

Verified QBF Solving

Axel Bergström and Tjark Weber

April 18, 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	2
1.1 QBF Datatype, Semantics, and Satisfiability	3
1.1.1 QBF Datatype	3
1.1.2 Formalisation of Semantics and Termination of Semantics	3
1.1.3 Formalisation of Satisfiability	4
1.2 Existential Closure	4
1.2.1 Formalisation of Free Variables	4
1.2.2 Formalisation of Existential Closure	5
1.2.3 Preservation of Satisfiability under Existential Quantification	5
1.2.4 Preservation of Satisfiability under Existential Closure	5
1.2.5 Non-Existence of Free Variables in Existential Closure	5
1.3 Sequence Utility Function	6
1.4 Naive Solver	6
1.4.1 Expanding Quantifiers	6
1.4.2 Expanding Formulas	7
1.4.3 Evaluating Expanded Formulas	8
1.4.4 Naive Solver	8

2	Prenex Conjunctive Normal Form Datatype	8
2.1	Prenex Conjunctive Normal Form Datatype	9
2.1.1	PCNF Predicate for Generic QBFs	9
2.1.2	Bijection with PCNF Subset of Generic QBF Datatype	9
2.1.3	Preservation of Semantics under the Bijection	12
3	QDIMACS Parser	14
4	Search-Based Solver Implementation and Verification	23
4.1	Formalisation of PCNF Assignment	24
4.2	Effect of PCNF Assignments on the Set of all Free Variables	25
4.2.1	Variables, Prefix Variables, and Free Variables	25
4.2.2	Free Variables is Variables without Prefix Variables	26
4.2.3	Set of Matrix Variables is Non-increasing under PCNF Assignments	27
4.2.4	PCNF Assignment Removes Variable from Prefix	27
4.2.5	Set of Free Variables is Non-increasing under PCNF Assignments	28
4.3	PCNF Existential Closure	28
4.3.1	Formalization of PCNF Existential Closure	28
4.3.2	PCNF Existential Closure Preserves Satisfiability	28
4.3.3	No Free Variables in PCNF Existential Closure	28
4.4	Search Solver (Part 1: Preliminaries)	29
4.4.1	Conditions for True and False PCNF Formulas	29
4.4.2	Satisfiability Equivalences for First Variable in Prefix	29
4.5	Cleansed PCNF Formulas	33
4.5.1	Predicate for Cleansed Formulas	33
4.5.2	The Cleansed Predicate is Invariant under PCNF Assignment	33
4.5.3	Cleansing PCNF Formulas	34
4.5.4	Cleansing Yields a Cleansed Formula	35
4.5.5	Cleansing Preserves the Set of Free Variables	35
4.5.6	Cleansing Preserves Semantics	36
4.6	Search Solver (Part 2: The Solver)	37
4.6.1	Correctness of the Search Function	38
4.6.2	Correctness of the Search Solver	38
5	Solver Export	39

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  $\Rightarrow$  bool  $\Rightarrow$  QBF  $\Rightarrow$  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  $\Rightarrow$  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>
```

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  $\in$  set qbf-list
  shows qbf-measure qbf < Suc (sum-list (map qbf-measure qbf-list))
<proof>
```

Semantics based on [2].

```
function qbf-semantics :: (nat  $\Rightarrow$  bool)  $\Rightarrow$  QBF  $\Rightarrow$  bool where
```

```

    qbf-semantic I (Var z) = I z
  | qbf-semantic I (Neg qbf) = (¬(qbf-semantic I qbf))
  | qbf-semantic I (Conj qbf-list) = list-all (qbf-semantic I) qbf-list
  | qbf-semantic I (Disj qbf-list) = list-ex (qbf-semantic I) qbf-list
  | qbf-semantic I (Ex x qbf) = ((qbf-semantic I (substitute-var x True qbf))
    ∨ (qbf-semantic I (substitute-var x False qbf)))
  | qbf-semantic I (All x qbf) = ((qbf-semantic I (substitute-var x True qbf))
    ∧ (qbf-semantic I (substitute-var x False qbf)))

