Validating QBF Validity in HOL4

Ramana Kumar and Tjark Weber*

Computer Laboratory, University of Cambridge, UK
{rk436,tw333}@cam.ac.uk

Abstract. The Quantified Boolean Formulae (QBF) solver Squolem can generate certificates of validity, based on Skolem functions. We present independent checking of these certificates in the HOL4 theorem prover. This enables HOL4 users to benefit from Squolem's automation for valid QBF problems. Detailed performance data shows that LCF-style checking of validity certificates is often (but not always) feasible even for large QBF instances. Additionally, our work provides high correctness assurances for Squolem's claims of validity and uncovered a soundness bug in a previous version of its certificate validator QBV.

1 Introduction

Quantified Boolean Formulae (QBF) extend propositional logic with universal and existential quantification over Boolean variables. QBF have numerous applications in adversarial planning and formal verification [1,2,3]; for instance, they enable succinct encodings of bounded and unbounded model checking problems [4]. As a simple example, consider the formula

$$\forall x\, \exists y\, \exists z.\ (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z), \quad (1)$$

which says for all x there is a y that implies $x \oplus y$.

Deciding the validity of QBF is an extension of the well-known Boolean satisfiability problem (SAT). A propositional formula ϕ in Boolean variables $x_1, \ldots, x_n$ is satisfiable if and only if the QBF $\exists x_1 \ldots \exists x_n.\, \phi$ is valid. However, SAT is merely NP-complete, while QBF is the canonical PSPACE-complete problem [5]. Satisfiable propositional formulae have short certificates—namely, their satisfying assignments—that can be validated in polynomial time. For valid QBF, there is no known way to even specify a solution succinctly.

Nevertheless, certain QBF solvers can produce certificates for their answers that can be checked independently [6]. Squolem is a state-of-the-art QBF solver that generates certificates for valid formulae in a unified format based on finitary Boolean Skolem functions [7].

In this paper, we present independent checking of these certificates in the HOL4 [8] theorem prover. HOL4 is a popular interactive theorem prover for higher-order logic [9]. It is based on a small LCF-style [10,11] kernel that provides an abstract data type of theorems, equipped with a fixed set of constructor functions. Each function corresponds to an axiom schema or inference rule

* This work was supported by the British EPSRC under grant EP/F067909/1.

of higher-order logic. Derived rules that are not provided by this kernel must be implemented by composing existing rules. This provides high correctness assurances: derived rules cannot produce inconsistent theorems, as long as the theorem data type itself is implemented correctly. On the other hand, it makes an efficient implementation of derived rules challenging.

Our work is motivated primarily by a desire for increased automation in interactive theorem proving. Systems like Coq [12], HOL4, Isabelle [13] and PVS [14] can greatly benefit from the reasoning power of automated tools. This has been demonstrated numerous times, e.g., by integrations of SAT [15] and SMT solvers [16,17] as well as automated first-order provers [18,19,20]. We envision that HOL4 users might invoke Squolem directly to solve suitable proof obligations, but also that our integration might serve as a foundation, on top of which decision procedures for richer logics can be implemented through QBF encodings. Since the results are checked by HOL4's inference kernel, no trust needs to be put in the QBF solver.

An additional motivation arises from the fact that correctness of QBF solvers is hard to establish. QBF solvers are complex software tools that employ sophisticated heuristics and optimizations [21,22]. Different solvers may disagree on the status of individual benchmarks. QBF-Eval competitions until 2006 resolved disagreements by majority vote [23]. This rather unsatisfactory approach (which has been replaced by certificate checking in recent years) confirms the importance of QBF benchmark certification. HOL4's inference kernel has been scrutinized by dozens of researchers for over two decades. By using HOL4 as an independent checker, we obtain high correctness assurances for Squolem's results.

We review related work in Section 2, before introducing relevant background material in Section 3. Our main contribution, an efficient LCF-style implementation of certificate checking for valid QBF, is presented in Section 4. We evaluate our implementation in Section 5, and conclude in Section 6.

2 Related Work

This paper complements previous work on LCF-style checking for QBF certificates of *in*validity. In [24], an algorithm was presented that, given a QBF ϕ, obtains a HOL4 theorem $\vdash \neg\phi$ from a Squolem-generated certificate of invalidity. Here we present an algorithm to obtain a HOL4 theorem $\vdash \phi$ from a certificate of validity. Certificates of validity and invalidity for QBF are quite different. The latter employ Q-resolution [25], a refutation-complete inference rule that extends propositional resolution to quantified Boolean logic. The former (as considered here) are based on Skolem functions (see Section 3). In principle, one could establish validity of ϕ from invalidity of $\neg\phi$. However, this approach is not feasible in practice: current QBF solvers usually find inverted valid instances considerably harder and often time out [7]. Therefore, it is a practical necessity to support certificates of validity directly. We do not know whether inverted invalid instances become easier to solve as validity problems. It would be interesting to

understand when one approach is superior to the other, based on the shape of a formula.

Other related work concerns the integration of automated solvers with LCF-style theorem provers, and certificate checking for QBF solvers. Integrations have been proposed, e.g., for first-order provers [18,19,20], for model checkers [26], for computer algebra systems [27,28,29], and more recently for SMT solvers [16,17]. We use a HOL4 integration of SAT solvers [15] in this work.

Narizzano et al. [6] give an overview of certificate checking for QBF solvers. Squolem's certificates show competitive performance, and they are relatively simple. Unsurprisingly, stand-alone proof checkers for QBF are typically much more efficient than the LCF-style proof checker presented here. However, they arguably do not provide the same degree of trustworthiness as the HOL4 kernel.

3 Background and Theory

We now introduce relevant terminology and describe the QBF certificate format (Section 3.3) as well as HOL4's inference calculus (Section 3.4). Propositional logic is presupposed.

