantics-eq-pcnf-semantics* **by** *simp*
**also have** ... =
(∃ *I*. *pcnf-semantics* (*I*(*y* := *True*)) (*prefix-pop ?pre*, *matrix*) ∧
*pcnf-semantics* (*I*(*y* := *False*)) (*prefix-pop ?pre*, *matrix*))
**by** (*induction ?pre rule*: *prefix-pop.induct*) *auto*
**also have** ... =
(∃ *I*. *pcnf-semantics* (*I*(*y* := *True*)) (*prefix-pop ?pre*, *matrix-assign* (*P y*) *matrix*)
∧
*pcnf-semantics* (*I*(*y* := *False*)) (*prefix-pop ?pre*, *matrix-assign* (*N y*) *matrix*))
**using** *pcnf-semantics-conj-iff-matrix-assign-conj assms* **by** *blast*
**also have** ... =
(∃ *I*. *pcnf-semantics* (*I*(*y* := *True*)) (*pcnf-assign* (*P y*) (*?pre*, *matrix*)) ∧
*pcnf-semantics* (*I*(*y* := *False*)) (*pcnf-assign* (*N y*) (*?pre*, *matrix*)))
**using** *pcnf-assign-free-eq-matrix-assgn*[*of P y*] *pcnf-assign-free-eq-matrix-assgn*[*of*
*N y*]
*assms* **by** *simp*
**also have** ... =
(∃ *I*. *pcnf-semantics I* (*pcnf-assign* (*P y*) (*?pre*, *matrix*)) ∧
*pcnf-semantics I* (*pcnf-assign* (*N y*) (*?pre*, *matrix*)))
**using** *interp-value-ignored-for-pcnf-N-assign interp-value-ignored-for-pcnf-P-assign*
**by** *blast*
**also have** ... =
(∃ *I*. *qbf-semantics I* (*convert* (*pcnf-assign* (*P y*) (*?pre*, *matrix*))) ∧
*qbf-semantics I* (*convert* (*pcnf-assign* (*N y*) (*?pre*, *matrix*))))
**using** *qbf-semantics-eq-pcnf-semantics* **by** *blast*
**also have** ... =
*satisfiable* (*Conj*
[*convert* (*pcnf-assign* (*P y*) (*?pre*, *matrix*)),
*convert* (*pcnf-assign* (*N y*) (*?pre*, *matrix*))])
**unfolding** *satisfiable-def* **by** *simp*
**finally show** *?thesis* **.**
**qed**

## 4.5 Cleansed PCNF Formulas

### 4.5.1 Predicate for Cleansed Formulas

**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** *prefix-variables-aux-matrix-inv*:
  *prefix-variables-aux* (*convert* (*prefix*, *matrix1*))
  = *prefix-variables-aux* (*convert* (*prefix*, *matrix2*))
  **by** (*induction* (*prefix*, *matrix1*) *arbitrary*: *prefix rule*: *convert.induct*) *auto*

**lemma** *eq-prefix-cleansed-p-add-all-inv*:
  *cleansed-p* (*add-universal-to-front y* (*prefix*, *matrix1*))
  = *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** *?case* **by** *simp*
**next**
  **case** (*2 x y ys qs*)
  **have** *prefix-variables-aux* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix1*))
      = *prefix-variables-aux* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix2*))
    **using** *prefix-variables-aux-matrix-inv* **by** *simp*
  **then show** *?case* **by** *simp*
**next**
  **case** (*3 x y ys qs*)
    **have** *prefix-variables-aux* (*convert* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix1*))
      = *prefix-variables-aux* (*convert* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix2*))
    **using** *prefix-variables-aux-matrix-inv* **by** *simp*
  **then show** *?case* **by** *simp*
**qed**

