Fine-grained SMT proofs for the theory of fixed-width bit-vectors *

Liana Hadarean[1], Clark Barrett[2], Andrew Reynolds[3],
Cesare Tinelli[4], and Morgan Deters[2]

[1] Oxford University
[2] New York University
[3] EPFL
[4] The University of Iowa

Abstract. Many high-level verification tools rely on SMT solvers to efficiently discharge complex verification conditions. Some applications require more than just a yes/no answer from the solver. For satisfiable quantifier-free problems, a satisfying assignment is a natural artifact. In the unsatisfiable case, an externally checkable proof can serve as a certificate of correctness and can be mined to gain additional insight into the problem. We present a method of encoding and checking SMT-generated proofs for the quantifier-free theory of fixed-width bit-vectors. Proof generation and checking for this theory poses several challenges, especially for proofs based on reductions to propositional logic. Such reductions can result in large resolution subproofs in addition to requiring a proof that the reduction itself is correct. We describe a fine-grained proof system formalized in the LFSC framework that addresses some of these challenges with the use of computational side-conditions. We report results using a proof-producing version of the CVC4 SMT solver on unsatisfiable quantifier-free bit-vector benchmarks from the SMT-LIB benchmark library.

1 Introduction

SMT solvers are often used to reason in theories whose satisfiability problem ranges in complexity from NP-complete to undecidable. To be able to do this, they implement complex algorithms combining efficient SAT solving with theory-specific reasoning, requiring many lines of highly optimized code.[5] Because the solvers' code base changes frequently to keep up with the state of the art, bugs are still found in mature tools: during the 2014 SMT competition, five SMT solvers returned incorrect results. In a field where correctness is paramount, this is particularly problematic. While great progress has been made in verifying complex software systems [18, 19], the verification of SAT and SMT solvers still remains a challenge [20].

One approach for addressing this concern is to instrument an SMT solver to emit a certificate of correctness. If the input formula is satisfiable and quantifier-free, a natural certificate is a satisfying assignment to its variables. Correctness can be checked

* Work partially supported by DARPA award FA8750-13-2-0241 and ERC project 280053 (CPROVER).

[5] For example, the CVC4 code base consists of over 250K lines of C++ code.

by evaluating the input formula under that assignment. In the unsatisfiable case, the solver could emit an externally-checkable proof of unsatisfiability. Proof checkers usually consist of a small trusted core that implements a set of simple rules. These can be composed to prove complex goals, while maintaining trustworthiness.

Proof-producing SMT solvers have been successfully used to improve the performance of sceptical proof assistants, as shown in several recent papers [1, 5, 6, 8, 9, 14]. The proof assistant can discharge complex sub-goals to the SMT solver. It can then check or reconstruct the proof returned by the solver without having to trust the result. In some applications, such as interpolant generation [26] and certified compilation [11], the proof object itself is used for more than just establishing correctness.

Proofs for the theory of fixed-width bit-vectors are of particular practical importance, with applications in both hardware and software verification. Previous work [7] shows how to reconstruct proofs from the Z3 SMT solver in HOL4 and Isabelle/HOL. However, due to the lack of detail in the Z3 bit-vector proofs, proof reconstruction is not always successful. In this paper, we seek to address this limitation by presenting a method of encoding and checking fine-grained SMT-generated proofs for the theory $\mathcal{T}_{\mathsf{bv}}$ of bit-vectors as formalized in the SMT-LIB 2 standard [3]. Proof generation and checking for the bit-vector theory poses several unique challenges. Algebraic reasoning is typically not sufficient by itself to decide most bit-vector formulas of practical interest, so often bitvector (sub)-problems are solved by reduction to SAT. However, such reductions usually result in very large propositional proofs. In addition, the reduction itself must be proven correct. LFSC is a meta-logic that was specifically designed to serve as a unified proof format for SMT solvers. Encoding the $\mathcal{T}_{\mathsf{bv}}$ proof rules in LFSC helps address some of these challenges.

We make the following contributions: (i) we develop an LFSC proof system for the quantifier-free theory of fixed-width bit-vectors that includes proof rules for bit-blasting and allows for a two-tiered DPLL($\mathcal{T}$) proof structure; (ii) we instrument the CVC4 SMT solver to output proofs in this proof system; and (iii) we report experimental results on an extensive set of unsatisfiable SMT-LIB benchmarks in the QF_BV logic.

We start with a discussion of related work in Section 2. Section 3 explains the structure of SMT-generated proofs, while Section 4 introduces the LFSC proof language and illustrates how to use it to encode the kinds of inferences routinely done by SMT solvers. We discuss how bit-vector constraints are decided in CVC4 and how to generate proofs for them in Section 5. Section 6 introduces the LFSC proof rules that are specific to the bit-vector theory. We show experimental results in Section 7 and conclude with future work in Section 8.

2 Related Work

Early approaches to proof-checking for SMT relied on using interactive theorem provers to certify proofs produced by SMT solvers. One effort [21] used HOL Light to certify proofs generated by the CVC Lite SMT solver. Another [13] generated proofs for quantifier-free problems in the logic of equality with uninterpreted symbols using the haRVey SMT solver and translated these into Isabelle/HOL. A contrasting approach [22] traded off assurance for speed by using a special-purpose external checker

to check proofs generated by the Fx7 solver. Our approach aims to balance trust and efficiency by using LFSC. Using a logical framework with a generic proof checker provides both trust and flexibility, while LFSC's computational side-conditions increase performance.

None of the work mentioned above supports proofs for the theory of bit-vectors. The work in [15] targets SMT-generated proofs for the theory of bit-vectors for the purpose of generating interpolants. It is similar to ours in that it uses a lazy bit-vector solver, integrated into a DPLL($\mathcal{T}$) framework and in that if algebraic reasoning fails, it falls back on a resolution proof generated by the SAT solver. However, the work is different in that its focus is on producing interpolants rather than proof-checking. They do not address the correctness of bit-blasting, for instance.

The work whose scope is most similar to ours is an effort that was undertaken to reconstruct bit-vector proofs produced by Z3 within Isabelle/Hol [7]. The main difference in that work is that Z3 does not produce full proofs, but rather "proof sketches." Specifically, Z3 provides some "large-step" inferences, lemmas that are valid in the theory of bit-vectors, without proof. As the authors remark, the coarse granularity of Z3's proofs makes proof reconstruction particularly challenging. A significant part of the proof checking time is spent re-proving large-step inferences that Z3 does not provide details for. In contrast, our approach is more fine-grained as it provides full details for every step. As we show below, this enables our approach to check more proofs.