3.1 Quantified Boolean Formulae

We assume an infinite set of Boolean variables. A *literal* is a possibly negated Boolean variable. When l is a literal, we write $\underline{l}$ to denote its variable. A *clause* is a disjunction of literals. We say that a propositional formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

Definition 1 (Quantified Boolean Formula). *A* Quantified Boolean Formula (QBF) *is of the form*

$$Q_1 x_1 \ldots Q_n x_n . \phi,$$

where $n \geq 0$, each x_i is a Boolean variable, each Q_i is either $\forall$ or $\exists$, and ϕ is a propositional formula in CNF.

$Q_1 x_1 \ldots Q_n x_n$ is called the *quantifier prefix*, and ϕ is called the *matrix*. Without loss of generality, we consider QBF in this *prenex form* only. Any formula involving only propositional connectives and quantifiers over Boolean variables can be transformed into prenex form through straightforward syntactic manipulations. (We have not yet implemented such a transformation in HOL4.) The *innermost* variable of the above QBF is x_n.

The QDIMACS format [30] is the standard input format of QBF solvers. It provides a textual means of encoding QBF in prenex form. It is a backward-compatible extension of the DIMACS format [31], the standard input format of SAT solvers. We have implemented a translation from (the QBF subset of) HOL4 terms to QDIMACS, and a simple recursive-descent parser for QDIMACS files that returns the corresponding QBF as a HOL4 term (see Section 3.4).

The QDIMACS format imposes further restrictions: all variables x_i must be distinct, all variables must appear in the matrix, and the innermost quantifier must be existential (i.e., $Q_n = \exists$). We further require all variables that appear in the matrix to be bound by some quantifier, i.e., we consider *closed* QBF only. This is to avoid confusion: in the QDIMACS format, free variables have existential semantics (to retain backward compatibility with DIMACS), while in HOL4, free variables in theorems have universal semantics (to permit instantiation). If a QBF has free variables, we consider its existential closure instead.

$\mathbb{B} = \{\top, \bot\}$ denotes the set of truth values. The semantics of closed QBF is defined recursively: $[\![\forall x.\, \phi]\!] = [\![\phi[x \mapsto \top] \wedge \phi[x \mapsto \bot]]\!]$, and similarly $[\![\exists x.\, \phi]\!] = [\![\phi[x \mapsto \top] \vee \phi[x \mapsto \bot]]\!]$. (Here $\phi[x \mapsto y]$ denotes substitution of y for all free occurrences of x in ϕ.) A QBF is called *valid* if its semantics is $\top$ (i.e., true).

3.2 Skolem Functions and Models

QBF of interest typically contain several dozen or even hundreds of quantifiers. A naive recursive computation of their semantics, which would be exponential in the number of quantifiers, is not feasible. Therefore, QBF solvers implement different algorithms. The certificates of validity that we consider here are based on finitary Boolean Skolem functions [7].

Definition 2 (Model). *A* model *of the QBF* $Q_1 x_1 \ldots Q_n x_n.\, \phi$ *maps each existentially quantified variable* x_k *to a function* $f_k \colon \mathbb{B}^{k-1} \to \mathbb{B}$.

A model provides a witness function for every existentially quantified variable. We identify each witness function f_k with a propositional formula in $k-1$ variables. Because Skolemization preserves satisfiability, we have

Theorem 1. *A QBF* $Q_1 x_1 \ldots Q_n x_n.\, \phi$ *with existentially quantified variables* $x_{e_1}, \ldots, x_{e_m}$ *(where* $e_1 < \cdots < e_m$*) is valid if and only if there is a model* $\{x_{e_k} \mapsto f_{e_k}\}_{k=1}^{m}$ *such that the propositional formula*

$$\phi[x_{e_m} \mapsto f_{e_m}(x_1, \ldots, x_{e_m - 1})] \cdots [x_{e_1} \mapsto f_{e_1}(x_1, \ldots, x_{e_1 - 1})]$$

obtained by replacing existential variables with their witness functions in ϕ *is valid.*

Thus, every valid QBF has a model that witnesses its validity, and conversely, a model that produces a valid propositional formula proves validity of the original QBF. This is the theoretical foundation for the certificate format that we describe in Section 3.3.

As a simple example, consider (1). A model is given by $f_y(x) = \bot$ and $f_z(x, y) = x$. From this model, we obtain the propositional formula $(x \vee \bot \vee \neg x) \wedge (x \vee \top \vee x) \wedge (\neg x \vee \bot \vee x) \wedge (\neg x \vee \top \vee \neg x) \wedge (\top \vee x)$. This formula is easily seen to be valid: each of its clauses is valid, containing either $\top$ or both x and $\neg x$. Hence (1) is valid by Theorem 1.

3.3 Certificates of Validity

Squolem generates certificates in a unified format that is described in detail in [32]. The format is ASCII-based. Clauses and variables are indexed by positive integers. Negative values stand for negated variables, i.e., integer negation denotes propositional negation. Indices do not necessarily correspond to variable positions in the quantifier prefix.

A certificate of validity encodes a model of a QBF, as defined in Section 3.2. Certificates introduce fresh *extension variables* as abbreviations for witness functions and other (sub-)formulae. For each extension variable, the certificate contains a line that defines the extension as either

- a conjunction of literals (with the empty conjunction denoting $\top$), or
- a formula $\mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ z$, where x, y, z are literals.

All variables that occur in the definiens must be extension variables that have been defined previously, or must come from the original QBF. From these two simple building blocks, extension variables can be defined for arbitrary propositional formulae (and hence, for arbitrary witness functions). The certificate's final line contains a list of (x_k, f_k) pairs that establishes the map from existential variables to witness functions, each function denoted by a (possibly negated) extension variable.