**lemma** *eq-prefix-cleansed-p-add-ex-inv*:
  *cleansed-p* (*add-existential-to-front x* (*prefix*, *matrix1*))
  = *cleansed-p* (*add-existential-to-front x* (*prefix*, *matrix2*))
**proof** (*induction x* (*prefix*, *matrix1*) *arbitrary*: *prefix rule*: *add-universal-to-front.induct*)
  **case** (*1 x*)
  **then show** *?case* **by** *simp*
**next**
  **case** (*2 x y ys qs*)
  **have** *prefix-variables-aux* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix1*))
      = *prefix-variables-aux* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix2*))
    **using** *prefix-variables-aux-matrix-inv* **by** *simp*
  **then show** *?case* **by** *simp*
**next**
  **case** (*3 x y ys qs*)
    **have** *prefix-variables-aux* (*convert* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix1*))
      = *prefix-variables-aux* (*convert* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix2*))
    **using** *prefix-variables-aux-matrix-inv* **by** *simp*
  **then show** *?case* **by** *simp*
**qed**

**lemma** *cleansed-p-matrix-inv*:
  *cleansed-p* (*prefix*, *matrix1*) = *cleansed-p* (*prefix*, *matrix2*)
**proof** (*induction* (*prefix*, *matrix1*) *arbitrary*: *prefix rule*: *convert.induct*)
  **case** (*4 x q qs*)

**have** (*UniversalFirst* (*x*, []) (*q* # *qs*), *matrix*)
= *add-universal-to-front x* (*ExistentialFirst q qs*, *matrix*)
**for** *matrix* **by** (*induction q*) *auto*
**then show** *?case* **using** *eq-prefix-cleansed-p-add-all-inv* **by** *simp*
**next**
**case** (*5 x q qs*)
**have** (*ExistentialFirst* (*x*, []) (*q* # *qs*), *matrix*)
= *add-existential-to-front x* (*UniversalFirst q qs*, *matrix*)
**for** *matrix* **by** (*induction q*) *auto*
**then show** *?case* **using** *eq-prefix-cleansed-p-add-ex-inv* **by** *simp*
**next**
**case** (*6 x y ys qs*)
**have** (*UniversalFirst* (*x*, *y* # *ys*) *qs*, *matrix*)
= *add-universal-to-front x* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
**for** *matrix* **by** *simp*
**then show** *?case* **using** *eq-prefix-cleansed-p-add-all-inv* **by** *metis*
**next**
**case** (*7 x y ys qs*)
**have** (*ExistentialFirst* (*x*, *y* # *ys*) *qs*, *matrix*)
= *add-existential-to-front x* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix*)
**for** *matrix* **by** *simp*
**then show** *?case* **using** *eq-prefix-cleansed-p-add-ex-inv* **by** *metis*
**qed** *auto*

**lemma** *cleansed-prefix-first-ex-unique*:
**assumes** *cleansed-p* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)
**shows** *x* ∉ *set* (*pcnf-prefix-variables* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*))
**using** *assms* **by** (*induction ExistentialFirst* (*x*, *xs*) *qs rule*: *prefix-pop.induct*) *auto*

**lemma** *cleansed-prefix-first-all-unique*:
**assumes** *cleansed-p* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
**shows** *y* ∉ *set* (*pcnf-prefix-variables* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*))
**using** *assms* **by** (*induction UniversalFirst* (*y*, *ys*) *qs rule*: *prefix-pop.induct*) *auto*

### 4.5.2 The Cleansed Predicate is Invariant under PCNF Assignment

**lemma** *cleansed-add-new-ex-to-front*:
**assumes** *cleansed-p pcnf*
**and** *x* ∉ *set* (*pcnf-prefix-variables pcnf*)
**shows** *cleansed-p* (*add-existential-to-front x pcnf*)
**using** *assms* **by** (*induction x pcnf rule*: *add-existential-to-front.induct*) *auto*

**lemma** *cleansed-add-new-all-to-front*:
**assumes** *cleansed-p pcnf*
**and** *y* ∉ *set* (*pcnf-prefix-variables pcnf*)

**shows** *cleansed-p* (*add-universal-to-front y pcnf*)
**using** *assms* **by** (*induction y pcnf rule*: *add-existential-to-front.induct*) *auto*