The LFSC meta-framework has been successfully used for encoding proofs generated by SMT solvers for other theories in [24, 25, 28]. The current paper extends this line of work to support LFSC proofs for the bit-vector theory. In [26] the authors show how to use LFSC to compute interpolants from unsatisfiability proofs in the theory of equality and uninterpreted function symbols. We believe this approach can be extended to generate bit-vector interpolants from LFSC bit-vector proofs.

3 Proofs in SMT

In the rest of the paper, we assume some familiarity with automated reasoning, many-sorted first-order logic, and the syntax of simply-typed lambda calculus. Let $\_^{\mathrm{P}}$ be an *abstraction* operator that replaces each atom (a predicate symbol applied to one or more terms) in a formula with a unique propositional variable. Most SMT solvers are based on some variant of the DPLL($\mathcal{T}$) architecture [23], which combines Boolean reasoning on the abstraction φ^{P} of a quantifier-free input formula φ with theory-specific reasoning in order to determine the satisfiability of φ with respect to a background theory $\mathcal{T}$.[6] Boolean reasoning on φ^{P} is performed by a SAT solver, while theory-specific reasoning is delegated to a *theory solver* for $\mathcal{T}$ (or $\mathcal{T}$-solver). The SAT solver enumerates satisfying assignments A^{P} for φ^{P}. The $\mathcal{T}$-solver checks whether the corresponding set of $\mathcal{T}$-literals A is $\mathcal{T}$-satisfiable. If A is not $\mathcal{T}$-satisfiable, a $\mathcal{T}$-valid clause is added that blocks the assignment A^{P}, and the process continues until either a satisfying assignment is found or a contradiction can be derived purely at the propositional level. From a proof-theoretic perspective, one can think of the $\mathcal{T}$-solver as refining the propositional

[6] For simplicity, we will ignore here the issue of whether the background theory is the combination of several more basic theories or not.

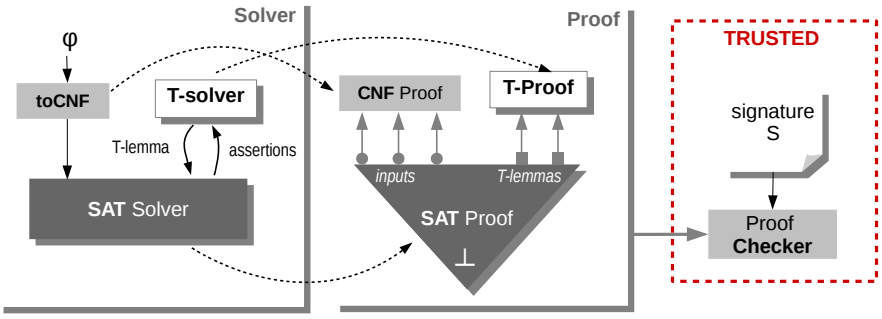


Fig. 1: DPLL($\mathcal{T}$) architecture, SMT proof structure, and proof checker.

abstraction φ^{P} with the addition of selected *theory lemmas* (clauses valid in $\mathcal{T}$) until a propositionally unsatisfiable formula is obtained [4].

The *resolution* calculus is refutationally complete for propositional clause logic [27] and has been successfully used as the basis for a common proof format for SAT solvers [30]. However, as we describe below, SMT proofs are significantly more sophisticated than SAT proofs (see, e.g., [2] for more details). First, SMT solvers convert their input to CNF; thus, a proof object produced by an SMT solver must incorporate a proof establishing that the CNF clauses used internally by the solver follow from the input formula. Second, the Boolean abstraction of the input formula is obtained by replacing $\mathcal{T}$-atoms with propositional variables. Hence, SMT proof generation must also rely on a mechanism that maintains a connection between input atoms and the propositional variables representing them in the SAT solver. Finally, each theory lemma generated by the theory solver must have a proof expressed in terms of $\mathcal{T}$-specific proof rules.

As a consequence, SMT proofs typically have a three-tiered structure: (i) a derivation of the internal CNF formula ψ from the input formula φ;[7] (ii) a resolution refutation of ψ in the form of a resolution tree whose root is the empty clause and whose leaves are either clauses from ψ or theory lemmas; and (iii) theory proofs of all the theory lemmas occurring in the resolution tree.

Figure 1 depicts the DPLL($\mathcal{T}$) architecture and how it relates to the structure of SMT proofs. In this paper, we consider proofs with this structure expressed as terms in the LFSC framework, which we discuss next.

4 LFSC

LFSC is an extension of the Edinburgh Logical Framework (LF) [17], a meta-framework based on a extension of simply-typed lambda calculus with dependent types. LF has been used extensively to encode various kinds of deductive systems. In general, a specific proof system P can be defined in LF by representing its proof rules as LF constants

[7] This step typically also includes the application of simplifying rewrite rules, which we ignore in this paper. Extending the approach here to include the many pre-processing rewrite rules used in real solvers is tedious but straightforward.

and encoding their premises and conclusions as a type. In this setting, a formal proof in the encoded proof system is represented as an LF term whose constants (in the sense of higher-order logic) are proof-rule names. A collection of type and term constant declarations is called a *signature* in LF. Checking the correctness of a proof then reduces to type checking: an LF proof checker takes as input both a signature S defining a proof system P and a proof term t encoding a proof in P. It verifies the correctness of the proof by checking that t is well-typed with respect to S. For example, the equality transitivity proof rule:

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \mathsf{trans} \tag{1}$$

in (unsorted) first-order logic can be encoded in LF as a constant with type:

$$\mathsf{trans} : \Pi t_1, t_2, t_3{:}\mathsf{tr}.\ \Pi p_1{:}\mathsf{holds}\,(\mathsf{eq}\ t_1\ t_2).\ \Pi p_2{:}\mathsf{holds}\,(\mathsf{eq}\ t_2\ t_3).\ \mathsf{holds}\,(\mathsf{eq}\ t_1\ t_3) \tag{2}$$

where Π is the binder for the dependently typed product, tr is the type of first-order terms, eq is a binary function of type $\mathsf{tr} \times \mathsf{tr} \to \mathsf{form}$ (where form is the type of first-order formulas), and holds is a unary (dependent) type parametrized by a first-order formula.[8] As a proof constructor, the proof rule (1) takes as arguments terms t_1, t_2 and t_3, as well as proofs p_1 of $t_1 = t_2$ and p_2 of $t_2 = t_3$, and returns a proof of $t_1 = t_3$. The LF declaration in (2) encodes this in the type of the constant trans. One possible proof that $a = d$ follows from the premises $a = b$, $b = c$, and $c = d$ is represented by the (well-typed) term:

$$\begin{array}{l}\lambda a, b, c, d{:}\mathsf{term}.\ \lambda p_1{:}\mathsf{holds}\,(\mathsf{eq}\ a\ b).\ \lambda p_2{:}\mathsf{holds}\,(\mathsf{eq}\ b\ c).\ \lambda p_3{:}\mathsf{holds}\,(\mathsf{eq}\ c\ d).\\ \quad (\mathsf{trans}\ a\ c\ d\ (\mathsf{trans}\ a\ b\ c\ p_1\ p_2)\ p_3)\end{array}$$

Using the wild-card symbol _, the body of the innermost lambda term can be simplified to $(\mathsf{trans}\ \_\ \_\ \_\ (\mathsf{trans}\ \_\ \_\ \_\ p_1\ p_2)\ p_3)$, since the omitted arguments can be inferred automatically during type-checking.

Purely declarative proof systems like those defined in LF cannot always efficiently model the kind of complex reasoning usually employed by SMT solvers. LFSC addresses this issue by extending LF types with computational *side conditions*, explicit computational checks defined as programs in a small but expressive functional first-order programming language. The language has built-in types for arbitrary precision integers and rationals, ML-style pattern matching over LFSC type constructors, recursion, limited support for exceptions, and a very restricted set of imperative features. A proof rule in LFSC may optionally include a side condition written in this language. When checking the application of such a proof rule, an LFSC checker computes actual parameters for the side condition and executes its code. If the side condition fails, the LFSC checker rejects the rule application.

As shown in Figure 1, when using LFSC, the trusted core includes both the (generic) LFSC checker and the specific LFSC signature which consists of a set of proof rules, each of which may have side conditions.

[8] Intuitively, an LF expression of dependent type $\Pi\varphi{:}\mathsf{form}.\ \mathsf{holds}(\varphi)$ represents a proof that the formula φ holds.

$$\mathsf{unit},\ \mathsf{var},\ \mathsf{lit},\ \mathsf{clause} : \mathsf{type} \qquad \mathsf{holds} : \mathsf{clause} \to \mathsf{type} \qquad \mathsf{cln} : \mathsf{clause}$$

$$\mathsf{ok} : \mathsf{unit} \qquad \mathsf{pos},\ \mathsf{neg} : \mathsf{var} \to \mathsf{lit} \qquad \mathsf{clc} : \mathsf{lit} \to \mathsf{clause} \to \mathsf{clause}$$

$$\mathsf{resolve}\ (c_1, c_2{:}\mathsf{clause}, v{:}\mathsf{var}){:}\mathsf{clause} = \mathsf{let}\ p\ (\mathsf{pos}\ v)\ \mathsf{in}\ \mathsf{let}\ n\ (\mathsf{neg}\ v)\ \mathsf{in}$$
$$\mathsf{let}\ \_\ (\mathsf{occurs}\ p\ c_1)\ \mathsf{in}\ \mathsf{let}\ \_\ (\mathsf{occurs}\ n\ c_2)\ \mathsf{in}\ \mathsf{merge}\ (\mathsf{remove}\ p\ c_1)\ (\mathsf{remove}\ n\ c_2)$$

$$\mathsf{Res} : \Pi c, c_1, c_2{:}\mathsf{clause}.\ \mathsf{holds}\ c_1 \to \mathsf{holds}\ c_2 \to \Pi v{:}\mathsf{var}\ \{(\mathsf{resolve}\ c_1\ c_2\ v) \downarrow\ c\}.\ \mathsf{holds}\ c$$

Fig. 2: LFSC declarations encoding propositional resolution.

We refer the reader to [28] for a detailed description of the LFSC language and its formal semantics. Here we introduce LFSC syntax via examples to illustrate the main features of the framework.

Example 1. An inference rule at the heart of SAT and SMT solvers is the propositional resolution rule:

$$\frac{l_1 \vee \ldots l_n \vee l \quad \neg l \vee l'_1 \vee \ldots l'_m}{l_1 \vee \ldots l_n \vee l'_1 \vee \ldots \vee l'_m}\ \mathsf{Res}$$

where l's are literals. This rule alone is actually not enough to express resolution derivations as formal objects, since one also has to account for the associativity, commutativity and idempotency of the $\vee$ operator. In LF, this problem can be addressed only by adding additional proof rules for those properties. Doing so makes it possible to move literals around in a clause and remove duplicate literals, but at the cost of requiring many proof rules for each resolution step, resulting in the generation of very large proofs. Alternative solutions [31] eschew the generic, declarative approach provided by meta-frameworks like LF and instead hard-code the clause data structure in the proof checker, requiring a proof-checker with higher complexity and lower generality.

In contrast, an LFSC proof rule for resolution can use a side condition to encode that the resulting clause is computed by removing the complementary literals in the two input clauses and then merging the remaining literals. One encoding of the rule and its side condition, together with all the necessary types and constants, is shown in Figure 2. In the figure and in the remainder of the paper, we write $\tau_1 \to \tau_2$ to abbreviate as usual a type of the form $\Pi x{:}\tau_1.\ \tau_2$ where τ_2 contains no occurrences of x. Clauses are encoded essentially as nil-terminated lists of literals. They are built with the constructors cln, for the empty clause, and clc, for non-empty clauses. Literals are built from propositional variables using the constructors pos and neg, for positive and negative literals. Variables do not have constructors because LFSC variables can be used directly.

The resolution rule Res takes as input the clauses c_1, c_2, and c, together with a proof of c_1 of type holds c_1, one of c_2 of type holds c_2, and a variable v to be used as the resolved atom. The resolve side condition function computes the resolvent of clause c_1 with c_2, provided that c_1 contains at least one occurrence of the positive literal $(\mathsf{pos}\ v)$ and c_2 contains at least one occurrence of the negative literal $(\mathsf{neg}\ v)$. The side condition $\{(\mathsf{resolve}\ c_1\ c_2\ v) \downarrow\ c\}$ succeeds if c is the result of resolving c_1 and c_2 on v. In that case, the proof rule returns a proof of c. The definitions of the auxiliary functions occurs, remove, and merge are omitted from Figure 2 due to space constraints. $(\mathsf{occurs}\ l\ c)$ does nothing if the literal l is in the clause c; otherwise, it raises a failure exception;

(remove l c) returns the result of removing the literal l from the clause c; (merge c_1 c_2) returns the clause with no repeated literals resulting from merging clauses c_1 and c_2.
□

LFSC has previously been successfully used to encode the constructs necessary for Boolean resolution, CNF conversion, and propositional abstraction of theory lemmas [28]. In this paper, we will not cover these constructs, but instead focus on how to encode bit-vector specific reasoning in LFSC.