For instance, mapping x, y and z to variable indices 1, 2 and 3, respectively, Squolem generates the following certificate for (1):

```
QBCertificate                  // explanatory comments:
E 4 A 2 0                      // v4 = v2 (0 ends the line)
E 5 A 1 -2 0                   // v5 = v1 ∧ ¬v2
E 6 I 4 4 5                    // v6 = if v4 then v4 else v5
E 7 A 0                        // v7 = ⊤ (empty conjunction)
CONCLUDE VALID 2 -7 3 6        // v2 = ¬v7, v3 = v6
```

There are four extension variables v_4 through v_7, defined as $v_4 = y$, $v_5 = x \wedge \neg y$, $v_6 = \mathsf{if}\ v_4\ \mathsf{then}\ v_4\ \mathsf{else}\ v_5$, and $v_7 = \top$. The witness for y is declared to be $\neg v_7$, i.e., $\bot$. The witness for z is declared to be v_6, which simplifies to x given that $v_4 = y = \bot$. The certificate thus encodes the model given in Section 3.2.

We have written a simple recursive-descent parser for this certificate format that returns the encoded information as a value in Standard ML.

3.4 Higher-Order Logic

HOL4 is a popular LCF-style [10,11] theorem prover for polymorphic higher-order logic [9]. It is based on Church's simple type theory [33] extended with Hindley-Milner style polymorphism [34]. Higher-order logic (HOL) contains a type of Booleans, propositional connectives, and quantifiers over arbitrary types. Hence, quantified propositional logic embeds straightforwardly into HOL.

HOL4 implements a natural-deduction calculus. Theorems represent *sequents* $\Gamma \vdash \phi$, where Γ is a finite set of *hypotheses*, and ϕ is the sequent's *conclusion*.

Instead of $\emptyset \vdash \phi$, we simply write $\vdash \phi$. Internally, the set of hypotheses is given by a red-black tree (for efficient search, insertion and deletion), with terms treated modulo α-equivalence.

Like other LCF-style provers, HOL4 has a small inference kernel. Theorems are implemented as an abstract data type, and new theorems can be constructed only through a fixed set of functions provided by this data type. These functions directly correspond to the axiom schemata and inference rules of higher-order logic. Figure 1 shows the rules of HOL that we use to validate certificates of QBF validity (our call to MiniSat [35], described in the next section, may use additional primitive rules involving negation).

$$\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume}_{\phi} \qquad \frac{\Gamma \vdash \phi}{\Gamma\theta \vdash \phi\theta}\ \textsc{Inst}_{\theta} \qquad \frac{}{\vdash t = t}\ \textsc{Refl}_{t}$$

$$\frac{\Gamma \vdash \psi}{\Gamma \setminus \{\phi\} \vdash \phi \implies \psi}\ \textsc{Disch}_{\phi} \qquad \frac{\Gamma \vdash \phi \implies \psi \qquad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\ \text{MP}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \forall x.\, \phi}\ \textsc{Gen}_{x}\ (x \text{ not free in } \Gamma) \qquad \frac{\Gamma \vdash \phi[t]}{\Gamma \vdash \exists x.\, \phi[x]}\ \textsc{Exists}_{(\exists x.\, \phi[x], t)}$$

Fig. 1. Selected HOL inference rules

The LCF-style architecture greatly reduces the trusted code base. Proof procedures, although they may implement arbitrarily complex algorithms, cannot produce unsound theorems, as long as the implementation of the theorem data type is correct. HOL4 is written in Standard ML [36], a type-safe functional language (with impure features, e.g., references) that has an advanced module system. To benefit from HOL4's LCF-style architecture, we must implement proof reconstruction in this language.

On top of its LCF-style inference kernel, HOL4 offers various automated proof procedures: e.g., a simplifier, which performs term rewriting, and various first-order provers. The performance of these procedures is hard to control, so we mostly avoid them by combining primitive inference rules directly. A major exception is our use of an existing HOL4 integration [15] of the SAT solver MiniSat to prove the propositional conjecture obtained from a QBF and its validity certificate. We now describe our certificate checking method, with this use of MiniSat, in more detail.

4 Checking Validity Certificates in HOL4

4.1 Overview

Given a QBF $\psi = Q_1 x_1 \ldots Q_n x_n.\, \phi$ and a certificate of its validity, our goal is to derive $\vdash \psi$ as a HOL4 theorem.

The certificate provides witnesses for the QBF's existential variables. However, unfolding the definition of witness functions in the QBF's matrix ϕ, as suggested by Theorem 1, could lead to an exponential blowup of the formula. Instead, we observe that we can use these definitions as hypotheses.

More specifically, the certificate gives a definition $\langle t_i \rangle$ for each extension variable v_i, and a witness literal f_{e_k} (where $\underline{f_{e_k}} = v_i$ for some i) for each existential variable x_{e_k}.[1] We convert definitions $\langle t_i \rangle$ to HOL4 terms t_i, and replace existential variables with their witnesses to ensure each t_i contains only universal and extension variables. We then prove the theorem $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$, where m and p are the number of existential and extension variables, respectively. To prove validity of the QBF from this theorem, we reintroduce quantifiers in order, from Q_n up to Q_1, and prove the hypotheses. We eliminate hypotheses eagerly, while ensuring we do not unfold the definition of any variable that occurs more than once in the sequent.

In the certificate, variables are indexed by positive integers. Our implementation maintains a one-to-one correspondence between variables and indices. For conceptual clarity, we describe the algorithm entirely in terms of variables. The implementation uses indices where possible to achieve better performance.

We maintain two maps keyed on variables. The first map, V, gives the variable's kind—universal, existential, or extension—along with a HOL4 term for its definition, if applicable: f_{e_k} for existential, and t_i for extension variables. The second map, D, maps each variable to a list of variables that it *depends* on. Dependency between variables is characterized as follows.