**lemma** *pcnf-assign-p-ex-eq*:
**assumes** *cleansed-p* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)
**shows** *pcnf-assign* (*P x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)
= (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix-assign* (*P x*) *matrix*)
**using** *assms* **by** (*metis cleansed-prefix-first-ex-unique lit-var.simps*(*1*)
*pcnf-assign.simps pcnf-assign-free-eq-matrix-assgn remove-var-prefix.simps*(*3*))

**lemma** *pcnf-assign-p-all-eq*:
**assumes** *cleansed-p* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
**shows** *pcnf-assign* (*P y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
= (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix-assign* (*P y*) *matrix*)
**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*(*3*))

**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* $\Longrightarrow$ *cleansed-p* (*pcnf-assign lit pcnf*)
**proof** (*induction pcnf rule*: *convert.induct*)
**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** $x \notin$ *set* (*pcnf-prefix-variables* (*remove-var-prefix ?z* (*ExistentialFirst q qs*), *?mat*))

```
    using 4 False prefix-assign-vars-eq-prefix-vars-minus-lit[of ?z] prefix-vars-matrix-inv
      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 x q qs matrix)
  let ?z = lit-var lit
  show ?case
  proof (cases x = ?z)
    case True
    then show ?thesis using 5 cleansed-p-matrix-inv
        pcnf-assign-n-ex-eq[of ?z] pcnf-assign-p-ex-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 (UniversalFirst q qs), ?mat)
      using 5 by simp
    moreover have x ∉ set (pcnf-prefix-variables (remove-var-prefix ?z (UniversalFirst
q qs), ?mat))
    using 5 False prefix-assign-vars-eq-prefix-vars-minus-lit[of ?z] prefix-vars-matrix-inv
      by fastforce
    ultimately have cleansed-p (add-existential-to-prefix x (remove-var-prefix ?z
(UniversalFirst q qs)), ?mat)
      using cleansed-add-new-ex-to-front add-ex-to-prefix-eq-add-to-front by simp
    then have cleansed-p (remove-var-prefix ?z (ExistentialFirst (x, []) (q # qs)),
?mat)
      using False by (induction q) auto
    then show ?thesis by simp
  qed
next
  case (6 x y ys qs matrix)
  let ?z = lit-var lit
  show ?case
  proof (cases x = ?z)
    case True
    then show ?thesis using 6 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 (UniversalFirst (y, ys) qs), ?mat)
```

**using** *6* **by** *simp*
**moreover have** $x \notin$ *set* (*pcnf-prefix-variables* (*remove-var-prefix ?z* (*UniversalFirst* (*y*, *ys*) *qs*), *?mat*))
**using** *6*(*2*) *False prefix-assign-vars-eq-prefix-vars-minus-lit*[*of ?z*] *prefix-vars-matrix-inv*
**by** *fastforce*
**ultimately have** *cleansed-p* (*add-universal-to-prefix x* (*remove-var-prefix ?z* (*UniversalFirst* (*y*, *ys*) *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*, (*y* # *ys*)) *qs*), *?mat*)
**using** *False* **by** *simp*
**then show** *?thesis* **by** *simp*
**qed**
**next**
**case** (*7 x y ys qs matrix*)
**let** *?z* = *lit-var lit*
**show** *?case*
**proof** (*cases x* = *?z*)
**case** *True*
**then show** *?thesis* **using** *7 cleansed-p-matrix-inv*
*pcnf-assign-n-ex-eq*[*of ?z*] *pcnf-assign-p-ex-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* (*y*, *ys*) *qs*), *?mat*)
**using** *7* **by** *simp*
**moreover have** $x \notin$ *set* (*pcnf-prefix-variables* (*remove-var-prefix ?z* (*ExistentialFirst* (*y*, *ys*) *qs*), *?mat*))
**using** *7*(*2*) *False prefix-assign-vars-eq-prefix-vars-minus-lit*[*of ?z*] *prefix-vars-matrix-inv*
**by** *fastforce*
**ultimately have** *cleansed-p* (*add-existential-to-prefix x* (*remove-var-prefix ?z* (*ExistentialFirst* (*y*, *ys*) *qs*)), *?mat*)
**using** *cleansed-add-new-ex-to-front add-ex-to-prefix-eq-add-to-front* **by** *simp*
**then have** *cleansed-p* (*remove-var-prefix ?z* (*ExistentialFirst* (*x*, (*y* # *ys*)) *qs*), *?mat*)
**using** *False* **by** *simp*
**then show** *?thesis* **by** *simp*
**qed**
**qed** *auto*

