Verified QBF Solving in Isabelle/HOL: A Trustworthy Baseline for Mechanised Quantified Boolean Reasoning

Axel Bergström[0009−0009−2968−9167] and Tjark Weber[0000−0001−8967−6987]

Department of Information Technology, Uppsala University, Sweden
{axel.bergstrom,tjark.weber}@it.uu.se

Abstract Quantified Boolean formulas (QBFs) compactly encode verification, synthesis, planning, and game-solving problems, but efficient QBF solvers are highly optimised tools whose correctness is difficult to assess. This paper presents, to our knowledge, the first mechanically verified QBF solvers in an interactive theorem prover. We formalise QBF syntax and semantics in Isabelle/HOL; prove correct an expansion-based decision procedure for arbitrary QBFs; formalise a prenex-CNF representation corresponding to the QDIMACS benchmark format; and prove correct a search-based solver for prenex-CNF QBFs. Executable Standard ML, OCaml, Haskell, and Scala code is generated from the verified definitions and connected to an Isabelle-implemented QDIMACS parser. Our aim is not to compete with state-of-the-art QBF solvers, but to provide a trustworthy baseline and reusable formal infrastructure for mechanised quantified Boolean reasoning. The development identifies verification issues that are easy to miss in informal presentations—notably the treatment of free variables, repeated binders, representation changes between general QBFs and prenex CNF, and totality/termination obligations for executable functions. Experiments confirm the expected performance gap to CAQE while demonstrating that the verified search solver solves non-trivial benchmark families and can serve as a reference implementation for future verified QBF algorithms.

Keywords: Quantified Boolean formulas · Verified solver · Isabelle/HOL · Code generation · QDIMACS

1 Introduction

Quantified Boolean formulas extend propositional logic with quantification over Boolean variables. A formula such as

$$\forall x.\ \exists y.\ ((x \vee z) \wedge y)$$

contains a universally quantified variable x, an existentially quantified variable y, and a free variable z. It is satisfiable because assigning z to true makes the quantified subformula true independently of the value of x: after x has been

chosen, the existential player can choose y to be true. This simple example already illustrates two aspects that are important for verified solvers. First, satisfiability is defined with respect to assignments of free variables, not only with respect to closed formulas. Second, quantifier order matters and generally cannot be eliminated by a naive appeal to propositional satisfiability; in a more complicated QBF, the value chosen for y may need to depend on the value of x.

QBF satisfiability is the canonical PSPACE-complete problem [2]. The succinctness of quantified Boolean logic makes it a useful target language for problems involving alternation, including planning [28,35], games [1], verification [9,16], FPGA synthesis [20], and many other applications [29]. In such settings, a QBF solver may be part of a larger verification or synthesis chain, and an incorrect answer may invalidate the result produced by the whole chain. Unfortunately, solver correctness is difficult to establish by testing alone; disagreement between solvers has occurred in QBF evaluations [25].

One response is certificate generation. Some QBF solvers can emit certificates whose validity can be checked independently [34]. Certificate checking is highly valuable, but it does not eliminate the need for verified decision procedures: certificates have worst-case super-polynomial size (unless PSPACE = NP = coNP) [8], not every solver run necessarily yields a usable certificate, and the trusted base of certificate-based verification includes a certificate format and checker. A fully verified solver offers a different point in the design space. It may be slower than a competitive unverified solver, but its answers are backed by a machine-checked proof of the solver's correctness. Such a solver can also serve as an executable specification, a regression oracle, and a platform for verifying more advanced algorithms.

Contributions and positioning. This paper reports a complete Isabelle/HOL development of trustworthy QBF solving. The algorithms are intentionally modest: the purpose is not to claim a performance advance in QBF solving, but to establish a carefully proved baseline that makes future verified QBF work possible. Concretely, we contribute:

- a deep embedding of QBF syntax and semantics, including free variables, substitution, quantifier depth, and existential closure;
- a verified expansion-based solver for arbitrary QBFs;
- a formal prenex-CNF representation corresponding to QDIMACS, with semantics-preserving conversions to and from the general QBF representation;
- a verified search-based solver for prenex-CNF QBFs, including a verified cleansing transformation that removes repeated prefix variables;
- executable solver pipelines obtained by Isabelle code generation; and
- an experimental evaluation of the generated solvers, comparing the expansion and search approaches and calibrating them against the QBF solver CAQE.

The work is, in this sense, a verification case study and an enabling infrastructure contribution. The mechanisation exposes proof obligations that paper descriptions of QBF procedures often hide: formulas may have free variables; repeated binders require care; QDIMACS input may require normalization; conversion between

syntactic representations must preserve meaning; and recursive search must be justified by a well-founded measure. These issues are not merely engineering details: each presents an opportunity for an implementation to silently diverge from the intended semantics.

Artifact status. The formalisation consists of executable definitions and machine-checked theorems in Isabelle/HOL. It is organised around reusable layers: a formula layer, a closure and expansion layer, a prenex-CNF layer, a parser layer, and the search-solver layer. The parser and solvers are executable and terminating, and the formalisation is available from the Archive of Formal Proofs [5].