(D_1) x_k depends on x_{k+1}, for all $1 \leq k < n$;
(D_2) $\underline{f_{e_k}}$ depends on x_{e_k}, for all $1 \leq k \leq m$; and
(D_3) each variable in t_i depends on v_i, for all $1 \leq i \leq p$.

We explain in Section 4.3 how dependencies are used for hypothesis elimination.

Our algorithm for checking validity certificates has four main steps.

1. Construct the formula $\phi' = (x_{e_1} \Leftrightarrow f_{e_1}) \Rightarrow \cdots \Rightarrow (x_{e_m} \Leftrightarrow f_{e_m}) \Rightarrow \phi$ and partially construct the maps V and D, omitting extension variables;
2. Add extension variable definitions, obtaining the formula $\phi'' = (v_1 \Leftrightarrow t_1) \Rightarrow \cdots \Rightarrow (v_p \Leftrightarrow t_p) \Rightarrow \phi'$, and finish constructing the maps V and D;
3. Prove the (purely propositional) theorem $\vdash \phi''$ using the MiniSat integration, then turn its antecedents into hypotheses to obtain $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$; and finally,
4. Topologically sort the variables according to D, then eliminate hypotheses and reintroduce quantifiers to obtain $\vdash \psi$.

4.2 Preparing the Formula for MiniSat

We first process the quantifier prefix of ψ, stripping off one quantifier at a time until we obtain ϕ. For each quantifier, we add $x_k \mapsto [x_{k+1}]$ to D (or $x_n \mapsto [\,]$ for

[1] Squolem may omit witnesses for variables whose value does not affect the QBF's validity. For these variables we use a dummy extension variable with definition $\top$.

the innermost variable). For each universal quantifier, we add $x_k \mapsto \forall$ to V. For each existential quantifier, we add $x_{e_k} \mapsto (\exists, f_{e_k})$ to V and $\underline{f_{e_k}} \mapsto [x_{e_k}]$ to D. When all the quantifiers have been stripped, V maps every quantified variable, and D accurately represents the first two dependency conditions. Iterating over V, we add each existential variable's definition, $x_{e_k} \Leftrightarrow f_{e_k}$, as an antecedent to ϕ to complete the algorithm's first main step, obtaining ϕ'.

Next, we process the certificate's definitions of extension variables. For each definition $(v_i, \langle t_i \rangle)$, we construct a term t_i by creating a HOL4 conjunction or if-then-else term, replacing references to existential variables with their witnesses. For each variable x that occurs in t_i (after replacing existential variables), we add v_i to the list associated with x in D. Thus, when all definitions have been processed, D accurately represents all three dependency conditions. We also add $v_i \mapsto (\mathsf{ext}, t_i)$ to V, and $v_i \Leftrightarrow t_i$ as an antecedent to ϕ', in the end obtaining ϕ''. This completes the second main step.

We now invoke MiniSat to prove ϕ'', which is a purely propositional formula with antecedents defining all existential and extension variables and the original matrix as consequent. MiniSat is an independent SAT solver that has been integrated into HOL4 just as we are now integrating Squolem. In particular, MiniSat logs proofs, and each proof is replayed via HOL4 inferences to produce a theorem that depends only on the trusted kernel [15]. When MiniSat returns a theorem $\vdash \phi''$, we turn all antecedents into hypotheses using ASSUME and MP.

4.3 Hypothesis Elimination

Given the theorem $\{x_{e_1} \Leftrightarrow f_{e_1}, \ldots, x_{e_m} \Leftrightarrow f_{e_m}, v_1 \Leftrightarrow t_1, \ldots, v_p \Leftrightarrow t_p\} \vdash \phi$ obtained from the previous step, our goal is to introduce quantifiers and eliminate hypotheses to obtain $\vdash \psi$. To introduce a universal (existential) quantifier, we use GEN (EXISTS, respectively). To eliminate a hypothesis of the form $x \Leftrightarrow t$, we use INST with a substitution mapping x to t. The hypothesis thus becomes $t \Leftrightarrow t$. We prove $\vdash t \Leftrightarrow t$ with REFL, then use DISCH and MP (see Figure 1).

However, care must be taken to introduce quantifiers and eliminate hypotheses in the correct order. The INST rule instantiates *all* free occurrences of a variable in a sequent. When eliminating a hypothesis, we want the variable on its left-hand side to occur only there, both to avoid changing the conclusion of the theorem, whose matrix should always be ϕ, and to prevent terms in the hypothesis set from growing too large. Therefore, before eliminating $x \Leftrightarrow t$, we ensure both that x is quantified in the conclusion (or is an extension variable), and that x does not appear on the right-hand side of any hypothesis. It is enough to consider right-hand sides, since the left-hand sides are all distinct.

A variable x has been *eliminated* if it has been quantified, if necessary, and the hypothesis with x on the left, if any, has been eliminated. Only existential variables require both treatments; for them, we quantify before eliminating the hypothesis. A variable x depends on another variable y if y must be eliminated before x can be eliminated. The last two dependency conditions defined in Section 4.1 effectively say that the left-hand side of a hypothesis must be eliminated before any variable on its right-hand side, which agrees with our

observations about INST. Dependency condition D_1 simply ensures that we introduce quantifiers in the correct order. To complete the algorithm's final main step, we topologically sort all variables according to their dependencies in D, then eliminate each variable in the order obtained.