### 4.5.3 Cleansing PCNF Formulas

**function** *pcnf-cleanse* :: *pcnf* ⇒ *pcnf* **where**
*pcnf-cleanse* (*Empty*, *matrix*) = (*Empty*, *matrix*)
| *pcnf-cleanse* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*) =
(*if* $y \in$ *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* $x \in$ *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* ($\lambda$(*pre*, *mat*). *prefix-measure pre*))
  (*auto simp add*: *prefix-pop-decreases-measure simp del*: *prefix-measure.simps*)

Simple tests.

**value** *pcnf-cleanse* (*UniversalFirst* (*0*, [*0*]) [(*0*, [*1*, *2*, *0*, *1*])], [])

### 4.5.4 Cleansing Yields a Cleansed Formula

**lemma** *prefix-pop-all-prefix-vars-set*:
 *set* (*pcnf-prefix-variables* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))
 = {*y*} ∪ *set* (*pcnf-prefix-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-prefix-vars-set*:
 *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))
 = {*x*} ∪ *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*

### 4.5.5 Cleansing Preserves the Set of Free Variables

**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*

### 4.5.6 Cleansing Preserves Semantics

**lemma** *pop-redundant-ex-prefix-var-semantics-eq*:
**assumes** $x \in$ *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*, *xs*) *qs*), *matrix*)
**have** *set* (*pcnf-prefix-variables ?pcnf*) = *set* (*pcnf-prefix-variables ?pop*)
**using** *assms prefix-pop-ex-prefix-vars-set* **by** *auto*
**hence** $x \notin$ *set* (*pcnf-free-variables ?pop*)
**using** *assms pcnf-free-eq-vars-minus-prefix* **by** *simp*
**hence** *0*: $\forall z \in$ *set* (*pcnf-free-variables ?pop*). $(I(x := b))\ z = I\ z$
**for** *b* **by** *simp*
**moreover have** *pcnf-semantics I ?pcnf*
$\longleftrightarrow$ *pcnf-semantics* ($I(x :=$ *True*)) *?pop*
$\vee$ *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** $y \in$ *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 prefix-pop-all-prefix-vars-set* **by** *auto*
**hence** $y \notin$ *set* (*pcnf-free-variables ?pop*)
**using** *assms pcnf-free-eq-vars-minus-prefix* **by** *simp*
**hence** *0*: $\forall z \in$ *set* (*pcnf-free-variables ?pop*). (*I*(*y* := *b*)) *z* = *I z*
**for** *b* **by** *simp*
**moreover have** *pcnf-semantics I ?pcnf*
$\longleftrightarrow$ *pcnf-semantics* (*I*(*y* := *True*)) *?pop*
$\wedge$ *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* $\vee$ *pcnf-semantics* (*I*(*y* := *False*)) *pcnf*
$\longleftrightarrow$ *pcnf-semantics I* (*add-existential-to-front y pcnf*)
**using** *convert-add-ex qbf-semantics-eq-pcnf-semantics qbf-semantics-substitute-eq-assign*
**by** *simp*

**lemma** *pcnf-semantics-conj-eq-add-all*:
*pcnf-semantics* (*I*(*y* := *True*)) *pcnf* $\wedge$ *pcnf-semantics* (*I*(*y* := *False*)) *pcnf*
$\longleftrightarrow$ *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-preserves-semantics*:
*pcnf-semantics I pcnf* = *pcnf-semantics I* (*pcnf-cleanse pcnf*)
**proof** (*induction pcnf arbitrary*: *I rule*: *pcnf-cleanse.induct*)
**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* $y \in$ *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*(*y* := *True*)) (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*)
∧ *pcnf-semantics* (*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*(*x* := *True*)) (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*)
∨ *pcnf-semantics* (*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** *auto*