Outline. Section 2 presents the formalised QBF syntax and semantics. Section 3 describes the verified expansion solver. Section 4 introduces the prenex-CNF representation, its connection to QDIMACS, and the conversion theorems. Section 5 gives the verified search solver and its key invariants. Section 6 reports executable-code generation and experiments. Section 7 discusses lessons for future verified QBF solvers, Section 8 discusses related work, and Section 9 concludes.

2 QBFs in Isabelle/HOL

We formalise QBFs by a deep embedding. Our definitions follow standard presentations in the literature [12,17] while adopting total functions and list-based representations to suit Isabelle/HOL. The QBF datatype is deliberately small:

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

Variables are natural numbers. Empty conjunctions and disjunctions represent true and false, respectively. This choice avoids special cases for constants and matches the usual interpretation of finite conjunctions and disjunctions. The type does not enforce any normal form and can represent arbitrary QBFs, including formulas with free variables, nested negations, non-prenex quantifiers, empty connectives, and repeated binders.

We avoid a well-formedness predicate and instead rely on total functions with invariants established by subsequent theorems. For example, the evaluator in Section 3 is total and returns an option type; it is proved to return a Boolean on the closed, quantifier-free formulas produced by expansion. This style fits Isabelle/HOL's logic, where all functions are total and executable definitions are most convenient when they do not depend on external side conditions.

2.1 Semantics and substitution

An interpretation is a function from variables to Booleans. The semantic function maps an interpretation and a formula to a Boolean:

```
function qbf_semantics :: "(nat ⇒ bool) ⇒ QBF ⇒ bool" where
  "qbf_semantics I (Var z) = I z"
| "qbf_semantics I (Neg F) = ¬(qbf_semantics I F)"
| "qbf_semantics I (Conj Fs) =
    list_all (qbf_semantics I) Fs"
| "qbf_semantics I (Disj Fs) =
    list_ex (qbf_semantics I) Fs"
| "qbf_semantics I (Ex x F) =
    qbf_semantics I (substitute_var x True F) ∨
    qbf_semantics I (substitute_var x False F)"
| "qbf_semantics I (All x F) =
    qbf_semantics I (substitute_var x True F) ∧
    qbf_semantics I (substitute_var x False F)"
```

For conjunctions, `list_all` is used to check if all conjuncts are true under the given interpretation. Dually, for disjunctions `list_ex` is used to check if at least one disjunct is true. It follows that the semantics of the empty conjunction (abbreviated by `1`) is `True`, while the semantics of the empty disjunction (abbreviated by `0`) is `False`. The quantifier equations use a substitution function `substitute_var` to replace occurrences of the quantified variable `x` within the scope of the quantifier by `0` or `1`. Since the formulas `0` and `1` contain no variables, variable capture cannot occur. Existential and universal quantification correspond to disjunction and conjunction, respectively. Shadowing is made explicit because substitution does not enter scopes that rebind the same variable.

All functions in Isabelle/HOL are total, and we need to prove that the recursive definition of `qbf_semantics` is terminating. To this end, we have shown that applications of `substitute_var` do not change the number of `QBF` constructors in a formula, and that this number is therefore decreasing in the second argument of `qbf_semantics` with each recursive call.

The central lemma connecting substitution and semantic update is

```
lemma qbf_semantics_substitute_eq_assign:
  "qbf_semantics I (substitute_var x b F) =
    qbf_semantics (I(x := b)) F"
```

Here, `I(x := b)` denotes the interpretation `I` except that `x` maps to `b`. The lemma, which is proved by induction over the formula, is a small but important result in our development. It is used to justify quantifier expansion and the assignment operation in the search solver. It also provides a sanity check on the definitions.

Satisfiability is defined by existential quantification over interpretations:

```
definition satisfiable :: "QBF ⇒ bool" where
  "satisfiable F = (∃I. qbf_semantics I F)
```

For closed formulas, the choice of interpretation is irrelevant; for open formulas, this definition implements the usual convention that free variables are existentially assigned. This definition is concise but not executable, because it quantifies over the infinite function type `nat ⇒ bool`. Both our solvers avoid that issue by turning the input formula into a closed problem.

2.2 Free variables

Free variables are formalised by a recursive auxiliary function that carries the set of variables considered bound:

```
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 F) =
    free_variables_aux bound F"
| "free_variables_aux bound (Conj Fs) =
    concat (map (free_variables_aux bound) Fs)"
| "free_variables_aux bound (Disj Fs) =
    concat (map (free_variables_aux bound) Fs)"
| "free_variables_aux bound (Ex x F) =
    free_variables_aux (insert x bound) F"
| "free_variables_aux bound (All x F) =
    free_variables_aux (insert x bound) F"

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

The wrapper removes duplicates and fixes a deterministic order. A fixed order ensures determinism of the generated solver. Lists are favoured over finite sets, as lists are more directly executable and can easily be converted to sets when convenient for proofs.

The definition also clarifies the treatment of repeated names. In the formula $(\forall x.\, x \wedge y) \vee (x \wedge z)$, the variable x is considered bound in the sub-formula $x \wedge y$ but not in $x \wedge z$, so x is a free variable. This form of lexical scoping occurring in arbitrary QBFs is exactly what the accumulator-based definition computes. In contrast, prenex representations lack the ability to directly express similar scoping; Section 5 addresses this via a verified cleansing transformation.

2.3 Existential closure

To obtain an executable decision procedure for satisfiability, the free variables of a formula are bound existentially:

```
fun existential_closure_aux :: "QBF ⇒ nat list ⇒ QBF" where
  "existential_closure_aux F Nil = F"
| "existential_closure_aux F (Cons x xs) =
    Ex x (existential_closure_aux F xs)"

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

If $x_1, \ldots, x_n$ are the free variables of Φ, then the closure is $\exists x_1 \ldots \exists x_n.\, \Phi$, up to the particular list order chosen by `free_variables`. The two fundamental theorems that characterise the existential closure are:

```
theorem sat_iff_ex_closure_sat:
  "satisfiable F ↔ satisfiable (existential_closure F)"

theorem ex_closure_no_free:
  "set (free_variables (existential_closure F)) = {}"
```

The first theorem follows from the fact that satisfiability already existentially quantifies over interpretations; adding explicit existential binders for all free variables preserves that property. The second theorem states that the closure contains no free variables. Although this result appears almost trivial, the formal proof requires precise lemmas about how the result of `free_variables_aux` changes when a variable is inserted into the bound set.

3 A Verified Expansion Solver

The expansion solver is the simplest complete decision procedure in our development. It demonstrates that the semantic infrastructure suffices for an end-to-end correctness proof and serves as a baseline for later solvers.

3.1 Quantifier expansion

The solver replaces existential quantification by disjunction over both Boolean values and universal quantification by conjunction over both Boolean values:

$$(\exists x.\, \Phi) \;\equiv\; \Phi[1/x] \vee \Phi[0/x], \qquad (\forall x.\, \Phi) \;\equiv\; \Phi[1/x] \wedge \Phi[0/x].$$

The corresponding Isabelle definition traverses the formula recursively:

```
fun expand_quantifiers :: "QBF ⇒ QBF" where
  "expand_quantifiers (Var x) = (Var x)"
| "expand_quantifiers (Neg F) =
    Neg (expand_quantifiers F)"
| "expand_quantifiers (Conj Fs) =
    Conj (map expand_quantifiers Fs)"
| "expand_quantifiers (Disj Fs) =
    Disj (map expand_quantifiers Fs)"
| "expand_quantifiers (Ex x F) =
    (Disj [substitute_var x True (expand_quantifiers F),
           substitute_var x False (expand_quantifiers F)])"
| "expand_quantifiers (All x F) =
    (Conj [substitute_var x True (expand_quantifiers F),
           substitute_var x False (expand_quantifiers F)])"
```

We establish the following key properties of this function: it produces a quantifier-free (i.e., purely propositional) formula, preserves the semantics of the original formula—and hence satisfiability—and leaves the set of free variables unchanged.

```
lemma no_quantifiers_after_expand:
  "qbf_quantifier_depth (expand_quantifiers F) = 0"
```

```
lemma semantics_inv_under_expand:
  "qbf_semantics I F =
    qbf_semantics I (expand_quantifiers F)"

lemma free_vars_inv_under_expand:
  "set (free_variables (expand_quantifiers F)) =
    set (free_variables F)"
```

Here, `qbf_quantifier_depth` is defined in the standard way by structural recursion on formulas. All three lemmas are proved by induction over the formula `F`. The quantifier cases in the semantic preservation theorem use the substitution/update lemma from Section 2. Together with existential closure, these results show that expansion produces a closed propositional formula with the same satisfiability status as the original QBF.

3.2 Evaluation of expanded formulas

After closure and expansion, a formula contains neither variables nor quantifiers. It can be evaluated by a structurally recursive function:

```
fun eval_qbf :: "QBF ⇒ bool option" where
  "eval_qbf (Var x) = None"
| "eval_qbf (Neg F) = map_option (λx. ¬x) (eval_qbf F)"
| "eval_qbf (Conj Fs) =
    map_option (list_all id) (sequence (map eval_qbf Fs))"
| "eval_qbf (Disj Fs) =
    map_option (list_ex id) (sequence (map eval_qbf Fs))"
| "eval_qbf (Ex x F) = None"
| "eval_qbf (All x F) = None"
```

The function returns `None` on variables or quantifiers. This makes the function total while explicitly indicating its intended domain. For conjunctions and disjunctions, the auxiliary function `sequence` converts a list of `option` values into an optional list. The correctness theorem shows that the exceptional cases do not arise for closed, quantifier-free formulas; on such inputs, `eval_qbf` computes the semantics:

```
lemma eval_qbf_implements_semantics:
  assumes "set (free_variables F) = {}"
    and "qbf_quantifier_depth F = 0"
  shows "eval_qbf F = Some (qbf_semantics I F)"
```

Since the formula contains neither variables nor quantifiers, its value is independent of the interpretation `I`.