5 Bit-vector proof generation in CVC4

Decision procedures for the theory $\mathcal{T}_{\mathsf{bv}}$ of bit-vectors almost always involve a reduction to propositional logic. One approach for encoding a bit-vector formula φ into an equisatisfiable propositional formula φ^{BB} is known as *bit-blasting*. For each variable v denoting a bit-vector of size n, bit-blasting introduces n fresh propositional variables, $v_0, \ldots v_{n-1}$, to represent each bit in the vector. To be able to encode this mapping in $\mathcal{T}_{\mathsf{bv}}$, we extend the $\mathcal{T}_{\mathsf{bv}}$ signature with a family of interpreted predicate symbols $(\mathsf{bitOf}_i : \mathsf{BV}_n \mapsto \mathsf{bool})_{0 \leq i < n}$, where bitOf_i takes a bit-vector x of width n and returns *true* iff the i^{th} bit of x is 1. Let φ be a bit-vector formula. For each atom a appearing in φ, let $bbAtom(a)$ denote a propositional formula consisting of the circuit representation of a. Let C^{BB} denote the conjunction of bit-blasting clauses obtained from converting to CNF the atom definitions:

$$C^{BB} \equiv CNF\left(\bigwedge_{a \in Atoms(\varphi)} a^{BB} \Leftrightarrow bbAtom(a)\right),$$

where a^{BB} is a fresh propositional variable representing atom a and CNF represents conversion to CNF. The formula $\varphi^{BB} := \varphi[a \mapsto a^{BB}]_{a \in Atoms(\varphi)} \wedge C^{BB}$ is a propositional formula equisatisfiable with φ. Most state-of-the-art solvers for $\mathcal{T}_{\mathsf{bv}}$ generate a formula like φ^{BB} and then rely on a single query to a SAT solver to check its satisfiability. Thus, a proof of unsatisfiability for φ could consist of: (i) a proof that φ is equisatisfiable with φ^{BB} in $\mathcal{T}_{\mathsf{bv}}$, (ii) a propositional proof that φ^{BB} is equisatisfiable with $CNF(\varphi^{BB})$, and (iii) a monolithic, potentially very large, resolution-based refutation of $CNF(\varphi^{BB})$.

CVC4 incorporates an *eager* bit-vector decision procedure (cvcE) based on the approach sketched above. It also provides, as an alternative, a lazy DPLL($\mathcal{T}$)-style bit-vector solver (cvcLz) that maintains the word-level structure of the input terms and separates reasoning over the propositional structure of the input formula φ from bit-vector term reasoning [16]. In cvcLz, the bit-vector theory is treated like any other theory: the main DPLL($\mathcal{T}$) SAT engine $\mathrm{SAT}_{\mathsf{main}}$ reasons on the propositional abstraction φ^{P} whereas a $\mathcal{T}_{\mathsf{bv}}$-solver BV decides conjunctions A of $\mathcal{T}_{\mathsf{bv}}$-literals. Essentially, BV corresponds to the $\mathcal{T}$-solver box in the DPLL($\mathcal{T}$) diagram in Figure 1.

Recall from Section 3 that the $\mathcal{T}_{\mathsf{bv}}$ solver BV must *repeatedly* decide the satisfiability of the $\mathcal{T}_{\mathsf{bv}}$-literals A and return a $\mathcal{T}_{\mathsf{bv}}$-valid clause over the atoms of A if A is $\mathcal{T}_{\mathsf{bv}}$-unsatisfiable. We achieve this by relying on a second SAT solver, $\mathrm{SAT}_{\mathsf{bb}}$, to decide the satisfiability of each assignment A. It does this by checking the propositional formula $A^{BB} \wedge C^{BB}$, where $A^{BB} = A[a \mapsto a^{BB}]_{a \in Atoms(A)}$. Note that this may be

significantly smaller than the formula $\varphi[a \mapsto a^{BB}]_{a \in Atoms(\varphi)} \wedge C^{BB}$ checked in the eager approach.

If $A^{BB} \wedge C^{BB}$ is unsatisfiable, SAT_{bb} returns a set of literals $L^{BB} \subseteq A^{BB}$ that is inconsistent with C^{BB}. The clause $\neg L$ is a $\mathcal{T}_{\mathsf{bv}}$-valid lemma, and the $\neg L^{\mathrm{P}}$ clause is added to $\text{SAT}_{\mathsf{main}}$. We can efficiently use SAT_{bb} to check the satisfiability of C^{BB} with different assumptions A^{BB} by using the *solve with assumptions* feature of SAT solvers [12].

The lazy solver cvcLz in CVC4 also has several algebraic word-level sub-solvers. However, we do not yet support proof production for these sub-solvers, so in this paper, we focus on the $\mathcal{T}_{\mathsf{bv}}$-lemmas generated by SAT_{bb}.

6 LFSC Bit-vector signature

In this section, we discuss proof generation for the lazy bit-vector solver cvcLz described in Section 5. Figure 3 shows the overall structure of the $\mathcal{T}_{\mathsf{bv}}$ proof by zooming in on the $\mathcal{T}_{\mathsf{bv}}$-lemmas that occur as leaves in the resolution SAT proof in Figure 1. We start with the bit-blasting proofs that each atom a is equivalent to its bit-blasted formula: $a \Leftrightarrow bbAtom(a)$. These proofs require no assumptions as $a \Leftrightarrow bbAtom(a)$ is $\mathcal{T}_{\mathsf{bv}}$-valid. [9] Next, the CNF proof establishes that the bit-blasting clauses C^{BB} follow from the atom definitions.[10] Note that this step also establishes the mapping from the $\mathcal{T}_{\mathsf{bv}}$-atom a to the abstract Boolean variable a^{BB} used in the SAT_{bb} SAT solver.