**theorem** *sat-ex-first-iff-assign-disj-sat′*:
**assumes** *cleansed-p* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)
**shows** *satisfiable* (*convert* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))
⟷ *satisfiable* (*Disj*
[*convert* (*pcnf-assign* (*P x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)),
*convert* (*pcnf-assign* (*N x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))])
**using** *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* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
**shows** *satisfiable* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))
⟷ *satisfiable* (*Conj*
[*convert* (*pcnf-assign* (*P y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)),
*convert* (*pcnf-assign* (*N y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))])
**using** *assms cleansed-prefix-first-all-unique sat-all-first-iff-assign-conj-sat* **by** *auto*

## 4.6 Search Solver (Part 2: The Solver)

**lemma** *add-all-inc-prefix-measure*:
*prefix-measure* (*add-universal-to-prefix y prefix*) = *Suc* (*prefix-measure prefix*)
**by** (*induction y prefix rule*: *add-universal-to-prefix.induct*) *auto*

**lemma** *add-ex-inc-prefix-measure*:
*prefix-measure* (*add-existential-to-prefix x prefix*) = *Suc* (*prefix-measure prefix*)
**by** (*induction x prefix rule*: *add-universal-to-prefix.induct*) *auto*

**lemma** *remove-var-non-increasing-measure*:
*prefix-measure* (*remove-var-prefix z prefix*) ≤ *prefix-measure prefix*
**proof** (*induction z prefix rule*: *remove-var-prefix.induct*)
**case** (*2 x y ys qs*)
**hence** *0*: *prefix-measure* (*remove-var-prefix x* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*)))
≤ *prefix-measure* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*))
**by** (*cases x* = *y*) (*cases prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*) = *Empty*,*simp-all*)+
**show** *?case*
**proof** (*cases x* = *y*)
**case** *True*
**hence** *prefix-measure* (*remove-var-prefix x* (*UniversalFirst* (*y*, *ys*) *qs*))
= *prefix-measure* (*remove-var-prefix x* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*))) **by** *simp*
**also have** ... ≤ *prefix-measure* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*)) **using** *0* **by** *simp*
**also have** ... ≤ *prefix-measure* (*UniversalFirst* (*y*, *ys*) *qs*)
**using** *prefix-pop-decreases-measure less-imp-le-nat* **by** *blast*
**finally show** *?thesis* **.**
**next**
**case** *False*
**hence** *prefix-measure* (*remove-var-prefix x* (*UniversalFirst* (*y*, *ys*) *qs*))
= *prefix-measure* (*add-universal-to-prefix y*
(*remove-var-prefix x* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*)))) **by** *simp*
**also have** ... ≤ *Suc* (*prefix-measure* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*)))
**using** *0 add-all-inc-prefix-measure* **by** *simp*
**also have** ... ≤ *prefix-measure* (*UniversalFirst* (*y*, *ys*) *qs*)
**using** *Suc-leI prefix-pop-decreases-measure* **by** *blast*
**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** *auto*

**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*
**finally show** *?case* **.**
**qed**
**next**
**case** (*ExistentialFirst q qs*)
**then show** *?case*
**proof** (*induction q*)
**case** (*Pair x xs*)

```
    let ?pre = ExistentialFirst (x, xs) 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
    finally show ?case .
  qed
qed auto

fun first-existential :: prefix ⇒ bool option where
  first-existential (ExistentialFirst q qs) = Some True
| first-existential (UniversalFirst q qs) = Some False
| first-existential Empty = None

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) (λz.
      Option.bind (first-existential prefix) (λe. 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 (λ(pre, mat). prefix-measure pre))
      apply simp
     apply (metis first-existential.simps(3) in-measure lit-var.simps(1) option.sel
option.simps(3) pcnf-assign.simps prod.simps(2) remove-first-var-decreases-measure)
    apply (metis case-prod-conv first-existential.simps(3) in-measure lit-var.simps(2)
option.sel option.simps(3) pcnf-assign.simps remove-first-var-decreases-measure)
  apply (metis case-prod-conv first-existential.simps(3) in-measure lit-var.simps(1)
option.sel option.simps(3) pcnf-assign.simps remove-first-var-decreases-measure)
   by (metis case-prod-conv first-existential.simps(3) in-measure lit-var.simps(2)
option.sel option.simps(3) pcnf-assign.simps remove-first-var-decreases-measure)
```