```

⟨proof⟩

termination

⟨proof⟩

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-semantic (λx. False) test-qbf
value qbf-semantic ((λx. False)(2 := True)) test-qbf
value qbf-semantic (((λx. False)(2 := True))(1 := True)) test-qbf

```

1.1.3 Formalisation of Satisfiability

definition *satisfiable* :: QBF ⇒ bool **where**
satisfiable qbf = (∃ I. qbf-semantic I qbf)

definition *logically-eq* :: QBF ⇒ QBF ⇒ bool **where**
logically-eq qbf1 qbf2 = (∀ I. qbf-semantic I qbf1 = qbf-semantic I qbf2)

1.2 Existential Closure

1.2.1 Formalisation of Free Variables

fun *free-variables-aux* :: nat set ⇒ QBF ⇒ nat list **where**
free-variables-aux bound (Var x) = (if x ∈ bound then [] else [x])
 | *free-variables-aux* bound (Neg qbf) = *free-variables-aux* bound qbf
 | *free-variables-aux* bound (Conj list) = concat (map (*free-variables-aux* bound) list)
 | *free-variables-aux* bound (Disj list) = concat (map (*free-variables-aux* bound) list)
 | *free-variables-aux* bound (Ex x qbf) = *free-variables-aux* (insert x bound) qbf
 | *free-variables-aux* bound (All x qbf) = *free-variables-aux* (insert x bound) qbf

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

lemma *bound-subtract-equiv*:

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

1.2.2 Formalisation of Existential Closure

fun *existential-closure-aux* :: $QBF \Rightarrow nat\ list \Rightarrow QBF$ **where**

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

fun *existential-closure* :: $QBF \Rightarrow QBF$ **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 $x1 \neq x2 \vee b1 = b2$

shows $substitute-var\ x1\ b1\ (substitute-var\ x2\ b2\ qbf) = substitute-var\ x2\ b2\ (substitute-var\ x1\ b1\ qbf)$

{proof}

lemma *remove-outer-substitute-var*:

assumes $x1 = x2$

shows $substitute-var\ x1\ b1\ (substitute-var\ x2\ b2\ qbf) = (substitute-var\ x2\ b2\ qbf)$

{proof}

lemma *qbf-semantics-substitute-eq-assign*:

$qbf-semantics\ I\ (substitute-var\ x\ b\ qbf) \longleftrightarrow qbf-semantics\ (I(x := b))\ qbf$

{proof}

lemma *sat-iff-ex-sat*: $satisfiable\ qbf \longleftrightarrow satisfiable\ (Ex\ x\ qbf)$

{proof}

1.2.4 Preservation of Satisfiability under Existential Closure

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

{proof}

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

{proof}

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}

theorem *ex-closure-no-free*: $set (free-variables (existential-closure qbf)) = \{\}$
 ⟨proof⟩

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* (λx. x ≠ None) list
shows ∃xs. *map Some xs* = list ⟨proof⟩

lemma *sequence-aux-content*: *sequence-aux (map Some xs) list* = Some (rev xs @ list)
 ⟨proof⟩

lemma *sequence-content*: *sequence (map Some xs)* = Some xs
 ⟨proof⟩

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⟩

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 $qbf\text{-quantifier-depth } z = list\text{-max } (map\ qbf\text{-quantifier-depth } qbf\text{-list})$ $\langle proof \rangle$

function $expand\text{-quantifiers} :: QBF \Rightarrow QBF$ **where**

$expand\text{-quantifiers } (Var\ x) = (Var\ x)$
| $expand\text{-quantifiers } (Neg\ qbf) = Neg\ (expand\text{-quantifiers } qbf)$
| $expand\text{-quantifiers } (Conj\ list) = Conj\ (map\ expand\text{-quantifiers } list)$
| $expand\text{-quantifiers } (Disj\ list) = Disj\ (map\ expand\text{-quantifiers } list)$
| $expand\text{-quantifiers } (Ex\ x\ qbf) = (Disj\ [substitute\text{-var } x\ True\ (expand\text{-quantifiers } qbf),$
 $substitute\text{-var } x\ False\ (expand\text{-quantifiers } qbf)])$
| $expand\text{-quantifiers } (All\ x\ qbf) = (Conj\ [substitute\text{-var } x\ True\ (expand\text{-quantifiers } qbf),$
 $substitute\text{-var } x\ False\ (expand\text{-quantifiers } qbf)])$
 $\langle proof \rangle$

termination

$\langle proof \rangle$

Property 1: no quantifiers after expansion.

lemma $no\text{-quants}\text{-after}\text{-expand}\text{-quants}$: $qbf\text{-quantifier-depth } (expand\text{-quantifiers } qbf) = 0$
 $\langle proof \rangle$

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

lemma $semantics\text{-inv}\text{-under}\text{-expand}$:

$qbf\text{-semantics } I\ qbf = qbf\text{-semantics } I\ (expand\text{-quantifiers } qbf)$
 $\langle proof \rangle$

lemma $sat\text{-iff}\text{-expand}\text{-quants}\text{-sat}$: $satisfiable\ qbf \longleftrightarrow satisfiable\ (expand\text{-quantifiers } qbf)$
 $\langle proof \rangle$

Property 3: free variables invariant under expansion.

lemma $set\text{-free}\text{-vars}\text{-subst}\text{-all}\text{-eq}$:

$set\ (free\text{-variables } (substitute\text{-var } x\ b\ qbf)) = set\ (free\text{-variables } (All\ x\ qbf))$
 $\langle proof \rangle$

lemma $set\text{-free}\text{-vars}\text{-subst}\text{-ex}\text{-eq}$:

$set\ (free\text{-variables } (substitute\text{-var } x\ b\ qbf)) = set\ (free\text{-variables } (Ex\ x\ qbf))$
 $\langle proof \rangle$

lemma $free\text{-vars}\text{-inv}\text{-under}\text{-expand}\text{-quants}$:

$set\ (free\text{-variables } (expand\text{-quantifiers } qbf)) = set\ (free\text{-variables } qbf)$
 $\langle proof \rangle$

1.4.2 Expanding Formulas

fun $expand\text{-qbf} :: QBF \Rightarrow QBF$ **where**

$expand\text{-qbf } qbf = expand\text{-quantifiers } (existential\text{-closure } qbf)$

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

lemma *sat-iff-expand-qbf-sat*: $\text{satisfiable } (\text{expand-qbf } qbf) \longleftrightarrow \text{satisfiable } qbf$
 ⟨proof⟩

lemma *expand-qbf-no-free*: $\text{set } (\text{free-variables } (\text{expand-qbf } qbf)) = \{\}$
 ⟨proof⟩

lemma *expand-qbf-no-quants*: $\text{qbf-quantifier-depth } (\text{expand-qbf } qbf) = 0$
 ⟨proof⟩

1.4.3 Evaluating Expanded Formulas

fun *eval-qbf* :: $QBF \Rightarrow \text{bool option}$ **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*: $\text{list-ex } Q (\text{map } f x) = \text{list-ex } (Q \circ f) x$
 ⟨proof⟩

The evaluation implements the semantics.

lemma *eval-qbf-implements-semantics*:
assumes $\text{set } (\text{free-variables } qbf) = \{\}$ **and** $\text{qbf-quantifier-depth } qbf = 0$
shows $\text{eval-qbf } qbf = \text{Some } (\text{qbf-semantics } I qbf)$ ⟨proof⟩

1.4.4 Naive Solver

fun *naive-solver* :: $QBF \Rightarrow \text{bool}$ **where**
naive-solver $qbf = \text{the } (\text{eval-qbf } (\text{expand-qbf } qbf))$

theorem *naive-solver-correct*: $\text{naive-solver } qbf \longleftrightarrow \text{satisfiable } qbf$
 ⟨proof⟩

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)*
⟨proof⟩

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*
⟨proof⟩

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

lemma *convert-clause-p: clause-p (convert-clause cl)*
⟨proof⟩

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⟩

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

lemma *convert-cnf-p: cnf-p (convert-matrix mat)*
⟨proof⟩

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⟩

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*))
⟨*proof*⟩

termination

⟨*proof*⟩

theorem *convert-pcnf-p*: *pcnf-p* (*convert* *pcnf*)

⟨*proof*⟩

fun *add-universal-to-front* :: *nat* ⇒ *pcnf* ⇒ *pcnf* **where**

add-universal-to-front *x* (*Empty*, *matrix*) = (*UniversalFirst* (*x*, []) [], *matrix*)
| *add-universal-to-front* *x* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*) = (*UniversalFirst* (*x*, *y* # *ys*) *qs*, *matrix*)
| *add-universal-to-front* *x* (*ExistentialFirst* (*y*, *ys*) *qs*, *matrix*) = (*UniversalFirst* (*x*, []) ((*y*, *ys*) # *qs*), *matrix*)

fun *add-existential-to-front* :: *nat* ⇒ *pcnf* ⇒ *pcnf* **where**

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

fun *convert-inv* :: *QBF* ⇒ *pcnf* *option* **where**

convert-inv (*All* *x* *qbf*) = *map-option* (λp . *add-universal-to-front* *x* *p*) (*convert-inv* *qbf*)
| *convert-inv* (*Ex* *x* *qbf*) = *map-option* (λp . *add-existential-to-front* *x* *p*) (*convert-inv* *qbf*)
| *convert-inv* *qbf* = *map-option* (λm . (*Empty*, *m*)) (*convert-matrix-inv* *qbf*)

lemma *convert-add-all*: $\text{convert } (\text{add-universal-to-front } x \text{ pcnf}) = \text{All } x \text{ (convert pcnf)}$
 ⟨proof⟩

lemma *convert-add-ex*: $\text{convert } (\text{add-existential-to-front } x \text{ pcnf}) = \text{Ex } x \text{ (convert pcnf)}$
 ⟨proof⟩

theorem *convert-inv*: $\text{convert-inv } (\text{convert pcnf}) = \text{Some pcnf}$
 ⟨proof⟩

theorem *convert-injective*: inj convert
 ⟨proof⟩

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

lemma *convert-literal-p-ex*:
assumes *literal-p lit*
shows $\exists l. \text{convert-literal } l = \text{lit}$
 ⟨proof⟩

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

lemma *convert-cnf-p-ex*:
assumes *cnf-p mat*
shows $\exists m. \text{convert-matrix } m = \text{mat}$
 ⟨proof⟩

theorem *convert-pcnf-p-ex*:
assumes *pcnf-p qbf*
shows $\exists \text{pcnf}. \text{convert pcnf} = \text{qbf}$ ⟨proof⟩

theorem *convert-range*: $\text{range convert} = \{p. \text{pcnf-p } p\}$
 ⟨proof⟩

theorem *convert-bijective-on*: $\text{bij-betw convert UNIV } \{p. \text{pcnf-p } p\}$
 ⟨proof⟩

2.1.3 Preservation of Semantics under the Bijection

fun *literal-semantics* :: $(\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{literal} \Rightarrow \text{bool}$ **where**
literal-semantics *I* (*P* *x*) = *I* *x*
 | *literal-semantics* *I* (*N* *x*) = (\neg *I* *x*)