The GEN_x rule has a side condition: x must not occur free in the hypotheses. We rely on the fact that if we eliminate hypotheses eagerly, i.e., as soon as their left-hand side is a lone occurrence, then the side condition holds as long as each witness function f_{e_j} represented by the certificate depends only on variables $x_1, \ldots, x_{e_j-1}$. In fact, Squolem may re-order existential variables, i.e., define a witness f_{e_j} in terms of x_{e_k} for some $e_k > e_j$, provided there is no intervening universal quantifier. However, only acyclic dependencies between existential variables are allowed; cycles are detected as failure of the topological sort.[2]

4.4 Example

Consider (1) again, where we have

$$\psi = \forall x\, \exists y\, \exists z.\ (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z).$$

Assume x, y, and z have variable indices 1, 2, and 3, respectively. Squolem provides the certificate of validity given in Section 3.3. Let v_1 through v_4 be extension variables with indices 4 through 7. Witnesses are given as $(y, \neg v_4)$ and (z, v_3). Definitions are given as $(v_1, \texttt{A}\ y\ \texttt{0})$, $(v_2, \texttt{A}\ x\ {-y}\ \texttt{0})$, $(v_3, \texttt{I}\ v_1\ v_1\ v_2)$, and $(v_4, \texttt{A}\ \texttt{0})$. After processing the quantifier prefix, we have

$$\begin{aligned}
\phi &= (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z),\\
V &= \{x \mapsto \forall, y \mapsto (\exists, \neg v_4), z \mapsto (\exists, v_3)\},\\
D &= \{x \mapsto [y], y \mapsto [z], z \mapsto [\,], v_3 \mapsto [z], v_4 \mapsto [y]\},\\
\phi' &= (y \Leftrightarrow \neg v_4) \Rightarrow (z \Leftrightarrow v_3) \Rightarrow \phi.
\end{aligned}$$

We process the definitions of extension variables (as described in Section 4.2) to obtain $t_1 = \neg v_4$, $t_2 = x \wedge v_4$, $t_3 = \mathsf{if}\ v_1\ \mathsf{then}\ v_1\ \mathsf{else}\ v_2$, $t_4 = \top$, and

$$\begin{aligned}
V = \{&x \mapsto \forall, y \mapsto (\exists, \neg v_4), z \mapsto (\exists, v_3),\\
&v_1 \mapsto (\mathsf{ext}, t_1), v_2 \mapsto (\mathsf{ext}, t_2), v_3 \mapsto (\mathsf{ext}, t_3), v_4 \mapsto (\mathsf{ext}, t_4)\},\\
D = \{&x \mapsto [y, v_2], y \mapsto [z], z \mapsto [\,], v_1 \mapsto [v_3], v_2 \mapsto [v_3], v_3 \mapsto [z], v_4 \mapsto [y, v_1, v_2]\},\\
\phi'' = &(v_1 \Leftrightarrow t_1) \Rightarrow (v_2 \Leftrightarrow t_2) \Rightarrow (v_3 \Leftrightarrow t_3) \Rightarrow (v_4 \Leftrightarrow t_4) \Rightarrow \phi'.
\end{aligned}$$

After passing ϕ'' to MiniSat and stripping all antecedents from the resulting theorem, we have

$$\{y \Leftrightarrow \neg v_4, z \Leftrightarrow v_3, v_1 \Leftrightarrow t_1, v_2 \Leftrightarrow t_2, v_3 \Leftrightarrow t_3, v_4 \Leftrightarrow t_4\} \vdash \phi$$

[2] Squolem's own certificate validator, QBV, did not implement this acyclicity check correctly in version 1.03. It would therefore accept certain invalid certificates. This bug has been fixed in the latest version (2.0) of QBV.

We now eliminate variables in a topological order according to D, say $z < v_3 < v_1 < v_2 < y < x < v_4$. After eliminating z, v_3, and v_1, we have

$$\{y \Leftrightarrow \neg v_4, v_2 \Leftrightarrow t_2, v_4 \Leftrightarrow t_4\} \vdash \exists z.\, \phi$$

We show the remainder in more detail. To eliminate v_2, we instantiate v_2 to t_2, then prove the hypothesis with a theorem $\vdash t_2 \Leftrightarrow t_2$:

$$\begin{aligned} \{y \Leftrightarrow \neg v_4, t_2 \Leftrightarrow t_2, v_4 \Leftrightarrow t_4\} &\vdash \exists z.\, \phi \\ \{y \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists z.\, \phi \end{aligned}$$

To eliminate y, we first quantify. Then we can instantiate without affecting the sequent's conclusion, before proving the hypothesis:

$$\begin{aligned} \{y \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \\ \{\neg v_4 \Leftrightarrow \neg v_4, v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \\ \{v_4 \Leftrightarrow t_4\} &\vdash \exists y.\, \exists z.\, \phi \end{aligned}$$

To eliminate x, we simply quantify, which is possible since all hypotheses mentioning x have been eliminated. And to eliminate v_4, we instantiate again before proving the hypothesis as before.

$$\begin{aligned} \{v_4 \Leftrightarrow t_4\} &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \\ \{t_4 \Leftrightarrow t_4\} &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \\ \emptyset &\vdash \forall x.\, \exists y.\, \exists z.\, \phi \end{aligned}$$

This sequent is now $\vdash \psi$ as required.

5 Experimental Results

We have evaluated our implementation on a set of 100 valid QBF problems that resulted from applying Squolem 2.02 to all 445 problems in the *2005 fixed instance* and *2006 preliminary QBF-Eval* data sets. With a time limit of 600 seconds per problem, Squolem solved 217 of these problems; 100 were determined to be valid.[3] (We did not consider inverting invalid problems.) The same set of problems was previously used (by the Squolem authors) to evaluate the performance of Squolem's certificate generation [7].

All experiments were conducted on a 64-bit Linux system with an Intel Core i7-920XM processor at 2.0 GHz clock speed. Memory usage was restricted to 4 GB. HOL4 was running on top of Poly/ML 5.4.1.

[3] In comparison, Squolem 1.03 only solved 142 problems, among them 73 valid ones.

5.1 Run-Times

Table 1 shows our experimental results for the first 50 of the 100 valid QBF problems. Our full results are available at `http://www.cl.cam.ac.uk/~tw333/qbf/`. The remainder of this section comprehensively covers all 100 problems.

