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*⟩

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*⟩

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*))
∨ (*qbf-semantics I* (*substitute-var x False qbf*)))
| *qbf-semantics I* (*All x qbf*) = ((*qbf-semantics I* (*substitute-var x True qbf*))
∧ (*qbf-semantics I* (*substitute-var x False qbf*)))
⟨*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-semantics* (λ*x. False*) *test-qbf*
value *qbf-semantics* ((λ*x. False*)(*2* := *True*)) *test-qbf*
value *qbf-semantics* (((λ*x. False*)(*2* := *True*))(*1* := *True*)) *test-qbf*

1.1.3 Formalisation of Satisfiability

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

definition *logically-eq* :: *QBF* ⇒ *QBF* ⇒ *bool* **where**
logically-eq qbf1 qbf2 = (∀ *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* ⇒ *QBF* ⇒ *nat list* **where**
free-variables-aux bound (*Var x*) = (*if x* ∈ *bound then* [] *else* [*x*])
| *free-variables-aux bound* (*Neg qbf*) = *free-variables-aux bound qbf*
| *free-variables-aux bound* (*Conj list*) = *concat* (*map* (*free-variables-aux bound*) *list*)
| *free-variables-aux bound* (*Disj list*) = *concat* (*map* (*free-variables-aux bound*) *list*)
| *free-variables-aux bound* (*Ex x qbf*) = *free-variables-aux* (*insert x bound*) *qbf*
| *free-variables-aux bound* (*All x qbf*) = *free-variables-aux* (*insert x bound*) *qbf*

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

lemma *bound-subtract-equiv*:
set (*free-variables-aux* (*bound* ∪ *new*) *qbf*) = *set* (*free-variables-aux bound qbf*) − *new*
⟨*proof*⟩

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*⟩

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*) ⟷ *qbf-semantics* (*I*(*x* := *b*)) *qbf*
⟨*proof*⟩

lemma *sat-iff-ex-sat*: *satisfiable qbf* ⟷ *satisfiable* (*Ex x qbf*)
⟨*proof*⟩

1.2.4 Preservation of Satisfiability under Existential Closure

lemma *sat-iff-ex-close-aux-sat*: *satisfiable qbf* ⟷ *satisfiable* (*existential-closure-aux qbf vars*)
⟨*proof*⟩

theorem *sat-iff-ex-close-sat*: *satisfiable qbf* ⟷ *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* (λ*x*. *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-quantifier-depth z* = *list-max* (*map qbf-quantifier-depth qbf-list*) ⟨*proof*⟩

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*)])
⟨*proof*⟩
termination
⟨*proof*⟩

Property 1: no quantifiers after expansion.

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

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

lemma *semantics-inv-under-expand*:
qbf-semantics I qbf = *qbf-semantics I* (*expand-quantifiers qbf*)
⟨*proof*⟩

lemma *sat-iff-expand-quants-sat*: *satisfiable qbf* ⟷ *satisfiable* (*expand-quantifiers qbf*)
⟨*proof*⟩

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*⟩

lemma *set-free-vars-subst-ex-eq*:
set (*free-variables* (*substitute-var x b qbf*)) = *set* (*free-variables* (*Ex x qbf*))
⟨*proof*⟩

lemma *free-vars-inv-under-expand-quants*:
set (*free-variables* (*expand-quantifiers qbf*)) = *set* (*free-variables qbf*)
⟨*proof*⟩

1.4.2 Expanding Formulas

fun *expand-qbf* :: *QBF* ⇒ *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*
⟨*proof*⟩

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

lemma *expand-qbf-no-quants*: *qbf-quantifier-depth* (*expand-qbf qbf*) = *0*
⟨*proof*⟩

1.4.3 Evaluating Expanded Formulas

fun *eval-qbf* :: *QBF* ⇒ *bool option* **where**
eval-qbf (*Var x*) = *None* |
eval-qbf (*Neg qbf*) = *map-option* (λ*x*. ¬*x*) (*eval-qbf qbf*) |
eval-qbf (*Conj list*) = *map-option* (*list-all id*) (*sequence* (*map eval-qbf list*)) |
eval-qbf (*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* ∘ *f*) *x*
⟨*proof*⟩