The final solver is the composition of closure, expansion, and evaluation:

```
fun expand_qbf :: "QBF ⇒ QBF" where
  "expand_qbf F = expand_quantifiers (existential_closure F)"

fun expansion_solver :: "QBF ⇒ bool" where
  "expansion_solver F = the (eval_qbf (expand_qbf F))"
```

The use of `the`—which extracts a Boolean from an optional value but is undefined on `None`—is safe here, as the preceding theorem guarantees that evaluation returns `Some b`. The following theorem shows the correctness of this solver:

```
theorem expansion_solver_correct:
  "expansion_solver F ↔ satisfiable F"
```

It holds because `eval_qbf` implements the semantics for closed, quantifier-free formulas, like those returned by `expand_qbf`, and expansion preserves satisfiability.

Limitations and advantages of the expansion solver. The expansion solver incurs an exponential blow-up in formula size in the number of quantified variables, since each quantifier duplicates its subformula. This reflects the inherent worst-case space complexity of full expansion. Nevertheless, verifying this solver is worthwhile: its proof is compact, it applies to arbitrary QBF syntax rather than only to prenex CNF, and it exercises the central semantic lemmas. It also provides a useful experimental baseline by establishing an end-to-end correctness theorem before introducing algorithmic sophistication.

4 Prenex CNF and QDIMACS Infrastructure

QBF benchmarks are typically supplied in QDIMACS format [26], which represents formulas in prenex conjunctive normal form: a quantifier prefix followed by a propositional matrix in conjunctive normal form. Although every PCNF formula can be embedded into the general `QBF` datatype, a dedicated representation makes parsing and search more natural.

4.1 The PCNF datatype

Literals are positive or negative variables. Clauses (i.e., disjunctions of literals) and matrices (i.e., conjunctions of clauses) are represented by lists:

```
datatype literal = P nat | N nat
type_synonym clause = "literal list"
type_synonym matrix = "clause list"
```

This suffices to represent propositional formulas in conjunctive normal form. A `quant_set` stores a quantifier block: the first component is the first variable of the block and the list component contains any other quantified variables. The `prefix` constructors indicate whether the first block is universal or existential; subsequent blocks alternate. This mirrors QDIMACS, where a prefix is a sequence of quantifier lines, each binding one or more variables.

```
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"
```

For example, the formula $\forall x_1\, x_2.\ \exists x_3.\ (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$ is represented by a universal-first prefix containing the block `(1, [2])`, followed by the existential block `(3, [])`, and by the matrix `[[P 1, N 3], [P 2, P 3]]`.

The datatype is close to the input format but does not itself enforce every convention that a benchmark suite might impose. For instance, it can represent repeated variables in a prefix. We deliberately handle such cases semantically rather than ruling them out at the type level. This choice keeps the parser simple and lets the solver perform a verified normalisation step before search.

4.2 Semantics-preserving conversion

The development defines an embedding `convert` of type `pcnf ⇒ QBF`. It translates the prefix into nested quantifiers and the matrix into a conjunction of disjunctions of literals. A partial inverse `convert_inv` of type `QBF ⇒ pcnf option` recognises QBFs that are in prenex CNF. A predicate `pcnf_p` characterises the same subset of the general representation. The main structural facts are:

```
theorem convert_pcnf_p:
  "pcnf_p (convert P)"

theorem convert_inv:
  "convert_inv (convert P) = Some P"

theorem convert_pcnf_p_ex:
  assumes "pcnf_p F" shows "∃P. convert P = F"
```

The PCNF semantics is defined directly over prefixes and matrices, again using interpretations. Following the conventions for conjunctions and disjunctions in the general QBF representation, the empty matrix is `True` under any interpretation while the empty clause is `False`. The connection to the general semantics is then established by:

```
theorem pcnf_semantics_eq_qbf_semantics:
  "pcnf_semantics I P = qbf_semantics I (convert P)"
```

This theorem is important because the search solver is most naturally expressed over `pcnf`, while the top-level notion of satisfiability is defined for `QBF`. The theorem allows us to prove algorithmic facts over the efficient representation and then state user-facing correctness in terms of the general semantics.

4.3 Parsing QDIMACS

The parser is implemented as a collection of executable HOL functions of type `string ⇒ ('a × string) option`. Each parser either fails or returns a parsed value together with the unconsumed suffix of the input. As an illustrative example, the parser for clause lists is given by:

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

```
fun clause_list :: "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'')))"
```

Termination of parser functions is proved by showing that successful subparsers consume input, so recursive calls are made on shorter strings. This is another example where executable verification exposes a proof obligation hidden by ordinary parser pseudo-code.

The parser is part of the executable tool chain, but its full functional correctness is not verified against an independent formal specification of QDIMACS. The theorems in this paper therefore justify the solver result for the `pcnf` value produced by parsing, not the parser's conformance to the textual standard. This limitation suggests a concrete future verification task: formalise the QDIMACS grammar and prove that parsing agrees with it. Importantly, the parser is not used as an oracle in the proof of solver correctness; it is merely an executable front end for producing values of the already formalised datatype.