The first column in Table 1 gives the name of the benchmark. The next three columns provide information about the size of the benchmark, giving the number of alternating quantifiers,[4] variables, and clauses, respectively. Column five shows the run-time of Squolem 2.02 (with certificate generation enabled) to solve the benchmark. Column six shows the number of extension variables in the resulting certificate.

The last two columns finally show the run-time of certificate validation in HOL4. The HOL4 system comes with two different implementations of its inference kernel: one uses de Bruijn indices (and explicit substitutions) to represent λ-terms [37], the other (by M. Norrish) uses a name-carrying implementation [38]. These implementations differ in the performance (and even complexity) of primitive operations. We present run-times for both implementations.

All run-times are given in seconds (rounded to the nearest tenth of a second). Timeouts are indicated by T. For comparison, we have also measured run-times of QBV [7], a stand-alone checker for Squolem's certificates that was developed by the authors of Squolem. QBV is written in C++ and uses MiniSat for tautology checking. Its run-times are given in column seven.

We observe that even for Squolem's stand-alone checker QBV, certificate validation is considerably harder than certificate generation on selected problems (e.g., adder-6-sat, qshifter_7, qshifter_8). This is in line with earlier results [7] and reflects the fact that certificate validation for valid QBF instances is, in general, co-NP-complete [39].

However, QBV times out on one problem only, while HOL4 times out on 13 problems (de Bruijn kernel) or 15 problems (name-carrying kernel). This corresponds to success rates of 87% and 85%, respectively. These rates are largely due to a number of relatively easy problems. The largest certificates that are validated successfully in HOL4 (k_d4_n-20, toilet_c_08_10.2) define just over 15000 extension variables each; QBV validates them in about a second. Figure 2 shows run-times for the de Bruijn kernel as a function of the number of extension variables. The dotted trend line ($R^2 = 0.96$) is given by $f(x) = 1.09 \cdot 10^{-5} \cdot x^{1.85}$.

If we count each timeout as 600 seconds, average run-times are 7.5 seconds for Squolem, 8.4 seconds for QBV, 134.1 seconds for the de Bruijn kernel, and 163.1 seconds for the name-carrying kernel. (Considering successfully validated problems only, average run-times are 2.4 seconds for QBV, 64.5 seconds for the de Bruijn kernel, and 86.1 seconds for the name-carrying kernel.) The de Bruijn kernel thus takes 16 times longer on average than QBV (and 18 times longer than Squolem's proof search), but is almost 18% faster than the name-carrying kernel.

[4] Counting successive quantifiers of the same kind, as in $\forall x \forall y \forall z \ldots$, as one quantifier only. The total number of quantifiers in each benchmark is typically identical to the number of variables.

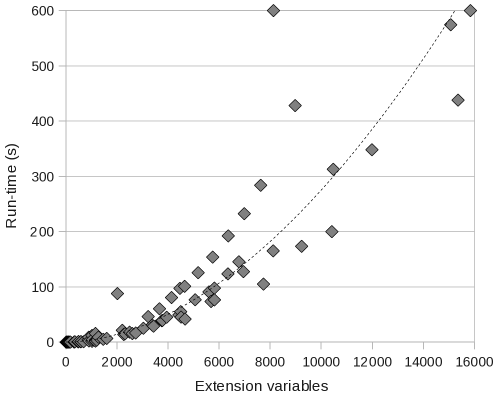

Fig. 2. Run-times/extension variables

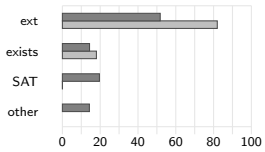

Fig. 3. de Bruijn

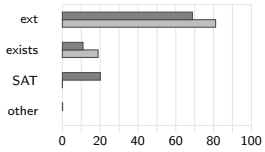

Fig. 4. Name-carrying

Interestingly, the picture is very different for invalid QBF [24]. For LCF-style validation of invalidity certificates, the name-carrying kernel is 75 times faster on average than the de Bruijn kernel, and 25 times faster than proof search with Squolem. This shows that LCF-style validation of QBF invalidity is not only easier in general, but also exercises different primitive inference rules of the HOL4 kernel.

5.2 Profiling and Future Improvements

To gain deeper insight into these results, we present profiling data for the de Bruijn kernel (Figure 3) and the name-carrying kernel (Figure 4).

For each kernel, we show the share of total run-time (dark bars) and relative number of function calls (light bars) for the following functions: elimination of extension variable definitions (ext), introduction of existential quantifiers into the conclusion (exists), and propositional tautology proving (SAT). Time spent on other aspects of certificate validation, e.g., parsing and pre-processing the certificate, is shown as well (other). The relative number of function calls (light bars) is similar for each kernel; small differences are caused by timeouts.

We observe that the bulk of run-time (52% for the de Bruijn kernel, 69% for the name-carrying kernel) is spent on eliminating extension variables. There are about four extension variables on average for every existential quantifier. Introducing the latter takes 11% (name-carrying kernel) to 14% (de Bruijn kernel) of total run-time. With either kernel, about 20% of run-time is spent on propositional tautology proving, i.e., on the call to MiniSat with pre-processing and proof checking in HOL4. Other inferences, e.g., introduction of universal quantifiers via Gen, take a much greater share of run-time in the de Bruijn kernel (14%) than in the name-carrying kernel (0%). As indicated by the run-times in Section 5.1, however, this effect is more than outweighed by the name-carrying kernel's slightly inferior performance on ext.