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*) ⟨*proof*⟩

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* $\longleftrightarrow$ *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* (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*)
⟨*proof*⟩

lemma *convert-add-ex*: *convert* (*add-existential-to-front x pcnf*) = *Ex x* (*convert pcnf*)
⟨*proof*⟩

theorem *convert-inv*: *convert-inv* (*convert pcnf*) = *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 ∃ *l*. *convert-literal l* = *lit*
⟨*proof*⟩

lemma *convert-clause-p-ex*:
assumes *clause-p cl*
shows ∃ *c*. *convert-clause c* = *cl*
⟨*proof*⟩

lemma *convert-cnf-p-ex*:
assumes *cnf-p mat*
shows ∃ *m*. *convert-matrix m* = *mat*
⟨*proof*⟩

theorem *convert-pcnf-p-ex*:
assumes *pcnf-p qbf*
shows ∃ *pcnf*. *convert pcnf* = *qbf* ⟨*proof*⟩

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

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

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*))
  ⟨*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 ⟹ *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'* $\Longrightarrow$ *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'*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*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'*) $\Longrightarrow$ *length s'* $\leq$ *length s*
⟨*proof*⟩

lemma *decreasing-match* [*simp*]:
xs $\neq$ [] $\Longrightarrow$ *match xs s* = *Some* ((), *s'*) $\Longrightarrow$ *length s'* $<$ *length s*

⟨*proof*⟩

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*
⟨*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′*) ⟹ *length s′* < *length s*
⟨*proof*⟩

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*⟩

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 ⇒ *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 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''
value *clause-list* ''1 2 −3 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''
value *clause-list* ''1 2 −3 0 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''

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 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''
value *matrix* ''1 2 −3 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''
value *matrix* ''1 2 −3 0 ↩ 1 −2 3 0 ↩ −1 2 3 0 ↩ ''

lemma *decreasing-matrix* [*simp*]: *matrix s* = *Some* (*mat*, *s'*) ⟹ *length s'* < *length s* ⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩

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⟩


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′*) ⟹ *s′* = *Nil*
  ⟨*proof*⟩

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* ⟨*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* ⇒ *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*
⟨*proof*⟩

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*))))
⟨*proof*⟩
termination
⟨*proof*⟩

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*
⟹ *free-assgn* ⊆ *free* − *lit*
⟨*proof*⟩

4.2.2 Free Variables is Variables without Prefix Variables

lemma *lit-p-free-eq-vars*:
literal-p qbf ⟹ *set* (*free-variables qbf*) = *set* (*variables qbf*)
⟨*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 ⟹ *set* (*free-variables qbf*) = *set* (*variables qbf*) − *set* (*prefix-variables-aux qbf*)
⟨*proof*⟩

lemma *pcnf-p-free-eq-vars-minus-prefix*:
pcnf-p qbf ⟹ *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-variables pcnf*) − *set* (*pcnf-prefix-variables pcnf*)
⟨*proof*⟩

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*⟩

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*}
⟨*proof*⟩

lemma *pcnf-vars-eq-matrix-vars*:
set (*pcnf-variables* (*prefix*, *matrix*))
= *set* (*variables* (*convert-matrix matrix*))
⟨*proof*⟩

lemma *pcnf-assign-vars-subseteq-vars-minus-lit*:
set (*pcnf-variables* (*pcnf-assign x pcnf*))
⊆ *set* (*pcnf-variables pcnf*) − {*lit-var x*}
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *add-ex-to-prefix-eq-add-to-front*:
(*add-existential-to-prefix x prefix*, *matrix*) = *add-existential-to-front x* (*prefix*, *matrix*)
⟨*proof*⟩

lemma *add-all-adds-prefix-var*:
set (*pcnf-prefix-variables* (*add-universal-to-front x pcnf*))
= *set* (*pcnf-prefix-variables pcnf*) ∪ {*x*}
⟨*proof*⟩

lemma *add-all-to-prefix-eq-add-to-front*:
(*add-universal-to-prefix x prefix*, *matrix*) = *add-universal-to-front x* (*prefix*, *matrix*)
⟨*proof*⟩

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*⟩

lemma *prefix-vars-matrix-inv*:
set (*pcnf-prefix-variables* (*prefix*, *matrix1*))
= *set* (*pcnf-prefix-variables* (*prefix*, *matrix2*))
⟨*proof*⟩

lemma *pcnf-prefix-vars-eq-prefix-minus-lit*:
set (*pcnf-prefix-variables* (*pcnf-assign x pcnf*))
= *set* (*pcnf-prefix-variables pcnf*) − {*lit-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-free-variables* (*pcnf-assign x pcnf*)) ⊆ *set* (*pcnf-free-variables pcnf*) − {*lit-var x*}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ex-closure-pcnf-p-inv*:
pcnf-p qbf ⟹ *pcnf-p* (*existential-closure qbf*)
⟨*proof*⟩

theorem *pcnf-sat-iff-ex-close-sat*:
satisfiable (*convert pcnf*) = *satisfiable* (*convert* (*pcnf-existential-closure pcnf*))
⟨*proof*⟩