Each $\mathcal{T}_{\mathsf{bv}}$-lemma has a corresponding resolution proof in SAT_{bb} with C^{BB} as leaves. The resolution proof constructs a clause over the a^{BB} SAT variables. To use this in $\text{SAT}_{\mathsf{main}}$, we need to map the lemma to $\mathcal{T}_{\mathsf{bv}}$ atoms, and then to the SAT variables a^{P} in $\text{SAT}_{\mathsf{main}}$. In the figure, circles denote $\mathcal{T}_{\mathsf{bv}}$-atoms and diamonds the propositional variables that abstract them (either in SAT_{bb} or in $\text{SAT}_{\mathsf{main}}$).

6.1 Encoding bit-vector formulas

Figure 4 shows the LFSC constructs needed to represent formulas in the theory of bit-vectors. Note that the encoding distinguishes between formulas and terms: formulas are represented by the simple type form and terms by the dependent type term, parametrized by the sort of the term: Πs:sort. term s. Formulas are constructed with the usual logical operators and with an equality operator over terms which is parametric in the terms' sort. The int type is LFSC's own built-in infinite precision integer type. Bit-vector sorts are represented by the dependent type Πn:int. BV n where n is the width of the bit-vector. Bit-vector constants are represented as lists of bits using the constBV type with the two constructors bvn and bvc, for the empty sequence and the list cons operator respectively. The constBV bit-vector constants are converted to bit-vector terms with the const2BV function. Bit-vector variables are represented as LFSC variables of type varBV and converted to terms with var2BV.

[9] Recall that $bbAtom(a)$ is a propositional formula encoding the semantics of atom a, and contains bitOf_i applications on the bit-vector variables in a.

[10] For details on how to use LFSC to encode proofs for CNF conversion, see [28]

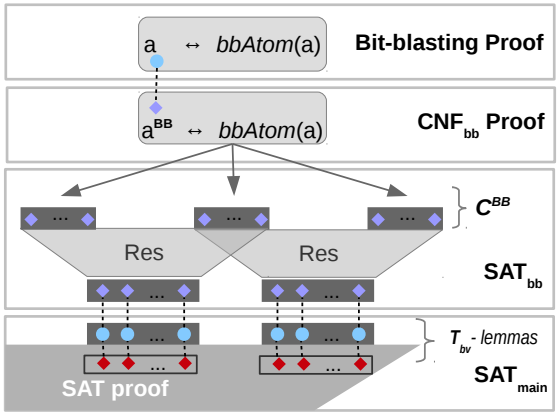


Fig. 3: Bit-vector proof structure.

sort : type	term : sort → type	BV : int → sort
form : type	true, false : form not : form → form	and, or, impl, iff : form → form → form = : Πs:sort. term s → term s → form
varBV : type	var2BV : Πn:int. varBV → term (BV n)	
bit : type	b0, b1 : bit	const2BV : Πn:int. constBV → term (BV n)
constBV : type	bvn : constBV	bvc : bit → constBV → constBV

Fig. 4: Partial LFSC signature for the theory $\mathcal{T}_{\mathsf{bv}}$ of bit-vectors.

Example 2. The bit-wise conjunction operator is encoded in LFSC as:

$$\mathsf{bvand} : \Pi n{:}\mathsf{int}.\ \mathsf{term}\ (\mathsf{BV}\ n) \to \mathsf{term}\ (\mathsf{BV}\ n) \to \mathsf{term}\ (\mathsf{BV}\ n)$$

Similarly, the unsigned comparison operator $<$ is encoded as:

$$\mathsf{bvult} : \Pi n{:}\mathsf{int}.\ \mathsf{term}\ (\mathsf{BV}\ n) \to \mathsf{term}\ (\mathsf{BV}\ n) \to \mathsf{form}$$

The $\mathcal{T}_{\mathsf{bv}}$ formula $(t_1 = t_2\ \&\ t_3) \vee (t_1 < 0_{[3]})$ where $\&$ is bvand, $0_{[3]}$ is the zero bit-vector of size 3, and t_1, t_2, t_3 have type (term (BV 3)) can be encoded in LFSC as

$$\begin{array}{l}(\mathsf{or}\ (=\ \_\ t_1\ (\mathsf{bvand}\ \_\ t_2\ t_3))\\ \quad (\mathsf{bvult}\ \_\ t_1\ (\mathsf{const2BV}\ 3\ (\mathsf{bvc}\ \mathsf{b0}\ (\mathsf{bvc}\ \mathsf{b0}\ (\mathsf{bvc}\ \mathsf{b0}\ \mathsf{bvn})))))),\end{array}$$

with b0 representing the zero bit. □

6.2 Bit-blasting

Recall that a bit-blasting proof (see Figure 3) makes the connection between a bit-vector formula and its propositional logic encoding by proving for each bit-blasted atom a in

$\mathsf{bbt} : \mathsf{type}$ $\quad$ $\mathsf{bbtn} : \mathsf{bbt}$ $\quad$ $\mathsf{bbtc} : \mathsf{formula} \to \mathsf{bbt} \to \mathsf{bbt}$

$\mathsf{bitOf} : \mathsf{varBV} \to \mathsf{int} \to \mathsf{form}$ $\quad$ $\mathsf{bbTerm} : \Pi n{:}\mathsf{int}.\, \mathsf{term}\ (\mathsf{BV}\ n) \to \mathsf{bbt} \to \mathsf{type}$

$\mathsf{bb\text{-}var}\ (v : \mathsf{varBV}, n : \mathsf{int}) : \mathsf{bbt} =$
$\quad$ *if* $n < 0$ *then* bbtn *else* $(\mathsf{bbtc}\ (\mathsf{bitOf}\ v\ n)\ (\mathsf{bb\text{-}var}\ v\ (n-1)))$

$\mathsf{bbVar} : \Pi n{:}\mathsf{int}.\, \Pi v{:}\mathsf{varBV}.$
$\quad \Pi vb{:}\mathsf{bbt}\,\{(\mathsf{bb\text{-}var}\ v\ (n-1)) \downarrow\ vb\}.\, (\mathsf{bbTerm}\ n\ (\mathsf{var2BV}\ n\ v)\ vb)$