Benchmark name	Quant.	Vars.	Clauses	Squolem (s)	Ext. vars.	QBV (s)	de Bruijn (s)	name-carry. (s)
Adder2-2-s	6	236	292	0.0	352	0.0	0.7	1.0
Adder2-4-s	6	1117	1405	1.3	5685	0.4	73.6	88.3
adder-2-sat	4	51	109	0.0	179	0.0	0.2	0.3
adder-4-sat	4	226	530	0.1	4422	0.2	49.1	56.8
adder-6-sat	4	525	1259	7.8	104897	83.2	T	T
CHAIN12v.13	3	925	4582	0.1	896	0.0	9.0	13.8
CHAIN13v.14	3	1080	5458	0.1	1023	0.0	12.1	19.2
CHAIN14v.15	3	1247	6424	0.1	1157	0.0	15.7	25.2
CHAIN15v.16	3	1426	7483	0.1	3221	0.1	46.3	86.5
CHAIN16v.17	3	1617	8638	0.2	3664	0.1	60.4	113.3
CHAIN17v.18	3	1820	9892	0.3	4136	0.1	80.7	145.4
CHAIN18v.19	3	2035	11248	0.3	4649	0.1	101.1	192.6
CHAIN19v.20	3	2262	12709	0.4	5178	0.1	125.7	247.2
CHAIN20v.21	3	2501	14278	0.5	5748	0.1	153.9	311.4
CHAIN21v.22	3	2752	15958	0.5	6358	0.1	192.3	381.0
CHAIN22v.23	3	3015	17752	0.7	6985	0.1	232.5	467.4
CHAIN23v.24	3	3290	19663	0.9	7628	0.1	283.9	567.8
comp.blif_0.10_0.20_0_1_inp_exact	7	311	833	1.3	9231	0.5	173.5	169.4
comp.blif_0.10_1.00_0_1_inp_exact	3	307	844	0.1	4667	0.2	41.8	46.8
counter_2	5	42	103	0.0	322	0.0	0.2	0.3
counter_4	9	130	333	0.4	7737	0.2	105.1	104.6
counter_e_2	5	50	123	0.0	692	0.0	0.8	1.0
counter_e_4	9	144	373	136.4	69771	7.3	T	T
counter_r_2	5	50	121	0.0	360	0.0	0.3	0.4
counter_r_4	9	144	369	0.8	10415	0.4	200.0	226.0
counter_re_2	5	58	141	0.0	583	0.0	0.6	0.9
counter_re_4	9	158	409	38.8	40716	3.4	T	T
impl02	5	10	18	0.0	14	0.0	0.0	0.0
impl04	9	18	34	0.0	28	0.0	0.0	0.0
impl06	13	26	50	0.0	42	0.0	0.0	0.0
impl08	17	34	66	0.0	56	0.0	0.0	0.0
impl10	21	42	82	0.0	70	0.0	0.0	0.1
impl12	25	50	98	0.0	84	0.0	0.0	0.1
impl14	29	58	114	0.0	98	0.0	0.1	0.1
impl16	33	66	130	0.0	112	0.0	0.1	0.1
impl18	37	74	146	0.0	126	0.0	0.1	0.1
impl20	41	82	162	0.0	140	0.0	0.1	0.2
k_branch_n-4	13	803	2565	33.7	8118	0.5	165.2	235.3
k_d4_n-16	41	1437	5140	0.9	11985	0.8	348.2	561.2
k_d4_n-20	49	1785	6416	1.3	15069	1.1	574.4	T
k_d4_n-21	51	1872	6735	1.4	15840	1.1	T	T
k_d4_n-4	17	393	1312	0.1	2733	0.1	16.6	28.3
k_d4_n-8	25	741	2588	0.3	5817	0.2	76.5	126.9
k_dum_n-12	35	620	1594	0.1	2315	0.1	15.1	25.0
k_dum_n-16	43	796	2062	0.1	3035	0.2	25.2	46.2
k_dum_n-20	51	972	2530	0.1	3755	0.2	38.7	70.0
k_dum_n-21	53	1016	2647	0.2	3945	0.2	45.1	79.1
k_dum_n-4	19	262	649	0.0	902	0.0	2.2	3.8
k_dum_n-8	27	444	1126	0.0	1599	0.0	6.7	12.6
k_grz_n-12	17	557	2003	7.5	4510	0.2	45.3	77.9

Table 1. Experimental results

At present, we convert the negation of the QBF's matrix ϕ into CNF. This is costly, despite the fact that HOL4 uses a Tseitin-style transformation [15]. It would be more apt to call MiniSat several times, to prove each clause of ϕ separately from the certificate's definitions. However, the overhead associated with calling MiniSat from HOL4 currently renders this approach infeasible: no incremental interface to MiniSat is available in HOL4.

More substantial improvements might be gained from a modified term data structure. The kernel could compute the set of a term's free variables when the term is built, and store it in memory along with the term itself. This would permit more efficient implementations of instantiation and generalization (see Figure 1) . However, it is difficult to predict the effect that such a major change in fundamental kernel data structures would have on other HOL4 applications.

As Figure 2 shows, eliminating extension variables is essentially quadratic. Harrison [40] presents an ingenious solution to this kind of problem using proforma theorems to reduce the complexity. Adapting his approach would require some effort, but could yield significant performance improvements.

6 Conclusions

We have presented LCF-style checking for certificates of QBF validity in HOL4. Detailed performance data shows that LCF-style certificate checking is often feasible even for large valid QBF instances: up to 87% of our benchmark certificates were checked successfully. With a time limit of 600 seconds, the algorithm succeeds on certificates that have at most some 15000 extension variables. Our implementation is freely available from the HOL4 repository [38].

Our work complements earlier work on LCF-style checking for certificates of QBF invalidity [24]. It has two main applications. First, it enables HOL4 users to benefit from Squolem's automation. QBF can simply be passed from the HOL4 system to Squolem. If Squolem proves that the QBF is valid, our method then derives it as a theorem in HOL4. Second, our work provides high correctness assurances for Squolem's results; in fact, we uncovered a soundness bug in an earlier version of Squolem's certificate validator QBV. Due to HOL4's LCF-style architecture, our proof checker cannot draw unsound inferences (provided HOL4's kernel is correct). Thus, it can be used for QBF benchmark certification.