```

fun clause-semantics :: (nat  $\Rightarrow$  bool)  $\Rightarrow$  clause  $\Rightarrow$  bool where
  clause-semantics I clause = list-ex (literal-semantics I) clause

fun matrix-semantics :: (nat  $\Rightarrow$  bool)  $\Rightarrow$  matrix  $\Rightarrow$  bool where
  matrix-semantics I matrix = list-all (clause-semantics I) matrix

function pcnf-semantics :: (nat  $\Rightarrow$  bool)  $\Rightarrow$  pcnf  $\Rightarrow$  bool where
  pcnf-semantics I (Empty, matrix) =
    matrix-semantics I matrix
| pcnf-semantics I (UniversalFirst (y, []) [], matrix) =
  (pcnf-semantics (I(y := True)) (Empty, matrix))
   $\wedge$  pcnf-semantics (I(y := False)) (Empty, matrix)
| pcnf-semantics I (ExistentialFirst (x, []) [], matrix) =
  (pcnf-semantics (I(x := True)) (Empty, matrix))
   $\vee$  pcnf-semantics (I(x := False)) (Empty, matrix)
| pcnf-semantics I (UniversalFirst (y, []) (q # qs), matrix) =
  (pcnf-semantics (I(y := True)) (ExistentialFirst q qs, matrix))
   $\wedge$  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))
   $\vee$  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))
   $\wedge$  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))
   $\vee$  pcnf-semantics (I(x := False)) (ExistentialFirst (xx, xs) qs, matrix)
  <proof>
termination
  <proof>

theorem qbf-semantics-eq-pcnf-semantics:
  pcnf-semantics I pcnf = qbf-semantics I (convert pcnf)
  <proof>

lemma convert-inv-inv:
  pcnf-p qbf  $\implies$  convert (the (convert-inv qbf)) = qbf
  <proof>

theorem qbf-semantics-eq-pcnf-semantics':
  assumes pcnf-p qbf
  shows qbf-semantics I qbf = pcnf-semantics I (the (convert-inv qbf))
  <proof>

end

```

3 QDIMACS Parser

```
theory Parser
  imports PCNF
begin
```

```
type-synonym 'a parser = string  $\Rightarrow$  ('a  $\times$  string) option
```

```
fun trim_ws :: string  $\Rightarrow$  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
  <proof>
```

```
lemma non-increasing-trim_ws-lemmas [intro]:
  shows length s  $\leq$  length s'  $\implies$  length (trim_ws s)  $\leq$  length s'
  and length s < length s'  $\implies$  length (trim_ws s) < length s'
  and length s  $\leq$  length (trim_ws s')  $\implies$  length s  $\leq$  length s'
  and length s < length (trim_ws s')  $\implies$  length s < length s'
  <proof>
```

```
lemma whitespace-and-parse-le [intro]:
  assumes  $\bigwedge s s' r. p s = \text{Some } (r, s') \implies \text{length } s' \leq \text{length } s$ 
  shows  $\bigwedge s s' r. p (\text{trim\_ws } s) = \text{Some } (r, s') \implies \text{length } s' \leq \text{length } s$  <proof>
```

```
lemma whitespace-and-parse-unit-le [intro]:
  assumes  $\bigwedge s s'. p s = \text{Some } ((), s') \implies \text{length } s' \leq \text{length } s$ 
  shows  $\bigwedge s s'. p (\text{trim\_ws } s) = \text{Some } ((), s') \implies \text{length } s' \leq \text{length } s$  <proof>
```

```
lemma whitespace-and-parse-less [intro]:
  assumes  $\bigwedge s s' r. p s = \text{Some } (r, s') \implies \text{length } s' < \text{length } s$ 
  shows  $\bigwedge s s' r. p (\text{trim\_ws } s) = \text{Some } (r, s') \implies \text{length } s' < \text{length } s$  <proof>
```

```
lemma whitespace-and-parse-unit-less [intro]:
  assumes  $\bigwedge s s'. p s = \text{Some } ((), s') \implies \text{length } s' < \text{length } s$ 
  shows  $\bigwedge s s'. p (\text{trim\_ws } s) = \text{Some } ((), s') \implies \text{length } s' < \text{length } s$  <proof>
```

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

```
lemma non-increasing-match [simp]: match xs s = Some ((), s')  $\implies \text{length } s' \leq$ 
length s
  <proof>
```

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

<proof>

fun *digit-to-nat* :: *char* \Rightarrow *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* \Rightarrow *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')* \implies *length s' \leq length s*

<proof>

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')* \implies *length s' < length s*

<proof>

fun *pnum* :: *nat parser* **where**

```
pnum str = (case pnum-raw str of  
  None  $\Rightarrow$  None |  
  Some (n, str')  $\Rightarrow$  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⟩

```

```

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⟩

```

```

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 "↔" (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 ↔"
value clause "1 2 -3 4 0 ↔↔"
value clause "1 2 -3 40 ↔"
value clause "1 2 -3 4 0↔"
value clause "1 2 -3 4 0"
value clause "1 2 -3 4 0 ↔"

```


lemma *decreasing-clause* [simp]:
assumes *clause s = Some (c, s')*
shows *length s' < length s* <proof>

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

Simple tests.

value *clause-list* "1 2 -3 0 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "
value *clause-list* "1 2 -3 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "
value *clause-list* "1 2 -3 0 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "

lemma *decreasing-clause-list* [simp]:
assumes *clause-list s = Some (cls, s')*
shows *length s' < length s* <proof>

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

Simple tests.

value *matrix* "1 2 -3 0 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "
value *matrix* "1 2 -3 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "
value *matrix* "1 2 -3 0 $\boxed{\leftarrow}$ 1 -2 3 0 $\boxed{\leftarrow}$ -1 2 3 0 $\boxed{\leftarrow}$ "

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

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

Simple tests.

value *atom-set* "1 2 3 4"
value *atom-set* "1 2 -3 4"
value *atom-set* "1 2 3 4 0 $\boxed{\leftarrow}$ "
value *atom-set* "1 2 3 4 0"
value *atom-set* "1 2 3 4 0 $\boxed{\leftarrow}$ "