4.3.3 No Free Variables in PCNF Existential Closure

theorem *pcnf-ex-closure-no-free*:
pcnf-free-variables (*pcnf-existential-closure pcnf*) = []
⟨*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*)
⟨*proof*⟩

lemma *empty-prefix-cons-matrix-variables*:
set (*pcnf-variables* (*Empty*, *Cons cl cls*))
= *set* (*pcnf-variables* (*Empty*, *cls*)) ∪ *set* (*map lit-var cl*)
⟨*proof*⟩

lemma *false-if-empty-clause-in-matrix*:
[] ∈ *set matrix* ⟹ *pcnf-semantics I* (*prefix*, *matrix*) = *False*
⟨*proof*⟩

lemma *true-if-matrix-empty*:
matrix = [] ⟹ *pcnf-semantics I* (*prefix*, *matrix*) = *True*
⟨*proof*⟩

lemma *matrix-shape-if-no-variables*:
pcnf-variables (*Empty*, *matrix*) = [] ⟹ (∃ *n*. *matrix* = *replicate n* [])
⟨*proof*⟩

lemma *empty-clause-or-matrix-if-no-variables*:
pcnf-variables (*Empty*, *matrix*) = [] ⟹ [] ∈ *set matrix* ∨ *matrix* = []
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *clause-semantics-inv-remove-true*:
clause-semantics (*I*(*z* := *False*)) *cl* = *clause-semantics* (*I*(*z* := *False*)) (*remove-lit-neg* (*N z*) *cl*)
⟨*proof*⟩

lemma *matrix-semantics-inv-remove-true*:
matrix-semantics (*I*(*z* := *True*)) (*matrix-assign* (*P z*) *matrix*)
= *matrix-semantics* (*I*(*z* := *True*)) *matrix*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *matrix-semantics-inv-remove-false*:
matrix-semantics $(I(z := False))$ (*matrix-assign* $(N\ z)$ *matrix*)
= *matrix-semantics* $(I(z := False))$ *matrix*
⟨*proof*⟩

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*
⟨*proof*⟩

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* $(I(z := False))$ (*matrix-assign* $(N\ z)$ *matrix*)
⟨*proof*⟩

lemma *matrix-semantics-conj-iff-true-assgn*:
($\forall\, b$. *matrix-semantics* $(I(z := b))$ *matrix*)
$\longleftrightarrow$ *matrix-semantics* $(I(z := True))$ (*matrix-assign* $(P\ z)$ *matrix*)
$\wedge$ *matrix-semantics* $(I(z := False))$ (*matrix-assign* $(N\ z)$ *matrix*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *neq-first-if-notin-all-prefix*:
$z \notin$ *set* (*pcnf-prefix-variables* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*)) $\Longrightarrow z \neq y$
⟨*proof*⟩

lemma *neq-first-if-notin-ex-prefix*:
$z \notin$ *set* (*pcnf-prefix-variables* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)) $\Longrightarrow z \neq x$
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pcnf-semantics-inv-matrix-assign-true*:
assumes $z \notin$ *set* (*pcnf-prefix-variables* (*prefix*, *matrix*))
shows *pcnf-semantics* (I(z := *True*)) (*prefix*, *matrix-assign* (P z) *matrix*)
= *pcnf-semantics* (I(z := *True*)) (*prefix*, *matrix*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
∨ *pcnf-semantics* (I(z := *False*)) (*prefix*, *matrix*)
⟷
pcnf-semantics (I(z := *True*)) (*prefix*, *matrix-assign* (P z) *matrix*)
∨ *pcnf-semantics* (I(z := *False*)) (*prefix*, *matrix-assign* (N z) *matrix*)
⟨*proof*⟩

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*)
∧ *pcnf-semantics* (I(z := *False*)) (*prefix*, *matrix*)
⟷
pcnf-semantics (I(z := *True*)) (*prefix*, *matrix-assign* (P z) *matrix*)
∧ *pcnf-semantics* (I(z := *False*)) (*prefix*, *matrix-assign* (N z) *matrix*)
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

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

$x \notin$ *set* (*pcnf-variables* (*pcnf-assign* (*P x*) *pcnf*))
⟨*proof*⟩

lemma *x-notin-assign-N-x*:
$x \notin$ *set* (*pcnf-variables* (*pcnf-assign* (*N x*) *pcnf*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟷ *satisfiable* (*convert* (*pcnf-assign* (*P x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)))
∨ *satisfiable* (*convert* (*pcnf-assign* (*N x*) (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*)))
⟨*proof*⟩

theorem *sat-ex-first-iff-assign-disj-sat*:
assumes $x \notin$ *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*))])
⟨*proof*⟩

theorem *sat-all-first-iff-assign-conj-sat*:
assumes $y \notin$ *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*⟩