$\mathsf{bbAnd} : \Pi n{:}\mathsf{int}.\, \Pi x, y{:}\mathsf{term}\ (\mathsf{BV}\ n).\, \Pi xb, yb, rb{:}\mathsf{bbt}.$
$\quad \Pi xbb{:}\mathsf{bbTerm}\ n\ x\ xb.$
$\quad \Pi ybb{:}\mathsf{bbTerm}\ n\ y\ xb\,\{(\mathsf{bb\text{-}bvand}\ xb\ yb) \downarrow\ rb\}.\, \mathsf{bbTerm}\ n\ (\mathsf{bvand}\ n\ x\ y)\ rb$

$\mathsf{bbEq} : \Pi n{:}\mathsf{int}.\, \Pi x, y{:}\mathsf{term}\ (\mathsf{BV}\ n).\, \Pi bx, by{:}\mathsf{bbt}.\, \Pi f{:}\mathsf{form}.$
$\quad \Pi bbx{:}\mathsf{bbTerm}\ n\ x\ bx.$
$\quad \Pi bby{:}\mathsf{bbTerm}\ n\ y\ by\,\{(\mathsf{bb\text{-}eq}\ bx\ by) \downarrow\ f\}.\, \mathsf{thHolds}\ (\mathsf{iff}\ (=\ (\mathsf{BV}\ n)\ x\ y)\ f)$

Fig. 5: Partial list of the LFSC bit-blasting rules for $\mathcal{T}_{\mathsf{bv}}$.

the input formula, the following formula:

$$a \Leftrightarrow bbAtom(a).$$

We represent a bit-blasted bit-vector term of width n as a sequence of n formulas, with the i^{th} formula in the sequence corresponding to the i^{th} bit. The bbt type encodes bit-blasted terms and has two type constructors bbtn and bbtc as shown in Figure 5. We introduce the dependent type constructor bbTerm to encode the fact that the bit-vector term $x{:}\mathsf{BV}\ n.$ corresponds to a bit-blasted term $y{:}\mathsf{bbt}$. For example, the following term encodes that $15_{[4]}$ is bit-blasted as $[true, true, true, true]$:

$$(\mathsf{bbTerm}\ \_\ (\mathsf{const2BV}\ 4\ (\mathsf{bvc}\ \mathsf{b1}\ (\mathsf{bvc}\ \mathsf{b1}\ (\mathsf{bvc}\ \mathsf{b1}\ (\mathsf{bvc}\ \mathsf{b1}\ \mathsf{bvn})))))$$
$$(\mathsf{bbtc}\ \mathsf{true}\ (\mathsf{bbtc}\ \mathsf{true}\ (\mathsf{bbtc}\ \mathsf{true}\ (\mathsf{bbtc}\ \mathsf{true}\ \mathsf{bbtn})))))$$

We can define proof rules for each piece of syntax in bit-vector terms and compose them in order to build up arbitrary bit-blasted terms. Figure 5 shows several such bit-blasting rules. The bbVar rule takes a bit-vector variable v, its width n, and a sequence of bit-blasted terms vb, and checks that the sequence computed by the side condition code in bb-var matches vb. The side condition code just builds a sequence of applications of the bitOf operator to v—with $(\mathsf{bitOf}\ v\ i)$ representing the $\mathcal{T}_{\mathsf{bv}}$ predicate bitOf_i introduced at the beginning of Section 5. Similarly, the rule that establishes how to bit-blast bit-wise conjunction ($\&$) takes a proof xbb that xb is the bit-blasted term corresponding to x as well as a proof ybb for yb corresponding to y and returns a proof that $x\&y$ is bit-blasted to rb. The rb term is constructed by the side condition code bb-bvand (not shown) which works similarly to bb-var. The bbEq rule for equality $\mathcal{T}_{\mathsf{bv}}$-atoms follows a similar pattern, but returns a formula instead of a bbTerm. Note that bit-blasting proof rules do not take any $\mathcal{T}_{\mathsf{bv}}$-assertions as assumptions: their conclusions are $\mathcal{T}_{\mathsf{bv}}$-valid.

Example 3. Encoding in LFSC the bit-blasting proof for the formula $a_{[8]} = x_{[8]} \& y_{[8]}$ requires the following proof rule applications:

$$(\mathsf{bbEq}\ \_\ \_\ \_\ \_\ \_\ \_\ (\mathsf{bbVar}\ 8\ a\ \_\ )\ (\mathsf{bbAnd}\ \_\ \_\ \_\ \_\ \_\ \_\ (\mathsf{bbVar}\ 8\ x\ \_\ )\ (\mathsf{bbVar}\ 8\ y\ \_\ )))$$

Assuming previously defined variables a, x, and y, the above term has type $\mathsf{thHolds}(\varphi)$ where φ is:

$$(a_{[8]} = x_{[8]} \& y_{[8]}) \Leftrightarrow \bigwedge_{0 \leq i < 8} (a_i \Leftrightarrow (\mathsf{bitOf}\ v\ i) \wedge (\mathsf{bitOf}\ v\ i))\,.$$

The bit-blasting LFSC proof rules rely on the side-condition code to build up the bit-blasted terms. This side-condition code thus becomes part of the trusted core and offers an efficient way to encode bit-blasting proofs.

6.3 Resolution in $\mathrm{SAT}_{\mathsf{bb}}$

A resolution refutation can be obtained from a SAT solver by instrumenting it to store resolution proofs of all the clauses learned during search. The empty clause is then derived by resolving input clauses and learned clauses. Recall that $\mathrm{SAT}_{\mathsf{bb}}$ uses "solve with assumptions" to identify a subset $L^{BB} \subseteq A^{BB}$ that is inconsistent with C^{BB} and thereby produce the theory lemma $\neg L$. Because the assumption literals are implemented as decisions in $\mathrm{SAT}_{\mathsf{bb}}$, all clauses learned in $\mathrm{SAT}_{\mathsf{bb}}$ follow from the bit-blasting clauses alone and can thus be reused in subsequent checks by $\mathrm{SAT}_{\mathsf{bb}}$. In particular, we can retrieve a resolution proof of the $\neg L^{BB}$ clause from $\mathrm{SAT}_{\mathsf{bb}}$ starting from the bit-blasting clauses C^{BB} and using the stored resolutions of the learned clauses. We are careful to reuse the resolution proofs of learned clauses in multiple $\mathcal{T}_{\mathsf{bv}}$ lemmas.