```

value atom-set "1 2 3 4"
value atom-set " 1  2  3  4 0  $\boxed{\leftrightarrow}$  "

```

```

lemma decreasing-atom-set [simp]:
  assumes atom-set s = Some (as, s')
  shows length s' < length s <math>\langle\text{proof}\rangle</math>

```

```

datatype quant = Universal | Existential

```

```

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

```

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'  $\leq$  length s
  <math>\langle\text{proof}\rangle</math>

```

```

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

```

Simple tests.

```

value quant-set "e 1 2 3 0 $\boxed{\leftrightarrow}$ "
value quant-set "a 1 2 3 0 $\boxed{\leftrightarrow}$ "
value quant-set "a 1 2 -3 0 $\boxed{\leftrightarrow}$ "

```

```

lemma decreasing-quant-set [simp]:

```

```

assumes quant-set s = Some (q-set, s')
shows length s' < length s
⟨proof⟩

```

```

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 ⟨proof⟩

```

```

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* ⟨proof⟩

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⟩

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') ⇒*
length s' ≤ length s
 ⟨proof⟩

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 "↔" str'' of
      None ⇒ None |
      Some (-, str''') ⇒ Some ((, str'''))))

```

Simple tests.

```

value comment-line "c e 1 2 3↔e 1 2 3"
value comment-line "e 1 2 3↔e 1 2 3"
value comment-line " c e 1 2 3 ↔e 1 2 3"

```

```

lemma decreasing-comment-line [simp]:
  assumes comment-line s = Some ((, s')
  shows length s' < length s
  ⟨proof⟩

```

```

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 ⟨proof⟩

```

```

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↔p cnf 4 5↔"
value preamble " c an example↔ p cnf 4 5↔"

```

```

lemma decreasing-preamble [simp]:
  assumes preamble s = Some (p, s')
  shows length s' < length s
  ⟨proof⟩

```

fun eof :: unit parser **where**

 eof Nil = Some ((), Nil)
 | eof (Cons x xs) = None

lemma eof-nil [simp]: eof s = Some ((), s') \implies s' = Nil
 ⟨proof⟩

fun input :: PCNF.pcnf parser **where**

 input str = (case preamble str of
 None \Rightarrow None |
 Some ((lits, clauses), str') \Rightarrow
 (case prefix str' of
 None \Rightarrow None |
 Some (pre, str'') \Rightarrow
 (case matrix str'' of
 None \Rightarrow None |
 Some (mat, str''') \Rightarrow
 (case eof str''' of
 None \Rightarrow None |
 Some (-, str''') \Rightarrow 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 ⟨proof⟩
```

```
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* \Rightarrow *literal* **where**

lit-neg (*P l*) = *N l*
 | *lit-neg* (*N l*) = *P l*

fun *lit-var* :: *literal* \Rightarrow *nat* **where**

lit-var (*P l*) = *l*
 | *lit-var* (*N l*) = *l*

fun *remove-lit-neg* :: *literal* \Rightarrow *clause* \Rightarrow *clause* **where**

remove-lit-neg *lit clause* = *filter* ($\lambda l. l \neq \text{lit-neg lit}$) *clause*

fun *remove-lit-clauses* :: *literal* \Rightarrow *matrix* \Rightarrow *matrix* **where**

remove-lit-clauses *lit matrix* = *filter* ($\lambda cl. \neg(\text{list-ex } (\lambda l. l = \text{lit}) cl)$) *matrix*

fun *matrix-assign* :: *literal* \Rightarrow *matrix* \Rightarrow *matrix* **where**

matrix-assign *lit matrix* = *remove-lit-clauses* *lit* (*map* (*remove-lit-neg lit*) *matrix*)

fun *prefix-pop* :: *prefix* \Rightarrow *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* \Rightarrow *prefix* \Rightarrow *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* \Rightarrow *prefix* \Rightarrow *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* \Rightarrow *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* \Rightarrow *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 \neq *Empty* \implies *prefix-measure* (*prefix-pop* *prefix*) < *prefix-measure* *prefix*
 <*proof*>

function *remove-var-prefix* :: *nat* \Rightarrow *prefix* \Rightarrow *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*)))
 <*proof*>

termination

<*proof*>

fun *pcnf-assign* :: *literal* \Rightarrow *pcnf* \Rightarrow *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* \Rightarrow *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* \Rightarrow *nat list* **where**

variables *qbf* = *sort* (*remdups* (*variables-aux* *qbf*))

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

fun *pcnf-variables* :: *pcnf* \Rightarrow *nat list* **where**
pcnf-variables *pcnf* = *variables* (*convert* *pcnf*)

fun *pcnf-prefix-variables* :: *pcnf* \Rightarrow *nat list* **where**
pcnf-prefix-variables *pcnf* = *prefix-variables* (*convert* *pcnf*)

fun *pcnf-free-variables* :: *pcnf* \Rightarrow *nat list* **where**
pcnf-free-variables *pcnf* = *free-variables* (*convert* *pcnf*)

lemma *free-assgn-proof-skeleton*:
free = *var* - *pre* \implies *free-assgn* = *var-assgn* - *pre-assgn*
 \implies *var-assgn* \subseteq *var* - *lit*
 \implies *pre-assgn* = *pre* - *lit*
 \implies *free-assgn* \subseteq *free* - *lit*
 <proof>

4.2.2 Free Variables is Variables without Prefix Variables

lemma *lit-p-free-eq-vars*:
literal-p *qbf* \implies *set* (*free-variables* *qbf*) = *set* (*variables* *qbf*)
 <proof>

lemma *cl-p-free-eq-vars*:
assumes *clause-p* *qbf*
shows *set* (*free-variables* *qbf*) = *set* (*variables* *qbf*)
 <proof>

lemma *cnf-p-free-eq-vars*:
assumes *cnf-p* *qbf*
shows *set* (*free-variables* *qbf*) = *set* (*variables* *qbf*)
 <proof>

lemma *pcnf-p-free-eq-vars-minus-prefix-aux*:
pcnf-p *qbf* \implies *set* (*free-variables* *qbf*) = *set* (*variables* *qbf*) - *set* (*prefix-variables-aux* *qbf*)
 <proof>

lemma *pcnf-p-free-eq-vars-minus-prefix*:
pcnf-p *qbf* \implies *set* (*free-variables* *qbf*) = *set* (*variables* *qbf*) - *set* (*prefix-variables* *qbf*)
 <proof>

lemma *pcnf-free-eq-vars-minus-prefix*:
set (*pcnf-free-variables* *pcnf*)