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*)
⟨*proof*⟩

lemma *prefix-variables-aux-matrix-inv*:
prefix-variables-aux (*convert* (*prefix*, *matrix1*))
= *prefix-variables-aux* (*convert* (*prefix*, *matrix2*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *cleansed-p-matrix-inv*:
cleansed-p (*prefix*, *matrix1*) = *cleansed-p* (*prefix*, *matrix2*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
  ⟨*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*)
      = (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix-assign* (*P x*) *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*)
      = (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix-assign* (*P y*) *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*)
      = (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix-assign* (*N x*) *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*)
      = (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix-assign* (*N y*) *matrix*)
  ⟨*proof*⟩

theorem *pcnf-assign-cleansed-inv*:
  cleansed-p pcnf ⟹ *cleansed-p* (*pcnf-assign lit pcnf*)
⟨*proof*⟩

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* ∈ *set* (*pcnf-prefix-variables* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*))
      then pcnf-cleanse (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*)
      else add-universal-to-front y
        (*pcnf-cleanse* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*)))
| *pcnf-cleanse* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*) =
    (*if x* ∈ *set* (*pcnf-prefix-variables* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*))
      then pcnf-cleanse (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*)
      else add-existential-to-front x
        (*pcnf-cleanse* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*)))

⟨*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-prefix-variables* (*UniversalFirst* (*y*, *ys*) *qs*, *matrix*))
= {*y*} ∪ *set* (*pcnf-prefix-variables* (*prefix-pop* (*UniversalFirst* (*y*, *ys*) *qs*), *matrix*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *cleanse-prefix-vars-inv*:
set (*pcnf-prefix-variables* (*prefix*, *matrix*))
= *set* (*pcnf-prefix-variables* (*pcnf-cleanse* (*prefix*, *matrix*)))
⟨*proof*⟩

theorem *pcnf-cleanse-cleanses*:
cleansed-p (*pcnf-cleanse pcnf*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *prefix-pop-ex-vars-inv*:
set (*pcnf-variables* (*ExistentialFirst* (*x*, *xs*) *qs*, *matrix*))
= *set* (*pcnf-variables* (*prefix-pop* (*ExistentialFirst* (*x*, *xs*) *qs*), *matrix*))
⟨*proof*⟩

lemma *add-all-vars-inv*:
set (*pcnf-variables* (*add-universal-to-front y pcnf*))
= *set* (*pcnf-variables pcnf*)
⟨*proof*⟩

lemma *add-ex-vars-inv*:
set (*pcnf-variables* (*add-existential-to-front x pcnf*))
= *set* (*pcnf-variables pcnf*)

⟨*proof*⟩

lemma *cleanse-vars-inv*:
set (*pcnf-variables* (*prefix*, *matrix*))
= *set* (*pcnf-variables* (*pcnf-cleanse* (*prefix*, *matrix*)))
⟨*proof*⟩

theorem *cleanse-free-vars-inv*:
set (*pcnf-free-variables pcnf*)
= *set* (*pcnf-free-variables* (*pcnf-cleanse pcnf*))
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *pcnf-cleanse-preserves-semantics*:
pcnf-semantics I *pcnf* = *pcnf-semantics* I (*pcnf-cleanse pcnf*)
⟨*proof*⟩

theorem *sat-ex-first-iff-assign-disj-sat'*:
assumes *cleansed-p* (*ExistentialFirst* (x, *xs*) *qs*, *matrix*)
shows *satisfiable* (*convert* (*ExistentialFirst* (x, *xs*) *qs*, *matrix*))
$\longleftrightarrow$ *satisfiable* (*Disj*
[*convert* (*pcnf-assign* (P x) (*ExistentialFirst* (x, *xs*) *qs*, *matrix*)),
convert (*pcnf-assign* (N x) (*ExistentialFirst* (x, *xs*) *qs*, *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*))
⟷ *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*) ≤ *prefix-measure prefix*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))))))
⟨*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* ⇒ *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* = []
⟨*proof*⟩

lemma *no-vars-if-no-free-empty-prefix*:
pcnf-free-variables (*Empty*, *matrix*) = [] ⟹ *pcnf-variables* (*Empty*, *matrix*) = []
⟨*proof*⟩

lemma *search-cleansed-closed-yields-Some*:
assumes *cleansed-p pcnf* **and** *pcnf-free-variables pcnf* = []
shows (∃ *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 ⟷ *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* ⇒ *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.