Simple tests.

```
value search (UniversalFirst (1, []) [(2, [3])], [])
value search (UniversalFirst (1, []) [(2, [3])], [[]])
value search (UniversalFirst (1, []) [(2, [3])], [[P 1]])
value search (UniversalFirst (1, []) [(2, [3])], [[P 1, N 2]])
value search (UniversalFirst (1, []) [(2, [3])], [[P 1, N 2], [N 1, P 3]])

fun search-solver :: pcnf ⇒ bool where
```

*search-solver pcnf* = *the* (*search* (*pcnf-cleanse* (*pcnf-existential-closure pcnf*)))

Simple tests.

**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [[]])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [[*P 1*]])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [[*P 1*, *N 2*]])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [[*P 1*, *N 2*], [*N 1*, *P 3*]])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*])], [[*P 1*, *N 2*], [*N 1*, *P 3*], [*P 4*]])
**value** *search-solver* (*UniversalFirst* (*1*, []) [(*2*, [*3*, *3*, *3*])], [[*P 1*, *N 2*], [*N 1*, *P 3*], [*P 4*]])

### 4.6.1 Correctness of the Search Function

**lemma** *no-vars-if-no-free-no-prefix-vars*:
*pcnf-free-variables pcnf* = [] ⟹ *pcnf-prefix-variables pcnf* = [] ⟹ *pcnf-variables 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*:
*pcnf-free-variables* (*Empty*, *matrix*) = [] ⟹ *pcnf-variables* (*Empty*, *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** (∃ *b*. *search pcnf* = *Some b*) **using** *assms*
**proof** (*induction pcnf rule*: *search.induct*)
**case** (*1 prefix matrix*)
**then show** *?case*
**proof** (*cases* [] ∈ *set matrix*)
**case** *2*: *False*
**then show** *?thesis*
**proof** (*cases matrix* = [])
**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*›
‹*pcnf-free-variables* (*prefix*, *matrix*) = []›
*empty-clause-or-matrix-if-no-variables*
*no-vars-if-no-free-empty-prefix* **by** *blast*
**then show** *?thesis* **by** *simp*
**next**
**case** *4*: (*Some z*)
**then show** *?thesis*
**proof** (*cases first-existential prefix*)
**case** *None*