= $set (pcnf\text{-}variables\ pcnf) - set (pcnf\text{-}prefix\text{-}variables\ pcnf)$
 ⟨proof⟩

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

lemma *lit-not-in-matrix-assign-variables:*

$lit\text{-}var\ lit \notin set (variables (convert\text{-}matrix (matrix\text{-}assign\ lit\ matrix)))$
 ⟨proof⟩

lemma *matrix-assign-vars-subseteq-matrix-vars-minus-lit:*

$set (variables (convert\text{-}matrix (matrix\text{-}assign\ lit\ matrix)))$
 $\subseteq set (variables (convert\text{-}matrix\ matrix)) - \{lit\text{-}var\ lit\}$
 ⟨proof⟩

lemma *pcnf-vars-eq-matrix-vars:*

$set (pcnf\text{-}variables (prefix, matrix))$
 $= set (variables (convert\text{-}matrix\ matrix))$
 ⟨proof⟩

lemma *pcnf-assign-vars-subseteq-vars-minus-lit:*

$set (pcnf\text{-}variables (pcnf\text{-}assign\ x\ pcnf))$
 $\subseteq set (pcnf\text{-}variables\ pcnf) - \{lit\text{-}var\ x\}$
 ⟨proof⟩

4.2.4 PCNF Assignment Removes Variable from Prefix

lemma *add-ex-adds-prefix-var:*

$set (pcnf\text{-}prefix\text{-}variables (add\text{-}existential\text{-}to\text{-}front\ x\ pcnf))$
 $= set (pcnf\text{-}prefix\text{-}variables\ pcnf) \cup \{x\}$
 ⟨proof⟩

lemma *add-ex-to-prefix-eq-add-to-front:*

$(add\text{-}existential\text{-}to\text{-}prefix\ x\ prefix, matrix) = add\text{-}existential\text{-}to\text{-}front\ x (prefix, matrix)$
 ⟨proof⟩

lemma *add-all-adds-prefix-var:*

$set (pcnf\text{-}prefix\text{-}variables (add\text{-}universal\text{-}to\text{-}front\ x\ pcnf))$
 $= set (pcnf\text{-}prefix\text{-}variables\ pcnf) \cup \{x\}$
 ⟨proof⟩

lemma *add-all-to-prefix-eq-add-to-front:*

$(add\text{-}universal\text{-}to\text{-}prefix\ x\ prefix, matrix) = add\text{-}universal\text{-}to\text{-}front\ x (prefix, matrix)$
 ⟨proof⟩

lemma *prefix-assign-vars-eq-prefix-vars-minus-lit:*

$set (pcnf\text{-}prefix\text{-}variables (remove\text{-}var\text{-}prefix\ x\ prefix, matrix))$
 $= set (pcnf\text{-}prefix\text{-}variables (prefix, matrix)) - \{x\}$

<proof>

lemma *prefix-vars-matrix-inv:*

$set (pcnf\text{-}prefix\text{-}variables (prefix, matrix1))$
 $= set (pcnf\text{-}prefix\text{-}variables (prefix, matrix2))$
<proof>

lemma *pcnf-prefix-vars-eq-prefix-minus-lit:*

$set (pcnf\text{-}prefix\text{-}variables (pcnf\text{-}assign\ x\ pcnf))$
 $= set (pcnf\text{-}prefix\text{-}variables\ pcnf) - \{lit\text{-}var\ x\}$
<proof>

4.2.5 Set of Free Variables is Non-increasing under PCNF Assignments

theorem *pcnf-assign-free-subseteq-free-minus-lit:*

$set (pcnf\text{-}free\text{-}variables (pcnf\text{-}assign\ x\ pcnf)) \subseteq set (pcnf\text{-}free\text{-}variables\ pcnf) - \{lit\text{-}var\ x\}$
<proof>

4.3 PCNF Existential Closure

4.3.1 Formalization of PCNF Existential Closure

fun *pcnf-existential-closure* :: *pcnf* \Rightarrow *pcnf* **where**

pcnf-existential-closure pcnf = the (convert-inv (existential-closure (convert pcnf)))

4.3.2 PCNF Existential Closure Preserves Satisfiability

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

$pcnf\text{-}p\ qbf \implies pcnf\text{-}p (existential\text{-}closure\text{-}aux\ qbf\ vars)$
<proof>

lemma *ex-closure-pcnf-p-inv:*

$pcnf\text{-}p\ qbf \implies pcnf\text{-}p (existential\text{-}closure\ qbf)$
<proof>

theorem *pcnf-sat-iff-ex-close-sat:*

$satisfiable (convert\ pcnf) = satisfiable (convert (pcnf\text{-}existential\text{-}closure\ pcnf))$
<proof>

4.3.3 No Free Variables in PCNF Existential Closure

theorem *pcnf-ex-closure-no-free:*

$pcnf\text{-}free\text{-}variables (pcnf\text{-}existential\text{-}closure\ pcnf) = \square$
<proof>

4.4 Search Solver (Part 1: Preliminaries)

4.4.1 Conditions for True and False PCNF Formulas

lemma *single-clause-variables*:

$set (pcnf-variables (Empty, [cl])) = set (map lit-var cl)$
 $\langle proof \rangle$

lemma *empty-prefix-cons-matrix-variables*:

$set (pcnf-variables (Empty, Cons cl cls))$
 $= set (pcnf-variables (Empty, cls)) \cup set (map lit-var cl)$
 $\langle proof \rangle$

lemma *false-if-empty-clause-in-matrix*:

$[] \in set\ matrix \implies pcnf-semantic\ I (prefix, matrix) = False$
 $\langle proof \rangle$

lemma *true-if-matrix-empty*:

$matrix = [] \implies pcnf-semantic\ I (prefix, matrix) = True$
 $\langle proof \rangle$

lemma *matrix-shape-if-no-variables*:

$pcnf-variables (Empty, matrix) = [] \implies (\exists n. matrix = replicate\ n\ [])$
 $\langle proof \rangle$

lemma *empty-clause-or-matrix-if-no-variables*:

$pcnf-variables (Empty, matrix) = [] \implies [] \in set\ matrix \vee matrix = []$
 $\langle proof \rangle$

4.4.2 Satisfiability Equivalences for First Variable in Prefix

lemma *clause-semantic-inv-remove-false*:

$clause-semantic (I(z := True))\ cl = clause-semantic (I(z := True)) (remove-lit-neg$
 $(P\ z)\ cl)$
 $\langle proof \rangle$

lemma *clause-semantic-inv-remove-true*:

$clause-semantic (I(z := False))\ cl = clause-semantic (I(z := False)) (remove-lit-neg$
 $(N\ z)\ cl)$
 $\langle proof \rangle$

lemma *matrix-semantic-inv-remove-true*:

$matrix-semantic (I(z := True)) (matrix-assign (P\ z)\ matrix)$
 $= matrix-semantic (I(z := True))\ matrix$
 $\langle proof \rangle$

lemma *matrix-semantic-inv-remove-true'*:

assumes $y \neq z$
shows $matrix-semantic (I(z := True, y := b)) (matrix-assign (P\ z)\ matrix)$

$= \text{matrix-semantic} (I(z := \text{True}, y := b)) \text{ matrix}$
 $\langle \text{proof} \rangle$

lemma *matrix-semantic-inv-remove-false*:
 $\text{matrix-semantic} (I(z := \text{False})) (\text{matrix-assign } (N z) \text{ matrix})$
 $= \text{matrix-semantic} (I(z := \text{False})) \text{ matrix}$
 $\langle \text{proof} \rangle$

lemma *matrix-semantic-inv-remove-false'*:
assumes $y \neq z$
shows $\text{matrix-semantic} (I(z := \text{False}, y := b)) (\text{matrix-assign } (N z) \text{ matrix})$
 $= \text{matrix-semantic} (I(z := \text{False}, y := b)) \text{ matrix}$
 $\langle \text{proof} \rangle$

lemma *matrix-semantic-disj-iff-true-assgn*:
 $(\exists b. \text{matrix-semantic} (I(z := b)) \text{ matrix})$
 $\longleftrightarrow \text{matrix-semantic} (I(z := \text{True})) (\text{matrix-assign } (P z) \text{ matrix})$
 $\vee \text{matrix-semantic} (I(z := \text{False})) (\text{matrix-assign } (N z) \text{ matrix})$
 $\langle \text{proof} \rangle$

lemma *matrix-semantic-conj-iff-true-assgn*:
 $(\forall b. \text{matrix-semantic} (I(z := b)) \text{ matrix})$
 $\longleftrightarrow \text{matrix-semantic} (I(z := \text{True})) (\text{matrix-assign } (P z) \text{ matrix})$
 $\wedge \text{matrix-semantic} (I(z := \text{False})) (\text{matrix-assign } (N z) \text{ matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-assign-free-eq-matrix-assgn'*:
assumes $\text{lit-var lit} \notin \text{set } (\text{prefix-variables-aux } (\text{convert } (\text{prefix}, \text{matrix})))$
shows $\text{pcnf-assign lit } (\text{prefix}, \text{matrix}) = (\text{prefix}, \text{matrix-assign lit matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-assign-free-eq-matrix-assgn*:
assumes $\text{lit-var lit} \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $\text{pcnf-assign lit } (\text{prefix}, \text{matrix}) = (\text{prefix}, \text{matrix-assign lit matrix})$
 $\langle \text{proof} \rangle$

lemma *neq-first-if-notin-all-prefix*:
 $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{UniversalFirst } (y, ys) \text{ qs}, \text{matrix})) \implies z \neq y$
 $\langle \text{proof} \rangle$

lemma *neq-first-if-notin-ex-prefix*:
 $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{ExistentialFirst } (x, xs) \text{ qs}, \text{matrix})) \implies z \neq x$
 $\langle \text{proof} \rangle$

lemma *notin-pop-prefix-if-notin-prefix*:

assumes $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop prefix}, \text{matrix}))$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantic-inv-matrix-assign-true:*

assumes $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $\text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix-assign } (P z) \text{ matrix})$
 $= \text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantic-inv-matrix-assign-false:*

assumes $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $\text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix-assign } (N z) \text{ matrix})$
 $= \text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantic-disj-iff-matrix-assign-disj:*

assumes $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $\text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix})$
 $\vee \text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix})$
 \longleftrightarrow
 $\text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix-assign } (P z) \text{ matrix})$
 $\vee \text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix-assign } (N z) \text{ matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantic-conj-iff-matrix-assign-conj:*

assumes $z \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix}, \text{matrix}))$
shows $\text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix})$
 $\wedge \text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix})$
 \longleftrightarrow
 $\text{pcnf-semantic } (I(z := \text{True})) (\text{prefix}, \text{matrix-assign } (P z) \text{ matrix})$
 $\wedge \text{pcnf-semantic } (I(z := \text{False})) (\text{prefix}, \text{matrix-assign } (N z) \text{ matrix})$
 $\langle \text{proof} \rangle$

lemma *semantic-eq-if-free-vars-eq:*

assumes $\forall x \in \text{set } (\text{free-variables } \text{qbf}). I(x) = J(x)$
shows $\text{qbf-semantic } I \text{ qbf} = \text{qbf-semantic } J \text{ qbf} \langle \text{proof} \rangle$

lemma *pcnf-semantic-eq-if-free-vars-eq:*

assumes $\forall x \in \text{set } (\text{pcnf-free-variables } \text{pcnf}). I(x) = J(x)$
shows $\text{pcnf-semantic } I \text{ pcnf} = \text{pcnf-semantic } J \text{ pcnf}$
 $\langle \text{proof} \rangle$

lemma *x-notin-assign-P-x:*

$x \notin \text{set } (\text{pcnf-variables } (\text{pcnf-assign } (P \ x) \ \text{pcnf}))$
<proof>

lemma *x-notin-assign-N-x:*

$x \notin \text{set } (\text{pcnf-variables } (\text{pcnf-assign } (N \ x) \ \text{pcnf}))$
<proof>

lemma *interp-value-ignored-for-pcnf-P-assign:*

$\text{pcnf-semantics } (I(x := b)) (\text{pcnf-assign } (P \ x) \ \text{pcnf})$
 $= \text{pcnf-semantics } I (\text{pcnf-assign } (P \ x) \ \text{pcnf})$
<proof>

lemma *interp-value-ignored-for-pcnf-N-assign:*

$\text{pcnf-semantics } (I(x := b)) (\text{pcnf-assign } (N \ x) \ \text{pcnf})$
 $= \text{pcnf-semantics } I (\text{pcnf-assign } (N \ x) \ \text{pcnf})$
<proof>

lemma *sat-ex-first-iff-one-assign-sat:*

assumes $x \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{ExistentialFirst } (x, \ xs) \ qs), \ \text{matrix}))$
shows $\text{satisfiable } (\text{convert } (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix}))$
 $\longleftrightarrow \text{satisfiable } (\text{convert } (\text{pcnf-assign } (P \ x) (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix})))$
 $\vee \text{satisfiable } (\text{convert } (\text{pcnf-assign } (N \ x) (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix})))$
<proof>

theorem *sat-ex-first-iff-assign-disj-sat:*

assumes $x \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{ExistentialFirst } (x, \ xs) \ qs), \ \text{matrix}))$
shows $\text{satisfiable } (\text{convert } (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix}))$
 $\longleftrightarrow \text{satisfiable } (\text{Disj}$
 $\quad [\text{convert } (\text{pcnf-assign } (P \ x) (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix})),$
 $\quad \text{convert } (\text{pcnf-assign } (N \ x) (\text{ExistentialFirst } (x, \ xs) \ qs, \ \text{matrix}))])$
<proof>

theorem *sat-all-first-iff-assign-conj-sat:*

assumes $y \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{UniversalFirst } (y, \ ys) \ qs), \ \text{matrix}))$
shows $\text{satisfiable } (\text{convert } (\text{UniversalFirst } (y, \ ys) \ qs, \ \text{matrix}))$
 $\longleftrightarrow \text{satisfiable } (\text{Conj}$
 $\quad [\text{convert } (\text{pcnf-assign } (P \ y) (\text{UniversalFirst } (y, \ ys) \ qs, \ \text{matrix})),$
 $\quad \text{convert } (\text{pcnf-assign } (N \ y) (\text{UniversalFirst } (y, \ ys) \ qs, \ \text{matrix}))])$
<proof>