One could extend this work to other QBF solvers (see [23] for an overview), and to other interactive theorem provers, e.g., Isabelle or Coq. Because seemingly minor differences in kernel data structures can have significant effects, it is not clear whether similar performance can be achieved in these systems.

An alternative approach that might yield better performance than the LCF-style implementation presented in this paper is the use of reflection [41], i.e., implementing and proving correct a checker for Squolem's certificates in the prover's logic, and then executing the verified checker without producing proofs. While this approach still provides relatively high correctness assurances, obtaining a theorem in HOL4 would require enhancing the inference kernel with a reflection rule that allows us to trust the result of such a verified computation.

Acknowledgments. The authors would like to thank Christoph Wintersteiger for assistance with Squolem and QBV.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS 1999. Volume 1579 of LNCS., Springer (1999) 193–207
2. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: Proc. CAV 2004. Volume 3114 of LNCS., Springer (2004) 47–49
3. Hanna, Z., Dershowitz, N., Katz, J.: Bounded model checking with QBF. In: Proc. SAT 2005. Volume 3569 of LNCS., Springer Verlag (2005) 408–414
4. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. JSAT **5**(1–4) (2008) 133–191
5. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proc. 5th Annual ACM Symp. on Theory of Computing. (1973) 1–9
6. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and certifying QBFs: A comparison of state-of-the-art tools. AI Communications **22**(4) (2009) 191–210
7. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Proc. SAT 2007. Volume 4501 of LNCS., Springer (2007) 201–214
8. Slind, K., Norrish, M.: A brief overview of HOL4. [42] 28–32
9. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems. Volume 2 of Real-Time Safety Critical Systems Series. Elsevier (1994) 49–70
10. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. Volume 78 of LNCS. Springer (1979)
11. Gordon, M.: From LCF to HOL: a short history. In: Proof, language, and interaction: essays in honour of Robin Milner. MIT Press (2000) 169–185
12. Bertot, Y.: A short presentation of Coq. [42] 12–16
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. [42] 33–38
14. Owre, S., Shankar, N.: A brief overview of PVS. [42] 22–27
15. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic **7**(1) (2009) 26–40
16. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: A preliminary report. In: 6th International Workshop on Satisfiability Modulo Theories (SMT 2008). (2008)
17. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Proc. ITP 2010. Volume 6172 of LNCS., Springer (2010) 179–194
18. Kumar, R., Kropf, T., Schneider, K.: Integrating a first-order automatic prover in the HOL environment. In Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J., eds.: Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, IEEE Computer Society (1992) 170–176
19. Hurd, J.: An LCF-style interface between HOL and first-order logic. In: Proc. CADE-18. Volume 2392 of LNCS., Springer (2002) 134–138
20. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning **40**(1) (2008) 35–60
21. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Automated Reasoning with Analytic Tableaux and Related Methods. Volume 2381 of LNCS., Springer (2002) 5–15

22. Pulina, L., Tacchella, A.: Learning to integrate deduction and search in reasoning about quantified boolean formulas. In: Proc. FroCoS 2009. Volume 5749 of LNCS., Springer (2009) 350–365
23. Narizzano, M., Pulina, L., Tacchella, A.: Report of the third QBF solvers evaluation. JSAT **2**(1–4) (2006) 145–164
24. Weber, T.: Validating QBF invalidity in HOL4. In: Proc. ITP 2010. Volume 6172 of LNCS., Springer (2010) 466–480
25. Büning, H.K., Karpinski, M., Flögel, A.: Resolution for quantified boolean formulas. Information and Computation **117**(1) (1995) 12–18
26. Amjad, H.: Combining model checking and theorem proving. Technical Report UCAM-CL-TR-601, University of Cambridge Computer Laboratory (2004)
27. Ballarin, C.: Computer algebra and theorem proving. Technical Report UCAM-CL-TR-473, University of Cambridge Computer Laboratory (1999)
28. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. In: Proc. MKM/Calculemus 2009. Volume 5625 of LNCS., Springer (2009) 59–74
29. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. J. Autom. Reasoning **44**(3) (2010) 175–205
30. : QDIMACS standard version 1.1 (2005) Released on December 21, 2005. Retrieved February 20, 2011 from http://www.qbflib.org/qdimacs.html.
31. : DIMACS satisfiability suggested format (1993) Retrieved February 20, 2011 from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc.
32. Kroening, D., Wintersteiger, C.M.: A file format for QBF certificates (2007) Retrieved February 20, 2011 from http://www.cprover.org/qbv/download/qbcformat.pdf.
33. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5** (1940) 56–68
34. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL. (1982) 207–212
35. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT 2003. Volume 2919 of LNCS., Springer (2003) 502–518
36. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML – Revised. MIT Press (1997)
37. Barras, B.: Programming and computing in HOL. In: Proc. TPHOLs 2000. Volume 1869 of LNCS., Springer (2000) 17–37
38. HOL4 contributors: HOL4 Kananaskis 6 source code (2011) Retrieved February 20, 2011 from http://hol.sourceforge.net/.
39. Büning, H.K., Zhao, X.: On models for quantified Boolean formulas. In Lenski, W., ed.: Logic versus Approximation. Volume 3075 of LNCS., Springer (2004) 18–32
40. Harrison, J.: Binary decision diagrams as a HOL derived rule. The Computer Journal **38** (1995) 162–170
41. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge (1995) Retrieved February 20, 2011 from http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz.
42. Mohamed, O.A., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. Volume 5170 of LNCS., Springer (2008)