**hence** *prefix* = *Empty* **by** (*induction prefix*) *auto*
**hence** *False* **using** ‹*matrix* ≠ []› ‹[] ∉ *set matrix*›
‹*pcnf-free-variables* (*prefix*, *matrix*) = []›
*empty-clause-or-matrix-if-no-variables*
*no-vars-if-no-free-empty-prefix* **by** *blast*
**then show** *?thesis* **by** *simp*
**next**
**case** *5*: (*Some e*)
**have** *6*: *pcnf-free-variables* (*pcnf-assign lit* (*prefix*, *matrix*)) = []
**for** *lit* **using** *pcnf-assign-free-subseteq-free-minus-lit 1*(*6*)
*Diff-empty set-empty subset-Diff-insert subset-empty*
**by** *metis*
**then show** *?thesis*
**proof** (*cases e*)
**case** *7*: *True*
**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*(*1*)[*of 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*)[*of 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*)[*of 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*)[*of z e*] **by** *blast*
**ultimately show** *?thesis* **by** *force*
**qed**
**qed**
**qed**
**qed** *auto*
**qed** *auto*
**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*›
‹*pcnf-free-variables* (*prefix*, *matrix*) = []›
*empty-clause-or-matrix-if-no-variables*
*no-vars-if-no-free-empty-prefix* **by** *blast*
**then show** *?thesis* **by** *simp*
**next**
**case** *4*: (*Some z*)
**then show** *?thesis*
**proof** (*cases first-existential prefix*)
**case** *None*
**hence** *prefix* = *Empty* **by** (*induction prefix*) *auto*
**hence** *False* **using** ‹*matrix* ≠ []› ‹[] ∉ *set matrix*›
‹*pcnf-free-variables* (*prefix*, *matrix*) = []›
*empty-clause-or-matrix-if-no-variables*
*no-vars-if-no-free-empty-prefix* **by** *blast*
**then show** *?thesis* **by** *simp*
**next**
**case** *5*: (*Some e*)
**have** *6*: *pcnf-free-variables* (*pcnf-assign lit* (*prefix*, *matrix*)) = []
**for** *lit* **using** *pcnf-assign-free-subseteq-free-minus-lit 1*(*6*)
*Diff-empty set-empty subset-Diff-insert subset-empty*
**by** *metis*
**hence** *7*: ∃ *b*. *search* (*pcnf-assign lit* (*prefix*, *matrix*)) = *Some b* **for** *lit*
**using** *search-cleansed-closed-yields-Some pcnf-assign-cleansed-inv 6 1*(*5*,*6*) **by** *blast*

**then show** *?thesis*
**proof** (*cases e*)
**case** *8*: *True*
**from** *this* **obtain** *x xs qs* **where** *prefix-def*: *prefix* = *ExistentialFirst* (*x*, *xs*) *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* (*P z*) (*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*(*1*)[*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*: ∀ *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*(*2*)[*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*: ∀ *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 4* **by** *simp*
**ultimately show** *?thesis* **using** *sat-ex-first-iff-assign-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* (*P z*) (*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*: ∀ *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*: ∀ *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* (*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 4* **by** *simp*
**ultimately show** *?thesis* **using** *sat-all-first-iff-assign-conj-sat′ prefix-def*

*1(5)* **by** *simp*
**qed**
**qed**
**qed**
**qed**
**qed**
**qed**

### 4.6.2 Correctness of the Search Solver

**theorem** *search-solver-correct*:
*search-solver pcnf* $\longleftrightarrow$ *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**

# 5 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))*

Simple tests.

**value** *run-naive-solver (String.implode*
*''c an extension of the example from the QDIMACS specification*
*c multiple*
*c lines*
*cwith*
*c comments*
*p cnf 40 4*
*e 1 2 3 4 0*

```
a 11 12 13 14 0
e 21 22 23 24 0
−1  2 0
2 −3 −4 0
40 −13 −24 0
12 −23 −24 0
'')

value run-search-solver (String.implode
''c an extension of the example from the QDIMACS specification
c multiple
c lines
cwith
c comments
p cnf 40 4
e 1 2 3 4 0
a 11 12 13 14 0
e 21 22 23 24 0
−1  2 0
2 −3 −4 0
40 −13 −24 0
12 −23 −24 0
'')

value parse (String.implode
''p cnf 7 12
e 1 2 3 4 5 6 7 0
−3 −1 0
3 1 0
−4 −2 0
4 2 0
−5 −1 −2 0
−5 1 2 0
5 −1 2 0
5 1 −2 0
6 −5 0
−6 5 0
7 0
−7 6 0
'')
```

**code-printing** — This fixes an off-by-one error in the OCaml export.

```
  code-module Str-Literal ⇀
    (OCaml) ‹module Str-Literal =
struct

let implode f xs =
  let rec length xs = match xs with
      [] −> 0
```

```
    | x :: xs -> 1 + length xs in
  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.asciis-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** *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**

## References

[1] A. Bergström. A verified QBF solver. Master's thesis, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, Mar. 2024.

[2] H. Kleine Büning and U. Bubeck. Theory of quantified Boolean formulas. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1131–1156. IOS Press, 2021.