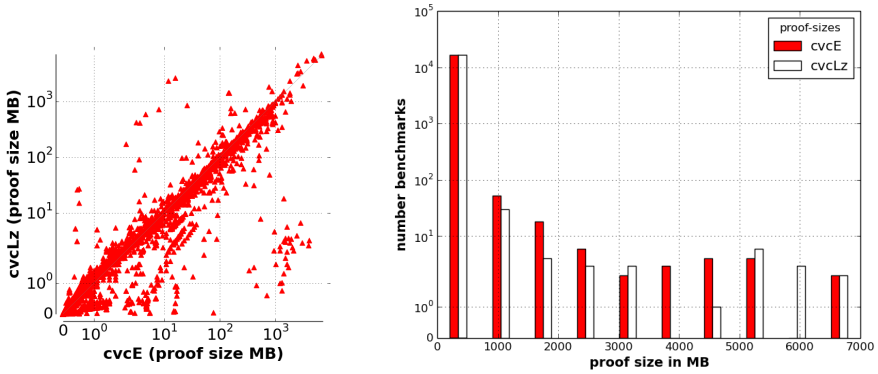
Stepping back and examining the overall $\mathcal{T}_{\mathsf{bv}}$ proof structure, it looks like we could obtain one big resolution proof if we could plug the $\mathrm{SAT}_{\mathsf{bb}}$ resolution trees into the $\mathrm{SAT}_{\mathsf{main}}$ resolution tree. However, this cannot be done directly as the SAT variable a^{BB} abstracting $\mathcal{T}_{\mathsf{bv}}$-atom a in the resolution proof in $\mathrm{SAT}_{\mathsf{bb}}$ is not the same as the a^{P} variable used to abstract the same atom in $\mathrm{SAT}_{\mathsf{main}}$. Therefore, we need a proof construct to map the proof of a clause c^{BB} to c^{P} (the dashed lines between $\mathrm{SAT}_{\mathsf{main}}$ and $\mathrm{SAT}_{\mathsf{bb}}$ in Figure 3).

In previous work on encoding SMT proofs in LFSC [28], we developed a specialized proof rule assump used to transform a $\mathcal{T}$-proof of $\bigwedge_{i=0}^{n} \neg l_i \models_{\mathcal{T}} \bot$ to a proof of the clause $c^{\mathrm{P}} = [l_1^{\mathrm{P}}, \ldots, l_n^{\mathrm{P}}]$ where we use the square brackets as a shorthand for the LFSC syntax for clauses. Chaining assump rules turns a term of type $\mathsf{thHolds}(\neg l_1) \rightarrow \ldots \rightarrow \mathsf{thHolds}(\neg l_n).\mathsf{holds}\ \mathsf{cln}$ into a term of type $\mathsf{holds}\ [l_1^{\mathrm{P}} \ldots l_n^{\mathrm{P}}]$. Our goal here is to build a proof that takes as assumptions the negation of each literal l_i as well as a proof of the clause $c^{BB} = [l_1^{BB}, \ldots, l_n^{BB}]$ and returns a term of type $\mathsf{holds}\ \mathsf{cln}$. We will do this using the $\mathsf{introUnit}$ rule: [11]

$$\begin{aligned}&\mathsf{introUnit} : \Pi f{:}\mathsf{form}.\, \Pi v{:}\mathsf{var}.\, \Pi c{:}\mathsf{clause}.\\ &\qquad \mathsf{thHolds}\ f \rightarrow \mathsf{atom}\ v\ f \rightarrow (\mathsf{holds}\ [v] \rightarrow \mathsf{holds}\ c) \rightarrow \mathsf{holds}\ c\end{aligned}$$

This natural deduction style rule states that if formula f holds ($\mathsf{thHolds}\ f$) and is abstracted by propositional variable v ($\mathsf{atom}\ v\ f$), and if we can derive clause c from the unit clause corresponding to f ($\mathsf{holds}\ [v] \rightarrow \mathsf{holds}\ c$), then we can derive clause c.

[11] For simplicity, $\mathsf{introUnit}$ only introduces literals in positive polarity. In reality, we also use a dual version that introduces literals in negative polarity.



(a) Size of generated proofs.

(b) Distribution of proof sizes.

Fig. 6: Proof sizes both cvcLz and cvcE

Example 4. We show how to put these rules together to lift a proof of a clause in SAT_{bb} to a proof of the corresponding clause in $\text{SAT}_{\mathsf{main}}$. In the sub-expression below, assume c has type holds $[\neg a_1^{BB}, \neg a_2^{BB}]$ and that at_1 and at_2 have types $\mathsf{atom}(a_1^{BB}, a_1)$ and $\mathsf{atom}(a_2^{BB}, a_2)$, respectively. The two resolution steps between the assumption unit clauses u_1 and u_2 derive the empty clause from c. Therefore, the computed type of the following term is $\mathsf{thHolds}(\mathsf{not}\ a_1) \to \mathsf{thHolds}(\mathsf{not}\ a_2) \to \mathsf{holds}\ \mathsf{cln}$, which is exactly what the assump rule requires:

$$\begin{array}{l}\lambda h_1 : \mathsf{thHolds}(\mathsf{not}\ a_1).\ \lambda h_2 : \mathsf{thHolds}(\mathsf{not}\ a_2).\\ \quad (\mathsf{introUnit}\ \_\ \_\ \_\ h_1\ at_1\ (\lambda u_1 : (\mathsf{holds}[a_1^{BB}]).\\ \qquad (\mathsf{introUnit}\ \_\ \_\ \_\ h_2\ at_2\ (\lambda u_2 : (\mathsf{holds}[a_2^{BB}]).\\ \qquad\quad (\mathsf{Res}\ \_\ \_\ (\mathsf{Res}\ \_\ \_\ c\ u_1\ v_1)\ u_2\ v_2)))))\end{array}$$

7 Experimental Results

All the experiments in this section were run on the StarExec [29] cluster infrastructure with a timeout of 600 seconds and a memory limit of 200GB.[12] We selected all of the 17,172 unsatisfiable QF_BV benchmarks used in the 2015 SMT-COMP competition and evaluated the overhead of proof generation for both the lazy cvcLz and the eager cvcE configurations of CVC4. CVC4 is a competitive bit-vector solver that placed second in the QF_BV division of the 2015 SMTCOMP by running cvcLz and cvcE in parallel.[13] The proof generated by cvcE uses the same proof signature as cvcLz but has

[12] Experiments were run on the queue all.q consisting of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz machines with 268 GB of memory.