5 A Verified Search Solver for PCNF

The search solver operates on PCNF formulas. It avoids constructing an exponentially larger expanded formula by recursively assigning the outermost quantified variable and simplifying the matrix.

5.1 Assignment and simplification

Let Φ_l denote the formula obtained from a PCNF formula Φ by assigning the literal l to true. Formally, we define Φ_l using a matrix transformation and a prefix transformation. The matrix transformation removes every clause containing l, because those clauses are satisfied, and removes the complementary literal from all remaining clauses, because those literal occurrences are false. The prefix transformation removes all occurrences of the variable of the assigned literal l. The Isabelle function `pcnf_assign` of type `literal ⇒ pcnf ⇒ pcnf` implements this operation using auxiliary functions for matrix simplification and prefix removal.

The base cases of search are standard for CNF. If the matrix is empty, all clauses have been satisfied and the branch is true. If the matrix contains an empty clause, one clause has become false and the branch is false. Otherwise the solver inspects the first variable in the prefix and combines the recursive results by disjunction for existential variables and conjunction for universal variables:

$$(\exists x.\, \Phi) \rightsquigarrow \Phi_x \vee \Phi_{\neg x}, \qquad (\forall x.\, \Phi) \rightsquigarrow \Phi_x \wedge \Phi_{\neg x}.$$

This corresponds to the core of Q-DLL [12], without propagation, learning, dependency schemes, watched literals, restarts, or branching heuristics.

5.2 Cleansed formulas

A subtle issue concerns repeated variables in the prefix. In this case, the innermost binding shadows all outer bindings. Such formulas have unambiguous semantics, but the equivalence between $\forall x.\, \Phi$ and $\Phi_x \wedge \Phi_{\neg x}$ assumes that x does not occur again in the remaining prefix. Otherwise, assigning x would conflate distinct binding occurrences. The QBF semantics handle repeated binders by shadowing, but the PCNF assignment operation is intentionally simple and syntax-based.

We therefore define a PCNF formula to be *cleansed* when all variables in its prefix are distinct. A preprocessing pass `pcnf_cleanse` removes outer binding occurrences that are shadowed by later occurrences of the same variable. The key verified properties are:

```
theorem pcnf_cleanse_cleanses:
  "cleansed_p (pcnf_cleanse P)"

theorem cleanse_free_vars_inv:
  "set (pcnf_free_variables (pcnf_cleanse P)) =
    set (pcnf_free_variables P)"

theorem pcnf_cleanse_preserves_semantics:
  "pcnf_semantics I (pcnf_cleanse P) = pcnf_semantics I P"
```

The semantics-preservation theorem is the essential one. It says that cleansing is a verified normalisation step that makes the invariant required by the search algorithm available without changing the formula's meaning. This point is easy to miss in informal descriptions because many benchmark formats assume variables are declared sensibly. A verified solver should either reject non-cleansed inputs or normalise them; our development takes the second route.

5.3 The search function and solver

The recursive search function returns an optional Boolean:

```
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) (λx.
        Option.bind (first_existential prefix) (λe.
          if e then combine_options (∨)
            (search (pcnf_assign (P x) (prefix, matrix)))
            (search (pcnf_assign (N x) (prefix, matrix)))
          else combine_options (∧)
            (search (pcnf_assign (P x) (prefix, matrix)))
            (search (pcnf_assign (N x) (prefix, matrix))))))"
```

The option type accounts for malformed calls, for example a non-empty matrix with an empty prefix. The top-level solver prevents this situation by existentially closing free variables before search. Termination is proved by showing that the

prefix length decreases after assigning the first prefix variable. The proof relies on the prefix-removal part of `pcnf_assign` and on the cleansedness invariant.

The solver first closes free variables and cleanses the prefix:

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

fun search_solver P =
  the (search (pcnf_cleanse (pcnf_existential_closure P)))
```

Using the general existential-closure function here has two advantages. It avoids duplicating semantic reasoning for PCNF, and it reuses the conversion theorems from Section 4. The use of `the` in `pcnf_existential_closure` is justified by the fact that existentially closing a PCNF formula yields another PCNF formula.

5.4 Correctness theorem

The correctness theorem for the `search` function exposes the invariants required by the algorithm:

```
theorem search_cleansed_closed_correct:
  assumes "cleansed_p P"
    and "set (pcnf_free_variables P) = {}"
  shows "search P = Some (satisfiable (convert P))"
```

The proof proceeds by induction following the recursive structure of `search`. The base cases use the semantics of CNF matrices. The recursive cases rely on equisatisfiability lemmas for assignment under an outermost quantifier. For an existential variable, satisfiability of the quantified formula is equivalent to satisfiability of at least one assigned branch; for a universal variable, it is equivalent to satisfiability of both assigned branches. The cleansedness assumption guarantees that assignment cannot accidentally affect a later binding occurrence of the same variable.

The top-level solver correctness theorem has no side conditions:

```
theorem search_solver_correct:
  "search_solver P ↔ satisfiable (convert P)"
```

Its proof composes the correctness theorem for the `search` function with existential closure, cleansing, and the PCNF/QBF semantic bridge. This compositional structure is useful for future extensions. For example, a verified unit-propagation rule could be inserted into `search` by proving that it preserves satisfiability under the same closed and cleansed invariants.

5.5 Modularity and extension points

The current solver branches on the outermost variable. This choice keeps the algorithm simple and makes the termination argument straightforward, but it is not intended as the final architecture for high-performance verified solving. Rather,

the formalisation is deliberately structured so that extensions can be incorporated with modest additional effort. For example, search could be parameterised by a verified decision-variable selection function; showing that any choice satisfying the required dependency and prefix conditions preserves correctness. Likewise, although matrix simplification is currently list-based, it could be refined to more efficient representations, such as occurrence lists or watched data structures, while reusing the existing correctness arguments as a specification. Overall, the development stays close to the abstract calculus to obtain a clear end-to-end proof, while providing a foundation on which more efficient, refined implementations can be built in a straightforward and principled way.

6 Executable Code and Experiments

Isabelle's code generator exports executable code from definitions written in executable HOL [14]. We generate versions of the parser and both solvers in Standard ML, OCaml, Haskell, and Scala. Small wrappers read a QDIMACS file, invoke the parser, and call the selected solver. The generated code is therefore an executable artifact derived from the verified definitions, modulo the trusted code generator and target-language compiler/runtime.

6.1 Experimental setup

We evaluated nine solver variants: the expansion solver in four target languages, the search solver in four target languages, and CAQE 4.0.2 as a state-of-the-art baseline [34]. The experiments were run on StarExec [31], with a CPU-time limit of 900 seconds and a memory limit of 32 GB. The benchmark families were:

- *Domino games*: QBF encodings of a two-player game in which players place 1×2 dominoes on a $1 \times n$ board. These instances exercise alternating choices and grow roughly quadratically, with linearly increasing alternation depth.
- *XOR formulas*: synthetic formulas with $2n$ alternating quantifiers. Satisfiable instances have exactly one satisfying assignment to the existential variables; unsatisfiable instances add a constraint excluding that assignment. Their input size grows linearly with n.
- *QBFEVAL 2022 prenex-CNF instances*: 434 benchmark instances designed for state-of-the-art solvers.

The Standard ML backend performed best or close to best among generated variants, so the compact Table 1 reports representative SML results. The full data contain the same qualitative picture across target languages [4].

6.2 Discussion of results

The results confirm the expected hierarchy. The expansion solver is quickly overwhelmed by formula duplication. It already runs out of memory on modest domino instances and on larger XOR instances. The search solver is far more

Table 1. Representative results for generated SML solvers and CAQE. Times are CPU seconds; t/o denotes timeout; oom denotes out-of-memory.

Instance	Expansion	Search	CAQE	Instance	Expansion	Search	CAQE
domino2 sat	0.004	0.004	0.013	domino2i unsat	0.004	0.004	0.008
domino3 sat	0.008	0.004	0.008	domino3i unsat	0.007	0.004	0.009
domino4 sat	oom	0.009	0.010	domino4i unsat	oom	0.009	0.009
domino5i sat	oom	0.030	0.008	domino5 unsat	oom	0.029	0.011
domino6 sat	oom	13.018	0.013	domino6i unsat	oom	12.856	0.016
domino7 sat	oom	139.567	0.013	domino7i unsat	oom	138.631	0.018
domino8 sat	oom	t/o	3.273	domino8i unsat	oom	t/o	0.273
xor5 sat	0.053	0.004	0.009	xor5 unsat	0.058	0.004	0.014
xor9 sat	44.920	0.006	0.010	xor9 unsat	48.310	0.006	0.022
xor10 sat	oom	0.006	0.010	xor10 unsat	275.724	0.008	0.024
xor15 sat	oom	0.094	0.013	xor15 unsat	oom	0.097	0.032
xor20 sat	oom	2.845	0.015	xor20 unsat	oom	2.845	0.042
xor25 sat	oom	90.431	0.017	xor25 unsat	oom	90.328	0.051
xor30 sat	oom	t/o	0.019	xor30 unsat	oom	t/o	0.063
QBFEVAL'22 (number of solved instances)					0	1	329

robust, because it avoids materializing the expanded formula and simplifies the matrix after each assignment. On the XOR family, for instance, the search solver remains very fast for small n and reaches $n = 25$ before becoming impractical.

CAQE is orders of magnitude faster on non-trivial instances. This is expected and should not be read as a failure of verification: CAQE implements advanced QBF-solving techniques, while our verified search solver implements only a basic branching core. The meaningful experimental question for this paper is different: does the artifact execute on standard inputs, and does the search formalisation improve on expansion? The answer to both is yes.

The experiments also guide future verification work. The search solver's poor performance on QBFEVAL instances suggests that the next verified steps should be unit propagation, monotone-literal detection, better branching, and more efficient data structures. These are among the main techniques used by practical search-based solvers. The present formalisation is designed so that such optimizations can be added as semantics-preserving refinements rather than by replacing the whole proof.

Threats to validity. The experiments are intended to calibrate the verified solvers, not to establish a new competitive solver. We therefore report representative results rather than tuning the backends aggressively. The parser is shared by all generated verified variants, whereas CAQE uses its own front end. The generated code also inherits overheads from high-level functional data structures. These choices are acceptable for the present purpose: the performance gap is so large that minor tuning would not change the conclusion, and the comparison still shows the qualitative advantage of search over expansion.