4.5 Cleansed PCNF Formulas

4.5.1 Predicate for Cleansed Formulas

fun *cleansed-p* :: *pcnf* \Rightarrow *bool* **where**
 cleansed-p pcnf = *distinct* (*prefix-variables-aux* (*convert pcnf*))

lemma *prefix-pop-cleansed-if-cleansed*:
 cleansed-p (*prefix*, *matrix*) \Longrightarrow *cleansed-p* (*prefix-pop prefix*, *matrix*)
 \langle *proof* \rangle

lemma *prefix-variables-aux-matrix-inv*:
 prefix-variables-aux (*convert* (*prefix*, *matrix1*))
 = *prefix-variables-aux* (*convert* (*prefix*, *matrix2*))
 \langle *proof* \rangle

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*))
 \langle *proof* \rangle

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*))
 \langle *proof* \rangle

lemma *cleansed-p-matrix-inv*:
 cleansed-p (*prefix*, *matrix1*) = *cleansed-p* (*prefix*, *matrix2*)
 \langle *proof* \rangle

lemma *cleansed-prefix-first-ex-unique*:
 assumes *cleansed-p* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)
 shows $x \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{ExistentialFirst } (x, xs) \text{ qs}), \text{matrix}))$
 \langle *proof* \rangle

lemma *cleansed-prefix-first-all-unique*:
 assumes *cleansed-p* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)
 shows $y \notin \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{UniversalFirst } (y, ys) \text{ qs}), \text{matrix}))$
 \langle *proof* \rangle

4.5.2 The Cleansed Predicate is Invariant under PCNF Assignment

lemma *cleansed-add-new-ex-to-front*:
 assumes *cleansed-p pcnf*
 and $x \notin \text{set } (\text{pcnf-prefix-variables } \text{pcnf})$
 shows *cleansed-p* (*add-existential-to-front x pcnf*)
 \langle *proof* \rangle

lemma *cleansed-add-new-all-to-front*:

assumes *cleansed-p pcnf*
and $y \notin \text{set } (\text{pcnf-prefix-variables } \text{pcnf})$
shows *cleansed-p (add-universal-to-front y pcnf)*
<proof>

lemma *pcnf-assign-p-ex-eq*:

assumes *cleansed-p (ExistentialFirst (x, xs) qs, matrix)*
shows *pcnf-assign (P x) (ExistentialFirst (x, xs) qs, matrix)*
 $= (\text{prefix-pop } (\text{ExistentialFirst } (x, xs) \text{ qs}), \text{matrix-assign } (P \ x) \ \text{matrix})$
<proof>

lemma *pcnf-assign-p-all-eq*:

assumes *cleansed-p (UniversalFirst (y, ys) qs, matrix)*
shows *pcnf-assign (P y) (UniversalFirst (y, ys) qs, matrix)*
 $= (\text{prefix-pop } (\text{UniversalFirst } (y, ys) \ \text{qs}), \text{matrix-assign } (P \ y) \ \text{matrix})$
<proof>

lemma *pcnf-assign-n-ex-eq*:

assumes *cleansed-p (ExistentialFirst (x, xs) qs, matrix)*
shows *pcnf-assign (N x) (ExistentialFirst (x, xs) qs, matrix)*
 $= (\text{prefix-pop } (\text{ExistentialFirst } (x, xs) \ \text{qs}), \text{matrix-assign } (N \ x) \ \text{matrix})$
<proof>

lemma *pcnf-assign-n-all-eq*:

assumes *cleansed-p (UniversalFirst (y, ys) qs, matrix)*
shows *pcnf-assign (N y) (UniversalFirst (y, ys) qs, matrix)*
 $= (\text{prefix-pop } (\text{UniversalFirst } (y, ys) \ \text{qs}), \text{matrix-assign } (N \ y) \ \text{matrix})$
<proof>

theorem *pcnf-assign-cleansed-inv*:

$\text{cleansed-p } \text{pcnf} \implies \text{cleansed-p } (\text{pcnf-assign lit } \text{pcnf})$
<proof>

4.5.3 Cleansing PCNF Formulas

function *pcnf-cleanse* :: *pcnf* \Rightarrow *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)))

<proof>
termination
 <proof>

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*:
 $set\ (pcnf\text{-}prefix\text{-}variables\ (UniversalFirst\ (y,\ ys)\ qs,\ matrix))$
 $=\ \{y\} \cup set\ (pcnf\text{-}prefix\text{-}variables\ (prefix\text{-}pop\ (UniversalFirst\ (y,\ ys)\ qs),\ matrix))$
 <proof>

lemma *prefix-pop-ex-prefix-vars-set*:
 $set\ (pcnf\text{-}prefix\text{-}variables\ (ExistentialFirst\ (x,\ xs)\ qs,\ matrix))$
 $=\ \{x\} \cup set\ (pcnf\text{-}prefix\text{-}variables\ (prefix\text{-}pop\ (ExistentialFirst\ (x,\ xs)\ qs),\ matrix))$
 <proof>

lemma *cleanse-prefix-vars-inv*:
 $set\ (pcnf\text{-}prefix\text{-}variables\ (prefix,\ matrix))$
 $=\ set\ (pcnf\text{-}prefix\text{-}variables\ (pcnf\text{-}cleanse\ (prefix,\ matrix)))$
 <proof>

theorem *pcnf-cleanse-cleanses*:
 $cleansed\text{-}p\ (pcnf\text{-}cleanse\ pcnf)$
 <proof>

4.5.5 Cleansing Preserves the Set of Free Variables

lemma *prefix-pop-all-vars-inv*:
 $set\ (pcnf\text{-}variables\ (UniversalFirst\ (y,\ ys)\ qs,\ matrix))$
 $=\ set\ (pcnf\text{-}variables\ (prefix\text{-}pop\ (UniversalFirst\ (y,\ ys)\ qs),\ matrix))$
 <proof>

lemma *prefix-pop-ex-vars-inv*:
 $set\ (pcnf\text{-}variables\ (ExistentialFirst\ (x,\ xs)\ qs,\ matrix))$
 $=\ set\ (pcnf\text{-}variables\ (prefix\text{-}pop\ (ExistentialFirst\ (x,\ xs)\ qs),\ matrix))$
 <proof>

lemma *add-all-vars-inv*:
 $set\ (pcnf\text{-}variables\ (add\text{-}universal\text{-}to\text{-}front\ y\ pcnf))$
 $=\ set\ (pcnf\text{-}variables\ pcnf)$
 <proof>

lemma *add-ex-vars-inv*:
 $set\ (pcnf\text{-}variables\ (add\text{-}existential\text{-}to\text{-}front\ x\ pcnf))$
 $=\ set\ (pcnf\text{-}variables\ pcnf)$