[13] CVC4 solved 26001 problems in that division compared to 26260 problems solved by the winning solver, Boolector [10].

	default		+log			+log+proof			+log+proof+check		
	solved	time (s)	solved	time (s)	%	solved	time (s)	%	solved	time (s)	%
cvcLz	16665	38575	16663	43684	11	16662	43729	14	14063	118544	973
cvcE	16601	65009	16583	78187	19	16582	78256	22	13734	137931	737

Table 1: Overhead of proof generation and its impact on the number of problems solved.

a single monolithic resolution proof as opposed to the modular two-tiered structure of cvcLz proofs.

Table 1 shows the results for both solvers. We ran the following configurations: solving with proof generation disabled (default); solving with proofs enabled (i.e., the solver logs the information needed to produce the proof) but without actually producing proofs (+log); solving with proof generation including writing the proof object to disk (+log + proof); and solving with proof generation as well as proof checking (+log + proof + check). For the lazy solver cvcLz, the overhead of proof logging results in 2 fewer problems solved while adding an 11% overhead to solving time.[14] The additional overhead of stitching the proof together and outputting it to a file is only 3% of the solving time. For the eager solver cvcE, proof logging adds a higher overhead of 19% and solves 18 fewer problems than the default configuration of cvcE. The overhead of proof generation is higher for the eager solver than for the lazy one.

To ensure the correctness of the proofs we generated, we checked them using our LFSC proof checker. Within the 600 sec time limit, we were able to succesfully check 84% of the problems we could solve with cvcLz and 82% of the ones solved with cvcE. Proof checking failed due to unsupported proof steps in our generated proof for 33 problems attempted by cvcLz, and for 92 attempted by cvcE. The other failures in proof checking were due to timeouts: proof checking is an order of magnitude slower than solving. We believe that with additional work on the LFSC proof checker, this can be improved.

Despite the slow checking times, we achieve higher proof checking rates for QF_BV than the proof reconstruction approach in Böhme et al. [7]. In that work, proofs could be produced for 735 of the 1377 QF_BV benchmarks available at the time. Out of these, the produced proofs were successfully checked only for 38.5% of the total; 48.4% timed out and 13.1% produced errors. The authors attribute the timeouts to the long time taken to reprove large-step Z3 inferences. Our experimental results indicate that fine-granularity bit-vector proofs enable proof checking for a significantly larger number of problems.

Finally, we compared the sizes of the proof files generated. Figure 6(a) is a log-scale scatter plot comparing the sizes of the proofs generated by the two solvers. Overall, the proofs generated by the two-tiered lazy approach are smaller: adding the sizes of all the lazy generated proofs results in 276GB while for the eager solver it is 328GB. Figure 6(b) shows, with the y-axis in log-scale, the distribution of the proof sizes over the benchmark selection. The majority of the benchmarks have relatively small proofs, well under 1GB.

[14] Overhead in each column is measured by comparing the time taken to solve only those problems solved by *both* the default and the column configuration.

8 Conclusion and Future Work

We have discussed a fine-grained LFSC proof system for the quantifier-free theory of bit-vectors. Our proof system takes advantage of LFSC's support for side conditions to efficiently check large resolution proofs and proofs of bit-blasting-based encodings to SAT. Used in the context of a lazy bit-vector solver, this proof system allows for modular two-tiered proofs that are smaller and more efficiently checked than a monolithic resolution proof, as shown by our experimental evaluation on a large set of QF_BV benchmarks.

The two-tiered proofs have several additional advantages we plan to investigate further in future work. For instance, it simplifies proof generation in the combination of $\mathcal{T}_{\mathsf{bv}}$ with other theories and allows more compact proofs through the use of algebraic proof rules for $\mathcal{T}_{\mathsf{bv}}$ conflicts. In addition to SAT reasoning, cvcLz also incorporates several word-level sub-solvers that use algebraic reasoning and equation solving to identify word-level conflicts. These conflicts can be expressed using proof rules that are bit-width independent and do not require reasoning about the bit-blasted terms.

One of the trade-offs of using side condition code in LFSC rules is that it becomes part of the trusted core. For future work we plan to look at a systematic approach for verifying the correctness of proof rules and their side condition code with the aid of theorem proving assistants such as Coq or Isabelle/HOL. Furthermore, we plan to develop infrastructure to export LFSC proofs to these tools as a way to integrate SMT solvers into interactive theorem provers and increase their level of automation.

References

1. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs*. 2011.
2. C. Barrett, L. de Moura, and P. Fontaine. Proofs in satisfiability modulo theories. In *All about Proofs, Proofs for All*, pages 23–44. 2015.
3. C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2015.
4. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Conference on Computer Aided Verification*, 2002.
5. F. Besson, P.-E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In *Certified Programs and Proofs*. 2011.
6. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning*, 2013.
7. S. Böhme, A. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *Certified Programs and Proofs*. 2011.
8. S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving*, 2010.
9. T. Bouton, D. Caminha B. De Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In *Conference on Automated Deduction*, 2009.
10. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2009.
11. J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Programming Language Design and Implementation*, 2010.

12. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, 2004.
13. P. Fontaine, J. Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *In Tools and Algorithms for Construction and Analysis of Systems*, 2006.
14. Y. Ge and C. Barrett. Proof translation and SMT-LIB benchmark certification: A preliminary report. In *Workshop on Satisfiability Modulo Theories*, 2008.
15. A. Griggio. Effective word-level interpolation for software verification. In *Formal Methods in Computer-Aided Design*, 2011.
16. L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Conference on Computer Aided Verification*, 2014.
17. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 1993.
18. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles*, 2009.
19. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages*, 2006.
20. S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Theorem Proving in Higher Order Logics*, 2008.
21. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, 2006.
22. M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
23. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.
24. D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In *Workshop on Satisfiability Modulo Theories*, 2009.
25. A. Reynolds, L. Hadarean, C. Tinelli, Y. Ge, A. Stump, and C. Barrett. Comparing proof systems for linear real arithmetic with LFSC. In *Workshop on Satisfiability Modulo Theories*, 2010.
26. A. Reynolds, C. Tinelli, and L. Hadarean. Certified interpolant generation for EUF. In *Workshop on Satisfiability Modulo Theories*, 2011.
27. J. A. Robinson. *Logic: Form and Function: The Mechanization of Deductive Reasoning*. Elsevier, 1980.
28. A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 2013.
29. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: a cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, 2014.
30. A. Van Gelder. http://users.soe.ucsc.edu/ avg/ProofChecker/ProofChecker-fileformat.txt.
31. N. Wetzler, M. J. Heule, and W. A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing*. 2014.