7 Lessons for Verified QBF Solving

Make free variables explicit. Many QBF presentations focus on closed formulas. Benchmarks and intermediate transformations, however, may naturally produce formulas with free variables. Our use of existential closure provides a clean interface: solver correctness is stated for satisfiability of arbitrary formulas, while the internal decision procedures can assume closed inputs after a verified preprocessing step.

Separate syntax representations from semantics. The general `QBF` datatype is convenient for semantic definitions and for arbitrary formulas; the `pcnf` datatype is convenient for QDIMACS parsing and search. Proving a semantics-preserving bridge between the two representations pays off repeatedly. It lets us reuse general lemmas for PCNF and state final theorems in a representation-independent way.

Normalise repeated binders. Repeated variable names are easy to ignore in informal descriptions, but they interact with assignment-based search. The verified cleansing pass makes the assumption of distinct prefix variables explicit and proves that enforcing it preserves semantics. This is a small but important example of how proof engineering can strengthen the robustness of the underlying implementation.

Use option types to expose partiality. Several functions are conceptually partial: evaluating a formula as propositional only makes sense after variables and quantifiers have been removed; search expects a closed formula whose prefix accounts for the matrix variables. Returning `option` values makes these preconditions explicit. Correctness theorems then show that the top-level solver calls the partial components only inside their verified domains.

Plan for refinement. The current solver uses lists and repeatedly scans them. This choice simplifies the correctness proof but is inefficient. A natural next phase is a refinement chain from the list-based solver state to representations using literal trails, watched data structures, and imperative state, following the pattern established by verified SAT solvers. The present development supplies the abstract specification and core invariants needed for such a chain.

8 Related Work

QBF solving has a rich algorithmic literature. Early search-based solvers such as QuBE implement branching and propagation [13]. QCDCL-style solvers add conflict and solution learning, dependency schemes, and sophisticated heuristics [21,22,24,30]. Variable-elimination and abstraction/refinement approaches include Quantor [6], RAReQS [15], and CAQE [27,33,34]. Our search solver is much simpler than these tools; its closest algorithmic ancestor is the basic Q-DLL procedure described in QBF surveys and handbook chapters [12].

Verified SAT solving is more mature than verified QBF solving. Marić verified a modern SAT solver in Isabelle/HOL [23]. Blanchette, Fleury, Lammich, and Weidenbach developed a refinement-based Isabelle framework for CDCL, leading to efficient verified SAT solvers with watched literals, restarts, learning, and imperative data structures [7,11,10]. Tan, Heule, and Myreen verified propagation-redundancy and compositional UNSAT checking in CakeML [32]. These projects show how to connect abstract calculi to efficient verified implementations. Our work begins an analogous path for QBF.

QBF proof checking provides a complementary approach. HOL4 and HOL Light have been used to validate QBF certificates based on Q-resolution and Skolem functions [36,18,19], and unified QBF certification has been studied by Balabanov and Jiang [3]. Proof checking can leverage fast unverified solvers when certificates are available; verified solving provides direct trust in the decision procedure, which can act as a small reference engine, and does not require generating and checking a potentially large certificate. The two approaches are not mutually exclusive. A mature trustworthy QBF ecosystem would likely include both certifying high-performance solvers and verified baseline procedures.

9 Conclusion and Future Work

We presented a mechanised Isabelle/HOL development for QBF solving. The development includes a general QBF semantics, a verified expansion solver for arbitrary QBFs, a PCNF representation corresponding to QDIMACS, semantics-preserving conversions between representations, and a verified search solver for prenex CNF. Isabelle's code generator produces executable solver pipelines, and experiments show that the search solver substantially improves on expansion while remaining far behind unverified state-of-the-art tools such as CAQE.

The contribution is a trustworthy baseline rather than a competitive solver. Its value lies in the reusable infrastructure, the machine-checked semantic foundation, and invariants supporting future verified QBF algorithms. The development also clarifies several subtle issues: free variables should be handled explicitly; repeated binders require normalisation before variable-based search; partial functions should expose their domains; and representation changes should be accompanied by semantic preservation theorems.

Future work should proceed in several directions. The most direct extension is a verified Q-DLL solver with unit propagation, monotone-literal detection, and more flexible branching. A second direction is refinement to efficient imperative data structures, drawing on the methodology of verified SAT solvers. A third direction is a verified QDIMACS front end based on an independent grammar specification, closing the remaining gap between text input and verified solver theorem. A fourth direction is a verified CEGAR-based QBF solver that reuses verified SAT solvers as subprocedures. Finally, reflection could turn the solver into an Isabelle proof method for quantified Boolean goals, providing a verified decision procedure for the QBF fragment of higher-order logic.

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA. pp. 275–281. AAAI Press/The MIT Press (2005), http://www.aaai.org/Library/AAAI/2005/aaai05-044.php
2. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009), http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264
3. Balabanov, V., Jiang, J.R.: Unified QBF certification and its applications. Formal Methods in System Design **41**(1), 45–65 (2012). https://doi.org/10.1007/S10703-012-0152-6
4. Bergström, A.: A Verified QBF Solver. Master's thesis, Uppsala University, Uppsala, Sweden (2024), https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Auu%3Adiva-524859
5. Bergström, A., Weber, T.: Verified QBF solving. Archive of Formal Proofs (March 2024), https://isa-afp.org/entries/QBF_Solver_Verification.html, Formal proof development
6. Biere, A.: Resolve and expand. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings (2004), http://www.satisfiability.org/SAT04/programme/93.pdf
7. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. Journal of Automated Reasoning **61**(1-4), 333–365 (2018). https://doi.org/10.1007/S10817-018-9455-7
8. Czerner, P., Esparza, J., Krasotin, V.: A resolution-based interactive proof system for UNSAT. In: Kobayashi, N., Worrell, J. (eds.) Foundations of Software Science and Computation Structures. pp. 116–136. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-57231-9_6
9. Dershowitz, N., Hanna, Z., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3569, pp. 408–414. Springer (2005). https://doi.org/10.1007/11499107_32
10. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 148–165. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_10
11. Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using imperative HOL. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 158–171. ACM (2018). https://doi.org/10.1145/3167080
12. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified Boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1157–1176. IOS Press (2021). https://doi.org/10.3233/FAIA201014

13. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE: A system for deciding quantified Boolean formulas satisfiability. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2083, pp. 364–369. Springer (2001). https://doi.org/10.1007/3-540-45744-5_27
14. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/node?id=886023
15. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. Artificial Intelligence **234**, 1–25 (2016). https://doi.org/10.1016/J.ARTINT.2016.01.004
16. Katz, J., Hanna, Z., Dershowitz, N.: Space-efficient bounded model checking. In: Design, Automation and Test in Europe. pp. 686–687 Vol. 2 (2005). https://doi.org/10.1109/DATE.2005.276
17. Kleine Büning, H., Bubeck, U.: Theory of quantified Boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1131–1156. IOS Press (2021). https://doi.org/10.3233/FAIA201013
18. Kumar, R., Weber, T.: Validating QBF validity in HOL4. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6898, pp. 168–183. Springer (2011). https://doi.org/10.1007/978-3-642-22863-6_14
19. Kuncar, O.: Proving valid quantified Boolean formulas in HOL Light. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6898, pp. 184–199. Springer (2011). https://doi.org/10.1007/978-3-642-22863-6_15
20. Ling, A.C., Singh, D.P., Brown, S.D.: FPGA logic synthesis using quantified Boolean satisfiability. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3569, pp. 444–450. Springer (2005). https://doi.org/10.1007/11499107_37
21. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. Journal on Satisfiability, Boolean Modeling and Computation **7**(2-3), 71–76 (2010). https://doi.org/10.3233/SAT190077
22. Lonsing, F., Egly, U.: DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 371–384. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_23
23. Maric, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science **411**(50), 4333–4356 (2010). https://doi.org/10.1016/J.TCS.2010.09.014
24. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. Journal of Artificial Intelligence Research **65**, 180–208 (2019). https://doi.org/10.1613/JAIR.1.11529
25. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artificial Intelligence **274**, 224–248 (2019). https://doi.org/10.1016/J.ARTINT.2019.04.002
26. QDIMACS standard version 1.1, https://www.qbflib.org/qdimacs.html, retrieved June 1, 2026

27. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 136–143 (2015). https://doi.org/10.1109/FMCAD.2015.7542263
28. Rintanen, J.: Constructing conditional plans by a theorem-prover. Journal of Artificial Intelligence Research **10**, 323–352 (1999). https://doi.org/10.1613/JAIR.591
29. Shukla, A., Biere, A., Pulina, L., Seidl, M.: A survey on applications of quantified Boolean formulas. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI). pp. 78–84 (2019). https://doi.org/10.1109/ICTAI.2019.00020
30. Slivovsky, F.: Quantified CDCL with universal resolution. In: Meel, K.S., Strichman, O. (eds.) 25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 236, pp. 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.SAT.2022.24
31. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8562, pp. 367–373. Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_28
32. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in CakeML. International Journal on Software Tools for Technology Transfer **25**(2), 167–184 (2023). https://doi.org/10.1007/S10009-022-00690-Y
33. Tentrup, L.: On expansion and resolution in CEGAR based QBF solving. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 475–494. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_25
34. Tentrup, L.: CAQE and QuAbS: Abstraction based QBF solvers. Journal on Satisfiability, Boolean Modeling and Computation **11**(1), 155–210 (2019). https://doi.org/10.3233/SAT190121
35. Turner, H.: Polynomial-length planning spans the polynomial hierarchy. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings. Lecture Notes in Computer Science, vol. 2424, pp. 111–124. Springer (2002). https://doi.org/10.1007/3-540-45757-7_10
36. Weber, T.: Validating QBF invalidity in HOL4. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 466–480. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_32