$\langle \text{proof} \rangle$

lemma *cleanse-vars-inv*:

$\text{set } (\text{pcnf-variables } (\text{prefix}, \text{matrix}))$
 $= \text{set } (\text{pcnf-variables } (\text{pcnf-cleanse } (\text{prefix}, \text{matrix})))$
 $\langle \text{proof} \rangle$

theorem *cleanse-free-vars-inv*:

$\text{set } (\text{pcnf-free-variables } \text{pcnf})$
 $= \text{set } (\text{pcnf-free-variables } (\text{pcnf-cleanse } \text{pcnf}))$
 $\langle \text{proof} \rangle$

4.5.6 Cleansing Preserves Semantics

lemma *pop-redundant-ex-prefix-var-semantics-eq*:

assumes $x \in \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{ExistentialFirst } (x, xs) qs), \text{matrix}))$
shows $\text{pcnf-semantics } I (\text{ExistentialFirst } (x, xs) qs, \text{matrix})$
 $= \text{pcnf-semantics } I (\text{prefix-pop } (\text{ExistentialFirst } (x, xs) qs), \text{matrix})$
 $\langle \text{proof} \rangle$

lemma *pop-redundant-all-prefix-var-semantics-eq*:

assumes $y \in \text{set } (\text{pcnf-prefix-variables } (\text{prefix-pop } (\text{UniversalFirst } (y, ys) qs), \text{matrix}))$
shows $\text{pcnf-semantics } I (\text{UniversalFirst } (y, ys) qs, \text{matrix})$
 $= \text{pcnf-semantics } I (\text{prefix-pop } (\text{UniversalFirst } (y, ys) qs), \text{matrix})$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantics-disj-eq-add-ex*:

$\text{pcnf-semantics } (I(y := \text{True})) \text{pcnf} \vee \text{pcnf-semantics } (I(y := \text{False})) \text{pcnf}$
 $\longleftrightarrow \text{pcnf-semantics } I (\text{add-existential-to-front } y \text{pcnf})$
 $\langle \text{proof} \rangle$

lemma *pcnf-semantics-conj-eq-add-all*:

$\text{pcnf-semantics } (I(y := \text{True})) \text{pcnf} \wedge \text{pcnf-semantics } (I(y := \text{False})) \text{pcnf}$
 $\longleftrightarrow \text{pcnf-semantics } I (\text{add-universal-to-front } y \text{pcnf})$
 $\langle \text{proof} \rangle$

theorem *pcnf-cleanse-preserves-semantics*:

$\text{pcnf-semantics } I \text{pcnf} = \text{pcnf-semantics } I (\text{pcnf-cleanse } \text{pcnf})$
 $\langle \text{proof} \rangle$

theorem *sat-ex-first-iff-assign-disj-sat'*:

assumes $\text{cleansed-p } (\text{ExistentialFirst } (x, xs) qs, \text{matrix})$
shows $\text{satisfiable } (\text{convert } (\text{ExistentialFirst } (x, xs) qs, \text{matrix}))$
 $\longleftrightarrow \text{satisfiable } (\text{Disj}$
 $\quad [\text{convert } (\text{pcnf-assign } (P x) (\text{ExistentialFirst } (x, xs) qs, \text{matrix})),$
 $\quad \text{convert } (\text{pcnf-assign } (N x) (\text{ExistentialFirst } (x, xs) qs, \text{matrix}))])$

<proof>

theorem *sat-all-first-iff-assign-conj-sat'*:

assumes *cleansed-p* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)

shows *satisfiable* (*convert* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))

\longleftrightarrow *satisfiable* (*Conj*

[*convert* (*pcnf-assign* (*P* *y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)),

convert (*pcnf-assign* (*N* *y*) (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))])

<proof>

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*)

<proof>

lemma *add-ex-inc-prefix-measure*:

prefix-measure (*add-existential-to-prefix* *x* *prefix*) = *Suc* (*prefix-measure* *prefix*)

<proof>

lemma *remove-var-non-increasing-measure*:

prefix-measure (*remove-var-prefix* *z* *prefix*) \leq *prefix-measure* *prefix*

<proof>

fun *first-var* :: *prefix* \Rightarrow *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* \neq *Empty*

shows *prefix-measure* (*remove-var-prefix* (*the* (*first-var* *prefix*)) *prefix*) < *prefix-measure* *prefix*

<proof>

fun *first-existential* :: *prefix* \Rightarrow *bool* *option* **where**

first-existential (*ExistentialFirst* *q* *qs*) = *Some* *True*

| *first-existential* (*UniversalFirst* *q* *qs*) = *Some* *False*

| *first-existential* *Empty* = *None*

function *search* :: *pcnf* \Rightarrow *bool* *option* **where**

search (*prefix*, *matrix*) =

(*if* [] \in *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* (\vee)

(*search* (*pcnf-assign* (*P* *z*) (*prefix*, *matrix*)))

(*search* (*pcnf-assign* (*N* *z*) (*prefix*, *matrix*)))

```

      else combine-options ( $\wedge$ )
        (search (pcnf-assign (P z) (prefix, matrix)))
        (search (pcnf-assign (N z) (prefix, matrix))))))
    <proof>
termination
  <proof>

```

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  $\Rightarrow$  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 = []  $\implies$  pcnf-prefix-variables pcnf = []  $\implies$  pcnf-variables
  pcnf = []
  <proof>

```

```

lemma no-vars-if-no-free-empty-prefix:
  pcnf-free-variables (Empty, matrix) = []  $\implies$  pcnf-variables (Empty, matrix) = []
  <proof>

```

```

lemma search-cleansed-closed-yields-Some:
  assumes cleansed-p pcnf and pcnf-free-variables pcnf = []
  shows ( $\exists$  b. search pcnf = Some b) <proof>

```

```

theorem search-cleansed-closed-correct:
  assumes cleansed-p pcnf and pcnf-free-variables pcnf = []
  shows search pcnf = Some (satisfiable (convert pcnf)) <proof>

```

4.6.2 Correctness of the Search Solver

```

theorem search-solver-correct:

```

```
    search-solver pcnf  $\longleftrightarrow$  satisfiable (convert pcnf)
  <proof>
```

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  $\Rightarrow$  bool where
```

```
  run-naive-solver qdimacs-str = naive-solver (convert (the (parse qdimacs-str)))
```

```
fun run-search-solver :: String.literal  $\Rightarrow$  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.